Parallel programming
MPI IO

Jan Thorbecke

Acknowledgments

- This course is partly based on the MPI courses developed by
  - Rolf Rabenseifner at the High-Performance Computing-Center Stuttgart (HLRS), University of Stuttgart in collaboration with the EPCC Training and Education Centre, Edinburgh Parallel Computing Centre, University of Edinburgh.
    http://www.hlrs.de/home/
    https://www.epcc.ed.ac.uk

H L R S
High-Performance Computing Center | Stuttgart

- CSC – IT Center for Science Ltd.
  - https://www.csc.fi
  - https://research.csc.fi

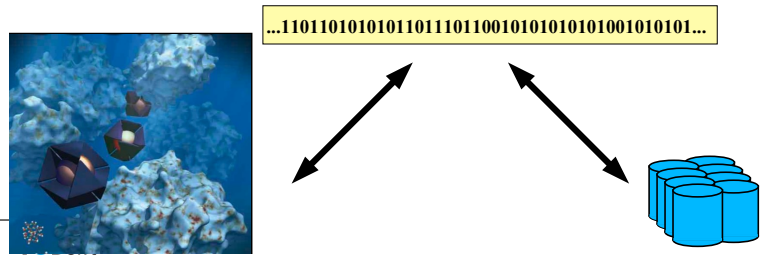
CSC - IT CENTER FOR SCIENCE
C S C

- http://mpitutorial.com

Contents

Parallel IO

- I/O (Input/output) is needed in all programs but is often overlooked
- Mapping problem: how to convert internal structures and domains to files which are a streams of bytes
- Transport problem: how to get the data efficiently from hundreds to thousands of nodes on the supercomputer to physical disks

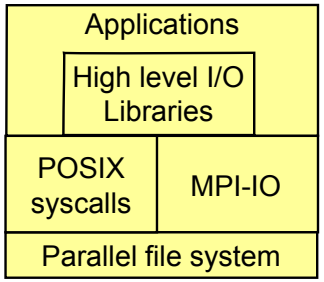


Parallel IO

- Good I/O is non-trivial
  - Performance, scalability, reliability
  - Ease of use of output (number of files, format)
- Portability
- One cannot achieve all of the above - one needs to decide what is most important
- New challenges
  - Number of tasks is rising rapidly
  - The size of the data is also rapidly increasing
- The need for I/O tuning is algorithm & problem specific
- Without parallelization, I/O will become scalability bottleneck for practically every application!

IO layers

- High-level
  - application: need to write or read data from disk
- Intermediate
  - libraries or system tools for I/O
  - high-level libraries
    - HDF5, netcdf
  - MPI-I/O
  - POSIX system calls (fwrite / WRITE)
- Low-level:
  - parallel filesystem enables the actual parallel I/O
  - Lustre, GPFS, PVFS, dCache

