国家超级计算天津中心
National SuperComputer Center in Tianjin

2019年用户培训交流会

IO优化案例详解

李云龙　应用研发工程师
国家超级计算天津中心　liyl@nscc-tj.cn

目 录 CONTENTS

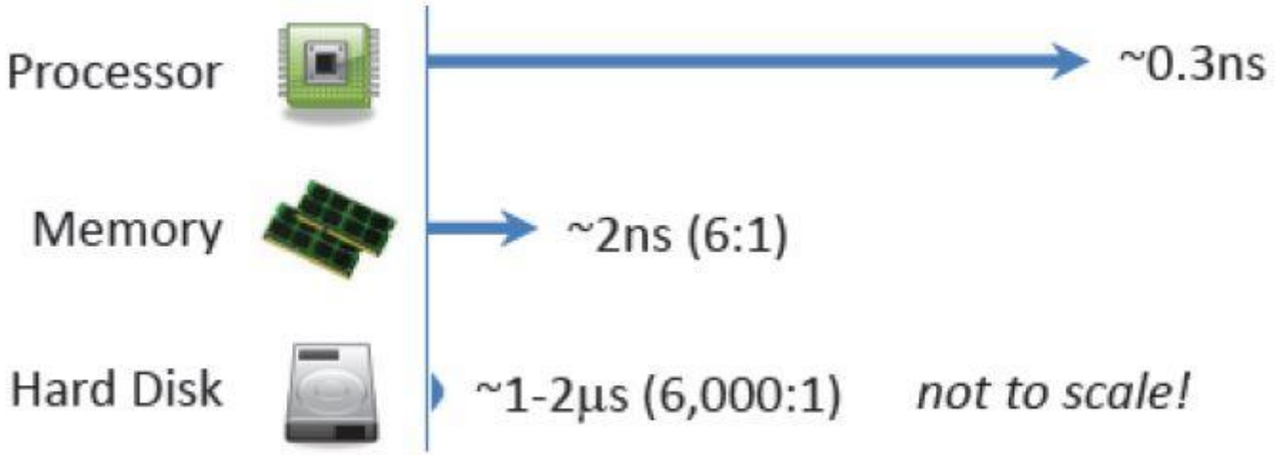
01 章节 PART

为什么要优化I/O（Why）

- I/O概述
- 串行IO

并行程序运行时间影响因素

- 程序总运行时间=计算时间+通信时间+I/O时间
- 实际上，大部分并行程序均包含如下过程：
- 程序运行开始，读取初始文件数据；
- 程序运行过程中，输出中间结果文件，以便程序报错后重新启动运行；
- 程序运行结束，输出最终结果数据；



I/O访问是影响并行程序运行效率的一个重要因素，其性能的好坏与并行计算的效率直接相关；

01 计算性能和I/O性能存在较大差异

- Case/Amdahl经验法则表明，1MIPS的计算能力需要有1MB的I/O带宽支持；
- 存储设备带宽增长要远远落后于计算能力的增长，从而单机系统的计算能力与I/O差距越来越大，CPU和I/O设备速度增长率不匹配是造成I/O瓶颈的主要原因；

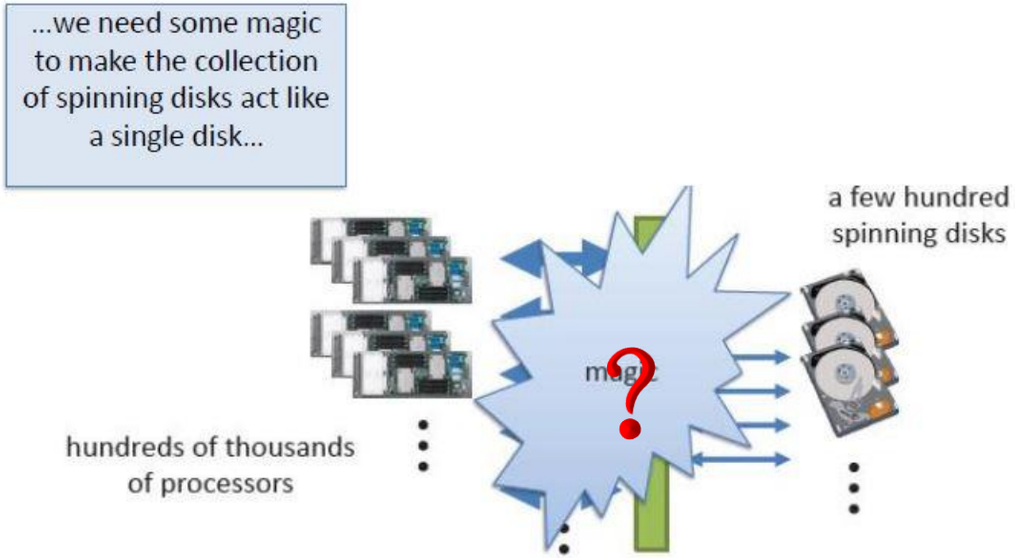
I/O访问是影响并行程序运行效率的一个重要因素，其性能的好坏与并行计算的效率直接相关；

02 对于分布式系统，差异更加明显和突出

- 分布式系统普遍采用多CPU模式，使得计算能力与I/O速度不匹配的问题变得更加严重；

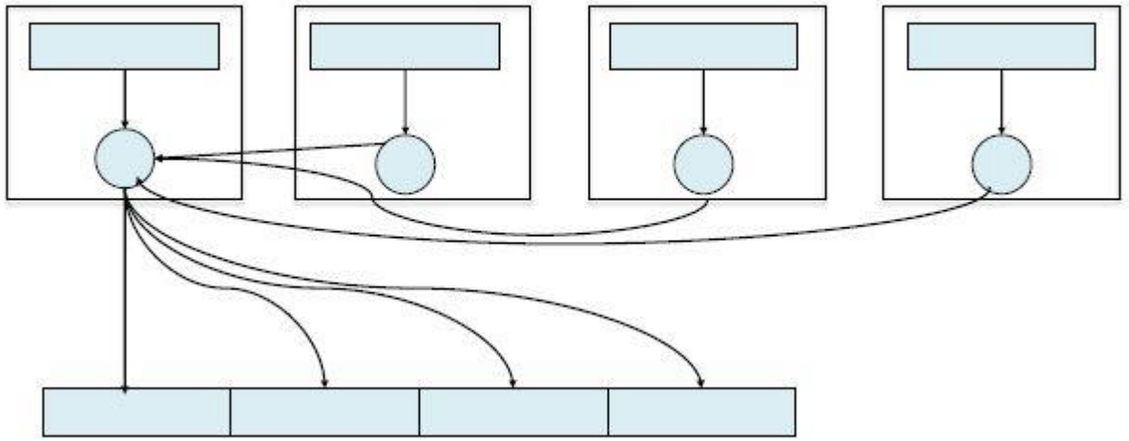
03 应用对I/O性能有更高要求

- 此外，一些应用领域模拟，如气候气象、流体力学等，对系统I/O性能提出了更高的要求，如大规模多数据输出等，并随时间呈现逐级递增的趋势；



串行I/O

- **基本流程**
- 首先，由主进程（通常为0 号进程）收集其他进程的数据；（收集）
- 然后，主进程再将所有数据一起输出；（输出）
- **存在问题**
- 耗时较长；对系统要求和压力较大（带宽）；无法充分发挥超算系统性能；扩展性差；；
- 表现为I/O时间在程序总体运行时间中占比较大，尤其是并行度较大的情况下；



