# 高级MPI编程技术

## Lecture 10-02: MPI编程——文件IO

肖俊敏

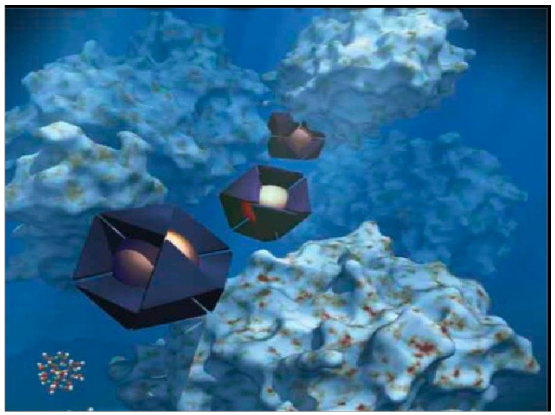
中国科学院计算技术研究所

# 目录

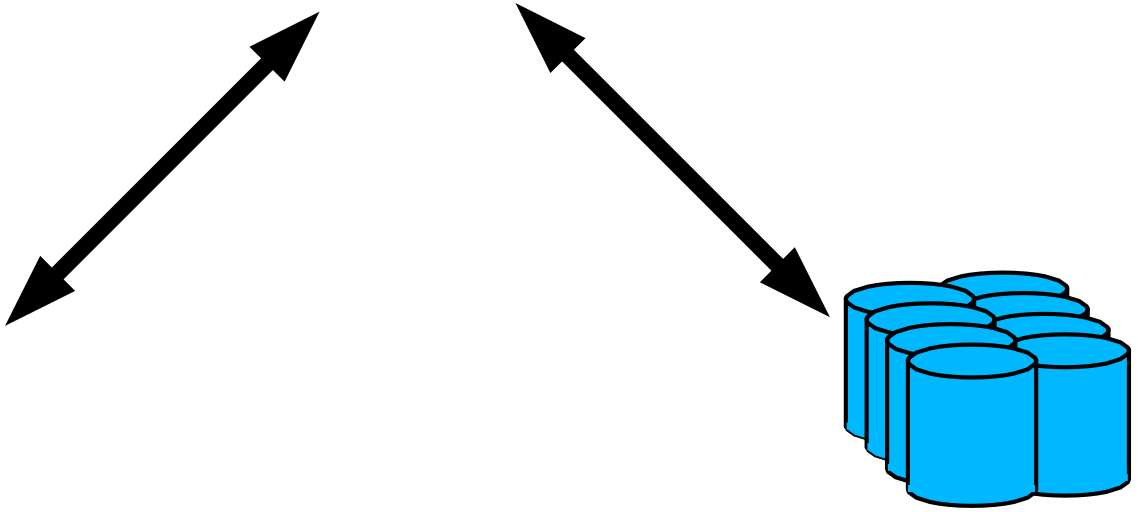
- 并行 IO
  - POSIX-IO
    - 单个任务处理所有文件
    - 单个任务处理一个文件
  - MPI-IO
    - 独立IO
    - 集合IO

# 并行 IO

- 所有程序都需要I/O (输入/输出)，但它经常被忽略
- 映射问题:如何将内部结构或域（domain）转换成字节流文件
- 传输问题:如何有效地将数据从超级计算机上成百上千个节点传输到物理硬盘



...1101101010101101110110010101010101001010101...

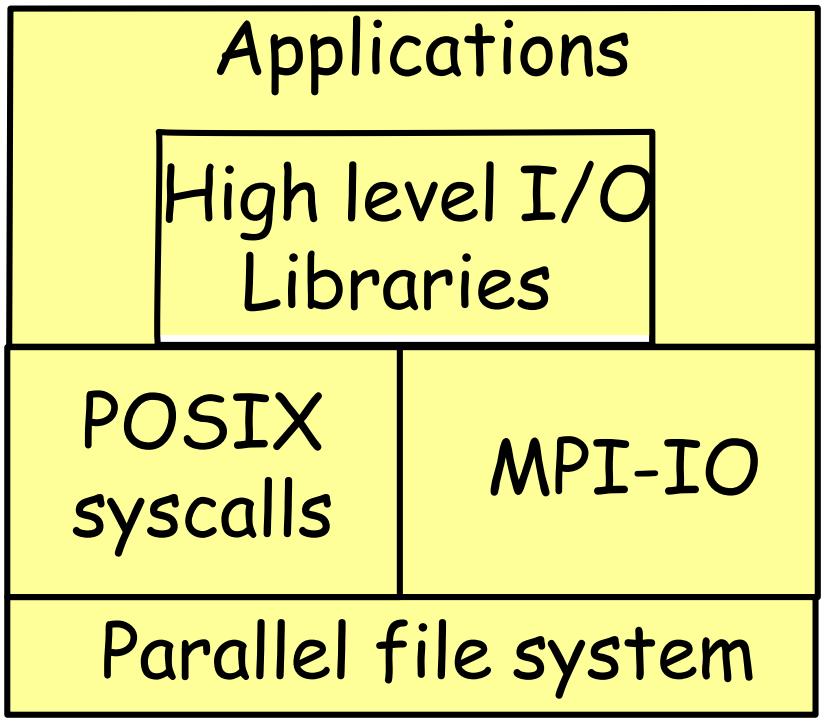


# 并行 IO

- I/O非常重要
  - 性能、可扩展性、可靠性
  - 输出的易用性（文件数量、格式）
- 可移植性
- 上述性质不能同时获得，我们需要决定什么是最重要的
- 新的挑战
  - 任务数量迅速增长
  - 数据规模迅速增长
- I/O 调节的需求要根据具体算法和问题进行分析
- 如果没有并行，I/O将成为几乎每个应用程序的可扩展性瓶颈!

# IO层次

- 上层
  - 应用: 需要从硬盘中读取数据
- 中间层
  - I/O库和系统工具
  - 库
    - HDF5, netcdf
  - MPI-I/O
  - POSIX system calls (fwrite / WRITE)
- 底层:
  - 并行文件系统实现真正的并行I/O
  - Lustre, GPFS, PVFS, dCache