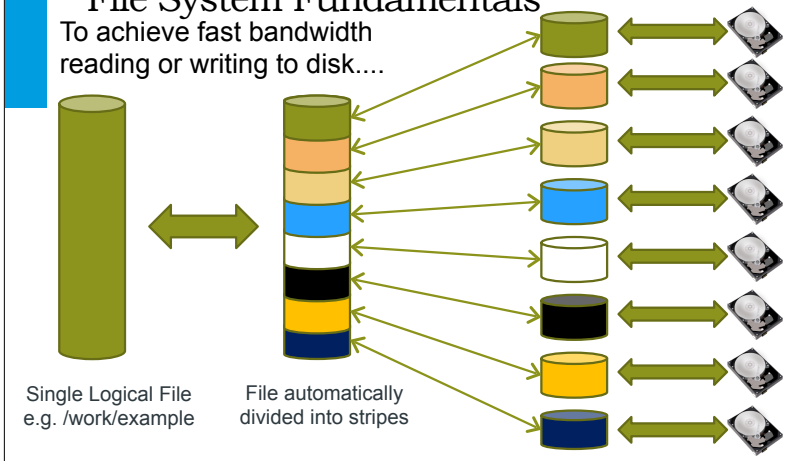


POSIX IO

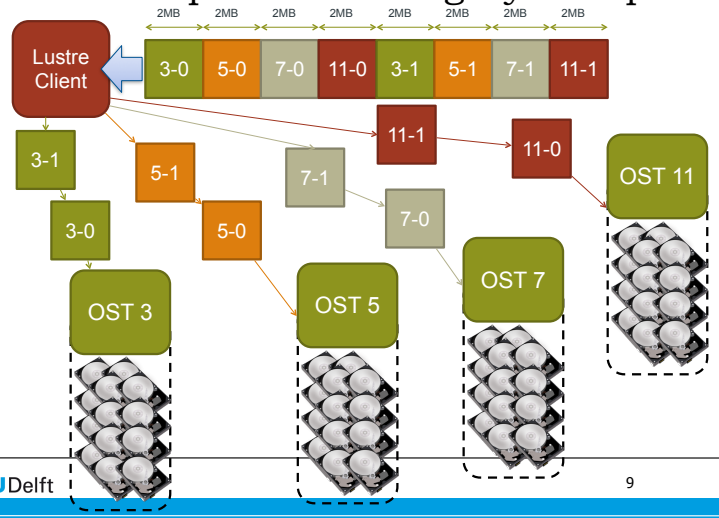
- Built in language constructs for performing I/O
  - WRITE/READ/OPEN/CLOSE in Fortran
  - stdio.h routines in C (fopen, fread, fwrite, ...)
- No parallel ability built in - all parallel I/O schemes have to be programmed manually
- Binary output not necessarily portable
- C and Fortran binary output not necessarily compatible
- Non-contiguous access difficult to implement efficiently
- Contiguous access can be very fast

File System Fundamentals

To achieve fast bandwidth reading or writing to disk....

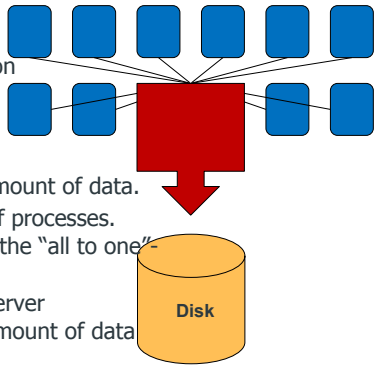


File decomposition – 2 Megabyte Stripes



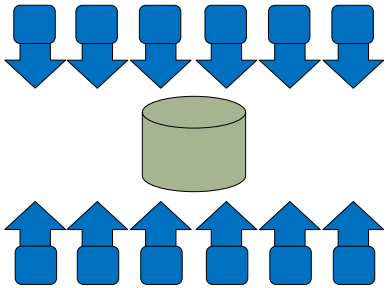
Spokesperson, basically serial I/O

- One process performs I/O.
  - Data Aggregation or Duplication
  - Limited by single I/O process.
- Easy to program
- Pattern does not scale.
  - Time increases linearly with amount of data.
  - Time increases with number of processes.
- Care has to be taken when doing the "all to one" - kind of communication at scale
- Can be used for a dedicated IO Server (not easy to program) for small amount of data



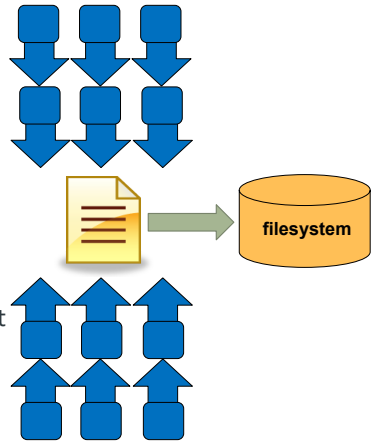
One file per process

- All processes perform I/O to individual files.
  - Limited by file system.
- Easy to program
- Pattern does not scale at large process counts.
  - Number of files creates bottleneck with metadata operations.
  - Number of simultaneous disk accesses creates contention for file system resources.



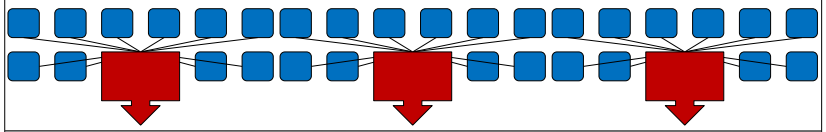
Shared File

- Each process performs I/O to a single file which is shared.
- Performance
  - Data layout within the shared file is very important.
  - At large process counts contention can build for file system resources (OST).
- Programming language might not support it
  - C/C++ can work with fseek
  - No real Fortran standard



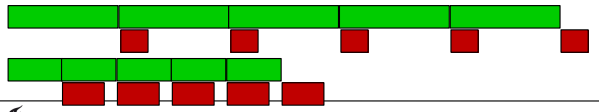
A little bit of all, using a subset of processes

- Aggregation to a processor group which processes the data.
  - Serializes I/O in group.
- I/O process may access independent files.
  - Limits the number of files accessed.
- Group of processes perform parallel I/O to a shared file.
  - Increases the number of shares to increase file system usage.
  - Decreases number of processes which access a shared file to decrease file system contention.