综上所述，针对大规模并行，无论是系统还是程序，I/O都是不可忽视，需要考虑的重要因素之一

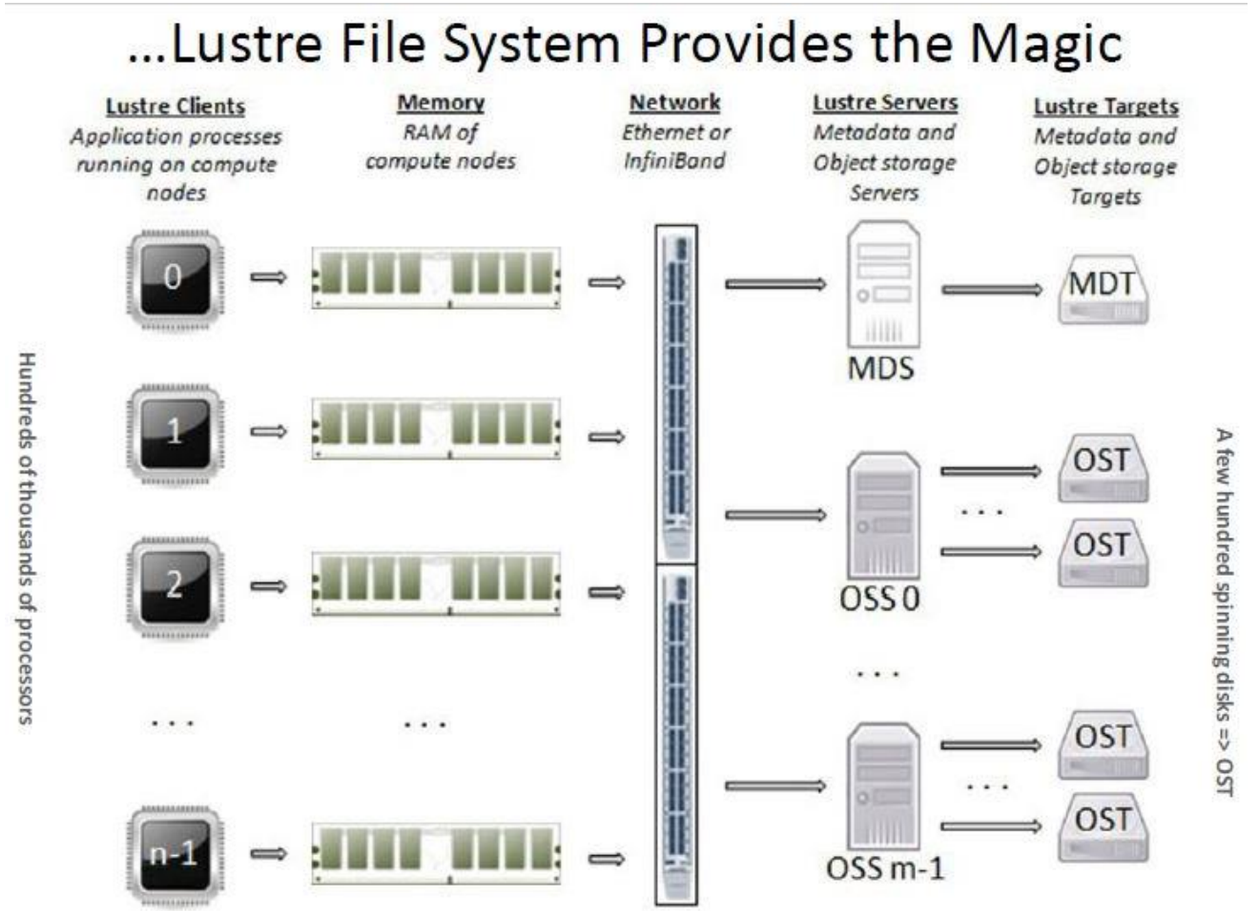
02 章节 PART

什么是IO优化（What）

- Lustre文件系统
- 并行IO

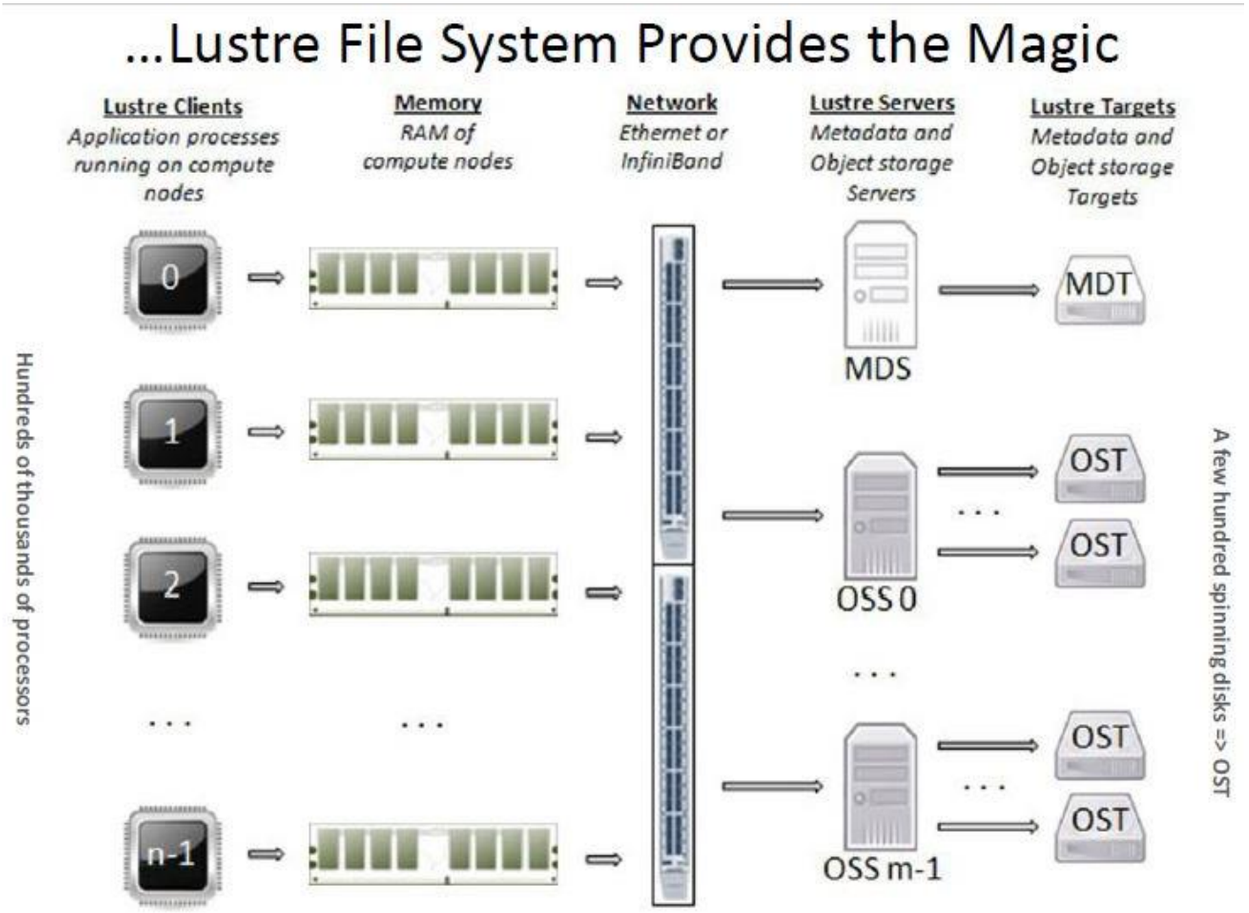
Lustre文件系统

- 一个开源的、全局单个命名空间的、符合POSIX标准的分布式并行文件系统；
- 在基于Linux的操作系统上运行，并采用基于对象的客户端-服务器模式的体系结构；
- 可将许多服务器的存储容量和I/O吞吐量进行聚合。因此，通过动态添置服务器，其可按需灵活方便的扩展容量和提升性能；（高可扩展性）
- Lustre支持数以万计的客户端，PB级存储和每秒数百GB的吞吐量；（高性能）
- 支持有效的数据管理机制、各种高性能低延迟的网络、全局数据共享共享、失效替代和系统快速配置等功能；（高可用性）
- 服务于许多全球最大的高性能计算(HPC)集群；（高可移植性）



