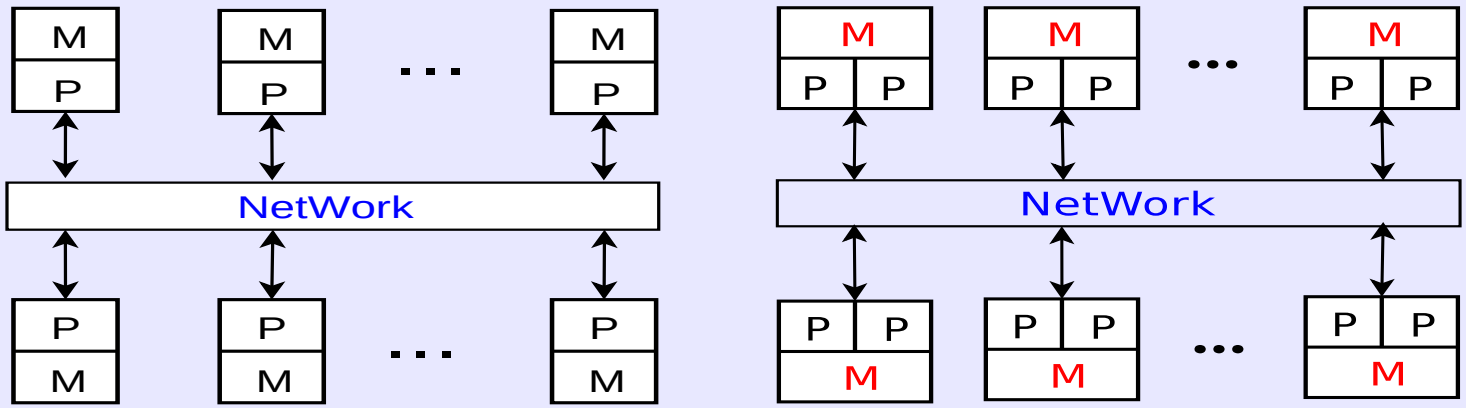
消息传递编程

崔 涛

tcui@lsec.cc.ac.cn

科学与工程计算国家重点实验室
中科院计算数学与科学工程计算研究所

MPI消息传递编程模型



(a) (b)

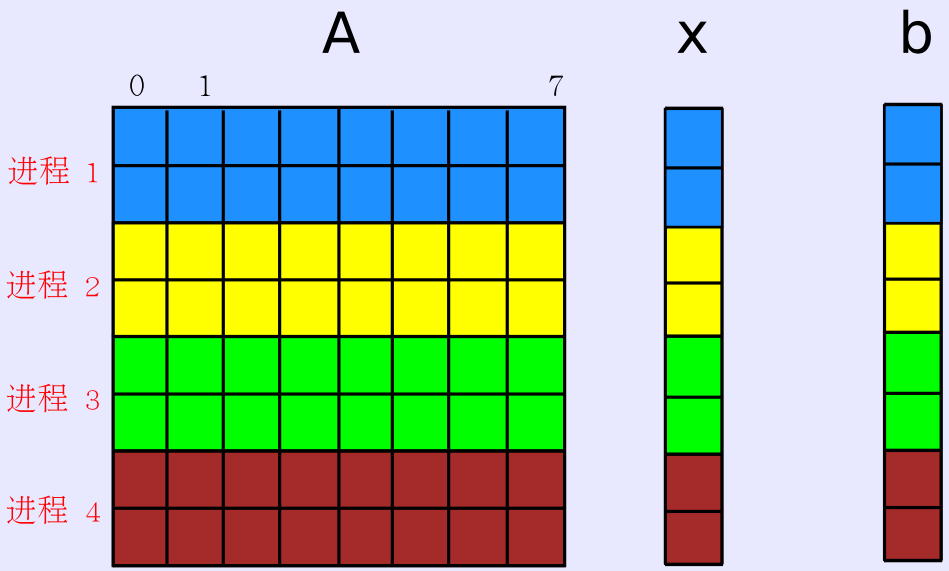
特点：

- 每个进程有独立的内存地址空间，数据分块独立存放；
- 编程复杂，需要显式并行操作；
- 可移植性强，可扩展性好。

分布式内存环境下的矩阵向量乘

$$Ax = b$$

$$b[i] = \sum_{j=0}^{7} A[i][j]x[j], i = 0, 1, \cdots, 7$$



MPI 是什么？

定义：

- 一个函数库，而非一种编程语言；
- 消息传递并行编程规范的代表；
- 消息传递编程模型。

特点：

- 很好的可移植性；
- 很好的可扩展性；
- 容易使用，提供明确、通用的函数接口；
- 完备的异步通信功能。

MPI历史 I

- MPI论坛 `http://www.mpi-forum.org`
  - MPI-1.0 1993.2，Dongarra, etc.
  - MPI-2 1997.7
  - MPI-3 2012.9
- MPI-1主要特征：
  - 点对点通信、聚合通信；
  - 定义新的数据类型（发送不连续的数据）；
  - 进程组、通信器及进程拓扑结构；
- MPI-2主要新增特征：
  - 动态进程管理；
  - 远程存储访问；

MPI历史 II

- 并行文件输入输出。

- MPI-3主要新增特征：
  - Topology Mapping(MPI-2.2)
  - Nonblocking and Neighborhood Collectives
  - Matched probe
  - MPI Tool Interface
  - New One Sided Functions and Semantics
  - New Communicator Creation Function
  - Impovements in Language Bindings
  - Fault Tolerance/Resiliency

MPI主要实现

Status of MPI-3 Implementations

	MPICH	MVAPICH	Cray	Tianhe	Intel	IBM PE	Open MPI	SGI	Fujitsu	MS
NB collectives	✔	✔	✔	✔	✔	Q1 '14	✔	✔		
Neighborhood collectives	✔	✔	✔	✔	✔	Q1 '14	Q4 '13 (in nightly snapshots)	Q1 '14		
RMA	✔	✔	✔	✔	✔	Q1 '14	Q1 '14	Q4 '13		
Shared memory	✔	✔	✔	✔	✔	Q1 '14	Q1 '14	Q4 '13		
Tools Interface	✔	Q1 '14	Q2 '14	Q1 '14	Q1 '14	Q1 '14	✔	Q1 '14		
Non-collective comm. create	✔	✔	✔	✔	✔	Q1 '14	Q4 '13 (in nightly snapshots)	✔		
F08 Bindings (Needs fixes to MPI-3)	(Q1 '14)	(Q2 '14)	(Q2 '14)	(Q1 '14)	(Q1 '14)	(Q1 '14)	(✔)	(Q1 '14)		
New Datatypes	✔	✔	✔	✔	✔	Q1 '14	✔	✔		
Large Counts	✔	✔	✔	✔	✔	Q1 '14	✔	Q1 '14		
Matched Probe	✔	✔	✔	✔	✔	Q1 '14	✔	✔		

Release dates are estimates and are subject to change at any time.
Empty cells indicate no *publicly announced* plan to implement/support that feature.

安装MPI（超级用户权限）

```
tar xzvf mpich-x.x.x.tgz
cd mpich-x.x.x
./configure --prefix=/usr/local/mpich-x.x.x \
            --with-comm=ch_p4 \
            --with-device=ch_p4 \
            --enable-sharedlib -rsh=ssh
make
make install
```

注意：

- 请参考文件：mpichman-chp4.pdf；
- 按需求确认C、C++、F77和F90编译器正确安装。

设置MPI（超级用户权限）

- 创建文件/etc/profile.d/user.sh，包含以下内容：

```
export PATH=${PATH}:/usr/local/mpi/bin
export MANPATH=${MANPATH}:/usr/local/mpi/man
```

- 创建文件/etc/ld.so.conf.d/mpi.conf，包含以下内容：

```
/usr/local/mpi/lib
/usr/local/mpi/lib/shared
```

- 执行以下命令，使上述文件生效：

```
source /etc/profile.d/user.sh
/sbin/ldconfig
```

设置SSH服务（普通用户权限）

在目录/home/*****/.ssh下执行以下命令，使得在运行并行程序时不用输入口令。

```
ssh-keygen -t dsa
cp id_dsa.pub authorized_keys
chmod go-rwx authorized_keys
ssh-agent $SHELL
ssh-add
```

一些名词与概念 I

程序与代码 我们这里说的程序不是指以文件形式存在的源代码、可执行代码等, 而是指为了完成一个计算任务而进行的一次运行过程.

进程(process) 一个MPI并行程序由一组运行在相同或不同计算机/计算结点上的进程或线程构成. 为统一起见, 我们将MPI程序中一个独立参与通信的个体称为一个进程. 在Unix系统中, MPI的进程通常是一个Unix 进程. 在共享内存/消息传递混合编程模式中, 一个MPI进程可能代表一组Unix线程.

进程组(process group) 指一个MPI程序的全部进程集合的一个有序子集. 进程组中每个进程被赋于一个在该组中唯一的序号(rank), 用于在该组中标识该进程. 序号的取值范围是0到进程数 -1.

一些名词与概念 II

通信器(communicator) 通信器(也有译成通信子的)是完成进程间通信的基本环境, 它描述了一组可以互相通信的进程以及它们之间的联接关系等信息. MPI的所有通信必须在某个通信器中进行. 通信器分域内通信器(intracommunicator) 和域间通信器(intercommunicator) 两类, 前者用于属于同一进程组的进程间的通信, 后者用于分属两个不同进程组的进程间的通信. 域内通信器由一个进程组和有关该进程组的进程间的拓扑联接关系构成.

MPI系统在一个MPI程序运行时会自动创建两个通信器, 一个称为`MPI_COMM_WORLD`, 它包含该MPI程序中的所有进程, 另一个称为`MPI_COMM_SELF`, 它指单个进程自己所构成的通信器. rank.c

一些名词与概念 III

序号(rank) 序号用来在一个进程组或通信器中标识一个进程. MPI程序中的进程由进程组/序号或通信器/序号所唯一确定. 序号是相对于进程组或通信器而言的: 同一个进程在不同的进程组或通信器中可以有不同的序号. 进程的序号是在进程组或通信器被创建时赋予的.

MPI系统提供了一个特殊的进程序号`MPI_PROC_NULL`, 它代表空进程(不存在的进程). 与`MPI_PROC_NULL`间的通信实际上没有任何作用.

消息(message) MPI程序中在进程间传送的数据称为消息. 一个消息由通信器、源地址、目的地址、消息标签和数据构成.

通信(communication) 通信指在进程之间进行消息的收发、同步等操作

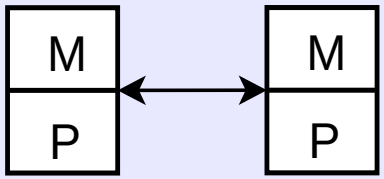
编程模式

SPMD编程模式 Single Program Multiple Data的缩写. 指构成一个程序的所有进程运行的是同一份可执行代码. 不同进程根据自己的序号可能执行该代码中的不同分支. 这是MPI编程中最常用的编程方式. 用户只需要编写、维护一份源代码.

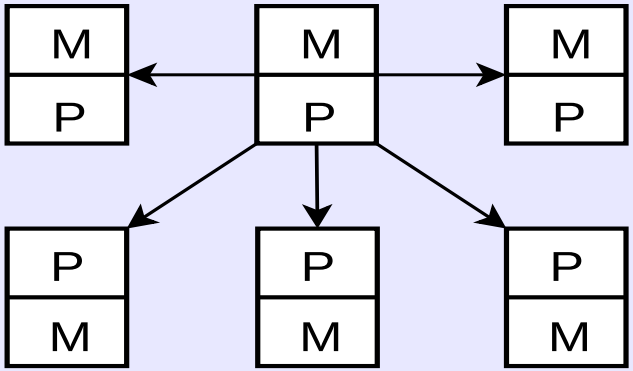
MPMD编程模式 Multiple Program Multiple Data的缩写. 指构成一个程序的不同进程运行不同的可执行代码. 用户需要编写、维护多份源代码.

主/从编程模式 英文为Master/Slave. 它是MPMD编程模式的一个特例, 也是MPMD编程模式中最常见的方式. 构成一个程序的进程之一负责所有进程间的协调及任务调度, 该进程称为主进程(Master), 其余进程称为从进程(Slave). 通常用户需要维护两份源代码.

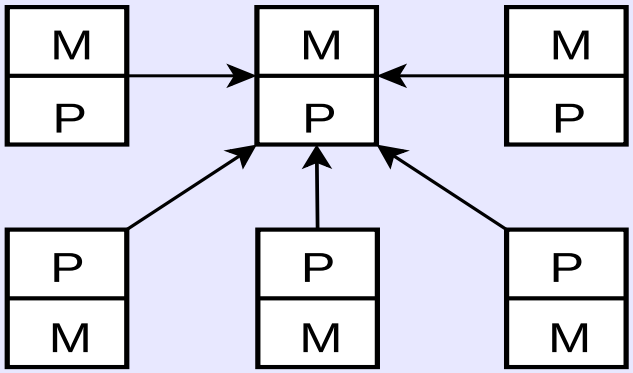
常见消息传递模式



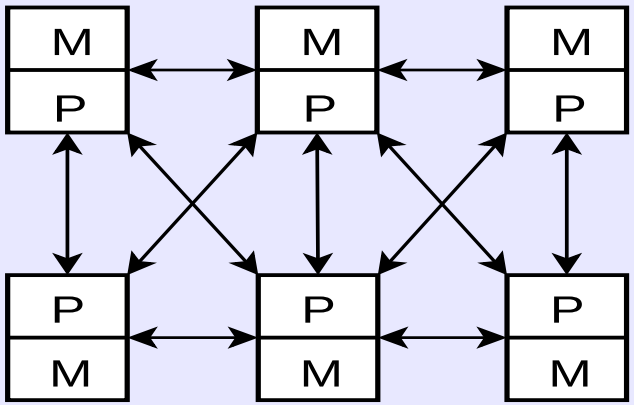
(c) 点对点通信



(d) 一对多通信（广播、散发）



(e) 多对一通信（收集、规约）



(f) 多对多通信（全收集、全交换、全规约、规约分发）

MPI并行程序和串行程序的区别

- MPI 仅是一个“函数库”而已
- 串行编译环境：`gcc g++ g77 gfortran f90 ifort`

  ```
  gcc -o execute source_file.c
  ```

  并行编译环境：`mpicc mpif77 mpif90 mpicxx`

  ```
  mpicc -o execute source_file.c
  ```

- 运行串行程序：

  ```
  ./execute
  ```

  运行并行程序：

  ```
  mpirun -np N execute
  ```

- 调试程序

  `gdb、dbx、totalview、...`

第一个MPI程序

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

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

FORTRAN 版本

```
program hello_world
 include ‘mpif.h’

 integer myrank, nprocs, ierr

 call MPI_Init(ierr)
 call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)
 call MPI_Comm_size(MPI_COMM_WORLD,nprocs,ierr)
 print *,’myrank’,myrank,’of’,nporcs,’:Hello!’
 call MPI_Finalize(ierr)
end program
```

MPI程序的基本要素

- 合适的头文件：mpi.h(C), mpif.h(Fortran)
- 初始化MPI 环境：MPI_Init
- 通信器：MPI_COMM_WORLD
- 进程号：(rank)
- 结束MPI 环境：MPI_Finalize

MPI函数的一般形式

C: 一般形式为:

```
int MPI_Xxxxxx(....)
```

MPI的C函数名中下划线后第一个字母大写, 其余字母小写.
除MPI_Wtime()和MPI_Wtick()外, 所有MPI的C函数均返回一个整型错误码, 当它等于MPI_SUCCESS (0) 时表示调用成功, 否则表示调用中产生了某类错误.