UM per Timestep I/O

- UM already had implemented a definable number of asynchronous I/O servers
  - Each handling a certain number of files (Fortran I/O units)
- When doubling the number of cores, ideally compute time AND amount of I/O per core is reduced to half
- I/O time should scale – but it doesn't – how come?
  - The single I/O server per file becomes overwhelmed
  - Increasing number of smaller packets
  - I/O server collects data in a prescribed order, compute tasks wait for completion



I/O Performance : To keep in mind

- There is no "One Size Fits All" solution to the I/O problem.
- Many I/O patterns work well for some range of parameters.
- Bottlenecks in performance can occur in many locations. (Application and/or File system)
- Going to extremes with an I/O pattern will typically lead to problems.
- I/O is a **shared** resource. Expect timing variation

BeeGFS

- Installed on DelftBlue
  - beegfs-ctl —getentryinfo .
  - beegfs-ctl --liststoragepools

MPI-IO

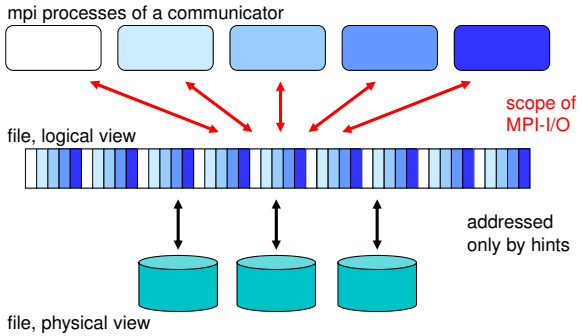
MPI-IO

- MPI I/O was introduced in MPI-2
- Defines parallel operations for reading and writing files
  - I/O to only one file and/or to many files
  - Contiguous and non-contiguous I/O
  - Individual and collective I/O
  - Asynchronous I/O
- Portable programming interface
- Potentially good performance
- Easy to use (compared with implementing the same algorithms on your own)
- Used as the backbone of many parallel I/O libraries such as parallel NetCDF and parallel HDF5
- By default, binary files are not necessarily portable

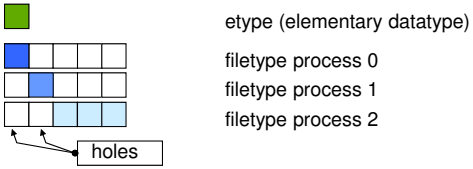
Basic concepts MPI-IO

- File handle
  - data structure which is used for accessing the file
- File pointer
  - position in the file where to read or write
  - can be individual for all processes or shared between the processes
  - accessed through file handle
- File view
  - part of a file which is visible to process
  - enables **efficient** non-contiguous access to file
- Collective and independent I/O
  - collective: MPI coordinates the reads and writes of processes
  - independent: no coordination by MPI

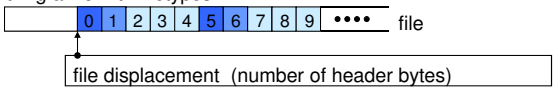
Logical view

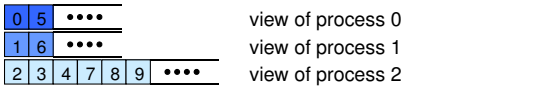


Definitions



tiling a file with filetypes:





Comments

file - an ordered collection of typed data items

etypes
- is the unit of data access and positioning / offsets
- can be any basic or derived datatype (with non-negative, monotonically non-decreasing, non-absolute displacem.)
- generally contiguous, but need not be
- typically same at all processes

filetypes
- the basis for partitioning a file among processes
- defines a template for accessing the file
- different at each process
- the etype or derived from etype (displacements: non-negative, monoton. non-decreasing, non-abs., multiples of etype extent)