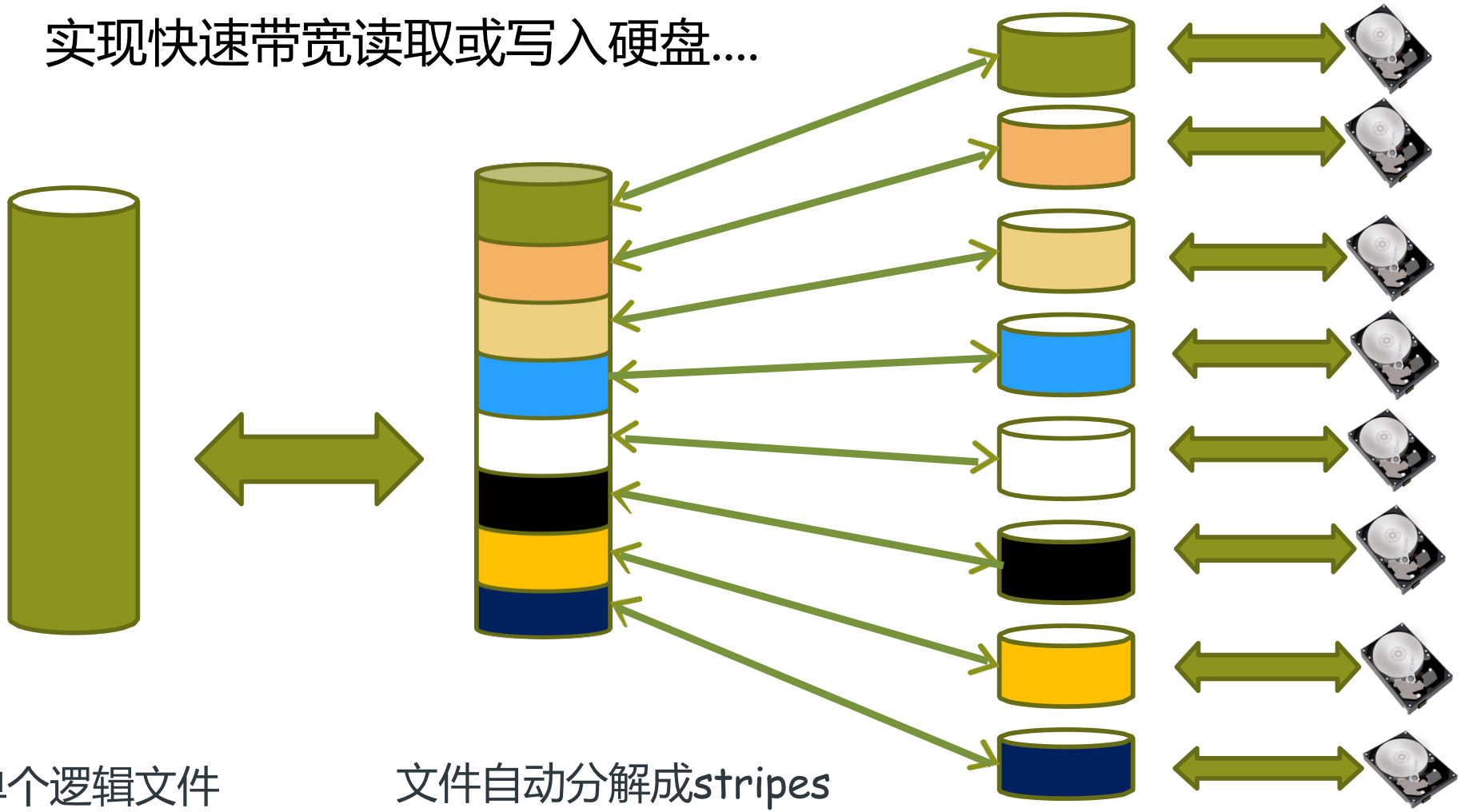


# POSIX IO

- 用于实现I/O的内置语言结构
  -WRITE/READ/OPEN/CLOSE in Fortran
  -stdio.h routines in C (fopen, fread, fwrite, ...)
- 没有内置并行能力-所有并行I/O方案必须手动编程
- 二进制输出不一定是可移植的
- C和Fortran的二进制输出不一定兼容
- 非连续访问难以高效实现
- 连续访问速度可以非常快

# 文件系统基础

实现快速带宽读取或写入硬盘....