Fortran 77: 一般形式为:

```
SUBROUTINE MPI_XXXXXX(...., IERR)
```

除MPI_WTIME和MPI_WTICK外, Fortran 77的MPI过程全部是Fortran 77子程序(SUBROUTINE), 它们与对应的MPI的C函数同名(但不区分大小写) , 参数表除最后多出一个整型参数IERR用于返回调用错误码及参数的类型不同外, 也和MPI的C函数一样.

MPI的原始数据类型

MPI系统中数据的发送与接收都是基于数据类型进行的. 数据类型可以是MPI系统预定义的, 称为原始数据类型, 也可以是用户在原始数据类型的基础上自己定义的数据类型

MPI数据类型	对应的Fortran 77数据类型
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER*1
MPI_BYTE	
MPI_PACKED	

MPI的原始数据类型

MPI数据类型	对应的C数据类型
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_SHORT	short
MPI_LONG	long
MPI_CHAR	char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned
MPI_UNSIGNED_LONG	unsigned long
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI 程序的基本模式

1. 初始化MPI环境：

```
MPI_Init(&argc, &argv);
```

2. 获取并行环境参数(总进程数、本地进程编号等)：

```
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

3. 执行计算、通信等操作：

```
printf("rank %d of %d: Hello, world!\n",
        myrank, nprocs);
```

4. 结束MPI环境：

```
MPI_Finalize();
```

常用MPI环境管理函数

- `MPI_Init()`
  启动MPI环境。
- `MPI_Comm_size()`, `MPI_Comm_rank()`
  获取进程总数和本地进程编号。
- `MPI_Get_processor_name()`
  获取处理器（主机）名称。
- `MPI_Initialized()`
  判断MPI环境是否已经启动。
- `MPI_Wtime()`, `MPI_Wtick()`
  获取系统时间及时间精度。
- `MPI_Finalize()`, `MPI_Abort()`
  终止MPI系统。

MPI 标准阻塞式消息传递函数

```
int MPI_Send(void *send_buffer,
             int send_count,
             MPI_Datatype datatype,
             int dest,
             int tag,
             MPI_Comm comm);

int MPI_Recv(void *recv_buffer,
             int recv_count,
             MPI_Datatype datatype,
             int src,
             int tag,
             MPI_Comm comm,
             MPI_Status * status);
```

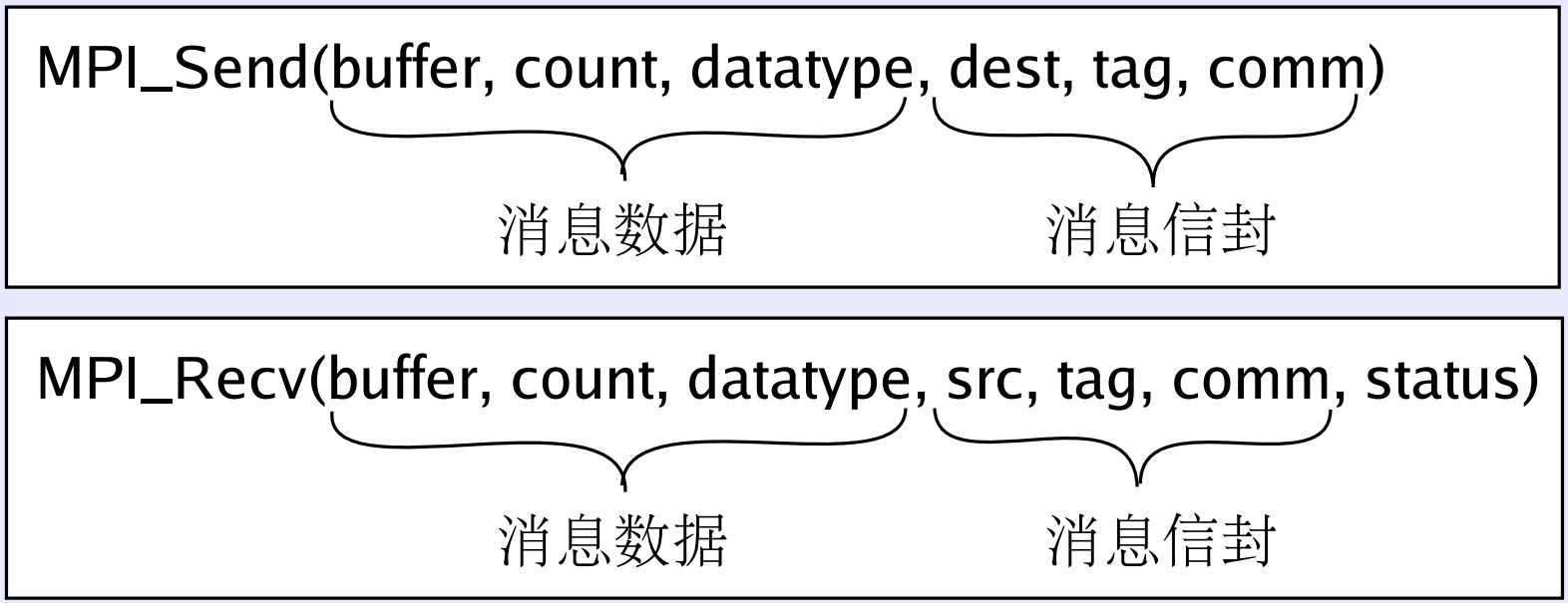
MPI 标准阻塞式收发消息程序

```
#include <mpi.h>
int main (int argc, char *argv[]) {
 int myrank, a;
 MPI_Status stat;

 MPI_Init (&argc, &argv);
 MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
 a = myrank + 10;
 if (myrank == 0)
   MPI_Send (&a, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
 else (myrank == 1)
   MPI_Recv (&a, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, &stat);
 MPI_Finalize ();
 return 0;
}
```

MPI消息的组成

MPI消息由消息信封和消息数据组成。

- 消息信封：<消息源/目的、消息标识、通信域>
- 消息数据：<数据起始地址、数据个数、数据类型>



“tag”在消息收发中的作用（I）

```
... ...
int aa, bb;
MPI_Status stat;

if (myrank == 0)
  aa = 10;
  bb = 20;
  MPI_Send (&aa, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
  MPI_Send (&bb, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
else
  MPI_Recv (&aa, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, &stat);
  MPI_Recv (&bb, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, &stat);
... ...
```

“tag”在消息收发中的作用（II）

```
... ...
int aa, bb;
MPI_Status stat;

if (myrank == 0)
  aa = 10;
  bb = 20;
  MPI_Send (&aa, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
  MPI_Send (&bb, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
else
  MPI_Recv (&bb, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, &stat);
  MPI_Recv (&aa, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, &stat);
... ...
```

“tag”在消息收发中的作用（III）

```
... ...
int aa, bb;
MPI_Status stat;

if (myrank == 0)
  aa = 10;
  bb = 20;
  MPI_Send (&aa, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
  MPI_Send (&bb, 1, MPI_INT, 1, 9, MPI_COMM_WORLD);
else
  MPI_Recv (&bb, 1, MPI_INT, 0, 9, MPI_COMM_WORLD, &stat);
  MPI_Recv (&aa, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &stat);
... ...
```

MPI程序的基本要素

- 头文件：mpi.h(C), mpi.h(Fortran)
- 初始化MPI 环境：MPI_Init
- 通信器：缺省为MPI_COMM_WORLD
- 进程号：(myrank)
- 结束MPI 环境：MPI_Finalize

MPI程序的基本流程

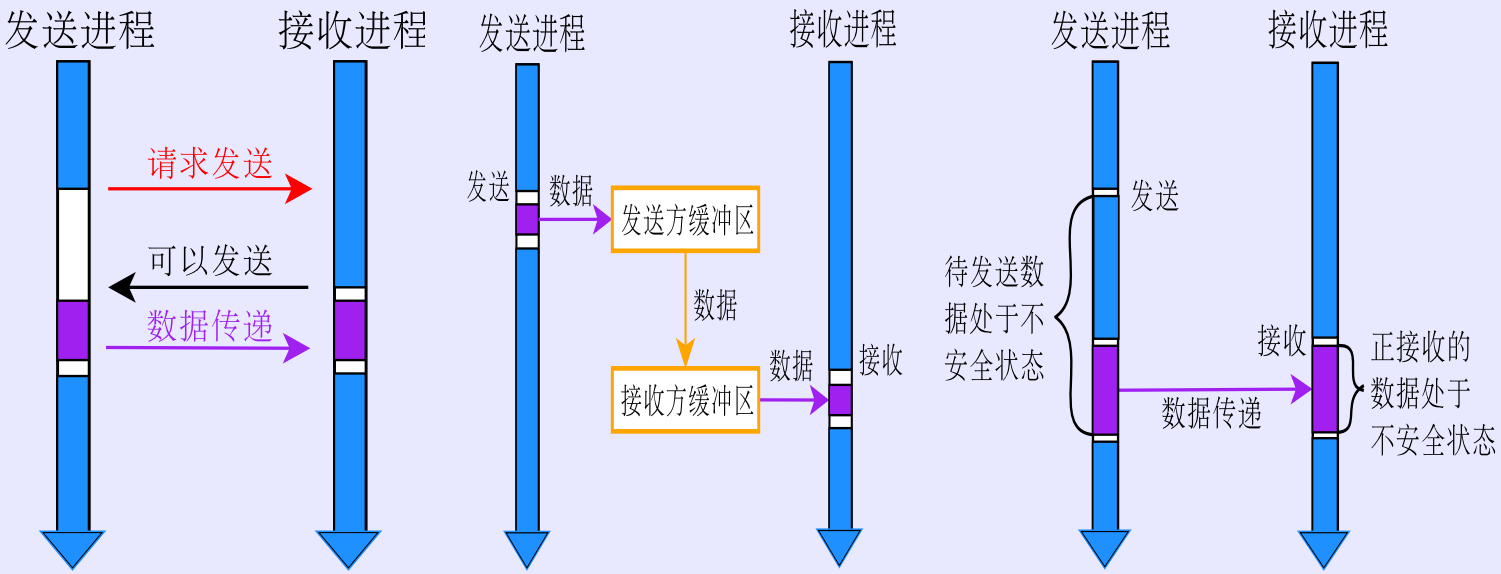
```
//启动MPI环境
MPI_Init(&argc, &argv);

//获取参与并行计算的进程数,
//或者说属于通信器MPI_COMM_WORLD的进程的数目
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

//获取本地进程的进程号,
//或者说本地进程在通信器MPI_COMM_WORLD中的序号
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

//执行计算、通信
... ...

//结束MPI环境
```

MPI通信类型

- 点对点通信：
  - 同一通信器内的两个进程之间的消息传递；
  - 有阻塞型、非阻塞型通信之分；
  - 消息传递函数中有“tag”。
- 聚合通信：
  - 同一通信器内所有进程参与的消息传递；
  - 均为阻塞型通信；
  - 通信函数中不需要“tag”；
  - 包括：传递数据、计算、同步

MPI 通信模式

- 阻塞型：无缓冲区、带缓冲区
- 非阻塞型：



(g) 阻塞型　(h) 阻塞型　(i) 非阻塞型

MPI 点对点消息发送模式

- 标准模式：自由发送、接收，不考虑其它进程状态；
- 缓存模式：由用户显式提供缓冲区，辅助通信；
  创建、释放缓冲区函数：

```
MPI_Buffer_attach(buffer, size);
MPI_Buffer_detach(buffer, size);
```

- 同步模式：通信双方首先建立联系，然后通信；
- 就绪模式：接收进程必须先于发送进程提交通信请求。

通信模式	阻塞通信函数名	非阻塞通信函数名
标准模式	MPI_Send()	MPI_Isend()
缓存模式	MPI_Bsend()	MPI_Ibsend()
同步模式	MPI_Ssend()	MPI_Issend()
就绪模式	MPI_Rsend()	MPI_Irsend()

MPI 标准阻塞式消息收发函数

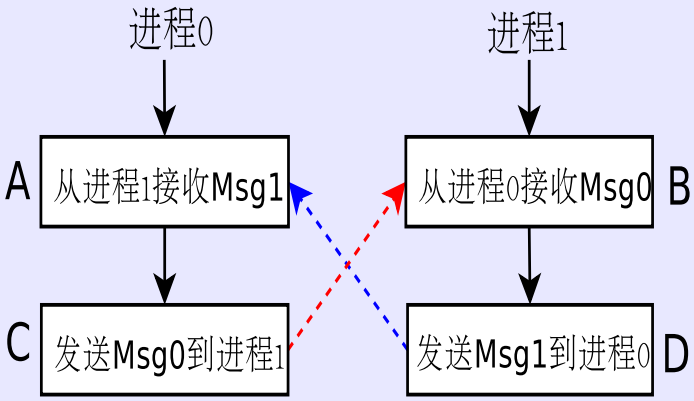
消息发送函数

```
int MPI_Send(void *send_buffer,
             int send_count,
             MPI_Datatype datatype,
             int dest,
             int tag,
             MPI_Comm comm);
```

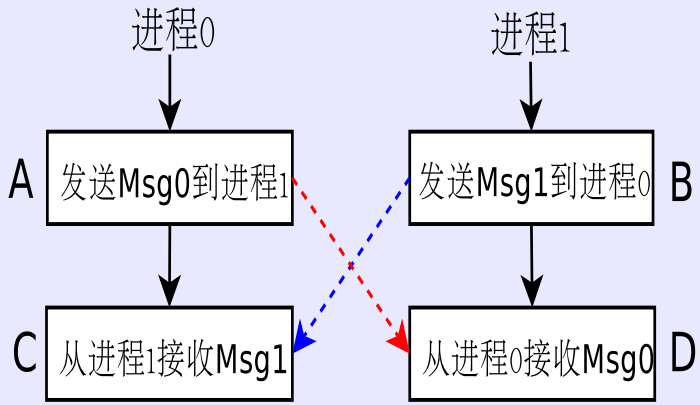
消息接收函数

```
int MPI_Recv(void *recv_buffer,
             int recv_count,
             MPI_Datatype datatype,
             int src,
             int tag,
             MPI_Comm comm,
             MPI_Status * status);
```

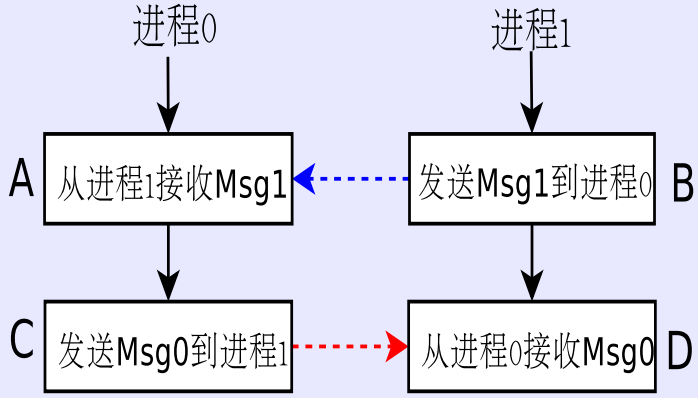
死锁现象



(j) 一定死锁



(k) 可能死锁



(l) 不会死锁

消息收发函数

```
int MPI_Sendrecv(void *send_buf,
                 int send_count,
                 MPI_Datatype send_type,
                 int dest,
                 int send_tag,
                 void *recv_buf,
                 int recv_count,
                 MPI_Datatype recv_type,
                 int source,
                 int recv_tag,
                 MPI_Comm comm,
                 MPI_Status *status);
```

消息收发替换函数