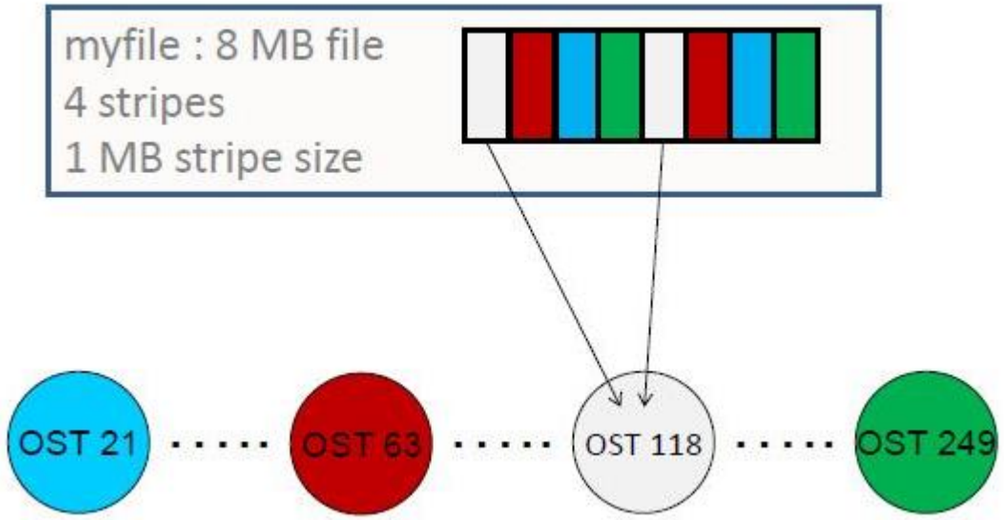
□ Lustre文件系统

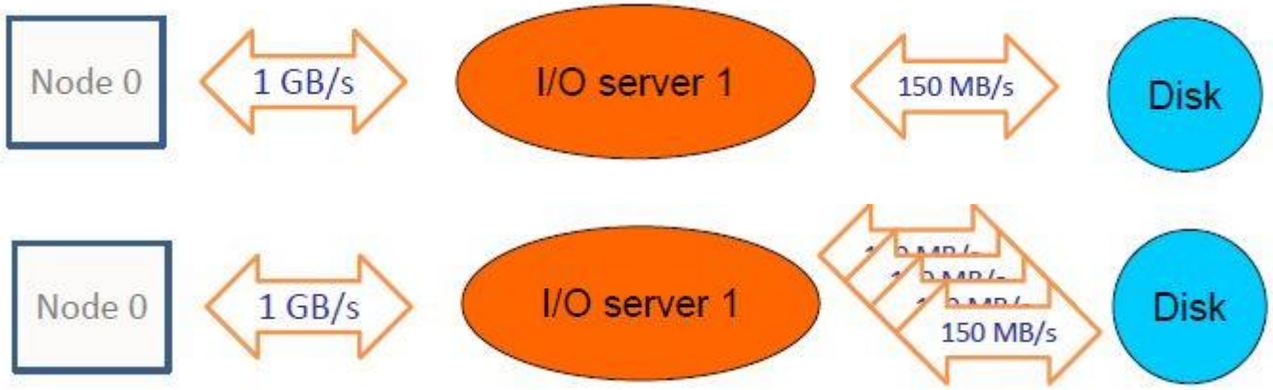
- **元数据服务器(MDS)**
- 为一个或多个本地MDT提供网络请求处理，使存储在一个或多个MDT中的元数据可供客户端使用；
- 负责元数据服务，并管理整个文件系统的命令空间；
- **元数据目标(MDT)**
- 在MDS的附加存储上存储元数据对象（例如文件名，目录，权限和文件布局）；
- **对象存储服务器(OSS)**
- 为一个或多个本地OST提供文件I／O服务和网络请求处理，负责客户端和物理存储之间的交互及数据存储；
- **对象存储目标(OST)**
- 用户文件数据存储在一个或多个对象中，每个对象位于Lustre文件系统的单独OST中。每个文件的对象数由用户配置，并可根据工作负载情况调试到最优性能；（条带化）
- **Lustre客户端**
- 是挂载了Lustre文件系统的任意节点；
- 用户可通过客户端透明的访问整个文件系统的数据；



口 条带化

- Lustre文件系统高性能的主要原因之一是能够以轮询方式跨多个OST将数据条带化；
- 所谓条带化，即允许将文件中的数据段或“块”存储在不同的OST中；
- 可根据需要为每个文件配置条带数量、条带大小和OST；
- 当单文件总带宽超过单OST带宽时，或当单OST没有足够空间容纳整个文件时，可以使用条带化来提高性能；
- 通过命令行和MPI函数均可实现设置；





并行I/O方法

01 **应用程序层I/O**

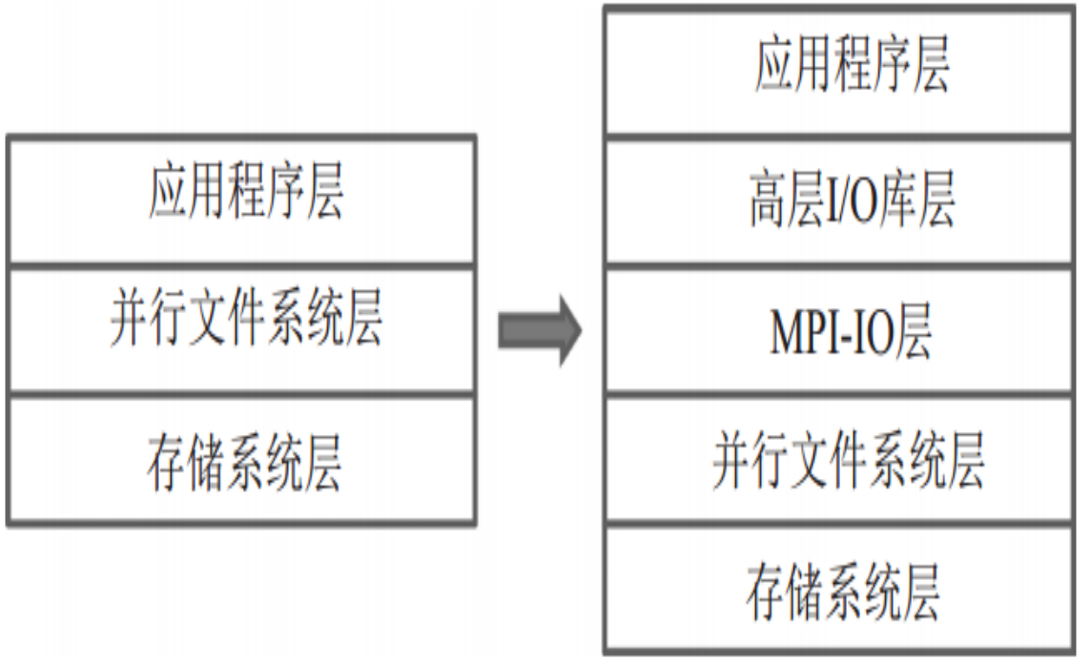
缺乏灵活性，每次使用新I/O方法时，不得不修改应用程序；

02 **并行文件系统层I/O**

缺乏可移植性，需在大量并行文件系统上分别实现I/O优化，增加难度；

03 **I/O中间件层**

包括MPI-IO和高层I/O库，不受应用程序和并行文件系统的限制，更容易被应用；



目前，已经发展了诸多并行I/O中间件方法，包括MPI-IO、pnetcdf、hdf5、adios等；

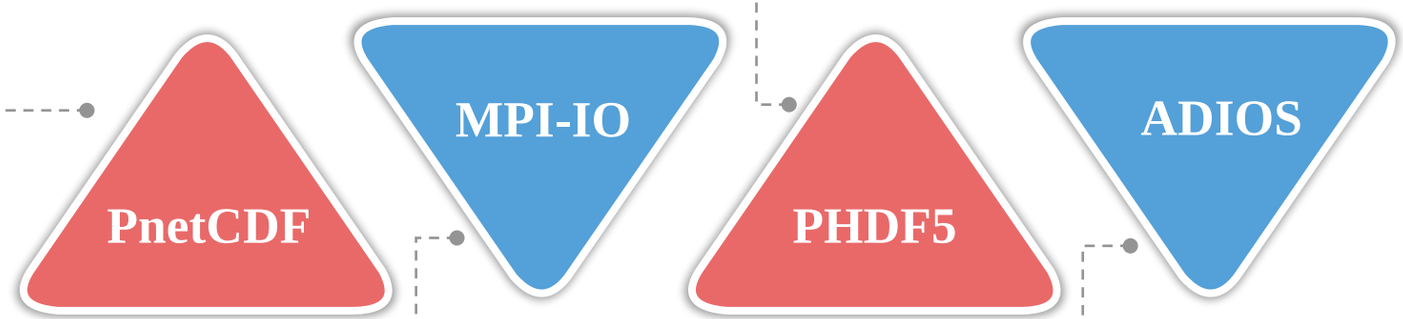
口 并行I/O方法

专为读写NetCDF格式设计的并行I/O库

- 所有进程并行访问一个共享文件；
- 只有6种基本数据类型，不支持自定义数据类型；
- 使用线性数据布局，追加数据会造成很大开销；

专为读写HDF5格式设计的并行I/O库

- 具有两个数据概念：数据集和组；
- 访问对象时，需遍历整个命名空间获得头部信息，而降低了效率；



通用基础并行I/O标准

- 一个标准、可移植的高性能并行文件I/O 接 口 ，为MPI-2标准的一部分；是大部分I/O库的基础库；
- 包括文件指针、文件视口、分布式数组读写、聚合 I/O等，以此实现灵活、高效的I/O行为；
- 输出多种格式文件；

多方法更灵活的并行I/O库

- 由橡树岭国立实验室开发,为一个包含不同 I/O 传输方法的 I/O 组件；
- 独立XML文件，更方便；自定义Bp格式，避免文件冲突；适用不同计算机结构和文件系统，更灵活；

章节 PART 03

怎样优化I/O（How）

- MPI-IO方法基本概念
- MPI-IO方法基础函数和代码样例
- MPI-IO方法基本流程

基础概念

MPI通信域

- 包括进程组和通信上下文；
- 进程组，即所有参加通信的进程的集合，若一共有N个进程参加通信则进程的编号从0到N-1；
- 通信上下文，提供一个相对独立的通信区域，不同的消息在不同的上下文中进行传递，将通信区分开；