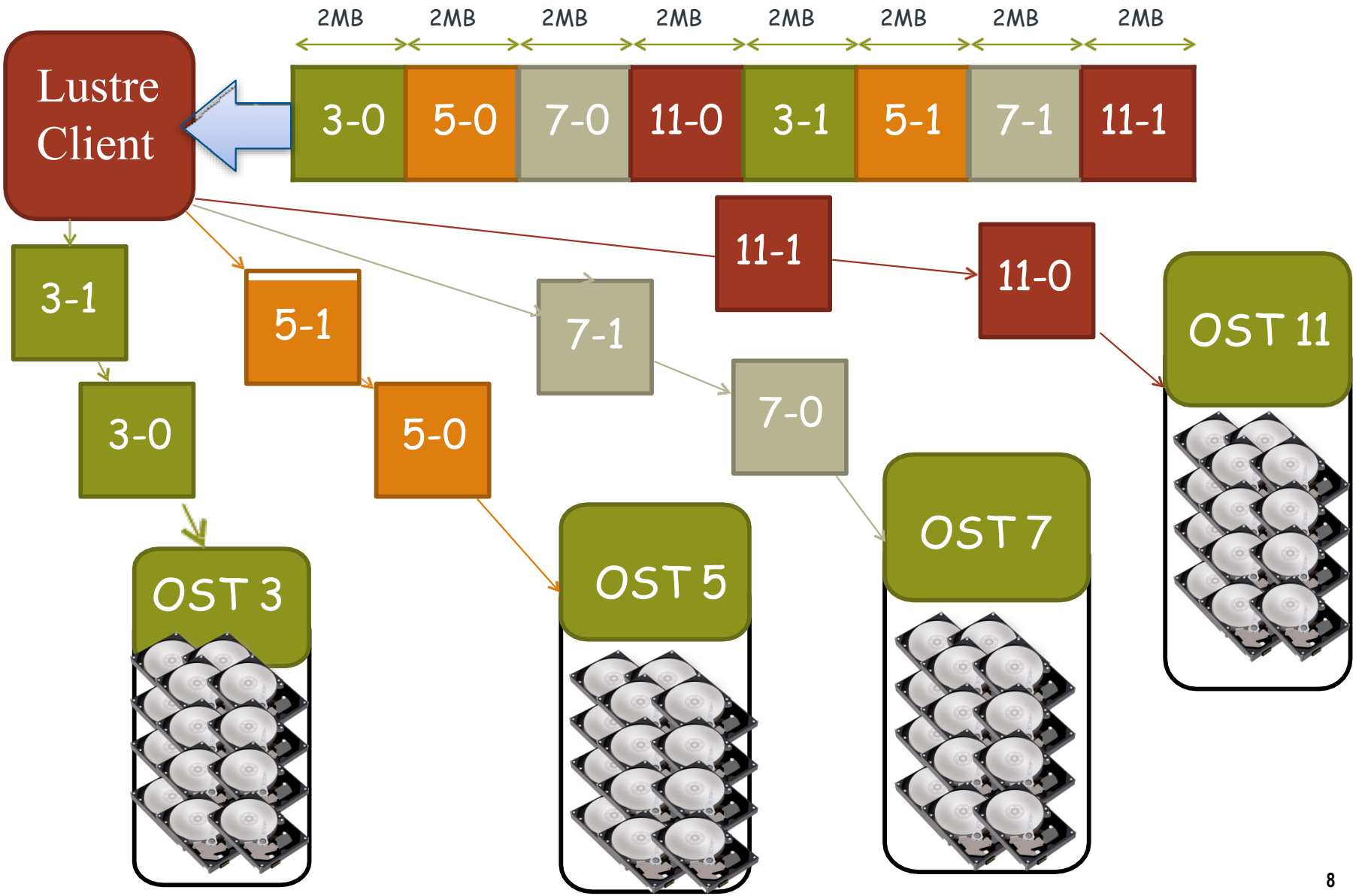


单个逻辑文件
e.g. /work/example

文件自动分解成stripes

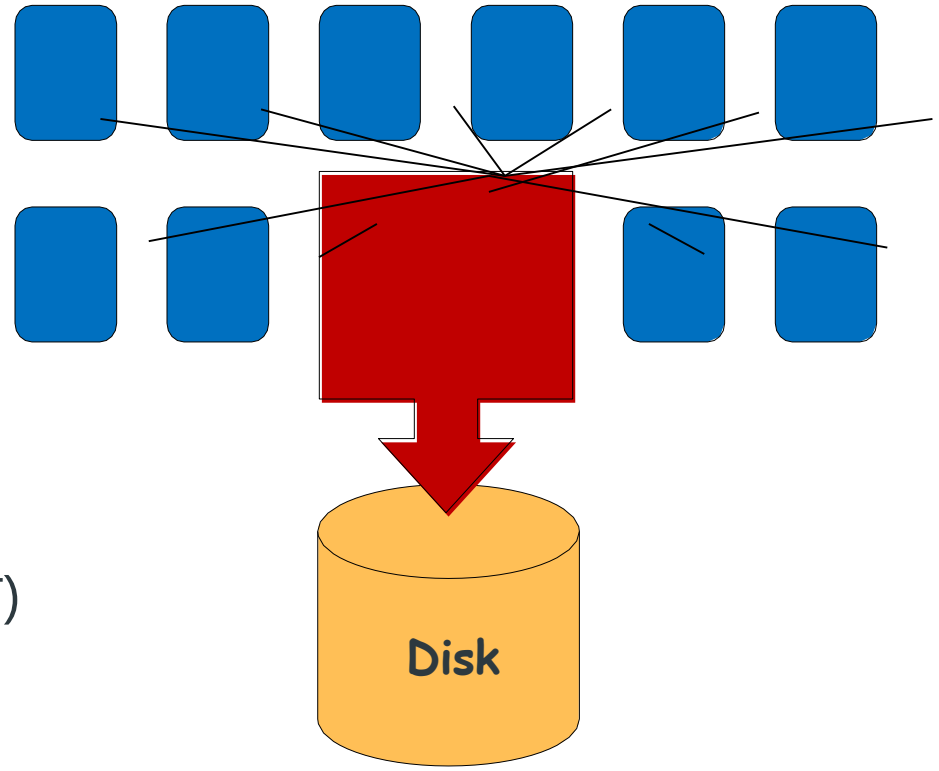
Stripes从多个驱动器中读/写

# 文件分解 – 2 Megabyte Stripes



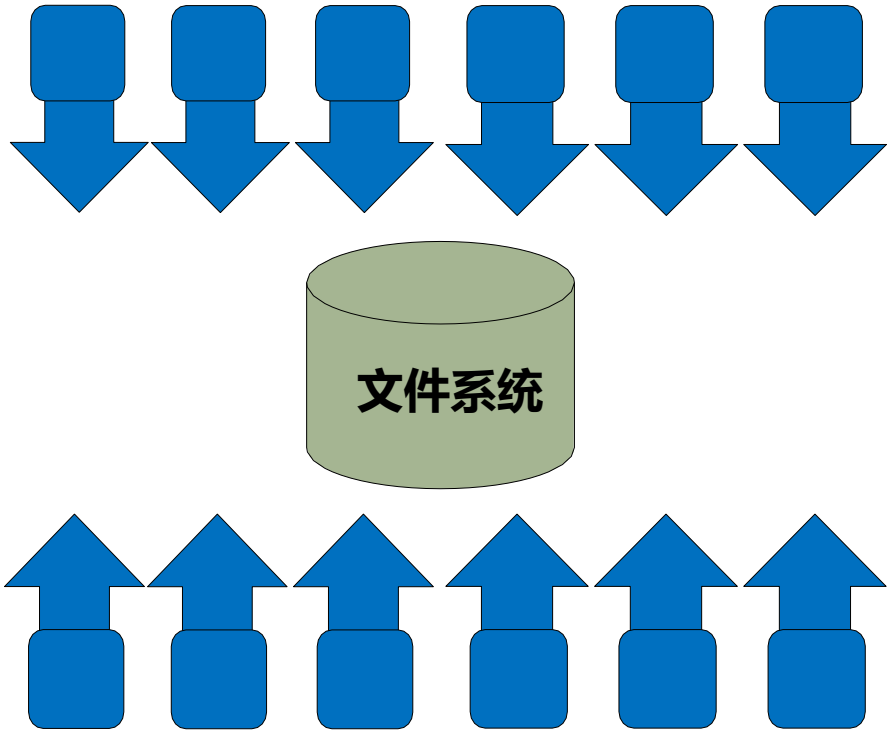
# 独立IO, 基本是串行 IO

- 一个进程执行I/O.
  - 数据聚合或复制受单个 I/O 进程限制
- 易于编程
- 模式不可扩展
  - 时间随数据量线性增加
  - 时间随进程数增加
- 要谨慎对待“全对一”(大规模通信)
- 可以用于专用IO服务器(不易编程),以处理少量数据



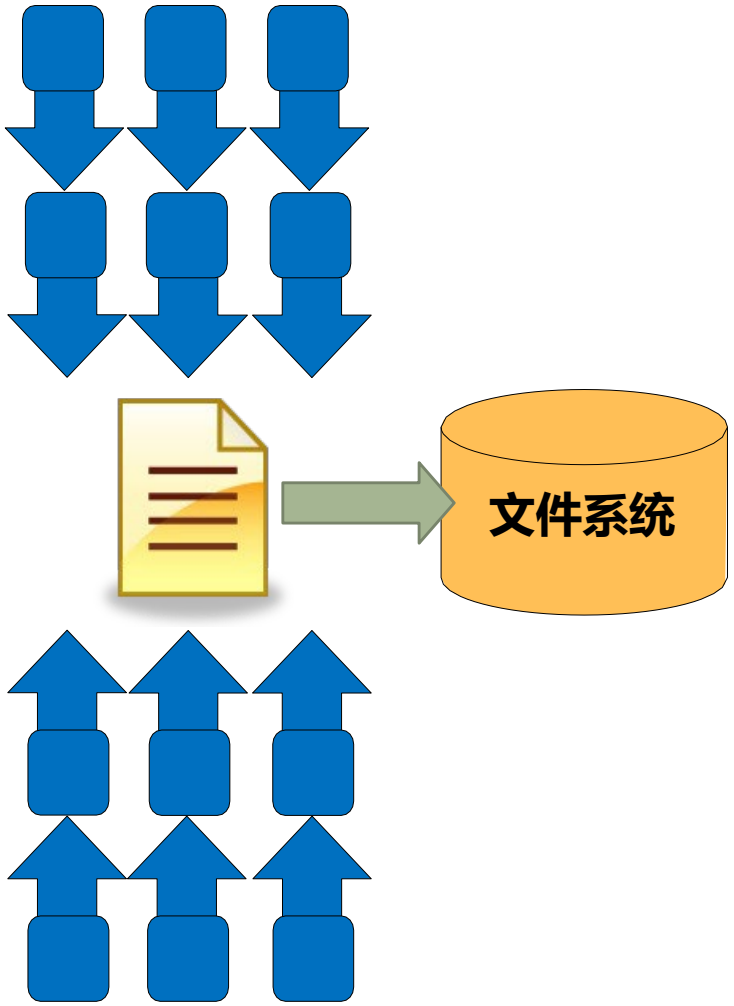
# 每个进程负责一个文件

- 所有进程对单个文件执I/O
  - 受限于文件系统.
- 易于编程
- 模式在大进程数时不可扩展.
  - 文件数量造成元数据操作瓶颈
  - 同时访问硬盘的数量对文件系统资源产生竞争.



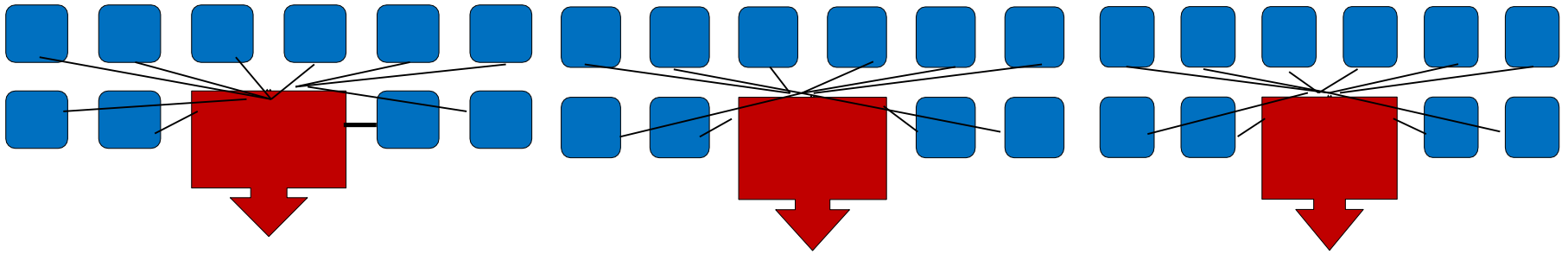
# 共享文件

- 每个进程对共享的单个文件执行I/O操作.
- 性能
  - 共享文件中的数据布局非常重要.
  - 在较大进程数下，可能会产生对文件系统资源的竞争使用(OST).
- 编程语言可能不支持
  - C/C++ 可以使用 fseek
  - 没有真正的Fortran 标准