```
int MPI_Sendrecv_replace(void *buf,
                         int count,
                         MPI_Datatype type,
                         int dest,
                         int send_tag,
                         int source,
                         int recv_tag,
                         MPI_Comm comm,
                         MPI_Status *status);
```

通信函数状态信息—status

- C语言：

  类型为MPI_Status的结构体

  - status.MPI_SOURCE
  - status.MPI_TAG
  - status.MPI_ERROR

- FORTRAN语言：

  大小为MPI_STATUS_SIZE的整型数组

  - status(MPI_SOURCE)
  - status(MPI_TAG)
  - status(MPI_ERROR)

消息传递的三个阶段

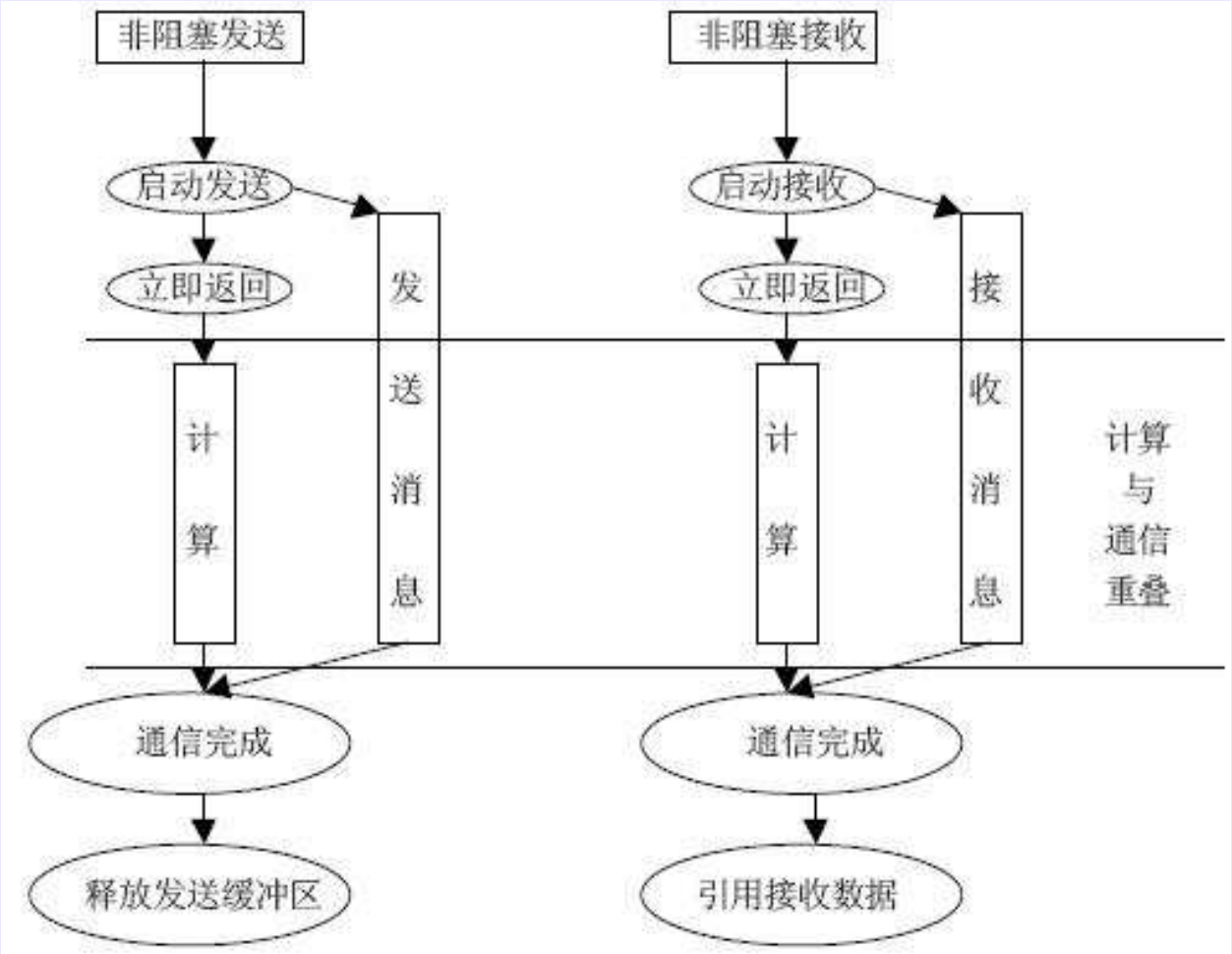
1. 在发送进程中，封装消息，从发送缓冲区到MPI系统；
2. 基于通信器，消息从发送进程到接收进程；
3. 在接收进程中，拆卸消息，从MPI系统到接收缓冲区。

Figure： 每个阶段都要保证数据类型匹配

MPI 标准非阻塞式消息收发函数

```
int MPI_Isend(void *send_buf,
              int send_count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);

int MPI_Irecv(void *recv_buf,
              int recv_count,
              MPI_Datatype datatype,
              int source,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);
```

计算与通信重叠


非阻塞发送
启动发送
立即返回
计
算
发
送
消
息
通信完成
释放发送缓冲区
非阻塞接收
启动接收
立即返回
计
算
接
收
消
息
通信完成
引用接收数据
计算与通信重叠

非阻塞通信请求完成函数(I)

等待指定通信请求关联的通信完成：

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

等待通信请求序列关联的多个通信中任意一个完成：

```
int MPI_Waitany(int incount,
                MPI_Request *request_array,
                int *index,
                MPI_Status *status);
```

非阻塞通信请求完成函数(II)

等待通信请求序列关联的多个通信中最少一个完成：

```
int MPI_Waitsome(int incount,
                 MPI_Request *request_array,
                 int *outcount,
                 int *indices,
                 MPI_Status *status_array);
```

等待通信请求序列关联的所有通信完成：

```
int MPI_Waitall(int incount,
                MPI_Request *request_array,
                MPI_Status *status_array);
```

非阻塞通信请求查询函数(I)

查询指定通信请求关联的通信是否已经完成：

```
int MPI_Test(MPI_Request *request,
             int *flag,
             MPI_Status *status);
```

查询通信请求序列关联的多个通信中是否有至少一个已经完成：

```
int MPI_Testany(int incount,
                MPI_Request *request_array,
                int *index,
                int *flag,
                MPI_Status *status);
```

非阻塞通信请求查询函数(II)

查询通信请求序列关联的多个通信中是否有部分已经完成：

```
int MPI_Testsome(int incount,
                 MPI_Request *request_array,
                 int *outcount,
                 int *indices,
                 MPI_Status *status_array);
```

查询通信请求序列关联的所有通信是否都已经完成：

```
int MPI_Testall(int incount,
                MPI_Request *request_array,
                int *flag,
                MPI_Status *status_array);
```

非阻塞通信的完成和查询函数

通信请求数目	完成函数	查询函数
一个	MPI_Wait()	MPI_Test()
任意一个	MPI_Waitany()	MPI_Testany()
至少一个	MPI_Waitsome()	MPI_Testsome()
所有	MPI_Waitall()	MPI_Testall()

消息查询函数

阻塞查询特定消息是否已经在MPI系统中：

```
int MPI_Probe(int source,
              int tag,
              MPI_Comm comm,
              MPI_Status *status);
```

非阻塞查询特定消息是否已经在MPI系统中：

```
int MPI_Iprobe(int source,
               int tag,
               MPI_Comm comm,
               int flag,
               MPI_Status *status);
```

持久（重复）非阻塞通信 (I)

创建持久发送通信请求：

```
int MPI_Send_init(void *send_buf,
                  int send_count,
                  MPI_Datatype datatype,
                  int dest,
                  int tag,
                  MPI_Comm comm,
                  MPI_Request *request);
```

创建持久接收通信请求：

```
int MPI_Recv_init(void *recv_buf,
                  int recv_count,
                  MPI_Datatype datatype,
                  int source,
                  int tag,
                  MPI_Comm comm,
                  MPI_Request *request);
```

持久（重复）非阻塞通信（II）

启动一个通信请求关联的通信：

```
int MPI_Start(MPI_Request *request);
```

启动一组通信请求关联的通信：

```
int MPI_Startall(int count,
                 MPI_Request *array_of_request);
```

需要通信完成函数MPI_Wait*()或者查询函数MPI_Test*()辅助完成通信。

MPI 聚合通信（I）

特点：

- 同一通信器内的所有进程都参加；
- 所有进程的函数调用形式相同，但部分参数意义不同；
- 阻塞通信方式；
- 不需要“tag”参数；
- 实现功能：通信、同步、计算；
- 通信模式：一对多、多对一、多对多。

MPI 聚合通信（II）

- 全局通信函数

  广播：`MPI_Bcast()`

  散发：`MPI_Scatter()`

  收集：`MPI_Gather()`

  全收集：`MPI_Allgather()`

  全交换：`MPI_Alltoall()`

- 全局规约函数

  规约：`MPI_Reduce()`

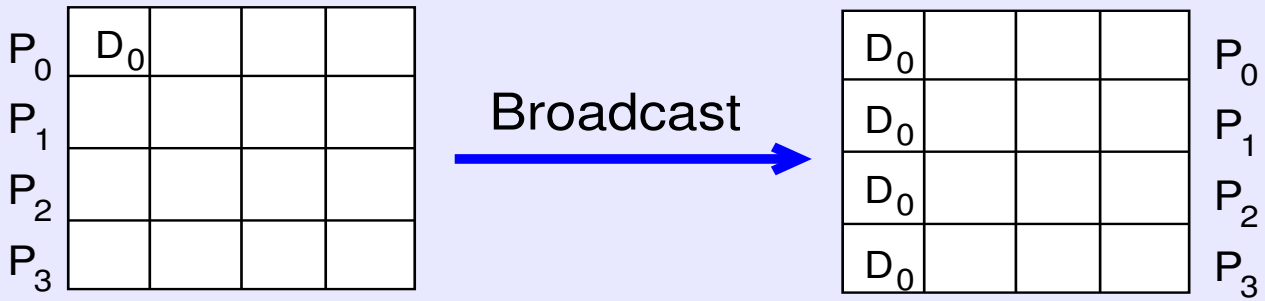
  全规约：`MPI_Allreduce()`

  规约散发：`MPI_Reduce_scatter()`

  前缀规约：`MPI_Scan()`

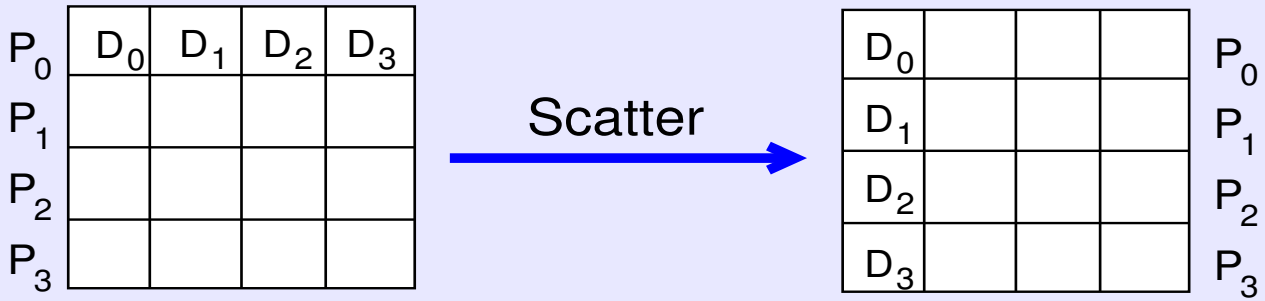
- 同步函数：`MPI_Barrier()`

数据广播




```
int MPI_Bcast(void *buffer,
              int count,
              MPI_Datatype datatype,
              int root,
              MPI_Common comm);
```

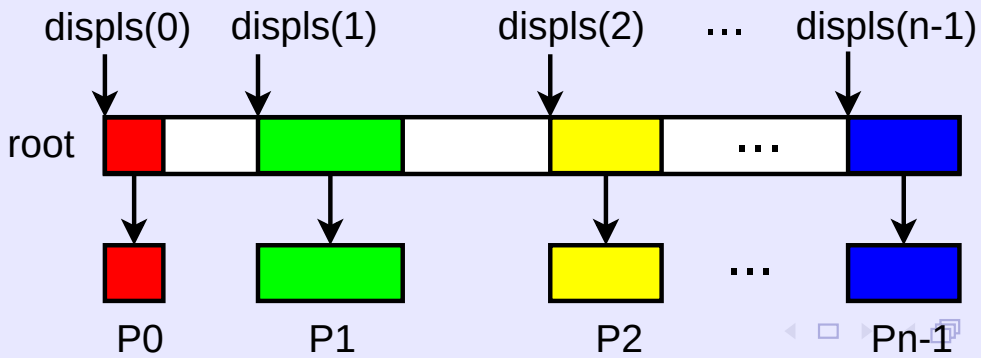
数据散发



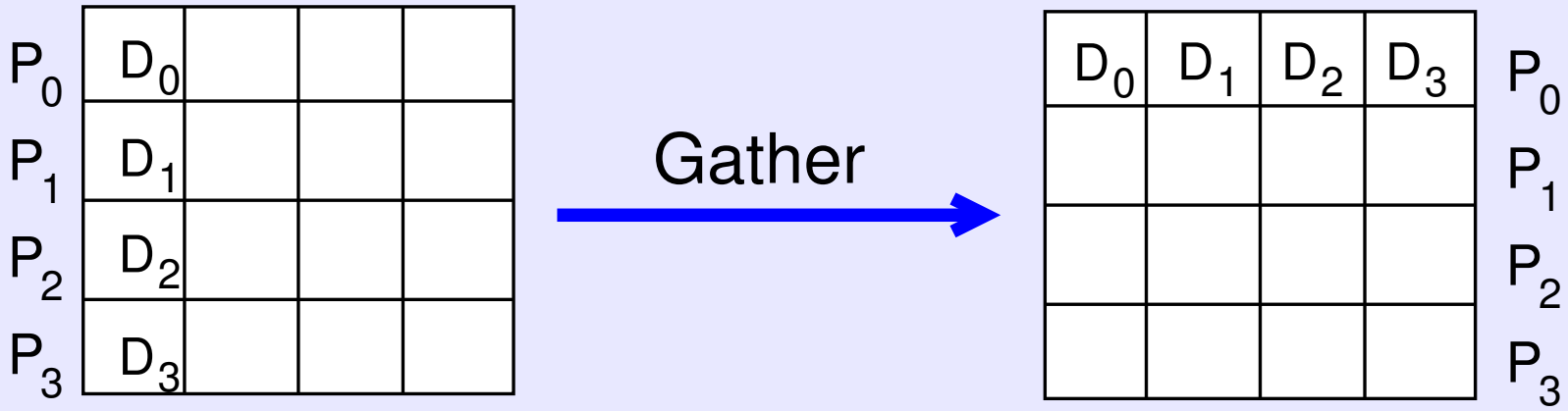

```
int MPI_Scatter(void *send_buf,
                int send_count,
                MPI_Datatype send_type,
                void *recv_buf,
                int recv_count,
                MPI_Datatype recv_type,
                int root,
                MPI_Common comm);
```

基于向量的数据散发