句柄

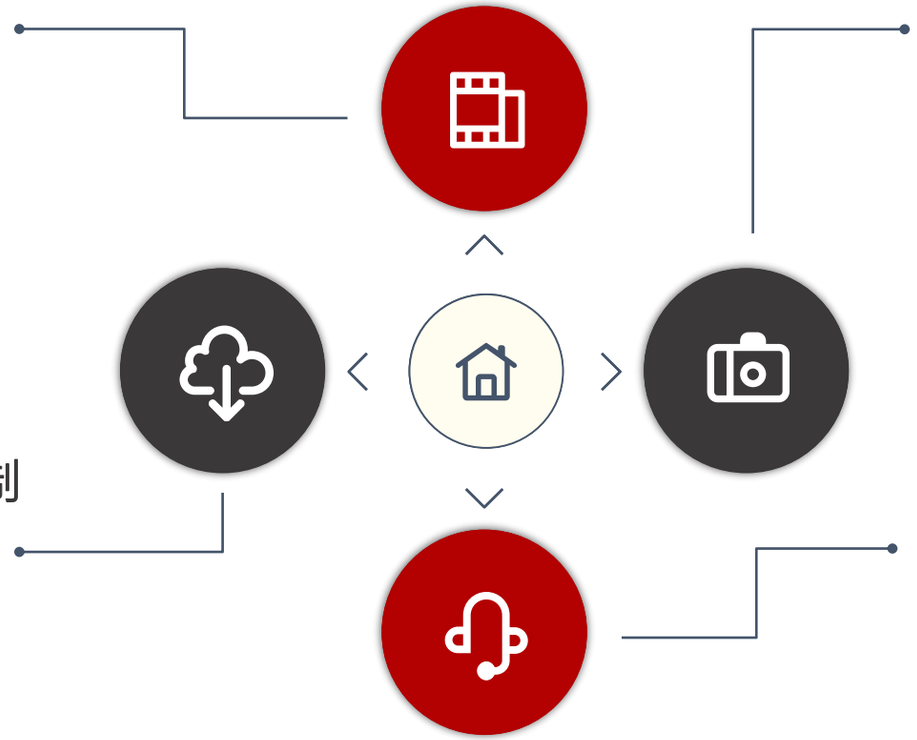
- 文件描述符，或者标识符；
- 用于标识打开的文件或存储等，并以此来对文件进行操作；
- 可理解为指针，但不完全是指针，无法直接使用其对文件数据进行访问；

根据读写定位方法

1. 指定显示偏移；
2. 各进程拥有独立的文件指针，需定义“视口”；
3. 共享文件指针；

根据对参加读写操作的进程的限制

1. 独立读写：单个进程可以实现的读写操作不需要其它进程的参与；
2. 组读写：要求所有的进程都必须执行相同的读写调用，但是提供给该调用的读写参数可以不同；



根据同步机制的不同

1. 阻塞调用：读写调用完成后，才进行下一步；
2. 非阻塞调用：读写调用执行后即进行下一步，无论是否完成；（单步和两步）

附加说明

1. 上述方式均有都有独立读写和组读写的调用；
2. 与文件句柄相联系的通信域就是组读写所使用的通信域；

MPI-2提供的关于并行文件I/O的调用十分丰富

- 显式偏移的并行文件读写
- 多视口的并行文件并行读写
- 共享文件读写
- 阻塞和非阻塞读写方式
- 独立读写和组读写

表格 20 各种并行文件I/O调用

读写定位方法	同步机制		各进程间的关系	
			独立读写	组读写
指定显式偏移	阻塞		READ_AT WRITE_AT	READ_AT_ALL WRITE_AT_ALL
	非阻塞	单步法	IREAD_AT IWRITE_AT	
		两步法		READ_AT_ALL_BEGIN READ_AT_ALL_END WRITE_AT_ALL_BEGIN WRITE_AT_ALL_END
独立的文件指针	阻塞		READ WRITE	READ_ALL WRITE_ALL
	非阻塞	单步法	IREAD IWRITE	
		两步法		READ_ALL_BEGIN READ_ALL_END WRITE_ALL_BEGIN WRITE_ALL_END
共享文件指针	阻塞		READ_SHARED WRITE_SHARED	READ_ORDERED WRITE_ORDERED
	非阻塞	单步法	IREAD_SHARED IWRITE_SHARED	
		两步法		READ_ORDERED_BEGIN READ_ORDERED_END WRITE_ORDERED_BEGIN WRITE_ORDERED_END

口 文件打开和关闭

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

IN comm 组内通信域

IN filename 将打开的文件名

IN amode 打开方式

IN info 传递给运行时的信息

OUT fh 返回的文件句柄

```
int MPI_File_open(MPI_Comm comm, char * filename, int amode, MPI_Info info,
                  MPI_File  * fh)
MPI_FILE_OPEN(COMM,FILENAME, AMODE, INFO, FH,IERROR)
    CHARACTER *(*)   FILENAME
  INTEGER COMM, AMODE, INFO, FH, IERROR
```

表格 21 文件打开方式

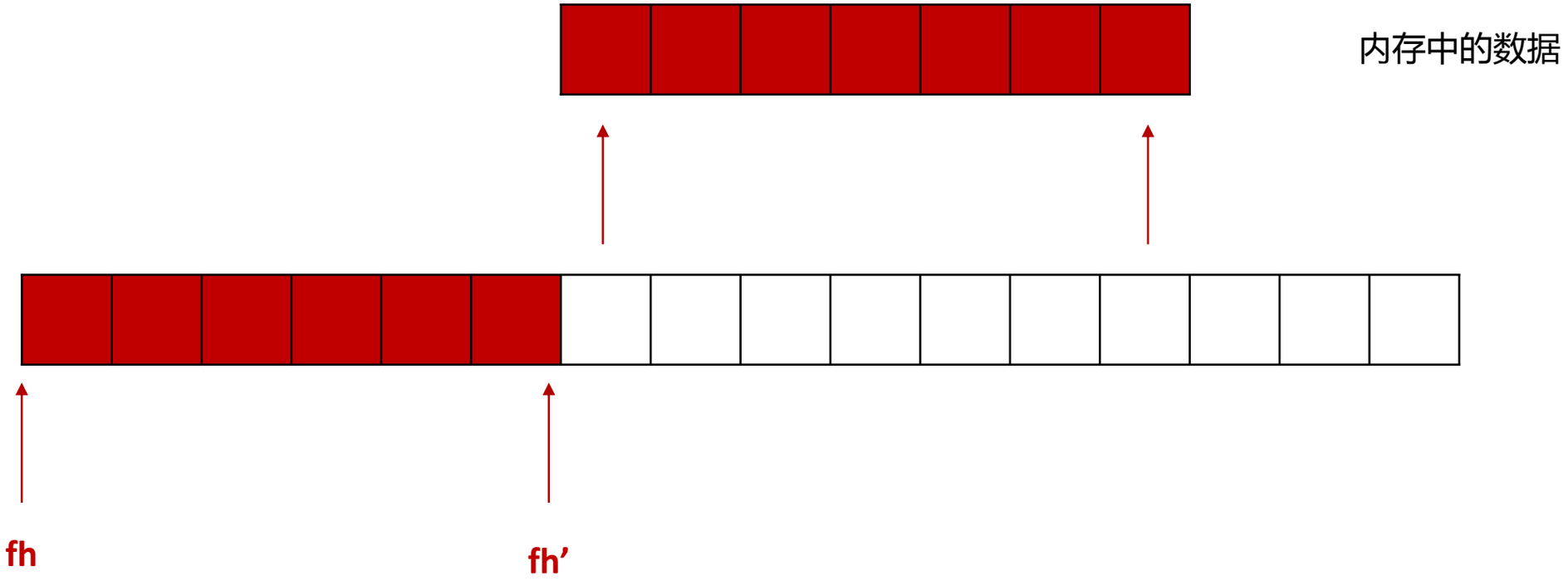
打开方式	含义
MPI_MODE_RDONLY	只读
MPI_MODE_RDWR	读写
MPI_MODE_WRONLY	只写
MPI_MODE_CREATE	若文件不存在则创建
MPI_MODE_EXCL	创建不存在的新文件，若存在则错
MPI_MODE_DELETE_ON_CLOSE	关闭时删除
MPI_MODE_UNIQUE_OPEN	不能并发打开
MPI_MODE_SEQUENTIAL	文件只能顺序存取
MPI_MODE_APPEND	追加方式打开，初始文件指针指向文件尾

MPI_FILE_CLOSE(fh)

INOUT fh 前面打开的文件句柄

```
int MPI_File_close(MPI_File * fh)
MPI_FILE_CLOSE(FH,IERROR)
    INTEGER FH, IERROR
```

连续数据操作



存储文件中数据结构

口 数据读取

```
MPI_FILE_SEEK(fh, offset, whence)
INOUT    fh        文件句柄
IN       offset    相对偏移位置
IN       whence    指出offset的参照位置
int MPI_File_seek(MPI_File fh, MPI_Offset offset, int whence)
MPI_FILE_SEEK(FH,OFFSET,WHENCE,IERROR)
   INTEGER FH, WHENCE, IERROR
    INTEGER (KIND=MPI_OFFSET_KIND)  OFFSET
```

表格 22 不同文件位置参照点的含义

参照位置取值	调用后文件指针的位置
MPI_SEEK_SET	offset
MPI_SEEK_CUR	当前指针位置+offset
MPI_SEEK_END	文件结束位置+offset

```
MPI_FILE_READ(fh, buf,count,datatype,status)
INOUT    fh        文件视口句柄
OUT      buf       读出数据存放的缓冲区
IN       count     读出数据个数
IN       datatype  读出数据的数据类型
OUT      status    操作完成后返回的状态
int  MPI_File_read(MPI_file fh, void * buf, int count, MPI_Datatype datatype,
        MPI_Status  * status)
MPI_FILE_READ(FH, BUF, COUNT,DATATYPE, STATUS, IERROR)
      <type>  BUF(*)
      INTEGER  FH,COUNT,DATATYPE,STATUS,IERROR
```

```
MPI_FILE_READ_AT(fh, offset,buf,count,datatype,status)
IN       fh        文件句柄
IN       offset    读取位置相对于文件头的偏移
OUT      buf       读取数据存放的缓冲区
IN       count     读取数据个数
IN       datatype  读取数据的数据类型
OUT      status    返回的状态参数
int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void * buf, int count,
        MPI_Datatype datatype, MPI_Statye * status)
MPI_FILE_READ_AT(FH, OFFSET, BUF, COUNT,DATATYPE,STATUS, IERROR)
     <type>  BUF(*)
     INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
     INTEGER(KIND=MPI_OFFSET_KIND) OFFSET
```