# 只使用进程的一个子集

- 聚合到处理数据的处理器组.
  - 串行化组中I/O .
- I/O进程可以访问独立的文件.
  - 限制访问文件的数量.
- 一组进程对共享文件执行并行I/O。
  - 增加共享数量，提高文件系统利用率.
  - 减少访问共享文件的进程数量，以减少文件系统竞争



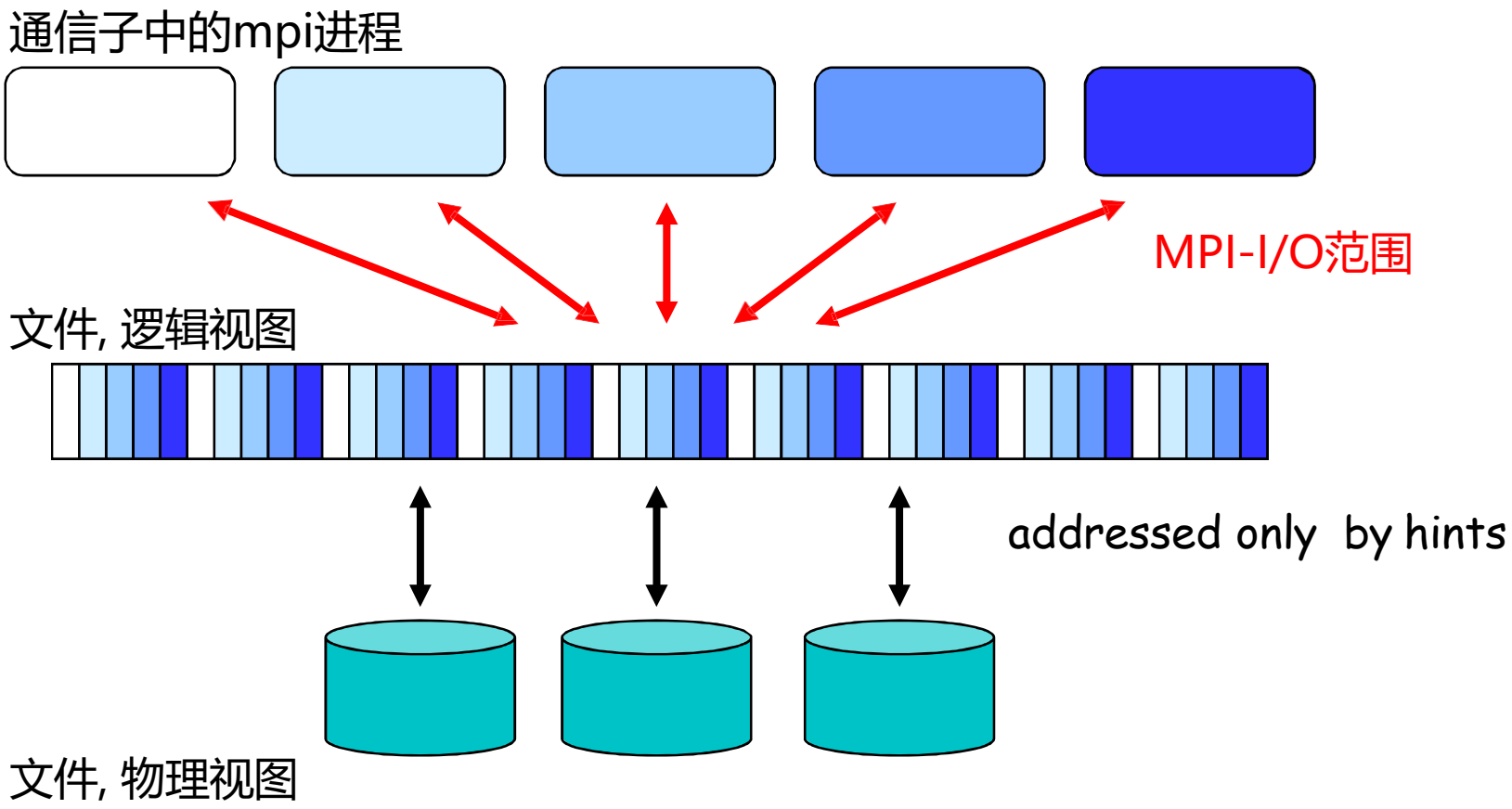
# MPI-IO

- MPI I/O 在MPI-2中才有
- 为读写文件进行定义并行操作
  -针对单个文件的I/O 和/或 针对多文件的I/O
  -连续和非连续 I/O
  -单独 I/O与集合I/O
  -异步I/O
- 可移植的编程接口
- 潜在的良好性能
- 易于使用 (相对于自己实现相同的算法)
- 用作许多并行I/O库的主干，如并行NetCDF和并行HDF5
- 默认情况下，二进制文件不一定是可移植的

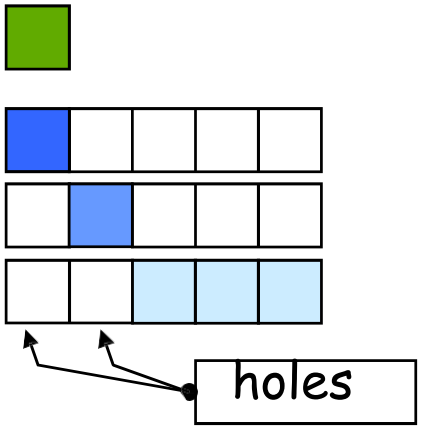
# MPI-IO 基本概念

- 文件句柄【File handle】
  -用于访问文件的数据结构
- 文件指针【File pointer】
  -文件中的读写位置
  -可以是所有进程单独的，也可以在进程之间共享
  -通过文件句柄访问
- 文件视图【File view】
  -文件中对进程可见的部分
  -支持对文件的**高效**非连续访问
- 集合和独立 I/O
  -集合: MPI协调进程的读写
  -独立: MPI不进行协调

# 逻辑视图



# 基本定义



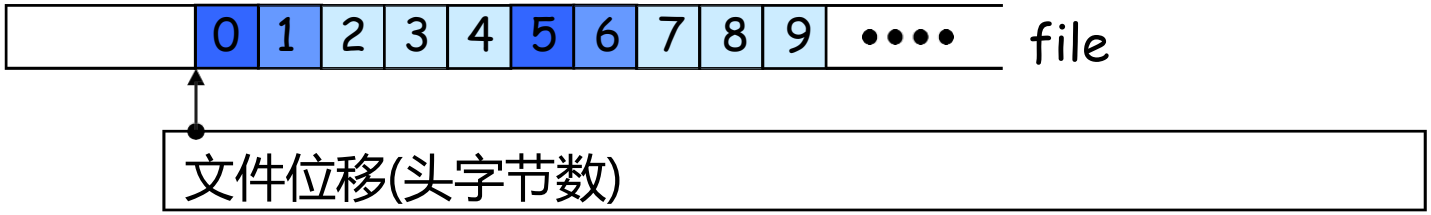

etype (基本数据类型)