```
int MPI_Scatterv(void *send_buf,
                 int *send_counts,
                 int *displs,
                 MPI_Datatype send_type,
                 void *recv_buf,
                 int recv_count,
                 MPI_Datatype recv_type,
                 int root,
                 MPI_Common comm);
```



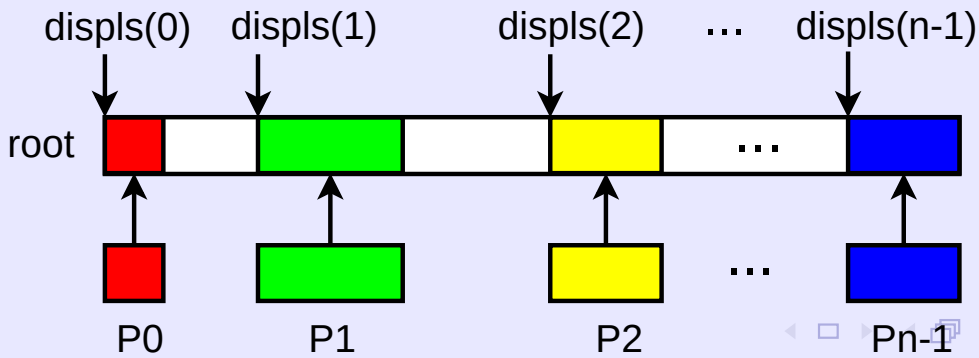
数据收集



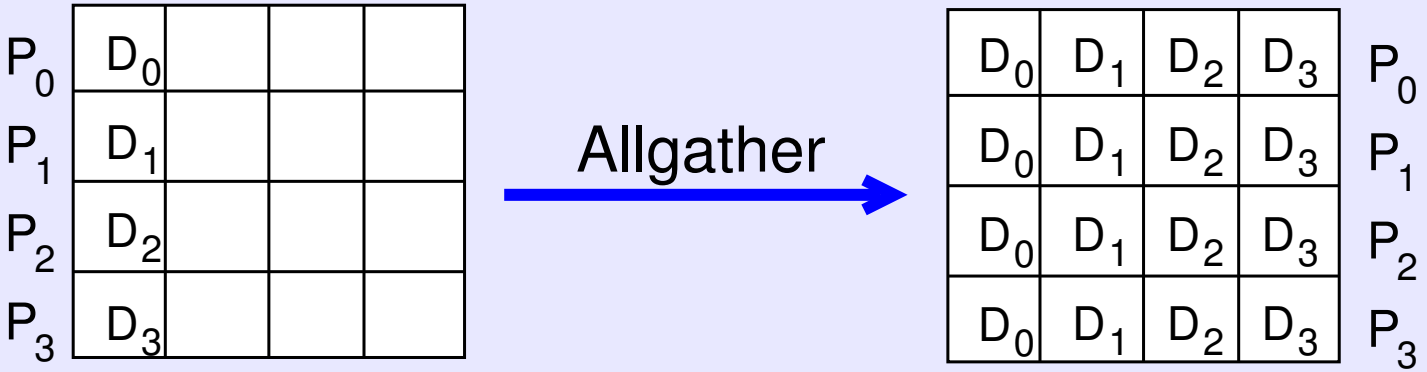

```
int MPI_Gather(void *send_buf,
               int send_count,
               MPI_Datatype send_type,
               void *recv_buf,
               int recv_count,
               MPI_Datatype recv_type,
               int root,
               MPI_Common comm);
```

基于向量的数据收集

```
int MPI_Gatherv(void *send_buf,
                int send_count,
                MPI_Datatype send_type,
                void *recv_buf,
                int *recv_counts,
                int *displs,
                MPI_Datatype recv_type,
                int root,
                MPI_Common comm);
```



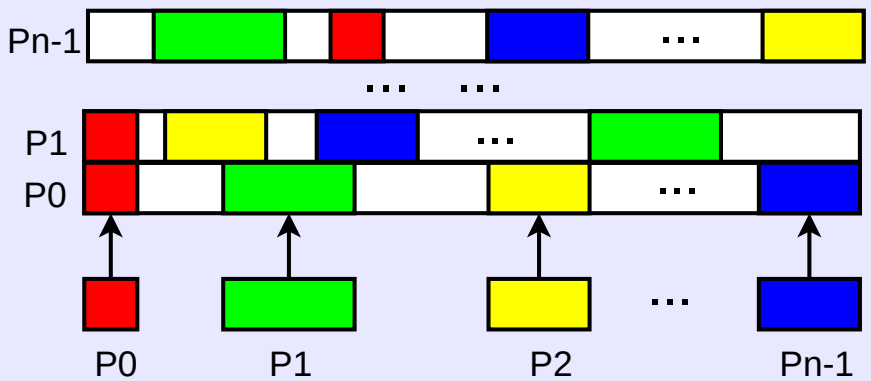
数据全收集



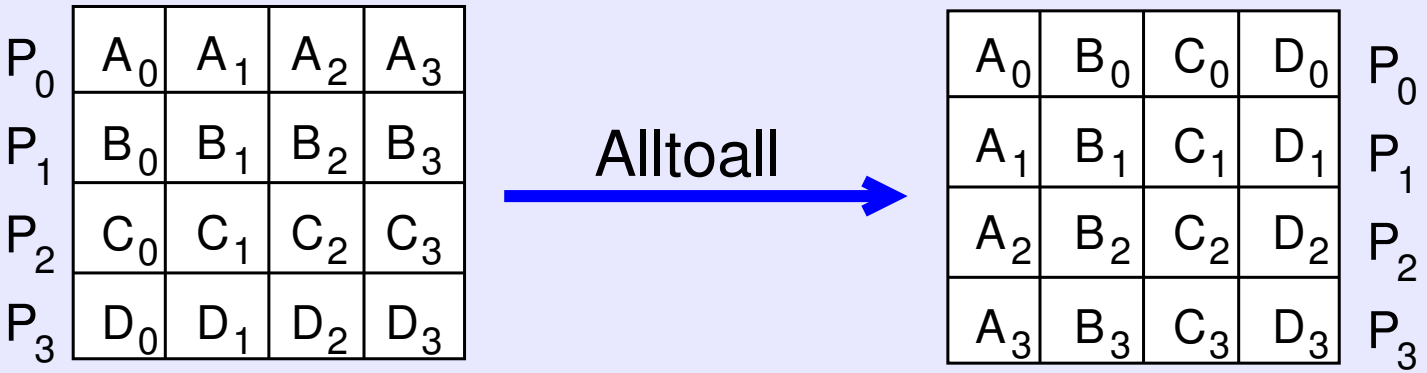

```
int MPI_Allgather(void *send_buf,
                  int send_count,
                  MPI_Datatype send_type,
                  void *recv_buf,
                  int recv_count,
                  MPI_Datatype recv_type,
                  MPI_Common comm);
```

基于向量的数据全收集

```
int MPI_Allgatherv(void *send_buf,
                   int send_count,
                   MPI_Datatype send_type,
                   void *recv_buf,
                   int *recv_counts,
                   int *displs,
                   MPI_Datatype recv_type,
                   MPI_Common comm);
```



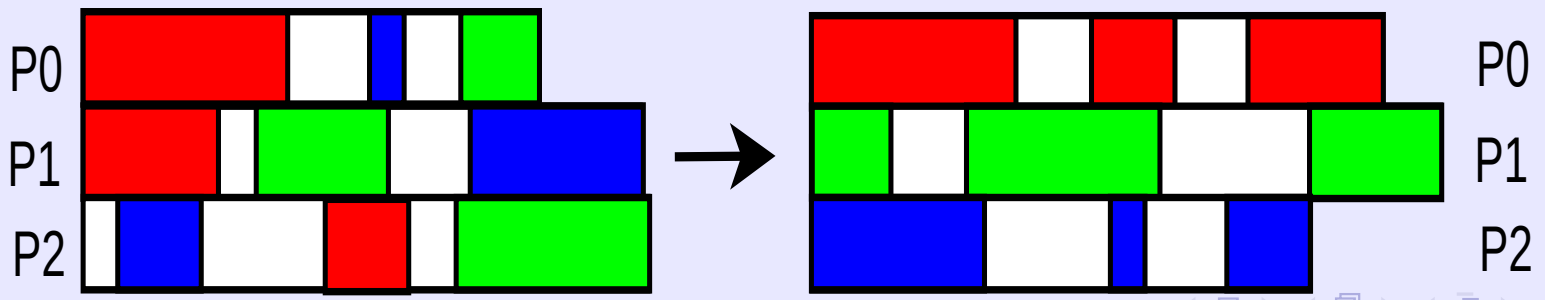
数据全交换



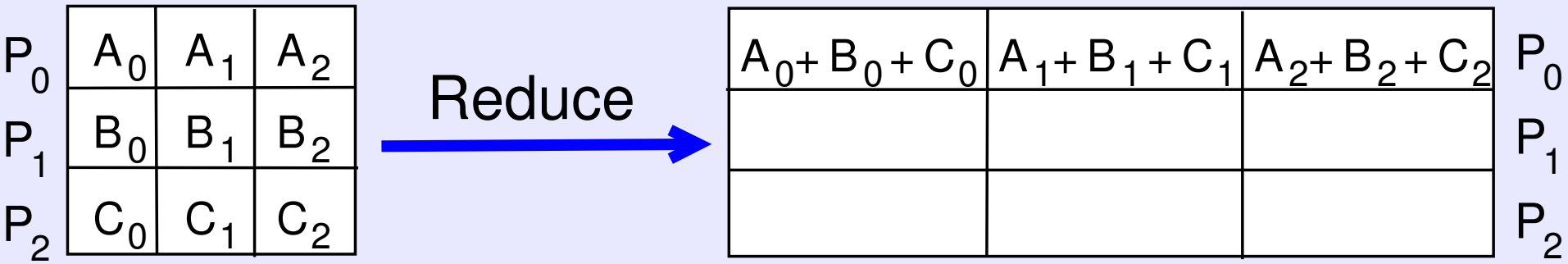

```
int MPI_Alltoall(void *send_buf,
                 int send_count,
                 MPI_Datatype send_type,
                 void *recv_buf,
                 int recv_count,
                 MPI_Datatype recv_type,
                 MPI_Common comm);
```

基于向量的数据全交换

```
int MPI_Alltoallv(void *send_buf,
                  int *send_counts,
                  int *sdispls,
                  MPI_Datatype send_type,
                  void *recv_buf,
                  int *recv_counts,
                  int *rdispls,
                  MPI_Datatype recv_type,
                  MPI_Common comm);
```



规约




```
int MPI_Reduce(void *send_buf,
               void *recv_buf,
               int count,
               MPI_Datatype datatype,
               MPI_Op op,
               int root,
               MPI_Common comm);
```

MPI系统定义的规约操作

操作	含义	操作	含义
MPI_MAX	最大值	MPI_MIN	最小值
MPI_SUM	求和	MPI_PROD	求内积
MPI_LAND	逻辑与	MPI_BAND	按位与
MPI_LOR	逻辑或	MPI_BOR	按位或
MPI_LXOR	逻辑异或	MPI_BXOR	按位异或
MPI_MAXLOC	最大值及位置	MPI_MINLOC	最小值及位置

与操作MPI_MAXLOC和MPI_MINLOC匹配的数据类型

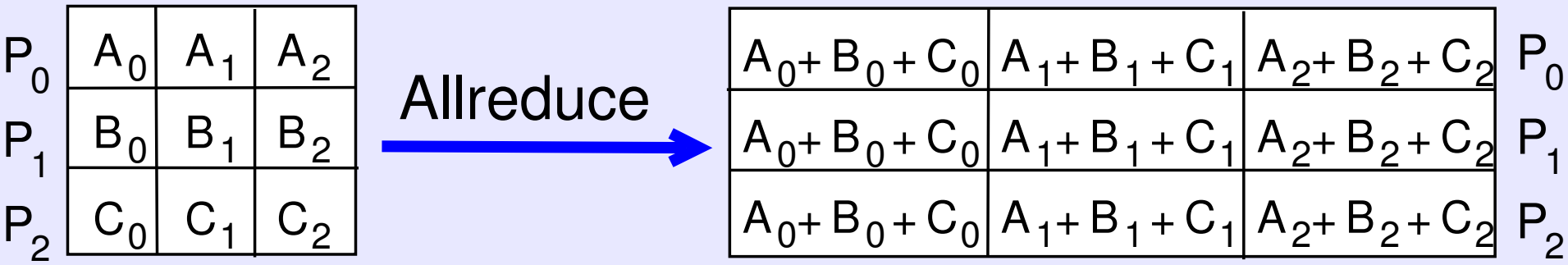
- C语言：

数据类型	描述
MPI_FLOAT_INT	一个float和一个int
MPI_DOUBLE_INT	一个double和一个int
MPI_2INT	两个int
...	...

- FORTRAN：

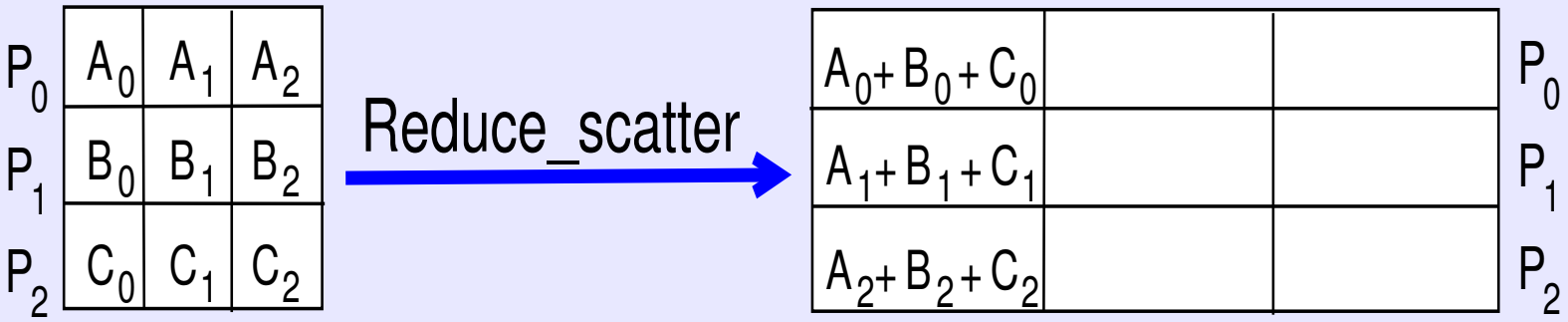
数据类型	描述
MPI_2REAL	两个REAL
MPI_2DOUBLE_PRECISION	两个DOUBLE_PRECISION
MPI_2INTEGER	两个INTEGER

全规约



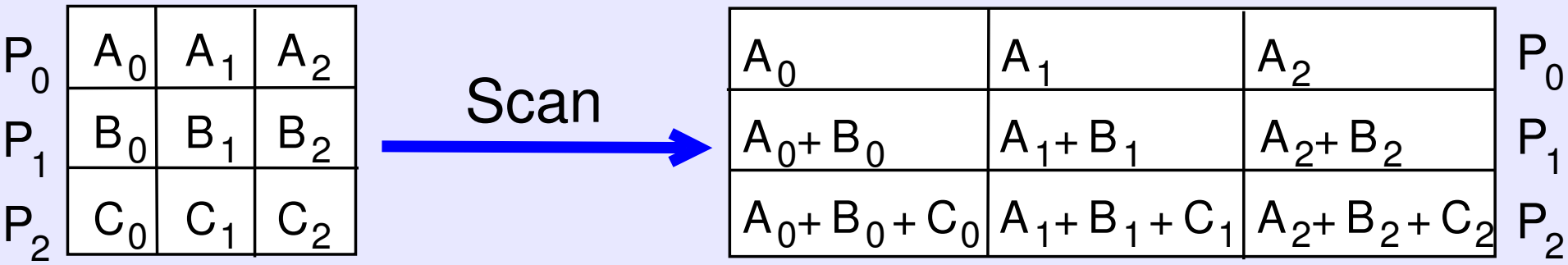