Example 1

```
#include<stdio.h>
#include "mpi.h"
#define FILESIZE 80
int main(int argc, char **argv){
  int rank, size, bufsize, nints;
  MPI_File fh;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  bufsize = FILESIZE/size;
  nints = bufsize/sizeof(int);
  int buf[nints];
  MPI_File_open(MPI_COMM_WORLD,"dfile",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);
  printf("\nrank: %d, buf[%d]: %d", rank, rank*bufsize, buf[0]);
  MPI_File_close(&fh);
  MPI_Finalize();
  return 0;
```

MPI_File_read_at(fh, rank*bufsize, buf, nints, MPI_INT, &status);

定义文件指针

打开文件

数据大小

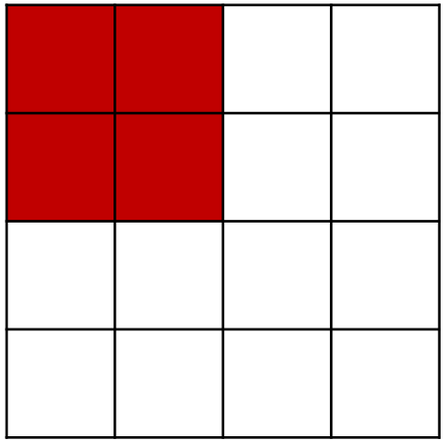
数据个数

设置指针位置

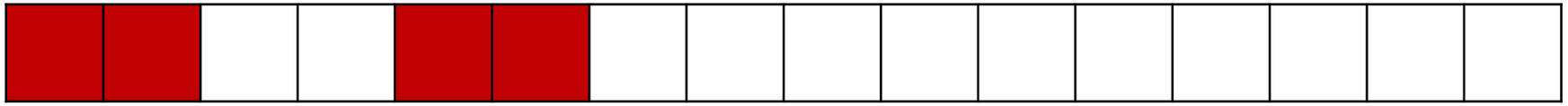
读取数据

关闭文件

不连续数据操作



实际处理数据数组结构



存储文件中数据结构

口 数据输出

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

INOUT	fh	视口对应文件的文件句柄
IN	disp	视口在文件中的偏移位置
IN	etype	视口基本数据类型
IN	filetype	视口文件类型
IN	datarep	视口数据的表示方法
IN	info	传递给运行时的信息

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp, MPI_Datatype etype,
    MPI_Datatype filetype, char * datarep, MPI_Info info)
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP, INFO,
    IERROR)
    CHARACTER *(*) DATAREP
    INTEGER FH, ETYPE, FILETYPE,INFO,IERROR
```

MPI_FILE_WRITE(fh, buf,count,datatype,status)

INOUT	fh	视口文件句柄
IN	buf	将要写入文件的数据存放的缓冲区
IN	count	写入数据的个数
IN	datatype	写入数据的数据类型
OUT	status	该调用返回的状态参数

```
int  MPI_File_write(MPI_file fh, void * buf, int count, MPI_Datatype datatype,
      MPI_Status  * status)
MPI_FILE_WRITE(FH, BUF, COUNT,DATATYPE, STATUS, IERROR)
     <type>  BUF(*)
     INTEGER  FH,COUNT,DATATYPE,STATUS,IERROR
```

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

INOUT	fh	文件句柄
IN	offset	写入文件数据的起始偏移地址
IN	buf	将写入数据存放缓冲区的起始地址
IN	count	写入数据的个数
IN	datatype	写入数据的数据类型
OUT	status	写入操作完成后返回的状态信息

```
int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void * buf, int count,
    MPI_Datatype datatype, MPI_Status * status)
MPI_FILE_WRITE_AT(FH, OFFSET, BUF, COUNT, DATATYPE, STATUS,
    IERROR)
  <type> BUF(*)
  INTEGER FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
  INTEGER (KIND=MPI_OFFSET_KIND) OFFSET
```

视口设置

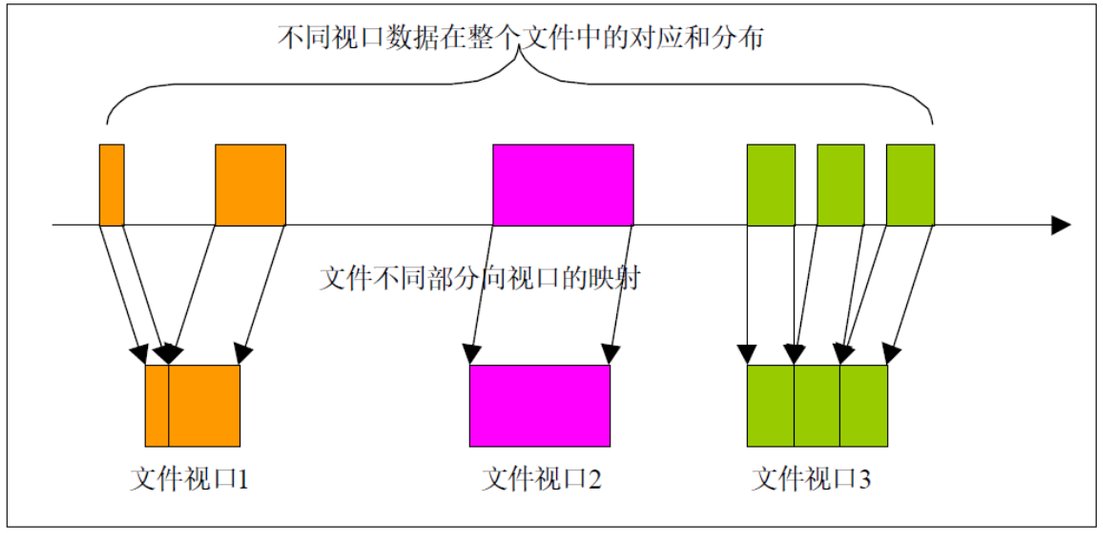


图 92 文件与视口的关系图示

文件与视口

- **相对于某一进程来说的它是特定进程所能看到的文件**
  - 某一进程的文件视口可以是整个文件；
  - 多数情况下文件视口只是整个文件的一个或几个部分；
- **在整个文件中对应的部分可以是不连续的但各个进程看到的其文件视口中的数据却是连续的；**
- **原文件句柄fh不再代表该文件而是代表本调用产生的文件视口，以后使用fh对文件的所有操作都是对其视口的操作；**

□ 视口设置

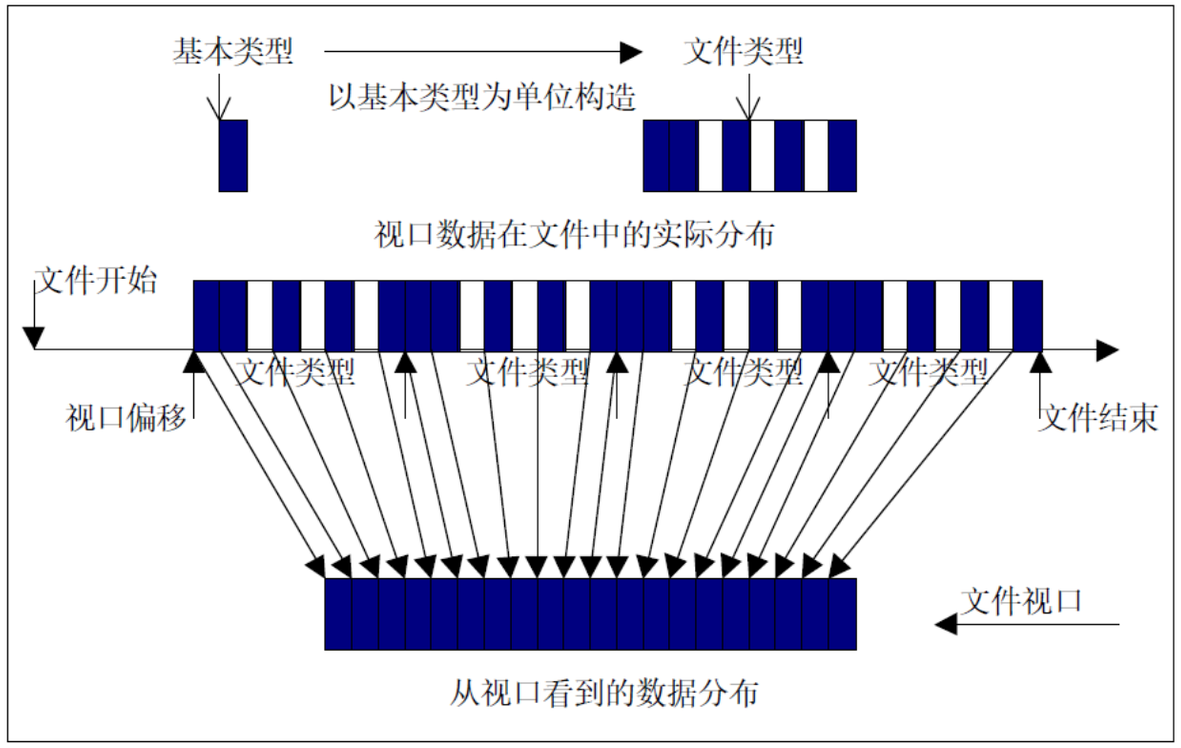


图 93 视口与基本类型、文件类型和文件的关系图示

视口指针

- **MPI_FILE_SET_VIEW(fh, disp,etype,filetype,datarep,info)**
- 是组调用，所有与fh相联系的进程组中进程都执行这一调用；
- **数据类型可以是基本类型，也可以是自定义类型；**
- **其中视口数据的表示方法共有三种：native internal和external32，其影响数据存取速率和一致性；**
- Native：数据在文件中存储方法和内存中一样，不损失精度，传输速率高，但不同系统数据无法完全匹配使用；
- Internal：放宽条件，相同MPI实现可以在不同类型的机器上实现数据转换；速率介于中间；
- External32：完全放宽条件，任何系统产生的数据文件均能相互使用，但效率最差，文件第一次读写时，均需要进行一次同一格式数据转换；

Example 2