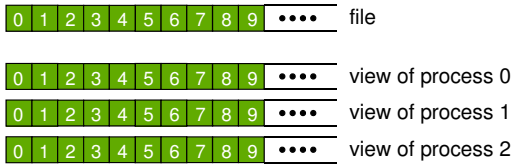
view
- each process has its own view, defined by: a displacement, an etype, and a filetype.
- The filetype is repeated, starting at **displacement**

offset - position relative to current view, in units of etype

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

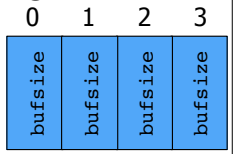
- Default:
  - displacement = 0
  - etype = MPI_BYTE
  - filetype = MPI_BYTE

  } each process has access to the whole file



- Sequence of MPI_BYTE matches with any datatype (see MPI-3.0, Section 13.6.5)
- Binary I/O (no ASCII text I/O)

A simple MPI-IO program in C




```
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);
```

Write instead of Read

- Use **MPI_File_write** or **MPI_File_write_at**
- Use **MPI_MODE_WRONLY** or **MPI_MODE_RDWR** as the flags to **MPI_File_open**
- If the file doesn't exist previously, the flag **MPI_MODE_CREATE must** be passed to **MPI_File_open**
- We can pass multiple flags by using bitwise-or '|' in C, or addition '+' or IOR in Fortran
- If not writing to a file, using MPI_MODE_RDONLY might have a performance benefit. Try it.

Parallel write

```
PROGRAM Output
 USE MPI
 IMPLICIT NONE
 INTEGER :: 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) = myid * count + i
 END DO
...
```

- Multiple processes write to a binary file test.
- First process writes integers 1-100 to the beginning of the file, etc.

```
...
 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 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
```

Note !
File (and total data) size depends on number of processes in this example

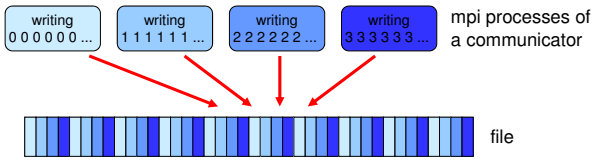
MPI_File_Seek and MPI_File_Write combined in one call

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

- writes **`count`** elements of **`datatype`** from memory **`buf`** to the file
- starting **`offset`** * units of **`etype`** from begin of view
- the elements are stored into the locations of the current view
- the sequence of basic datatypes of **`datatype`** (= signature of **`datatype`**) must match contiguous copies of the **`etype`** of the current view

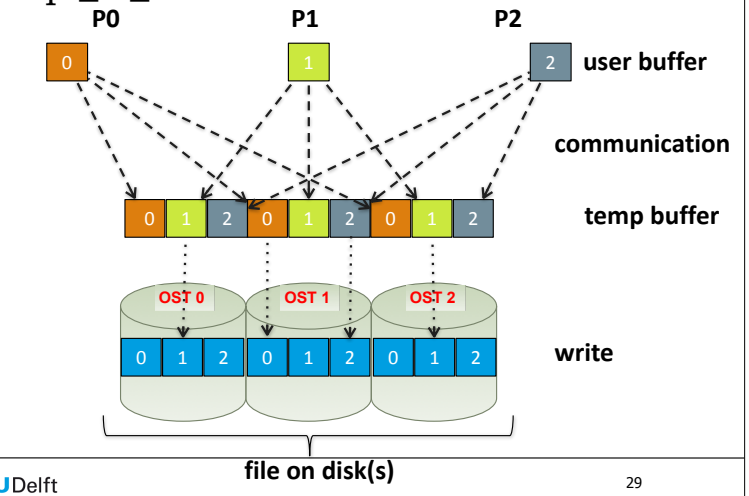
Exercise: MPI-IO exa1: Four processes write a file in parallel

- each process should write its rank (as one character) ten times to the offsets = my_rank + i * size_of_MPI_COMM_WORLD, i=0..9
- Result: "0123012301230123012301230123012301230123"
- Each process uses the default view



- cd MPI_IO
  cp mpi_io_exa1_skel.c my_exa1.x

mpi_io_exa1.c



File Views

- Provides a set of data visible and accessible from an open file
- A separate view of the file is seen by each process through triple := (displacement, etype, filetype)
- User can change a view during the execution of the program - but collective operation
- A linear byte stream, represented by the triple (0, MPI_BYTE, MPI_BYTE), is the default view.