```
int MPI_Allreduce(void *send_buf,
                  void *recv_buf,
                  int count,
                  MPI_Datatype datatype,
                  MPI_Op op,
                  MPI_Common comm);
```

规约散发




```
int MPI_Reduce_scatter(void *send_buf,
                       void *recv_buf,
                       int *recv_count,
                       MPI_Datatype datatype,
                       MPI_Op op,
                       MPI_Common comm);
```

前缀规约




```
int MPI_Scan(void *send_buf,
             void *recv_buf,
             int count,
             MPI_Datatype datatype,
             MPI_Op op,
             MPI_Common comm);
```

用户自定义规约操作

自定义规约操作创建函数：

```
int MPI_Op_create(MPI_User_function *function,
                  int commute,
                  MPI_Op *op);
```

用户自定义函数：

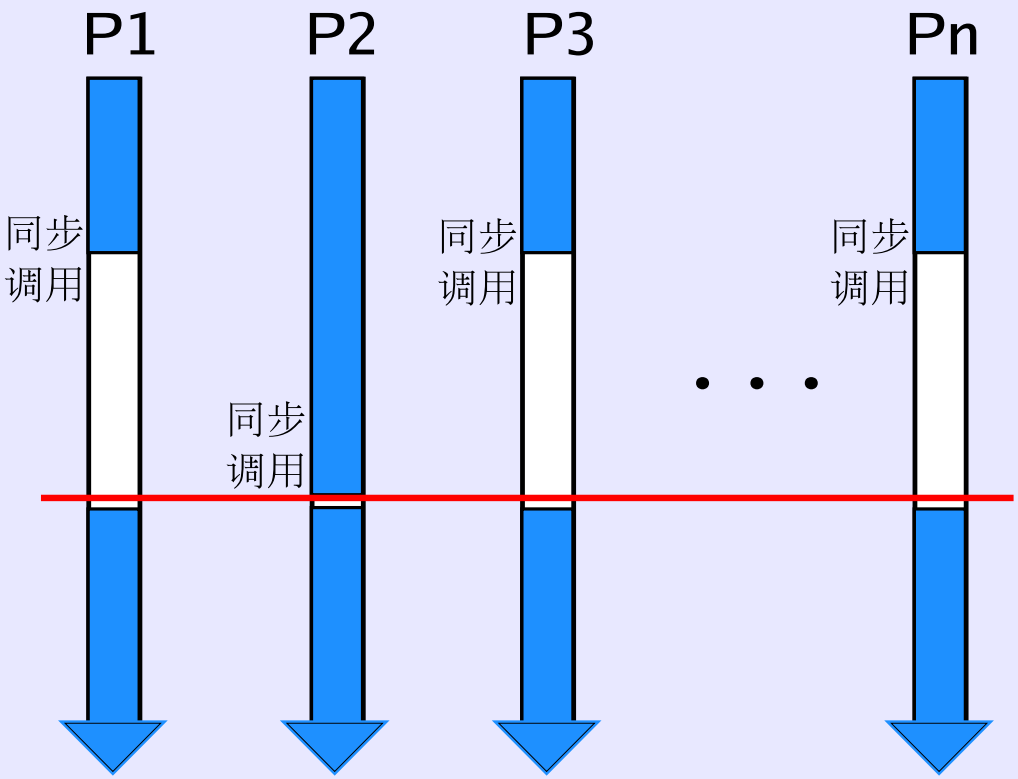
```
void function(void *invec,
              void *inoutvec,
              int *len,
              MPI_Datatype *datatype);
```

自定义规约操作释放函数：

```
int MPI_Op_free(MPI_Op *op);
```

同步

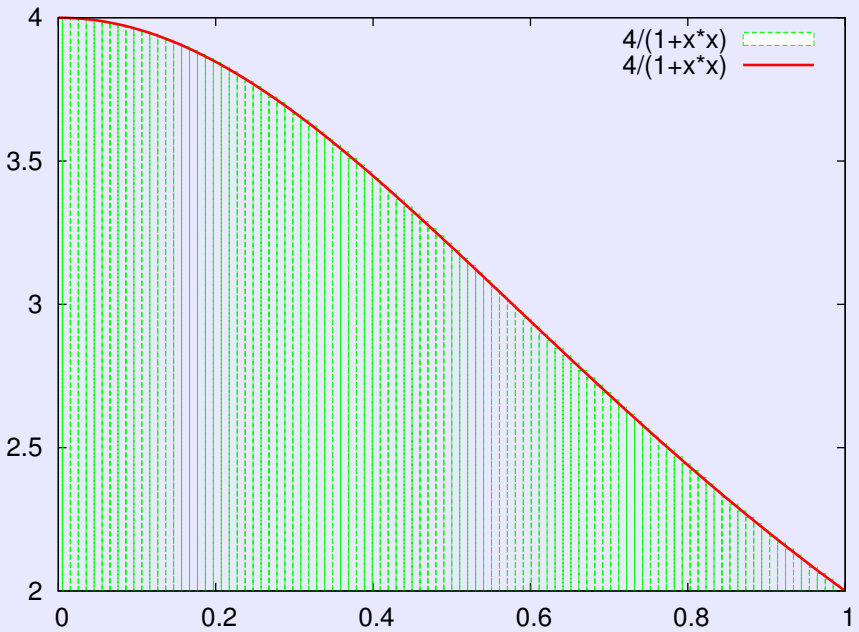
```
int MPI_Barrier(MPI_Common comm);
```



求π 值

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

$$\pi \approx \sum_{i=1}^{N}(\frac{4}{1+x^2} \times \frac{1}{N}) = \frac{1}{N} \times \sum_{i=1}^{N} \frac{4}{1+x^2}$$



数据类型

MPI的消息收发函数只能处理连续存储的同一类型的数据. 当需要在一个消息中发送或接收具有复杂结构的数据时, 可以通过定义数据类型(datatype) 来实现. 数据类型是MPI的一个重要特征, 它的使用可有效地减少消息传递的次数, 增大通信粒度, 并且在收/发消息时避免或减少数据在内存中的拷贝、复制.

- 自定义数据类型
  将内存地址相对固定的不连续或者不同类型的数据定义为新的数据类型。

- 数据封装（打包）
  将任意内存地址上的不连续或者不同类型的数据打包到连续的内存空间。

数据类型定义

一个MPI数据类型由两个n元序列构成, n为正整数. 第一个序列包含一组数据类型, 称为类型序列 (type signature):

$$|Typesig| = \{|type|_0, |type|_1, \ldots, |type|_{n-1}\}.$$

第二个序列包含一组整数位移, 称为位移序列 (type displacements):

$$|Typedisp| = \{|disp|_0, |disp|_1, \ldots, |disp|_{n-1}\}.$$

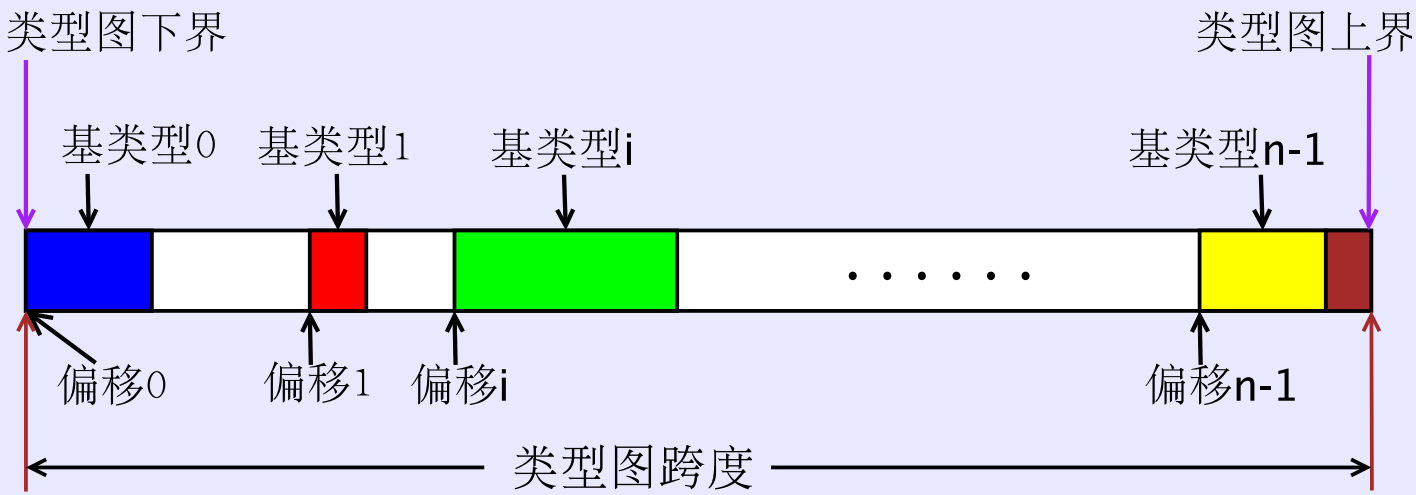
注意, 位移序列中位移总是以字节为单位计算的.

构成类型序列的数据类型称为基本数据类型, 它们可以是原始数据类型, 也可以¿ 是任何已定义的数据类型, 因此MPI的数据类型是嵌套定义的. 为了以后叙述方便, 我们称非原始数据类型为复合数据类型.

类型图

类型序列刻划了数据的类型特征. 位移序列则刻划了数据的位置特征. 类型序列和位移序列的元素一一配对构成的序列称为类型图 (type map).

类型图={<基类型0，偏移0>，<基类型1，偏移1>，... ...，<基类型n-1，偏移n-1>}



位移序列中的位移不必是单调上升的, 表明数据类型中的数据块不要求按顺序排放. 位移也可以是负的, 即数据类型中的数据可以位于缓冲区起始地址之前

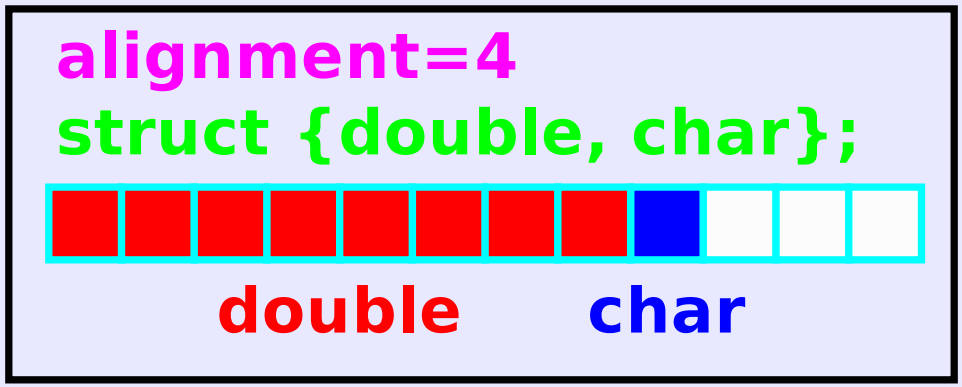
类型图的基本概念

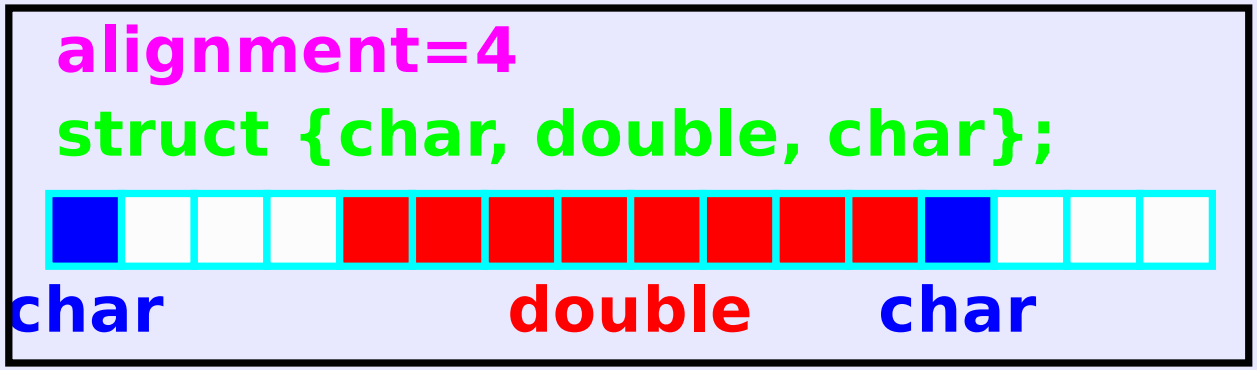
$$typemap = \{< type_0, disp_0 >, \cdots, < type_{n-1}, disp_{n-1} >\}$$

- 类型图下界：
  $lb(typemap) = \min\{disp_i\}, 0 \le j < n$
- 类型图上界：
  $ub(typemap) = \max\{disp_i + sizeof(type_i)\} + \varepsilon, 0 \le j < n$
- 类型图跨度：
  $extent(typemap) = ub(typemap) - lb(typemap)$
- 最小对界：ε 使得extent 能被对界量整除的最小非负整数。
- 一个数据类型的对界量定义如下: 原始数据类型的对界量由编译系统决定, 而复合数据类型的对界量则定义为它的所有基本数据类型的对界量的最大值. 地址对界要求指一个数据类型在内存中的(字节)地址必须是它的对界量的整数倍。

C语言中的对界

C语言中的地址对界: 04-ex0.c





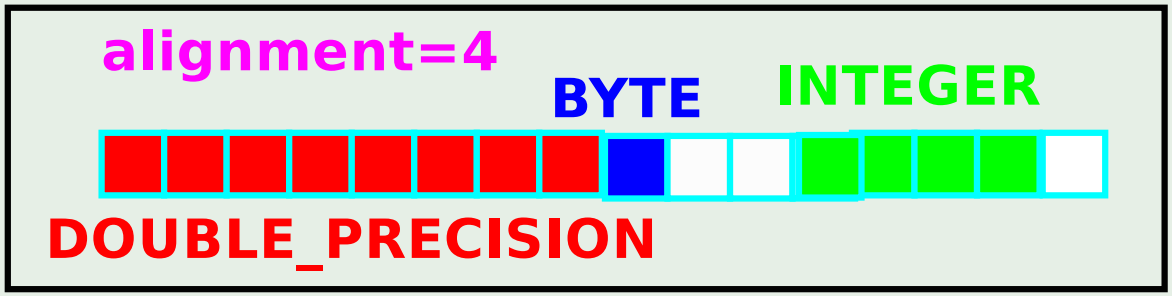
Example

Example

假设MPI_DOUBLE_PRECISION和MPI_INTEGER的对界量均为4, MPI_BYTE的对界量为1, 则类型图:

```
{(MPI_DOUBLE_PRECISION,0), (MPI_BYTE, 8), (MPI_INTEGER, 11) }
```

的对界量为4, 下界为0, 上界为16, 域为16, $\varepsilon = 1$. dtexample.f 数据类型的下界、上界、域和大小



MPI_LB和MPI_UB

MPI系统提供了两个特殊的数据类型MPI_LB和MPI_UB, 称为伪数据类型(pseudo datatype). 它们的大小是0, 并且当它们出现在一个数据类型的类型图中时对该数据类型的实际数据内容不起任何作用. 它们的作用是让用户可以人工指定一个数据类型的上下界. MPI规定: 如果一个数据类型的基本类型中含有MPI_LB, 则它的下界定义为:

$$lb(type) = \min_i \{disp_i \,|\, type_i = \texttt{MPI\_LB}\};$$

类似地, 如果一个数据类型的基本类型中含有MPI_UB, 则它的上界定义为:

$$ub(type) = \max_i \{disp_i \,|\, type_i = \texttt{MPI\_UB}\}.$$

MPI_LB和MPI_UB例子

Example

类型图