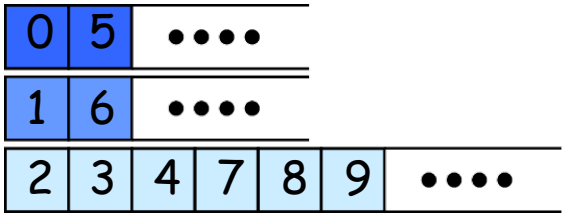
filetype process 0

filetype process 1

filetype process 2

用文件类型平铺(tiling)文件:




view of process 0

view of process 1

view of process 2

# 注记

**file**
- 类型化数据项的有序集合

**etypes**
- 数据访问和定位/偏移的单位
- 可以是任何基本或派生的数据类型
  (非负的，单调非递减的，非绝对的位移)
- 通常是连续的, 但并不一定
- 通常在所有进程中是一样的

**filetypes**
- 在进程之间划分文件的基础
- 定义访问文件的模板
- 每个进程之间是不同的
- etype或由etype派生
  (displacements:非负，单调，非递减，非绝对的，etype范围的倍数)

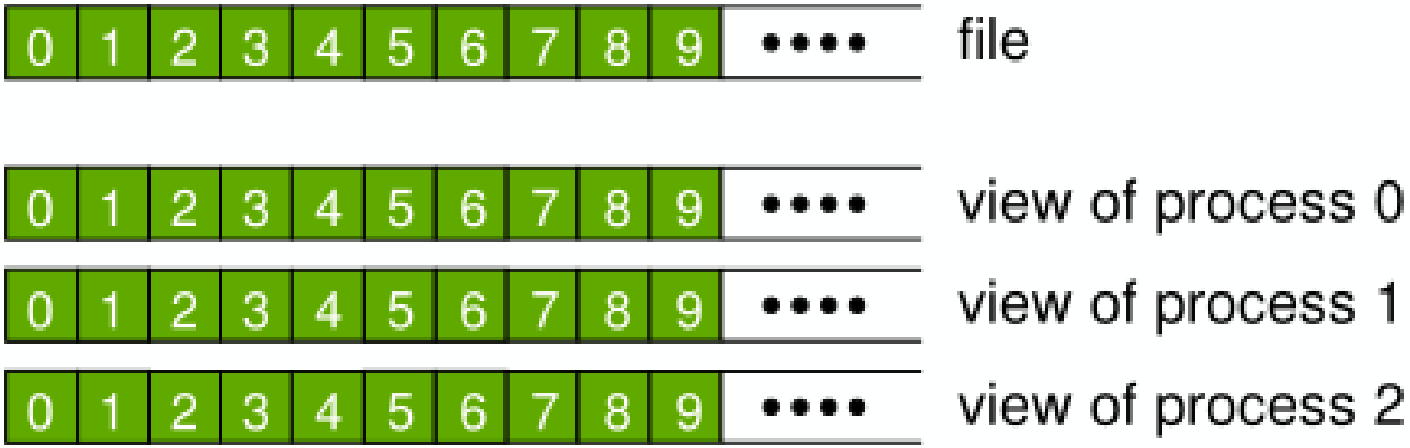
**view**
- 每个进程都有自己的view，由位移、etype和Filetype定义
- 文件类型是重复的，从位移开始

**offset**
- 相对于当view的位置，单位为etype

# MPI_FILE_OPEN(comm, filename, amode, info, *fh*)

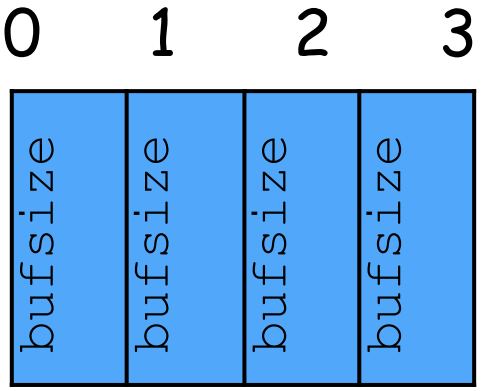
- 默认:
  - displacement = 0
  - etype = MPI_BYTE
  - filetype = MPI_BYTE

  } 每个进程都可以访问整个文件




- MPI_BYTE序列匹配任何数据类型 ( MPI-3.0)
- 二进制 I/O (no ASCII text I/O)

# 一个简单的C语言MPI-IO程序




```
MPI_File fh;
MPI_Status status;
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
bufsize = FILESIZE/nprocs;
nints = bufsize/sizeof(int);
```

```
MPI_File_open(MPI_COMM_WORLD, 'FILE',
     MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
MPI_File_seek(fh, rank * bufsize, MPI_SEEK_SET);
MPI_File_read(fh, buf, nints, MPI_INT, &status);
MPI_File_close(&fh);
```

# MPI的写与读

- 使用 **MPI_File_write** 或者 **MPI_File_read**
- 使用 **MPI_MODE_WRONLY** 或者 **MPI_MODE_RDWR**作为 **MPI_File_open**的flag
- 如果文件之前不存在, **MPI_MODE_CREATE** 必须作为flag被传递给**MPI_File_open**
- 可以用 C语言中的 ‘bitwise-’ 或者 ‘|’, 或者 Fortan语言中的 addition ‘+’ 和 ‘IOR’ 传递多个flag
- 如果不写入文件, 使用**MPI_MODE_RDONLY**可能会产生性能收益. 请尝试一下!

# 并行写入

```
PROGRAM Output
USE MPI
IMPLICIT NONEINTEGER :: err, i, myid, file, intsize
INTEGER :: status(MPI_STATUS_SIZE)
INTEGER, PARAMETER :: count=100
INTEGER,DIMENSION(count) ::buf
INTEGER(KIND=MPI_OFFSET_KIND) :: disp
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid,&err)
DO i = 1, count
      buf(i)
END DO
```

- 多进程写入文件测试.
- 第一个进程在文件开始处中写入整数1-100，等等.

注意!
在本例中，文件(和总数据)大小取决于进程的数量

```
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test', &
     MPI_MODE_WRONLY + MPI_MODE_CREATE,
     &MPI_INFO_NULL, file, err)
CALL MPI_TYPE_SIZE(MPI_INTEGER, intsize, err)
disp = myid * count * intsize

CALL MPI_FILE_SEEK(file,disp,&  MPI_SEEK_SET,  err)
CALL MPI_FILE_WRITE(file, buf, count,
     &MPI_INTEGER, status, err)
CALL MPI_FILE_CLOSE(file, err) CALL
MPI_FINALIZE(err)
END PROGRAM Output
```

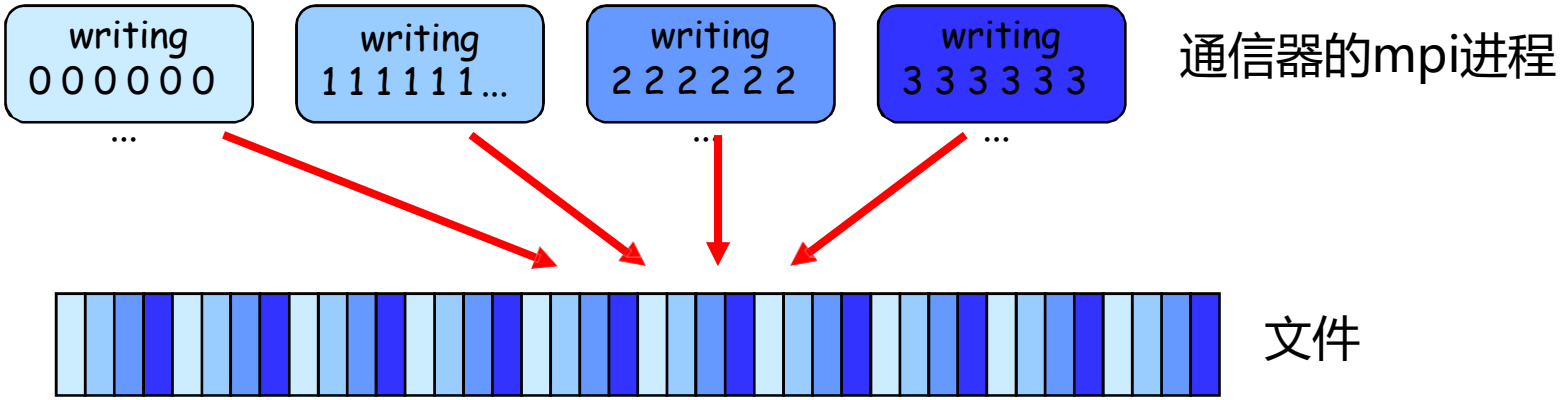
# MPI_FILE_WRITE_AT(fh,offset,buf,count,datatype,*status*)