```
#include<stdio.h>
#include "mpi.h"
int main(int argc, char **argv){
  int i, rank, size, offset, nints, N=16 ;
  MPI_File fhw;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  int buf[N];
  for ( i=0;i<N;i++){
        buf[i] = i ;
  }
```

```
  offset = rank*(N/size)*sizeof(int);

  MPI_File_open(MPI_COMM_WORLD, "datafile",
MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &fhw);

  printf("\nRank: %d, Offset: %d\n", rank, offset);

  MPI_File_write_at(fhw, offset, buf, (N/size),
MPI_INT, &status);

  MPI_File_close(&fhw);

  MPI_Finalize();
  return 0;
```

```
MPI_File_open(MPI_COMM_WORLD, "datafile3",
MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL,
&fhw);
  printf("\nRank: %d, Offset: %d\n", rank,
offset);
  MPI_File_set_view(fhw, offset, MPI_INT,
MPI_INT, "native", MPI_INFO_NULL);
  MPI_File_write(fhw, buf, (N/size), MPI_INT,
&status);
  MPI_File_close(&fhw);
  MPI_Finalize();
  return 0;
```

打包和解包

- 打包(Pack)和解包(Unpack)操作是为了发送不连续的数据，在发送前显式地把数据包装到一个连续的缓冲区，在接收之后从连续缓冲区中解包；
- MPI_PACK(inbuf, incount, datatype, outbuf, outcount, position, comm)
- MPI_UNPACK(inbuf, insize, position, outbuf, outcount, datatype, comm)

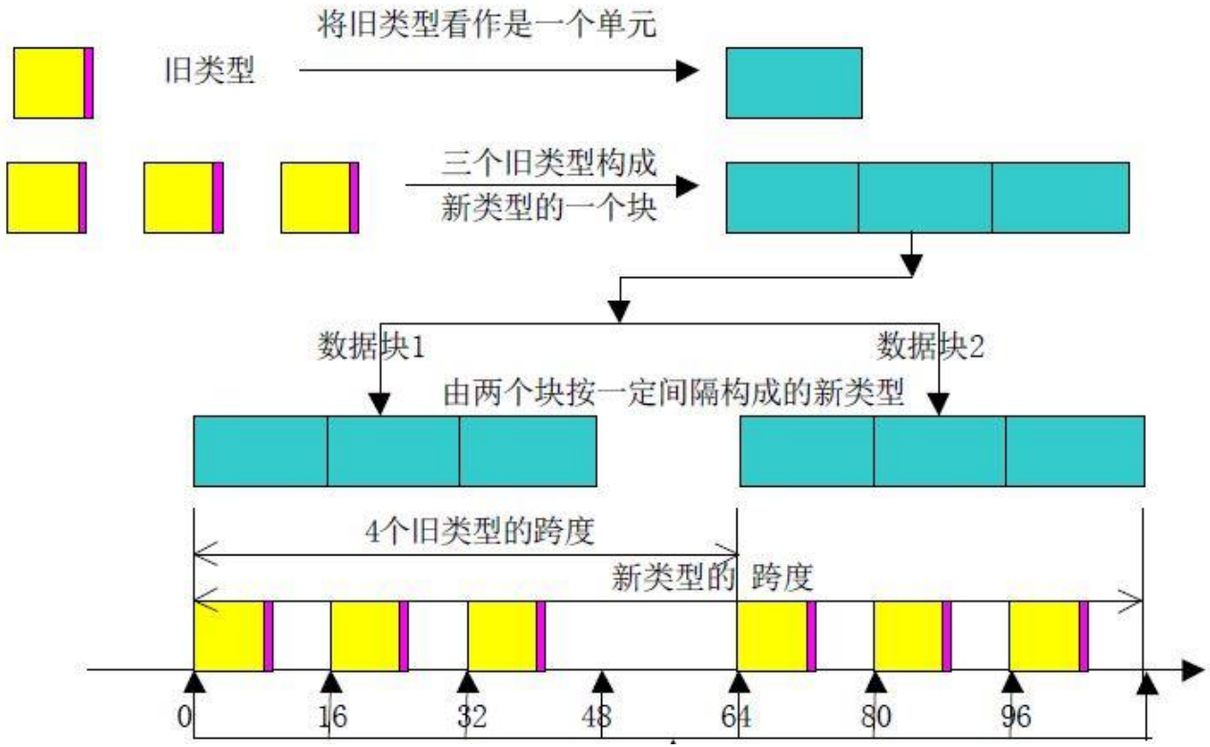
新数据类型自定义

- 连续复制的类型生成
- 向量数据类型的生成
- 索引数据类型的生成
- 结构数据类型的生成

新数据类型的递交和释放

```
MPI_TYPE_FREE(datatype)
      INOUT datatype  释放的数据类型(句柄)
int MPI_Type_free(MPI_Datatype *datatype)
MPI_TYPE_FREE(DATATYPE,IERROR)
      INTEGER  DATATYPE,IERROR
```

```
MPI_TYPE_COMMIT(datatype)
      INOUT datatype  递交的数据类型(句柄)
int MPI_Type_commit(MPI_Datatype *datatype)
MPI_TYPE_COMMIT(DATATYPE,IERROR)
      INTEGER  DATATYPE,IERROR
```

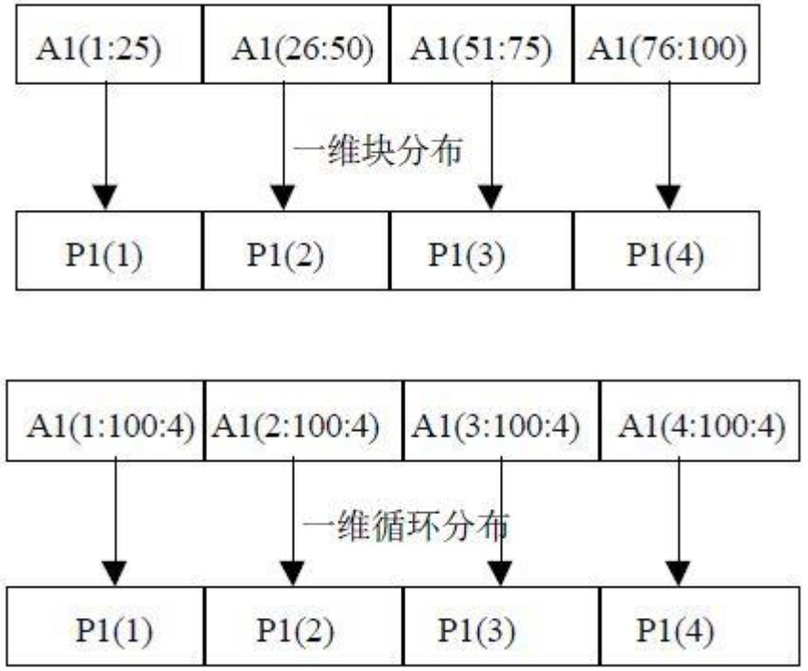


分布式数组定义

MPI_TYPE_CREATE_SUBARRAY(ndims,array_of_sizes, array_of_subsizes,
array_of_starts, order, oldtype, newtype)

IN	ndims	数组维数
IN	array_of_sizes	每一维数组的大小
IN	array_of_subsizes	子数组每一维的大小
IN	array_of_starts	子数组起始坐标在数组中每一维的相对偏移
IN	order	与创建分布式数组类型时使用的优先方式相同
IN	oldtype	数组元素的类型
OUT	newtype	返回的子数组类型