```
{ (MPI_LB,-4),
  (MPI_UB,20),
  (MPI_DOUBLE_PRECISION, 0),
  (MPI_INTEGER,8),
  (MPI_BYTE, 12) }
```

的下界为-4, 上界为20, 域为24. lb_ub.c

自定义数据类型相关函数 I

- 获取一个变量的“绝对”地址，或一个内存地址相对于MPI_BOTTOM的位置：

```
MPI_Address (void *location,
             MPI_Aint *address);
```

- 返回一个数据类型的跨度：

```
MPI_Type_extent (MPI_Datatype datatype,
                 MPI_Aint *extent);
```

自定义数据类型相关函数 II

- 返回一个数据类型的下界：

```
MPI_Type_lb (MPI_Datatype datatype,
             MPI_Aint *type_lb);
```

- 返回一个数据类型的上界：

```
MPI_Type_ub (MPI_Datatype datatype,
             MPI_Aint *type_ub);
```

自定义数据类型相关函数 III

- 以通信函数中指定的数据类型为单位返回消息中的数据单元个数：

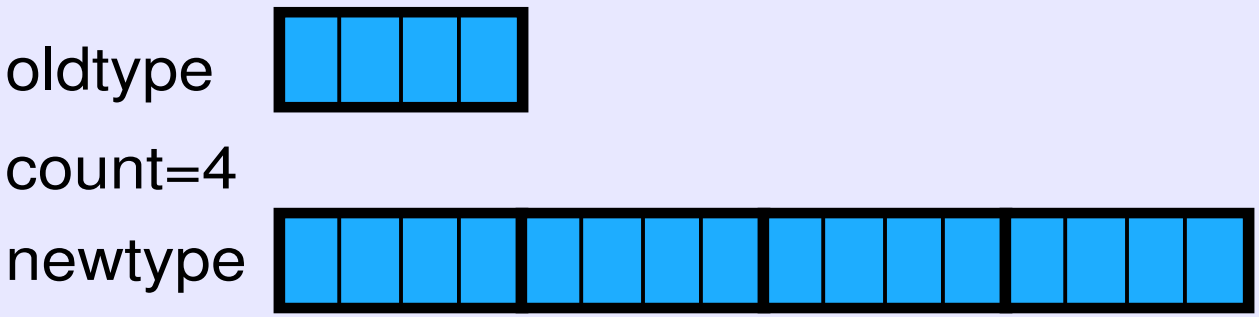
```
MPI_Get_count (MPI_Status *status,
               MPI_Datatype datatype,
               int *count);
```

- 以基本数据类型为单位返回消息中的数据单元个数

```
MPI_Get_elements (MPI_Status *status,
                  MPI_Datatype datatype,
                  int *count);
```

函数MPI_Get_elements与MPI_Get_count类似, 但它返回的是消息中所包含的MPI原始数据类型的个数. MPI_Get_elements返回的count值如果不等于MPI_UNDEFINED的话, 则必然是MPI_Get_count返回的count值的倍数.

在连续的内存空间上创建一个新数据类型。

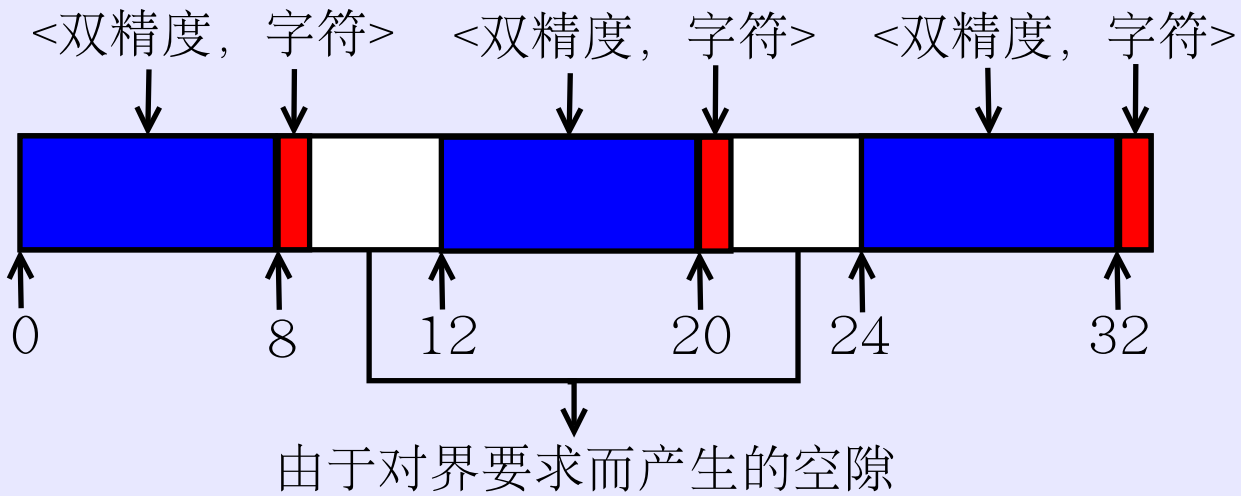



```
MPI_Type_contiguous (int count,
                     MPI_Datatype oldtype,
                     MPI_Datatype *newtype);
```

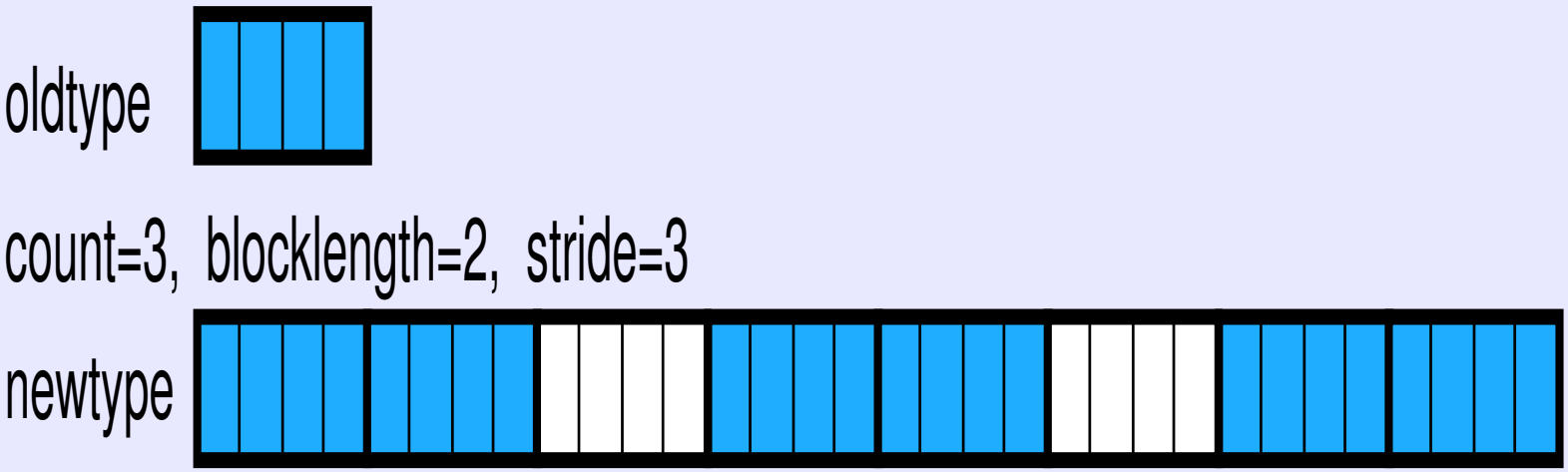
MPI_Type_contiguous示例

$$oldtype = \{< double, 0 >, < char, 8 >\}$$

```
MPI_Type_contiguous (3, oldtype, *newtype);
```

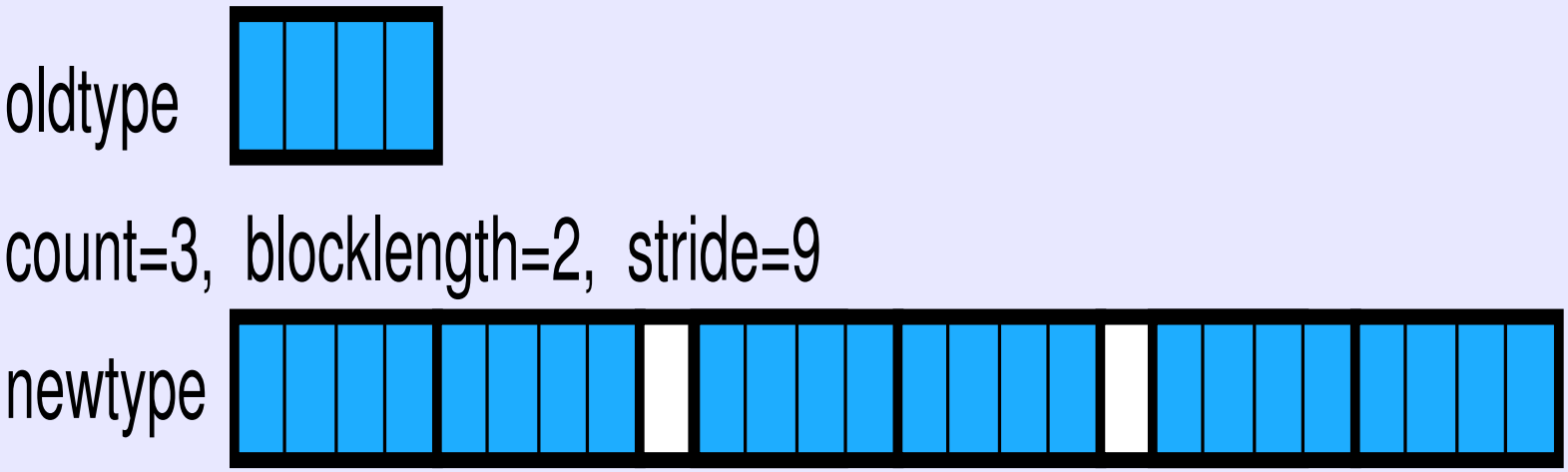


在非连续的内存空间上创建一个新数据类型，包含多个长度相同的数据块。各数据块类型相同。数据块之间的间隔固定，且以旧数据类型长度为单位。



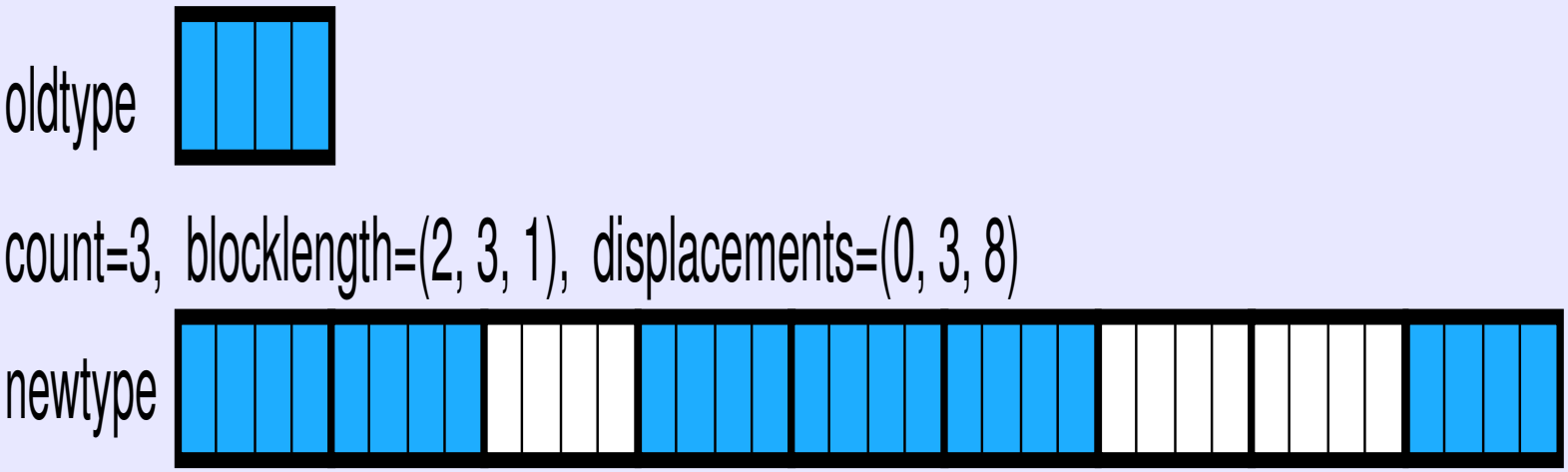

```
MPI_Type_vector (int count,
                 int blocklength,
                 int stirde,
                 MPI_Datatype oldtype,
                 MPI_Datatype *newtype);
```

在非连续的内存空间上创建一个新数据类型，包含多个长度相同的数据块。数据块类型相同。数据块之间的间隔相同，且以字节为单位。



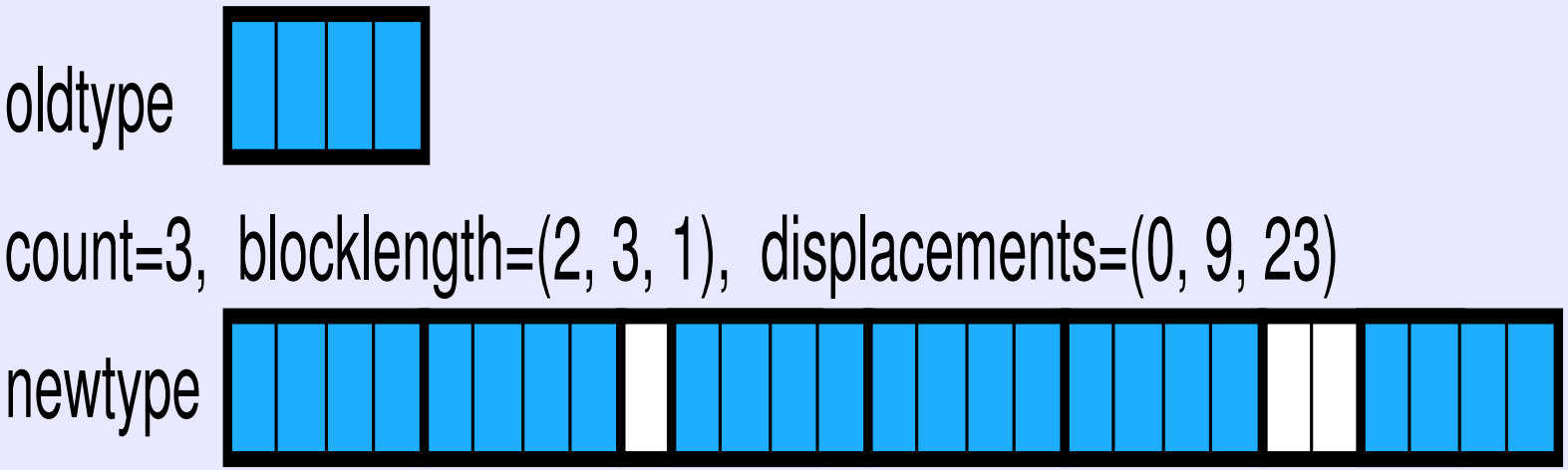