- 从内存buf将数据类型的count元素写入文件
- 起始offset*etype单位，从view开始
- 元素被存储在当前view的位置中
- Datatype的基本数据类型序列(= signature of **datatype**) 必须匹配当前view的etype的连续拷贝

# 练习: MPI-IO exa:

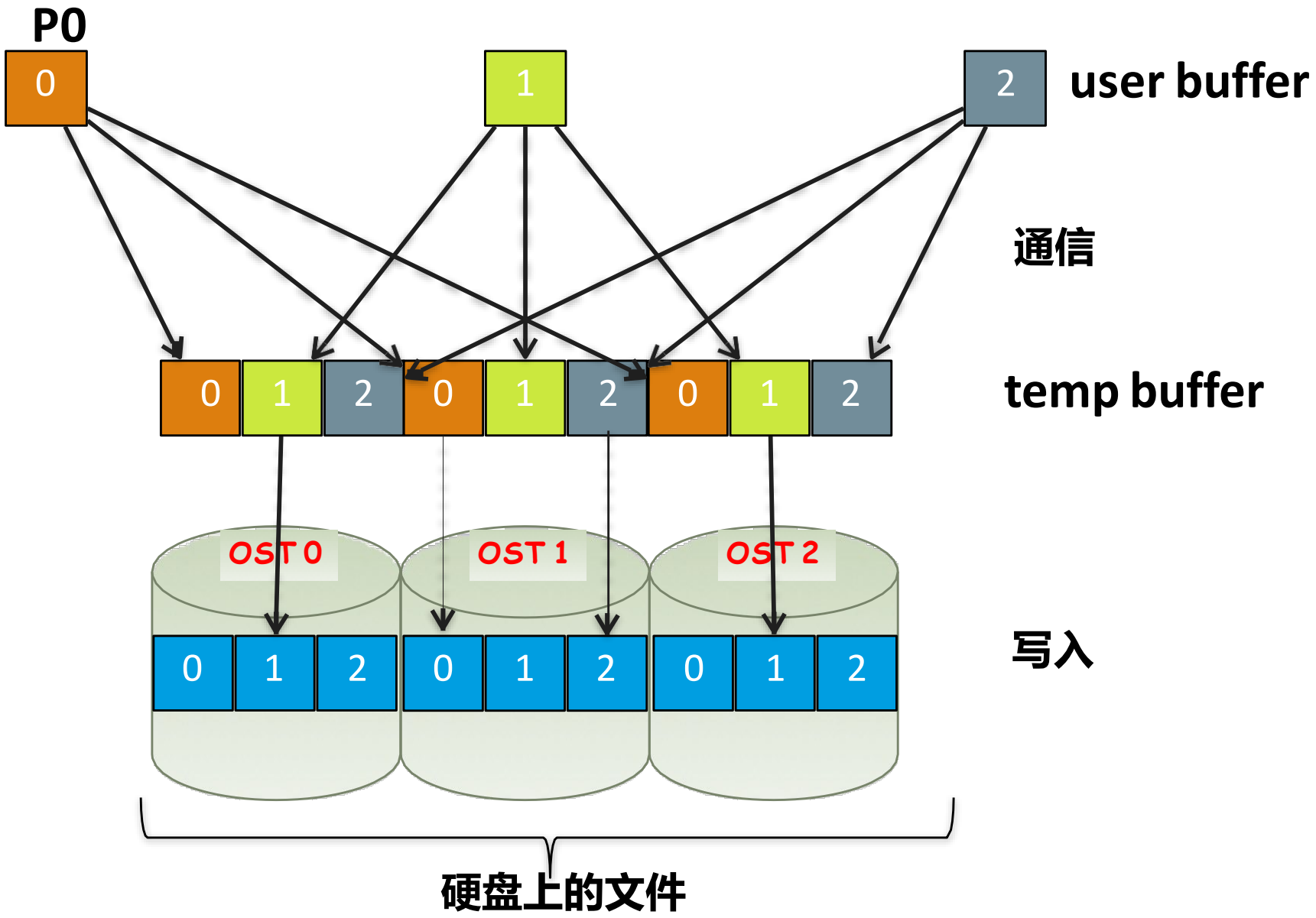
## 四个进程写入一个文件

- 每个进程应该将其rank(作为一个字符)写入10次
  offsets = my_rank + i * size_of_MPI_COMM_WORLD, i=0..9
- 结果: “0123012301230123012301230123012301230123”
- 每个进程使用默认 view




- cd MPI_IO
  cp mpi_io_exa1_skel.c my_exa1.x

# mpi_io_exa.c


P0
0
1
2
user buffer
通信
0 1 2 0 1 2 0 1 2
temp buffer
OST 0
OST 1
OST 2
0 1 2 0 1 2 0 1 2
写入
硬盘上的文件

# 文件视图

- 提供一组从打开的文件中可见和可访问的数据
- 通过triple := (displacement, etype, filetype)， 每个进程都可以看到文件的一个独立视图
- 用户可以在程序执行期间更改视图
  - 但需要集合操作
- 线性字节流, 用三元组(0,MPI_BYTE, MPI_BYTE)表示, 是默认视图

# MPI_File_set_view

```
MPI_File_set_view(fhandle, disp, etype, filetype, datarep,info)
```

*disp* 从文件开始的偏移量，总是以字节为单位

*etype* 基本MPI类型或用户定义类型

数据访问的基本单位

I/O命令中的偏移是etype单位

*filetype* 与etype或由etype构造的用户定义类型相同

指明文件按中哪部分是可见的

*datarep* 数据表示，有时对可移植性有帮助

“native”:与内存中相同格式存储

*info* 可以提升性能的提示

MPI_INFO_NULL: No hints

# MPI_File_set(get)_view

- 设置 view
  - 更改进程的数据视图
  - 本地和共享文件指针被重置为零
  - 集合操作
  - etype 和 filetype 必须被提交【committed】
  - datarep 参数是一个字符串，指明数据写入文件的格式:
    “native”，“internal”，“external32”，或者用户定义格式
  - 所有进程中etype extent和datarep都相同
- 获取 view
  - 返回进程的数据视图

MPI_FILE_SET_VIEW(fh, disp, etype, filetype, datarep, info)

MPI_FILE_GET_VIEW(fh, disp, etype, filetype,datarep)

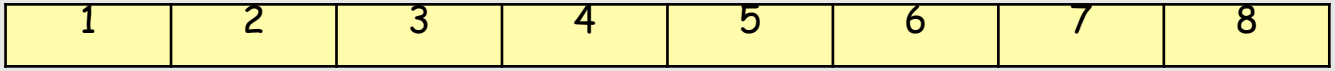
# 参数datarep 指定文件中的数据格式

- "native" 文件中数据完全按其在内存中的格式存放。使用该数据格式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。使用该数据格式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 通用数据表示格式
- external data representation (简称XDR)。使用 该数据格式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格式不兼容的计 算机间交换数据。

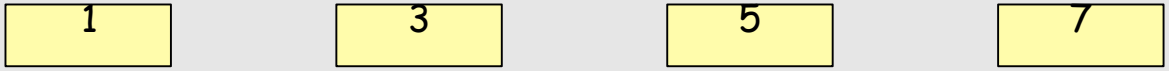
# 文件视图

- 文件视图定义了文件的哪些部分对进程是“可见的”
- 文件视图还定义了文件中数据的类型 (byte, integer,float, ...)
- 默认情况下，文件被视为由字节组成，进程可以访问(读或写)文件中的任何字节
- 文件视图由以下三部分组成
  - **displacement**: 从文件开始跳过的字节数
  - **etype**: 访问的数据类型，定义偏移量的单位
  - **filetype**:文件中对进程可见的部分