```
int MPI_Type_create_subarray(int ndims, int array_of_sizes[], int array_of_subsizes[],
    int array_of_starts[], int order, MPI_Datatype oldtype, MPI_Datatype * newtype)
MPI_TYPE_CREAT_SUBARRAY(NDIMS, ARRAY_OF_SIZES,
      ARRAY_OF_SUBSIZES,ARRAY_OF_STARTS,ORDER,OLDTYPE,NEWTYPE,
       IERROR)
INTEGER  NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
         ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR
```



- 用newtype作为文件类型来定义文件视口，这样将内存中连续排列的局部数组写入文件视口时会以正确的方式将该块局部数据在全局数组文件的指定部分进行更新；
- 程序员就不必去计算复杂的全局与局部数组的对应关系；

第一步

通过**分布式数组构造符**定义了与原程序各进程所处理网格单元数据数组大小相同的**分布式子数组类型；**

第二步

以此为基本数据单元类型，设置每个进程的文件视口；
每个进程所可见和操作的文件大小为分布式子数组大小，位置为子数组所对应于全局数组的起始位置；

第三步

调用**阻塞的独立文件指针的聚合I/O 函数**来实现向文件相应位置**高效并行写入或者读取分布式**子数组大小的数据；

MPI-IO技术要点和流程

口参考资料

《高性能计算之并行编程技术—— MPI并行程序设计》
都志辉　编著
清华大学出版社

04 章节 PART

IO优化案例（Case）

- 案例背景
- 前期工作
- 结果分析
- 问题分析
- 研究方法

地球系统模型（CESM）

- 由美国国家大气中心（NCAR）在共同气候系统模式（CCSM）基础上改进研发，用于模拟地球气候系统的耦合气候模型；
- 由大气（atm）、海冰（ice）、陆地（lnd）、河流径流（rof）、海洋（ocn）、陆地冰（glc）和海浪（wav-sub only）等七个独立模型，以及一个中心耦合器组成；
- 支持包括陆地碳循环、动态植被等多种物理、生物和化学过程模拟，支持诸多分辨率、组件和方法的不同配置，便于移植和自定义开发；

LASG/IAP气候系统海洋模式（LICOM3）

- 由中国科学院大气物理研究所大气科学和地球流体力学数值模拟国家重点实验室（LASG）研制、发展、建立的**模块化耦合气候模式中的海洋分模式**（第三版）；
- **主要通过对大尺度风生环流和热盐环流的模拟，实现对海洋变化规律及其与气候系统其它分量相互作用的研究；**
- 基于CESM的并行海洋分模式（POP），能够与CESM耦合运行；

I/O时间占比情况

在天河系统，水平10 公里分辨率下，测试得到：

- 2400 并行度时，串行I/O 部分占总运行时间的**48.2%**；
- 4400 并行度时，串行I/O 部分占比可达到**50%以上**；

IO瓶颈随进程数的增加而变得更突出；

I/O基本过程

- 首先，由主进程（通常为0 号进程）**读入整个全局多维数据**；（读取）
- 之后，主进程将全局数据**按需分发给其他进程**，各进程根据划分的子数据块计算结果后，**再各自发送给主进程**；（分发和收集）
- 最后，主进程将**输出保存所有计算结果的全局多维数组**；（输出）

I/O基本特点

- **规则网格划分**

以经度、纬度方向按照网格均匀划分；每个进程的数据块都是规则多维数组形式；

- **单次IO数据规模大**

在较高分辨率下，LICOM3系统单次输出数据规模达18GB以上；

- **输出频率高，多次输出变量相同**

系统每次输出时，变量相同，只是多次迭代后变量有差异，因此，只需统一定义输出过程函数；

研究团队

- 清华大学薛巍老师团队；

研究方法