```
MPI_Type_hvector (int count,
                  int blocklength,
                  MPI_Aint stirde,
                  MPI_Datatype oldtype,
                  MPI_Datatype *newtype);
```

在非连续的内存空间上创建一个新数据类型，包含多个任意长度的数据块。各数据块类型相同。数据块距离起始位置的偏移量可以不同，且以旧数据类型长度为单位。



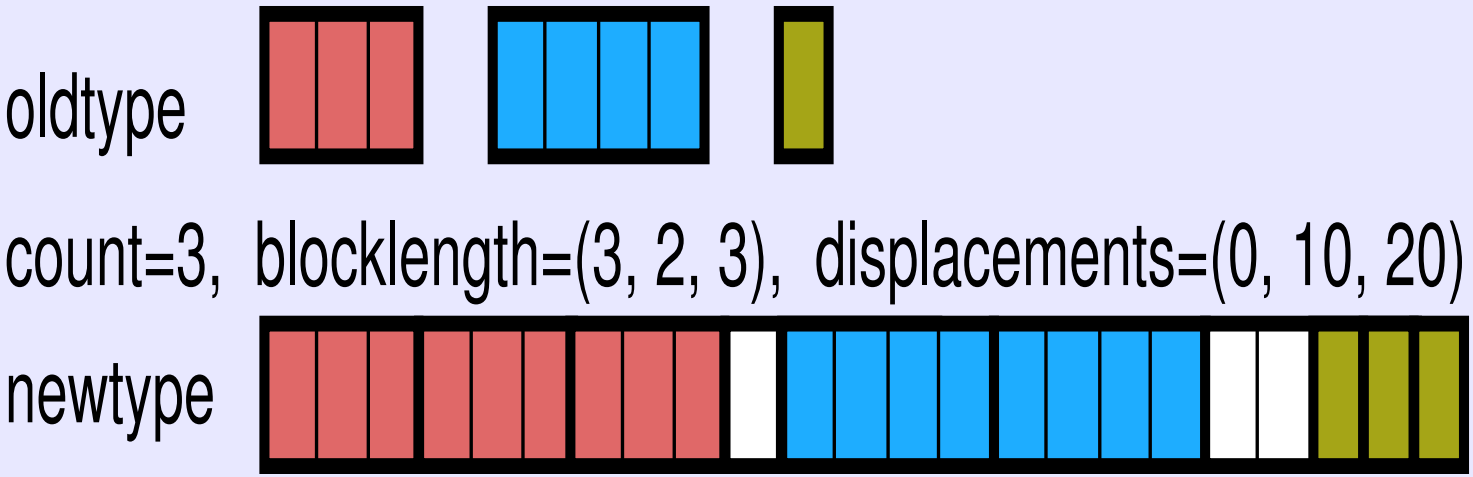

```
MPI_Type_indexed (int count,
                  int array_of_blocklength[],
                  int array_of_displacements[],
                  MPI_Datatype oldtype,
                  MPI_Datatype *newtype);
```

在非连续的内存空间上创建一个新数据类型，包含多个任意长度的数据块。各数据块类型相同。数据块距离起始位置的偏移量可以不同，且以字节为单位。




```
MPI_Type_hindexed (int count,
        int array_of_blocklength[],
        MPI_Aint array_of_displacements[],
        MPI_Datatype oldtype,
        MPI_Datatype *newtype);
```

在非连续的内存空间上创建一个新数据类型，包含多个任意长度的数据块。各数据块类型不同。数据块距离起始位置的偏移量依赖于各数据块的类型，且以字节为单位。




```
MPI_Type_struct (int count,
        int array_of_blocklength[],
        MPI_Aint array_of_displacements[],
        MPI_Datatype oldtype[],
        MPI_Datatype *newtype);
```

自定义数据类型函数总结

函数	特点
MPI_Type_contiguous()	连续、同类型、同大小
MPI_Type_vector()	非连续、同类型、同大小、同间隔
MPI_Type_hvector()	非连续、同类型、同大小、同间隔
MPI_Type_indexed()	非连续、同类型、不同大小、不同偏移
MPI_Type_hindexed()	非连续、同类型、不同大小、不同偏移
MPI_Type_struct()	非连续、不同类型、不同偏移

数据类型的使用

1. 提交：

```
MPI_Type_commit (MPI_Datatype *datatype);
```

2. 使用：

```
... ...
MPI_Send (..., datatype, ...);
MPI_Recv (..., datatype, ...);
... ...
```

3. 释放：

```
MPI_Type_free (MPI_Datatype *datatype);
```

数据封装和拆卸（I）

在MPI中，通过使用特殊数据类型MPI_PACKED，用户可以以类似于PVM中的方式将不同的数据进行打包后再一次发送出去，接收方在收到消息后再进行拆包.

数据封装：

```
MPI_Pack (void *inbuf, int incount,
          MPI_Datatype datatype,
          void *outbuf,
          int outsize,
          int *position,
          MPI_Comm comm);
```

查询封装指定数据所需的最大内存空间：

```
MPI_Pack_size (int incount,
               MPI_Datatype datatype,
               MPI_Comm comm,
               int *size);
```

数据封装和拆卸（II）

数据拆卸：

```
MPI_Unpack (void *inbuf, int insize,
            int *position,
            void *outbuf,
            int outcount,
            MPI_Datatype datatype,
            MPI_Comm comm);
```

附注 I

- MPI 1.1中位移与数据大小的限制

  本章介绍的许多函数中(如`MPI_Address`, `MPI_Type_hvector`等), C与Fortran 77对一些参数使用了不同的变量类型: 它们在C函数中的类型为`MPI_Aint`, 而在Fortran 77中的类型为`MPI_INTEGER`.

  `MPI_Aint`是MPI定义的一个用于与位移及数据大小有关的操作的C变量类型, 因为在64位系统中用`int`存储地址或数据大小可能不够, 在这些系统上`MPI_Aint`通常被定义成`long`, 而在32位系统中`MPI_Aint`则通常定义为`int`. 用`MPI_Aint`来说明存储地址的整型变量便于保证MPI代码中的地址操作在32位与64位操作系统间的可移植性.

  MPI 1.1没有为Fortran 77提供相对应的变量类型, 而是统一使用`INTEGER`. 因此如果一个Fortran 77程序使用了这些函数的话, 它在32与64位操作系统间的可移植性可能会受到影响, 并且用Fortran 77接口处理超过2GB(或4GB)的位移、数据大小等参数也会有困难.

附注 II

此外, 一些C接口的定义也有问题, 如`MPI_Type_size` 中`size`说明为`int *`, 这就隐含限制了它返回的大小不能超过2GB.

- MPI 2中定义了一组新的函数, 如`MPI_Get_address`, `MPI_Type_create_hvector`等等, 目的就是澄清部分这类问题并且为C和Fortran (90)提供统一的接口参数. 有兴趣者可参看MPI 2.0标准第2.6节

基本概念

- 进程组：一组进程的有序集合，每个进程具有唯一编号。MPI中进程组与通信器类似通过句柄来进行操作. 一个MPI程序创建时预定义了两个进程组句柄
  - MPI_GROUP_EMPTY: 由空进程组集合构成的进程组.
  - MPI_GROUP_NULL:非法进程组.
- 通信器：定义在进程组上，具有一定的拓扑信息。
  - 域内通信器: 域内通信器由进程组和上下文构成.通信器中还可以定义一些附加属性, 如进程间的拓扑联接方式等. 域内通信器可以用于点对点通信, 也可以用于聚合通信. 预定义域内通信器：MPI_COMM_WORLD MPI_COMM_SELF MPI_COMM_NULL
  - 域间通信器: 用于分属于不同进程组的进程间的点对点通信. 一个域间通信器由两个进程组构成. 域间通信器不能定义进程的拓扑联接信息, 也不能用于聚合通信.

进程组查询

- 获取指定进程组包含的进程数目：

```
MPI_Group_size (MPI_Group group,
                int *size);
```

- 返回本地进程在指定进程组中的序号：

```
MPI_Group_rank (MPI_Group group,
                int *rank);
```

查询一个进程组中的若干进程在另一个进程组中的序号：

```
MPI_Group_translate_ranks (MPI_Group group1,
                           int n,
                           int *ranks1,
                           MPI_Group group2,
                           int *ranks2);
```

比较两个进程组：

```
MPI_Group_compare (MPI_Group group1,
                   MPI_Group group2,
                   int *result);
```

返回值：

- `result == MPI_IDENT`
  包含的进程以及进程的序号完全相同。
- `result == MPI_SIMILAR`
  包含的进程相同，但是进程的序号不同。
- `result == MPI_UNEQUAL`
  包含不同的进程。

创建进程组 I

- 从已存在的通信器获取进程组：

```
MPI_Comm_group (MPI_Comm comm,
                MPI_Group *group);
```

- 基于两个进程组的并集，创建一个进程组：

```
MPI_Group_union (MPI_Group group1,
                 MPI_Group group2,
                 MPI_Group *newgroup);
```

创建进程组 II

- 基于两个进程组的交集，创建一个进程组：

```
MPI_Group_intersection (MPI_Group group1,
                        MPI_Group group2,
                        MPI_Group *newgroup);
```

- 基于两个进程组的差集，创建一个进程组：

```
MPI_Group_difference (MPI_Group group1,
                      MPI_Group group2,
                      MPI_Group *newgroup);
```

创建进程组 III

- 基于某进程组，创建包含指定进程集合的进程组：

```
MPI_Group_incl (MPI_Group group,
                int n,
                int *ranks,
                MPI_Group *newgroup);
```

- 基于某进程组，创建不包含指定进程集合的进程组：

```
MPI_Group_excl (MPI_Group group,
                int n,
                int *ranks,
                MPI_Group *newgroup);
```

创建进程组 IV

- 基于某进程组, 创建包含的多个指定进程集合的进程组:

```
MPI_Group_range_incl (MPI_Group group,
                      int n,
                      int ranges[][3],
                      MPI_Group *newgroup);
```

将由一组进程序号范围构成的子集组成一个新进程组. 每个范围由一个三元数对(起始序号, 终止序号, 步长) 描述, n为范围个数. 即构成新进程组的进程集合由老进程组中序号属于下述集合的进程组成:

$$\{r \mid r = ranges[i][0] + k \cdot ranges[i][2],$$
$$k = 0, \ldots, \lfloor \frac{ranges[i][1] - ranges[i][0]}{ranges[i][2]} \rfloor, \quad i = 0, \ldots, n-1\}$$

上式中所有计算出的r必须互不相同, 否则调用出错. 新进程组中进程的序号按上式中先k再i的顺序编排(Fortran 77中数组标号应该加1).

创建进程组 V

- 基于某进程组, 创建不包含的多个指定进程集合的进程组:

```
MPI_Group_range_excl (MPI_Group group,
                      int n,
                      int ranges[][3],
                      MPI_Group *newgroup);
```

将老进程组减去由一组进程序号范围给出的进程而得到一个新的进程组, 它相当于取函数MPI_Group_range_incl给出的子集的补集. 各项参数的含义与MPI_Group_range_incl类似.

释放进程组

```
MPI_Group_free (MPI_Group *group);
```

函数返回时会将group置成MPI_GROUP_NULL以防止以后被误用. 实际上, 函数只是将该进程组加上释放标志. 只有基于该进程组的所有通信器均已被释放后才会真的将其释放.

注意! 不能释放以下进程组:

- MPI_GROUP_NULL
- MPI_GROUP_EMPTY
- MPI_GROUP_WORLD
- MPI_GROUP_SELF

创建通信器 I

- 从已存在的通信器复制新通信器：

```
MPI_Comm_dup (MPI_Comm comm,
              MPI_Comm *newcomm);
```

生成一个与comm具有完全相同属性的新通信器newcomm. 注意, 新通信器newcomm与老通信器comm代表着不同的通信域, 因此在它们之间不能进行通信操作。

- 基于某通信器的进程子集，创建新通信器：

```
MPI_Comm_create (MPI_Comm comm,
                 MPI_Group group,
                 MPI_Comm *newcomm);
```

创建一个包含指定进程组group的新通信器newcomm. 这个函数并不将comm的其它属性传递给newcomm, 而是为newcomm建立一个新的上下文. 返回时, 属于进程

创建通信器 II

组group的进程中newcomm 等于新通信器的句柄, 而不属于进程组group的进程中newcomm则等于MPI_COMM_NULL.

- 将某通信器分裂成多个通信器:

```
MPI_Comm_split (MPI_Comm comm,
                int color,
                int key,
                MPI_Comm *newcomm);
```

该函数按照由参数color给出的颜色将通信器中的进程分组, 所有具有相同颜色的进程构成一个新通信器. 新通信器中进程的序号按参数key的值排序, 两个进程的key值相同时则按它们在老通信器comm中的序号排序. 返回时, newcomm等于进程所属的新通信器的句柄. color必须是非负整数或MPI_UNDEFINED. 如果color = MPI_UNDEFINED, 则表示进程不属于任何新通信器, 返回时newcomm = MPI_COMM_NULL.

通信器查询

- 获取指定通信器包含的进程数目：

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

- 返回本地进程在指定进程组中的序号：

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

比较两个通信器：

```
MPI_Comm_compare (MPI_Group comm1,
                  MPI_Group comm2,
                  int *result);
```

返回值：

- `result == MPI_IDENT`
  完全相同。
- `result == MPI_CONGRUENT`
  基于相同的进程组，但定义的通信环境不同。
- `result == MPI_SIMILAR`
  包含相同的进程，但进程序号不同。
- `result == MPI_UNEQUAL`
  包含不同的进程。

释放进程组和通信器

```
MPI_Comm_free (MPI_Group *comm);
```

函数返回时会将comm置成MPI_COMM_NULL以防止以后被误用. 实际上, 该函数只是将通信器加上释放标志. 只有所有引用该通信的操作均已完成后才会真的将其释放.

注意! 不能释放以下通信器:

- MPI_COMM_WORLD
- MPI_COMM_SELF

进程组创建函数总结

函数	功能
MPI_Comm_group()	基于通信器创建
MPI_Group_union()	两个进程组的并集
MPI_Group_intersection()	两个进程组的交集
MPI_Group_difference()	两个进程组的差集
MPI_Group_incl()	包含某进程组的一个子集
MPI_Group_excl()	排除某进程组中的一个子集
MPI_Group_range_incl()	包含某进程组的多个子集
MPI_Group_range_excl()	排除某进程组中的多个子集

通信器创建函数总结

函数	功能
`MPI_Comm_dup()`	复制通信器
`MPI_Comm_create()`	基于某通信器的进程子集

进程拓扑结构

进程拓扑结构是(域内)通信器的一个附加属性, 它描述一个进程组各进程间的逻辑联接关系. 进程拓扑结构的使用一方面可以方便、简化一些并行程序的编制, 另一方面可以帮助MPI系统更好地将进程映射到处理器以及组织通信的流向, 从而获得更好的并行性能.

MPI的进程拓扑结构定义为一个无向图, 图中结点(node)代表进程, 而边(edge) 代表进程间的联接. MPI进程拓扑结构也被称为虚拟拓扑结构, 因为它不一定对应处理器的物理联接.应用问题中较为常见、也是较为简单的一类进程拓扑结构具有网格形式, 这类结构中进程可以用迪卡尔坐标来标识, MPI中称这类拓扑结构为迪卡尔(Cartesian)拓扑结构.

- 图(Graph)拓扑结构–非规则网格
- 笛卡尔(Cartesian)拓扑结构–规则网格

创建笛卡尔拓扑结构

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,
                    int *periods, int reorder, MPI_Comm *comm_cart);
```

ndims给出进程网格的维数. 数组dims给出每维中的进程数. 数组periods则说明进程在各个维上的联接是否具有周期性, 即该维中第一个进程与最后一个进程是否相联, 周期的迪卡尔拓扑结构也称为环面(torus)结构, periods[i] = true 表明第i维是周期的, 否则则是非周期的. reorder指明是否允许在新通信器comm_cart中对进程进行重新排序. 在某些并行机上, 根据处理器的物理联接方式及所要求的进程拓扑结构对进程重新排序有助于提高并行程序的性能.

comm_cart中各维的进程数之积必须不大于comm_old中的进程数, 即:

$$\prod_{i=0}^{\texttt{ndims}-1} \texttt{dims}[i] \leq \texttt{NPROCS}$$

其中NPROCS为comm_old的进程数. 当$\prod_{i=0}^{\texttt{ndims}-1} \texttt{dims}[i] < \texttt{NPROCS}$时一些进程将不属于comm_cart, 这些进程的comm_cart参数将返回MPI_COMM_NULL.