Default file view



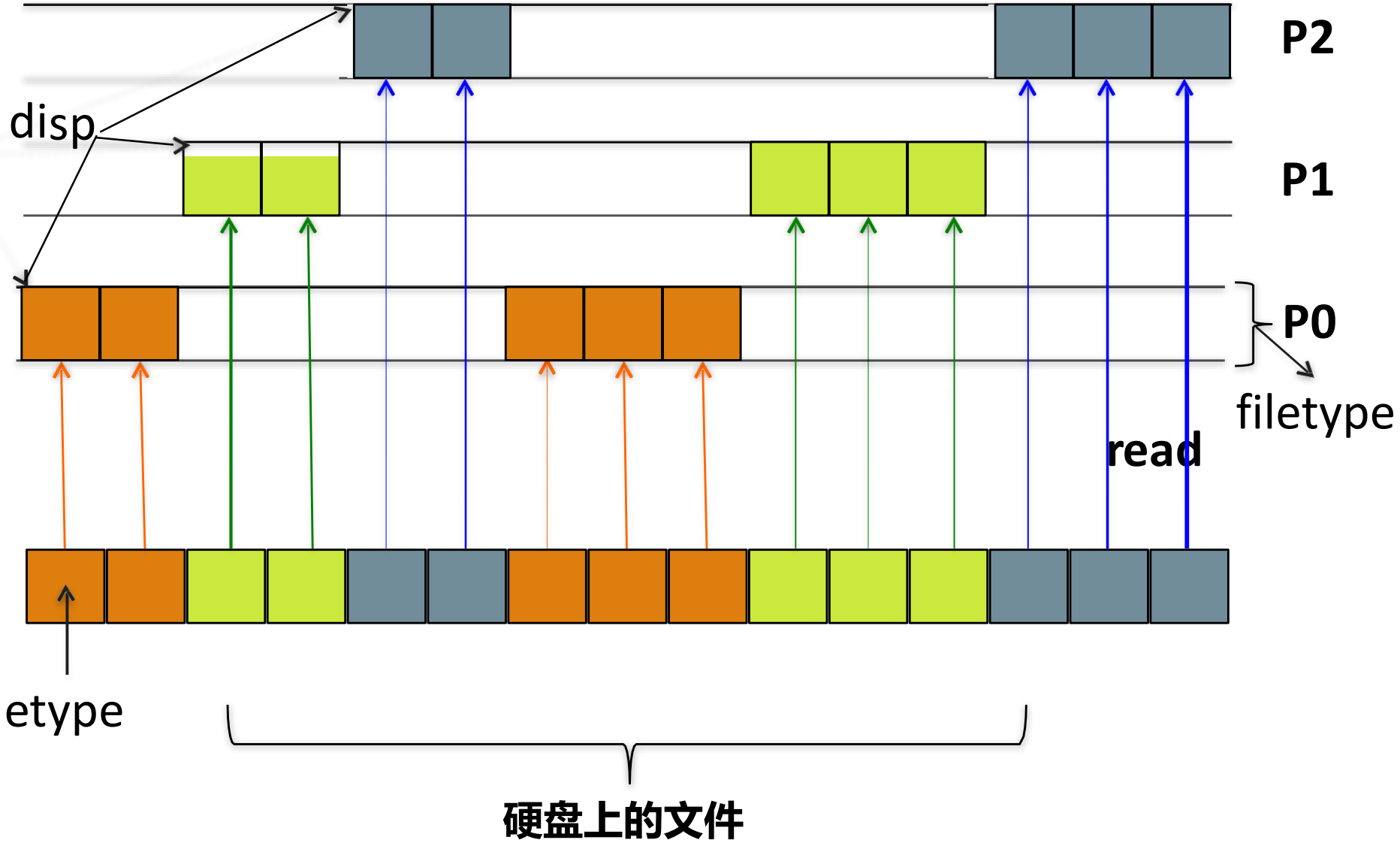

**etype**=MPI_INT
**filetype**=MPI_INT



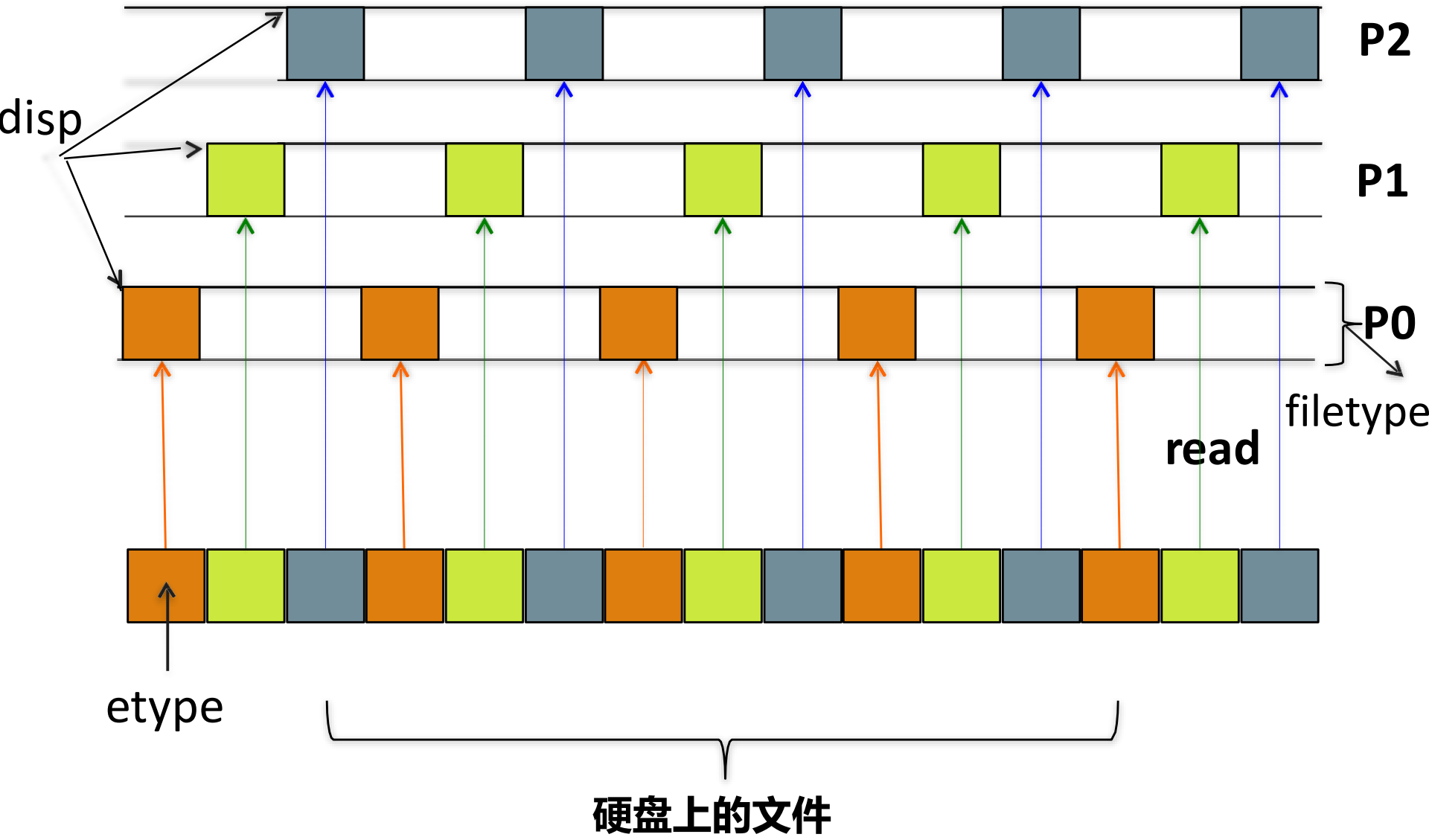

**etype**=MPI_INT

**filetype**=MPI_Type_vector(4, 1, 2, MPI_INT, &filetype);

# MPI_File_set_view (图 1)

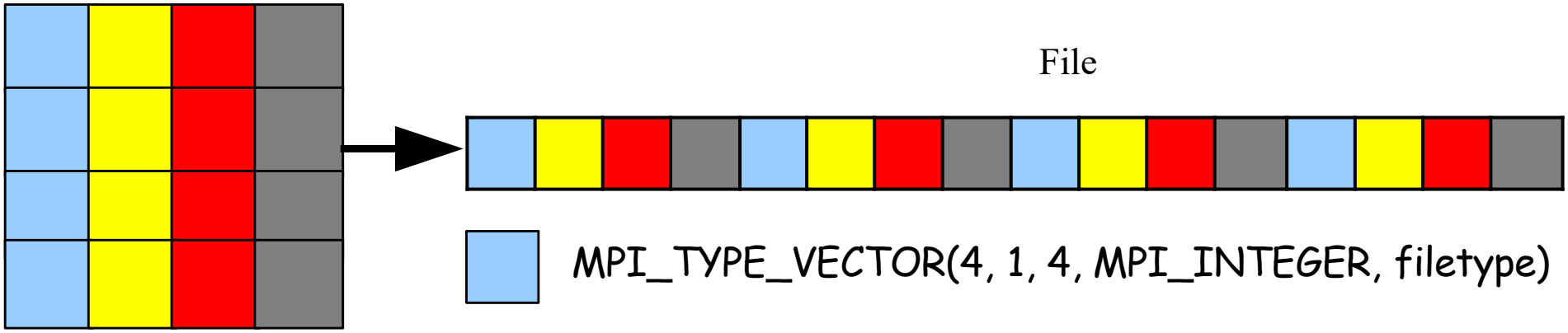


# MPI_File_set_view (图 2)



# 非连续数据的文件视图

2维数组-按列分布




```
...
 INTEGER :: count = 4   INTEGER,
 DIMENSION(count) :: buf
 ...
 CALL MPI_TYPE_VECTOR(4, 1, 4, MPI_INTEGER, filetype, err)
 CALL MPI_TYPE_COMMIT(filetype, err)
 disp = myid * intsize
 CALL CALL MPI_FILE_SET_VIEW(file, disp, MPI_INTEGER, filetype, "native", &
      MPI_INFO_NULL, err)
  CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)
...
```

# MPI IO

- **MPI 接口支持两种IO类型:**

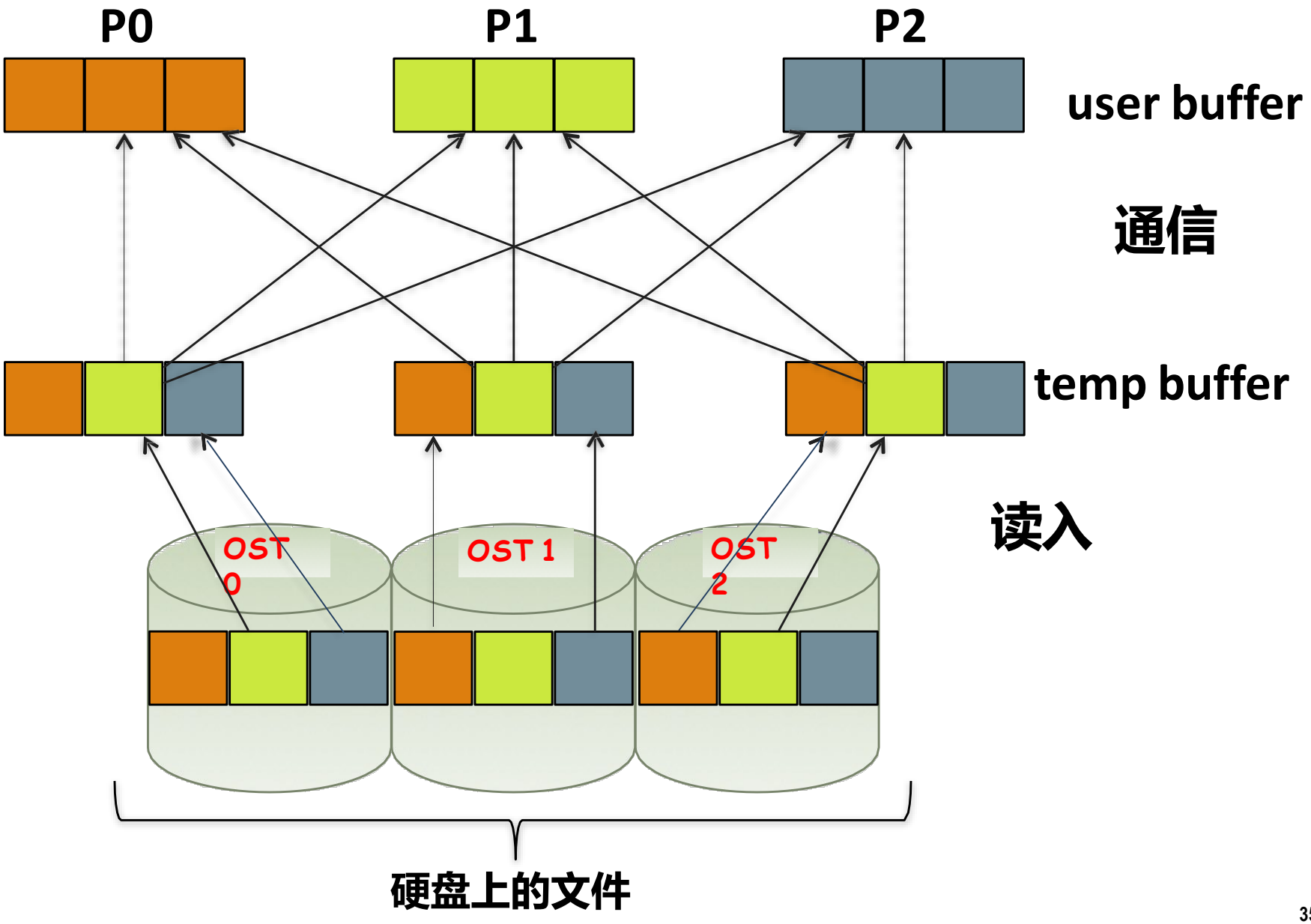