- ADIOS（1.3版）方法；
- LICOM第二版（LICOM2）；
- 在天河1A系统上进行了验证和测试；

研究结果

- 明显减少 I/O 运行时间，提高了运行效率；

不足和差异

- ADIOS使用自带的Bp 格式，区别于通用格式，不利于数据的处理；
- LICOM2是单独运行的，而LICOM3是耦合在CESM中运行的，一些数据结构和运行机制等有差异；

MPI-IO

01

标准I/O接口

一个标准、可移植的高性能并行文件I/O 接 口 ，为MPI-2标准的一部分；**是大部分I/O库的基础库；**

02

特殊设计

包括文件指针、文件视口、分布式数组读写、聚合 I/O等，以此**实现灵活、高效的I/O行为；**

03

通用格式

采用通用二进制格式进行数据的输入和输出；

ADIOS

开发部门

Adaptive I/O System (ADIOS）由橡树岭国立实验室开发,**为一个包含不同 I/O 传输方法的 I/O 组件；**

01

诸多优点

独立XML文件，更方便；**自定义Bp格式，**避免文件冲突；**适用不同计算机结构和文件系统，**更灵活；

02

实验验证

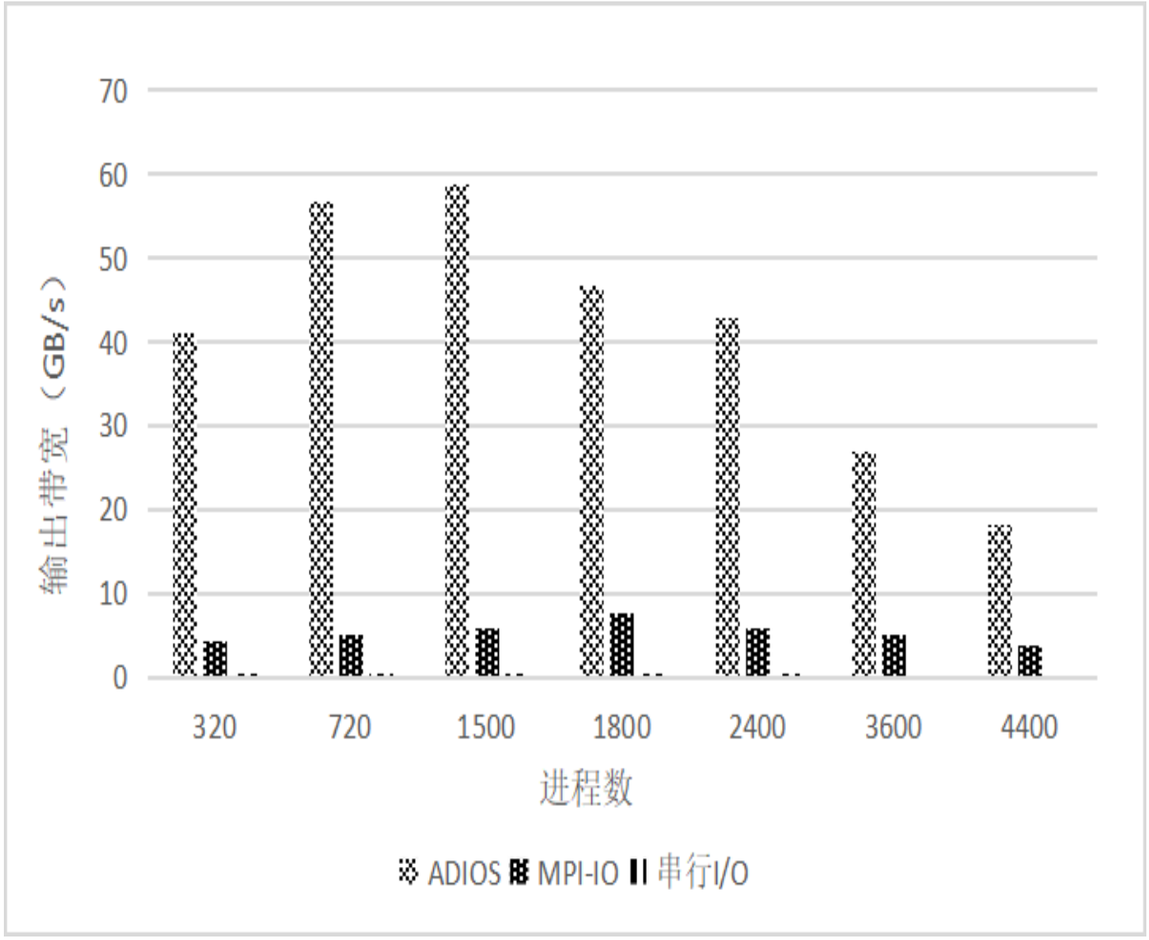
在LICOM上有良好的I/O优化效果；编写数据格式转换程序，以解决格式通用性问题；

03

- **从趋势上，输出带宽均随进程数先增加后减小；**
- **原因：**
- 进程数增加，并行度增加；
- 单个OST上进程增多，达到瓶颈，造成对I/O 资源的竞争；

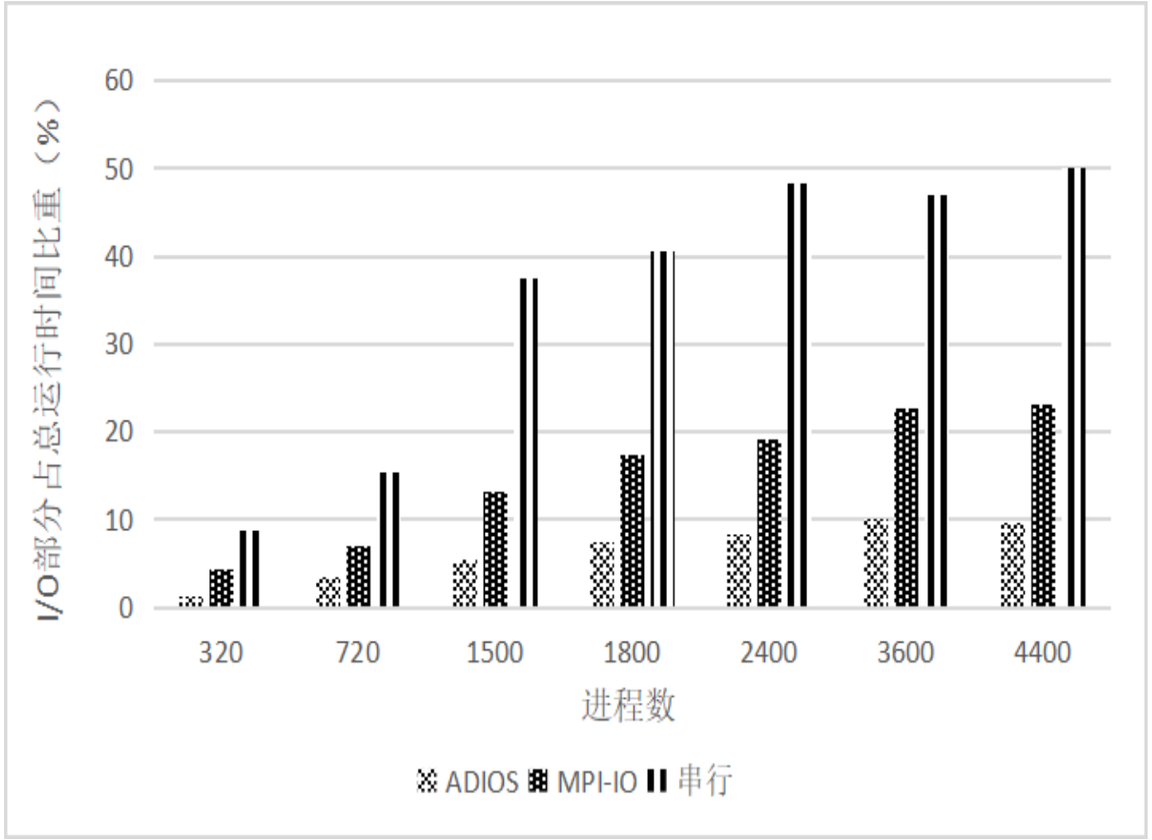
- **最高输出带宽：**
- ADIOS约为58.68GB/s，约为原串行输出的279 倍；
- MPI-IO约7.75GB/s，约原有串行输出的38.75倍；

- **差异原因：**
- bp格式为每个进程输出文件申请单独存储空间，无多进程存储竞争；
- 每个进程都输出单独文件中，无多进程文件访问影响；



ADIOS（POSIX）、MPI-IO和串行I/O
在各并行度下的输出带宽

- 原串行模式下，其IO时间占比随进程数的增加而增大；当进程规模达到2K时，I/O 部分的时间占比达到48.2%之多，严重影响了程序的运行效率；
- 两种优化方法则远远低于原串行模式下的时间占比；
- ADIOS约为其1/5；MPI-IO则约为其1/2左右；



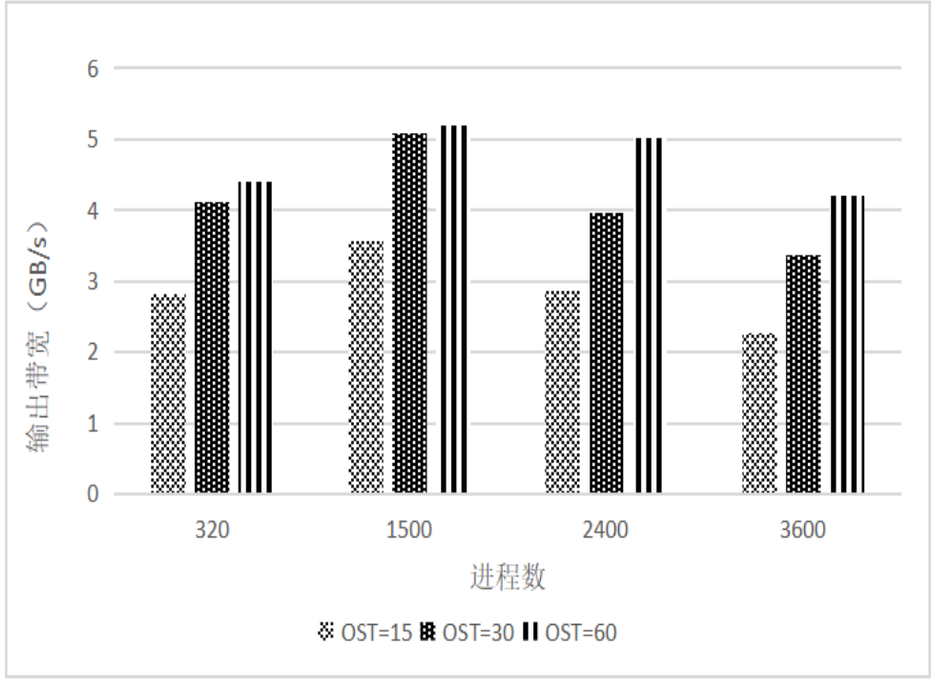
ADIOS（POSIX方法）、MPI-IO和串行I/O
在各并行度下的I/O时间占比

- 对于MPIIO方法，随着OST数的提高，输出带宽不断增加；

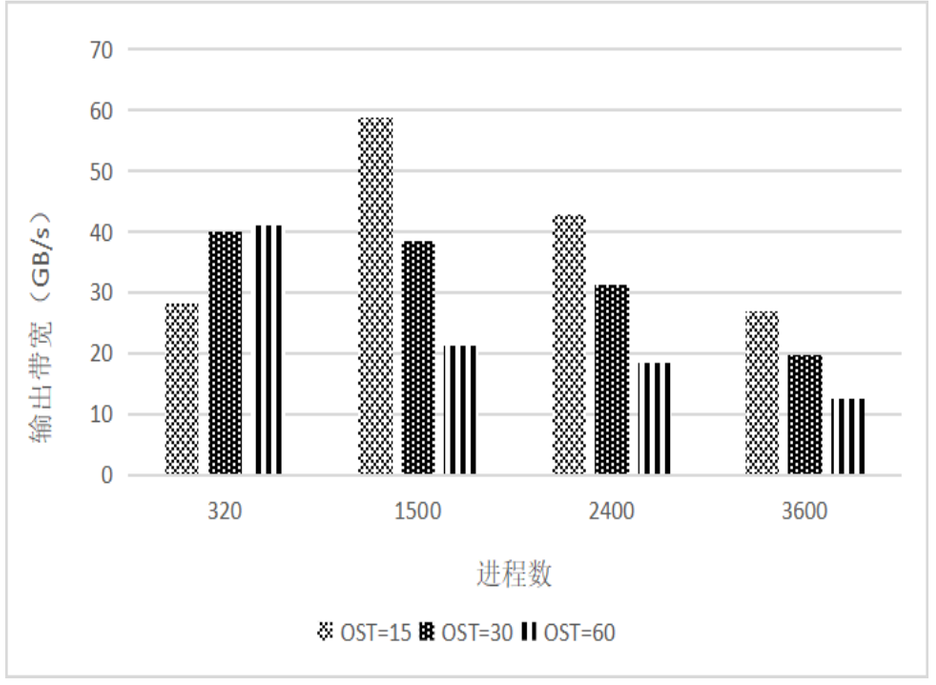
原因：在同进程下，数据总量不变，OST 数越多代表并行度越高，带宽也就越高；

- 对于ADIOS的POSIX方法，除了320进程外，其他规模的结果趋势均与之前的相反；

原因：其为多文件输出，随着OST数增加，单文件会再被分散到更多OST，数据变得太小，反而不利于输出；



MPI-IO不同OST数下输出带宽



ADIOS（POSIX方法）在不同OST数下的输出带宽

小结

针对海洋环流模式LICOM3的I/O 瓶颈问题，实现了基于ADIOS 和MPI-IO 技术的两种并行优化方案。实验表明：

1. 两种并行优化方案与原有的串行方式相比，均得到了显著的I/O性能提升；
2. ADIOS 并行优化方案总体优于MPI-IO 并行优化方案；
3. 关于进程数和OST数对优化性能的影响，分析可得：
   - 事实上，单个OST的输出带宽与输出数据量是类似正态分布的关系；
   - 只有保证在单个OST上的输出量达到一定规模，使得单OST的输出带宽可被充分利用时，才可获得更显著的性能；

国家超级计算天津中心
National SuperComputer Center in Tianjin

2019年用户培训交流会

感谢观看！

李云龙　应用研发工程师
国家超级计算天津中心　liyl@nscc-tj.cn