笛卡尔拓扑结构辅助函数

```
int MPI_Dims_create(int nnodes, int ndims, int *dims);
```

该函数当给定总进程数及维数时自动计算各维的进程数, 使得它们的乘积等于总进程数, 并且各维上的进程数尽量接近. 确切地说, 给定nnodes和ndims, 函数计算正整数dims[i], $i = 0, \ldots,$ ndims $- 1$, 使得

$$\prod_{i=0}^{\texttt{ndims}-1} \texttt{dims}[i] = \texttt{nnodes}$$

并且各dims[i]的值尽量接近.

该函数要求输入时dims的中元素的值为非负整数, 并且它仅修改dims中输入值为0的元素. 因此用户可以指定一些维的进程数而仅要求计算其它维的进程数.

局限: 没有考虑实际数据在各维上的大小. 例如, 假如进程数为4, 差分网络为100×400, 则理想的进程拓扑结构应为1×4, 此时无法调用MPI_DIMS_CREATE自动计算最优进程划分.

笛卡尔拓扑结构查询函数(I)

获取指定通信器的拓扑结构的维数。

```
int MPI_Cartdim_get(MPI_Comm comm,
                    int *ndims);
```

获取指定通信器的拓扑结构在每维方向的进程数、是否周期，以及本地进程的坐标。

```
int MPI_Cart_get(MPI_Comm comm,
                 int maxdims,
                 int *dims,
                 int *periods,
                 int *coords);
```

笛卡尔拓扑结构查询函数(II)

将进程坐标转换为进程序号：

```
int MPI_Cart_rank(MPI_Comm comm,
                  int *coords,
                  int *rank);
```

将进程号转换为进程坐标：

```
int MPI_Cart_coords(MPI_Comm comm,
                    int rank,
                    int maxdims,
                    int *coords);
```

数据平移(shift)操作中源地址与目的地址的计算 I

在一个具有迪卡尔拓扑结构的通信器中经常在一个给定维上对数据进行平移(shift), 如用MPI_SENDRECV将一块数据发送给该维上后面一个进程, 同时接收从该维上前面一个进程发送来的数据. MPI提供了一个函数来方便这种情况下目的地址和源地址的计算.

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
                   int *rank_source, int *rank_dest);
```

输入参数direction 是进行数据平移的维号($0 \leq$ direction $<$ ndims), disp给出数据移动的“步长”(绝对值)和“方向”(正负号). 输出参数rank_source和rank_dest分别是平移操作的源地址和目的地址.

假设指定维上的进程数为d, 当前进程该维的坐标为i, 源进程rank_source该维的坐标为i_s, 目的进程rank_dest该维的坐标为i_d, 如果该维是周期的, 则:

$$i_s = i - \texttt{disp} \mod d$$

$$i_d = i + \texttt{disp} \mod d$$

数据平移(shift)操作中源地址与目的地址的计算 II

否则:

$$i_s = \begin{cases} i - \texttt{disp}, & \texttt{if } 0 \le i - \texttt{disp} < d \\ \texttt{MPI\_PROC\_NULL}, & \text{otherwise} \end{cases}$$

$$i_d = \begin{cases} i + \texttt{disp}, & \texttt{if } 0 \le i + \texttt{disp} < d \\ \texttt{MPI\_PROC\_NULL}, & \text{otherwise} \end{cases}$$

创建图(graph)拓扑结构 I

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes,
                     int *index, int *edges, int reorder,
                     MPI_Comm *comm_graph)
```

新通信器的拓扑结构图由参数nnodes, index和edges描述: nnodes是图的结点数(如果nnodes小于通信器comm_old的进程数, 则一些进程将不属于新通信器comm_graph, 这些进程中参数comm_graph 的返回值将为MPI_COMM_NULL), index[i] ($i = 0, \ldots,$ nnodes $- 1$) 给出结点$0, \ldots, i$的邻居数之和, edges则顺序给出所有结点的邻居的序号. 用Neighbor(i)表示第i个结点的邻居的序号集合, 则:

$$\texttt{Neighbor}(0) = \{\texttt{edges}[j] \mid 0 \le j < \texttt{index}[0]\}$$

$$\texttt{Neighbor}(i) = \{\texttt{edges}[j] \mid \texttt{index}[i-1] \le j < \texttt{index}[i]\}$$

$$i = 1, \ldots, \texttt{nnodes} - 1$$

(注意: Fortran 77中所有数组下标应该加1).

并行I/O

MPI的输入输出(I/O)函数属于MPI 2.0. 在MPI 2.0中, 函数接口定义包含C, C++, 和Fortran三种. 出于严谨性考虑, Fortran接口中一些参数使用了Fortran 90 类型(如`INTEGER(MPI_OFFSET_KIND)`), 在Fortran 77代码中这些参数在不同的平台上可能需要采用不同的写法(如`INTEGER*4`, `INTEGER*8`等等), 因而会影响到代码的可移植性.

基本术语 I

文件(file) MPI的“文件”可以看成由具有相同或不同类型的数据项构成的序列. MPI支持对文件的顺序和随机访问. MPI的文件是和进程组相关联的: MPI打开文件的函数(`MPI_File_open`)中要求指定一个通信器, 并且该通信器中所有进程必须同时对文件进行打开或关闭操作.

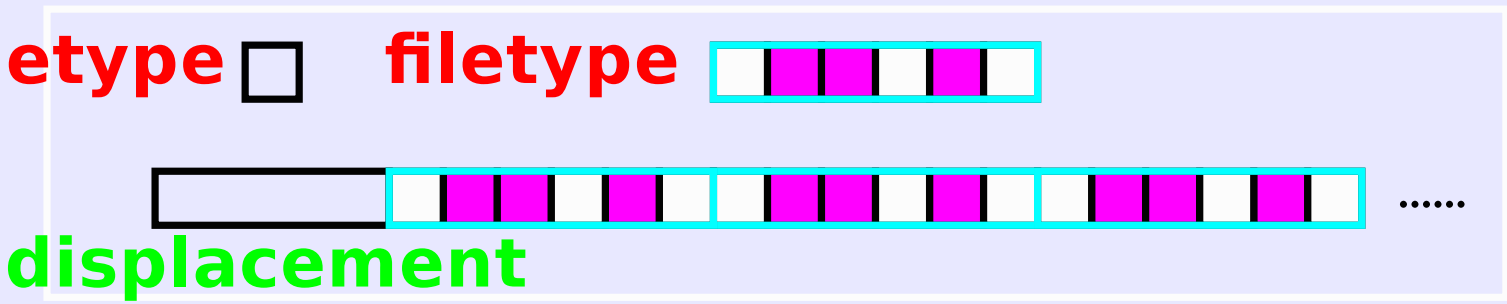
起始位置(displacement) 一个文件的起始位置指相对于文件开头以字节为单位的一个绝对地址, 它用来定义一个“文件视窗”的起始位置.

基本单元类型(etype) 基本单元类型(elementary type)是定义一个文件最小访问单元的MPI数据类型. 一个文件的基本单元类型可以是任何预定义或用户构造的并已经递交的MPI数据类型, 但其类型图中的位移必须非负并且位移序列是(非严格)单调上升的. MPI的文件操作完全以基本单元类型为单位: 文件中的位移(offset)以基本单元类型的个数而非字节数为单位, 文件指针总是指向一个基本单元的起始地址.

基本术语 II

文件单元类型(filetype) 文件单元类型也是一个MPI数据类型, 它定义了对一个文件的存取图案. 文件单元类型可以等于基本单元类型, 也可以是在基本单元类型基础上构造并已递交的任意MPI数据类型. 文件单元类型的域必须是基本单元类型的域的倍数, 并且文件单元类型中间的“洞”的大小也必须是基本单元类型的域的倍数.

视窗(view) 文件视窗指一个文件中目前可以访问的数据集. 文件视窗由三个参数定义: 起始位置, 基本单元类型, 文件单元类型. 文件视窗指从起始位置开始将文件单元类型连续重复排列构成的图案, MPI对文件进行存取操作时将“跳过”图案中的“空洞”, 见下图.



基本术语 III

位移(offset) MPI的I/O函数中位移总是相对于文件起始位置(当前视窗)计算, 并且以基本单元类型的域为单位.

文件大小(file size) 文件大小指从文件开头到文件结尾的总字节数.

文件指针(file pointer) 文件指针是MPI管理的两个内部位移(隐式位移). MPI在每个进程中为每个打开的文件定义了两个文件指针, 一个供本进程独立使用, 称为独立文件指针(individual file pointer), 另一个供打开文件的进程组中所有进程共同使用, 称为共享文件指针(shared file pointer).

文件句柄(file handle) MPI打开一个文件后, 返回给调用程序一个文件句柄, 供以后访问及关闭该文件时用. MPI的文件句柄在文件关闭时被释放.

并行I/O例子

Example

假设ext(MPI_INT) = 4, etype = MPI_INT, 打开文件fh的进程组包括4个进程p_i, $i = 0, 1, 2, 3$, 四个进程中文件单元类型分别定义如下:

$$p_0: \quad \text{filetype} = \{(\texttt{INT}, 0), (\texttt{LB}, 0), (\texttt{UB}, 16)\}$$

$$p_1: \quad \text{filetype} = \{(\texttt{INT}, 4), (\texttt{LB}, 0), (\texttt{UB}, 16)\}$$

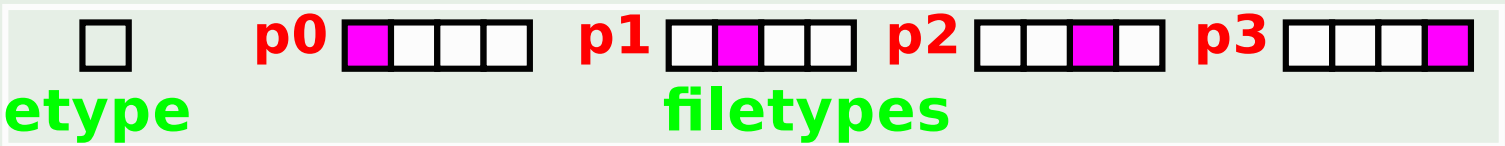
$$p_2: \quad \text{filetype} = \{(\texttt{INT}, 8), (\texttt{LB}, 0), (\texttt{UB}, 16)\}$$

$$p_3: \quad \text{filetype} = \{(\texttt{INT}, 12), (\texttt{LB}, 0), (\texttt{UB}, 16)\}$$

如果四个进程中独立文件指针均为0, 则调用:

```
MPI_File_read(fh, &A, 1, MPI_INT, status)
```

将文件开头的四个数依次赋给四个进程中的变量A.



打开一个文件

```
int MPI_File_open(MPI_Comm comm,
                  char *filename,
                  int mode,
                  MPI_Info info,
                  MPI_File *fh);
```

mode取值:

- MPI_MODE_RDONLY: 只进行读操作
- MPI_MODE_RDWR: 同时进行读操作和写操作
- MPI_MODE_WRONLY: 只进行写操作
- MPI_MODE_CREATE：如果文件不存在则创建一个新文件
- MPI_MODE_EXCL：创建文件时若文件存在则打开失败
- MPI_MODE_DELETE_ON_CLOSE：关闭文件后将其删除
- MPI_MODE_UNIQUE_OPEN：用户可以确保只有当前程序访问该文件
- MPI_MODE_SEQUENTIAL：只能对文件进行顺序读写

关闭文件

```
int MPI_File_close(MPI_File *fh);
```

MPI_FILE_CLOSE是聚合型函数, 进程组中所有进程必须同时调用并且提供同样的参数.

设定文件视窗 I

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
                      MPI_Datatype etype, MPI_Datatype filetype,
                      char *datarep, MPI_Info info)
```

将文件视窗的起始位置设为disp (从文件开头以字节为单位计算), 基本单元类型设为etype, 文件单元类型设为filetype. 参数datarep给出文件中的数据表示格式. 参数info用来重新指定附加提示信息. MPI_FILE_SET_VIEW是聚合型函数, 进程组中所有进程必须同时调用. 不同进程可以提供不同的disp, filetype和info参数, 但必须提供相同的datarep参数和具有相同域的etype参数.

如果文件打开时使用了MPI_MODE_SEQUENTIAL模式, 则disp参数必须写成MPI_DISPLACEMENT_CURRENT (代表文件的当前位置).

参数datarep是一个字符串, 给出文件中使用的数据表示格式. 它有下面一些可能值:

设定文件视窗 II

`"native"`	文件中数据完全按其在内存中的表示形式存放. 使用该数据表示的文件不能在数据格式不兼容的计算机间交换使用.
`"internal"`	指MPI内部格式, 具体由MPI的实现定义. 使用该数据表示的文件可以确保能在使用同一MPI系统的计算机间进行交换使用, 即使这些计算机的数据格式不兼容.
`"external32"`	使用IEEE定义的一种通用数据表示格式, external data representation (简称XDR). 使用该数据表示的文件可以在所有支持MPI的计算机间交换使用. 该格式可用于在数据表示不兼容的计算机间交换数据.

许多MPI系统目前尚未全部实现上述三种格式(它们通常只支持`"native"`格式).

MPI不将有关数据表示格式的信息写在文件中, 因此用户须保证在设定文件窗口时指定的数据表示格式与文件中的实际数据表示格式相符.

各进程读/写不同文件

```
int MPI_File_read(MPI_File fh,
                  void *buf,
                  int count,
                  MPI_Datatype datatype,
                  MPI_Status *status);

int MPI_File_write(MPI_File fh,
                   void *buf,
                   int count,
                   MPI_Datatype datatype,
                   MPI_Status *status);
```

各进程读/写相同文件

```
int MPI_File_read_at(MPI_File fh,
                     MPI_Offset offset,
                     void *buf,
                     int count,
                     MPI_Datatype datatype,
                     MPI_Status *status);

int MPI_File_write_at(MPI_File fh,
                      MPI_Offset offset,
                      void *buf,
                      int count,
                      MPI_Datatype datatype,
                      MPI_Status *status);
```

计算文件偏移位置

```
int MPI_File_seek(MPI_File fh,
                  MPI_Offset offset,
                  int whence);
```

whence取值：

- MPI_SEEK_SET：起始位置

课后练习 I

1. 安装MPI环境
2. 熟悉MPI程序的编译、运行
3. 修改hello world 程序，让每次屏幕打印输出结果均按进程号由小到大顺序输出。
4. 采用流水线的思想设计并行高斯-赛德尔迭代算法。
5. 将mpisum.c 改为打印出最大值和包含最大值的进程号及数组中的位置。给出不同进程数、不同数组长度的性能测试结果。(提示：用`MPI_MAXLOC`)
6. 编写一个环形通信程序ring.c。
7. 将mpisum.c 中的`MPI_Reduce`改为用点对点通信来完成。给出不同进程数、不同数组长度的性能测试结果并与原始程序进行比较。
8. 编写一个测试点对点通信的程序pingpong.c
9. 用点对点通讯实现：`MPI_Allgather()`