- **独立IO**
  - 每个进程独立处理自己的I/O
  - 支持派生数据类型

- **集合IO**
  - 调用必须由参与特定I/O序列的所有进程发起
  - 采用“文件共享，全写”策略，通过MPI库进行动态优化。

# MPI-IO 集合IO

- MPI_File_read_all, MPI_File_read_at_all, …
  "_all"表示由传递给MPI_File_open的通信器指定的进程组中的所有进程都将调用此函数。
- 每个进程仅指定自己的访问信息 - 参数列表与非集合函数相同。
- 在这种情况下，MPI-IO库提供了很多信息：
  - 读取或写入数据的进程集合
  - 区域的结构化描述
- 对于如何使用这些数据，库有一些选项
  - 非连续数据访问优化
  - 集体I/O优化

# 集合读入: 两阶段 IO



# 集合 IO

- I/O 可以由通信器中的所有进程共同执行

  **MPI_File_read_all**

  **MPI_File_write_all**

  **MPI_File_read_at_all**

  **MPI_File_write_at_all**

- 与独立I/O函数中的参数相同

  **MPI_File_read, MPI_File_write, MPI_File_read_at, MPI_File_write_at**

- 通信器中打开文件的所有进程都必须调用函数
- 性能可能比单个函数更好

  -即使每个处理器读取非连续的段

# 总结

- POSIX
  - 单个读/写，全部读/写，子集读/写
  - 用户负责通信
- MPI I/O
  - MPI库负责通信
  - 文件视图支持非连续访问模式
  - 集合I/O可以使实际的硬盘访问保持连续

# THANKS