MPI_File_set(get)_view

- Set view
  - changes the process's view of the data
  - local and shared file pointers are reset to zero
  - collective operation
  - etype and filetype must be committed
  - datarep argument is a string that specifies the format in which data is written to a file: "native", "internal", "external32", or user-defined
  - same etype extent and same datarep on all processes
- Get view
  - returns the process's view of the data

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

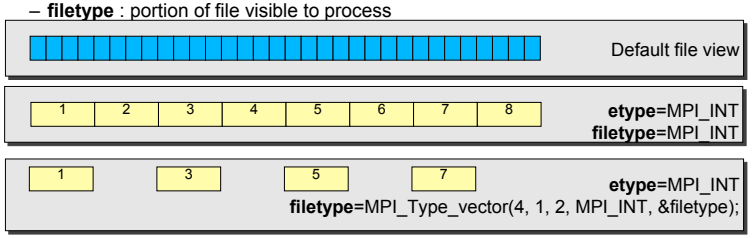
MPI_File_set_view

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

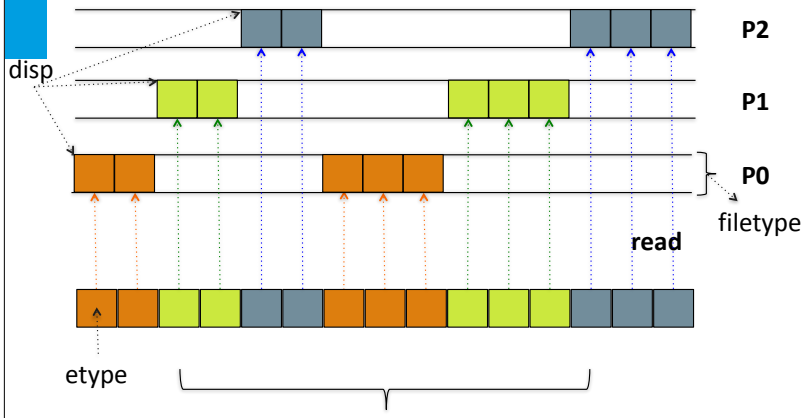
disp	Offset from beginning of file. Always in bytes
etype	Basic MPI type or user defined type Basic unit of data access Offsets in I/O commands in units of etype
filetype	Same type as etype or user defined type constructed of etype Specifies which part of the file is visible
datarep	Data representation, sometimes useful for portability "native": store in same format as in memory
info	Hints for implementation that can improve performance MPI_INFO_NULL: No hints

File View

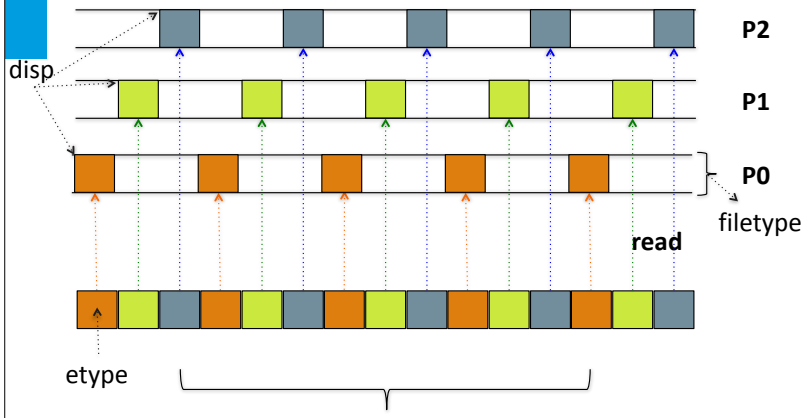
- A file view defines which portion of a file is “visible” to a process
- File view defines also the type of the data in the file (byte, integer, float, ...)
- By default, file is treated as consisting of bytes, and process can access (read or write) any byte in the file
- A file view consists of three components
  - **displacement** : number of *bytes* to skip from the beginning of file
  - **etype** : type of data accessed, defines unit for offsets
  - **filetype** : portion of file visible to process



MPI_File_set_view (picture 1)

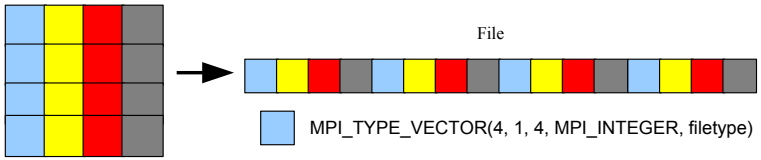


MPI_File_set_view (picture 2)



File view for non-contiguous data

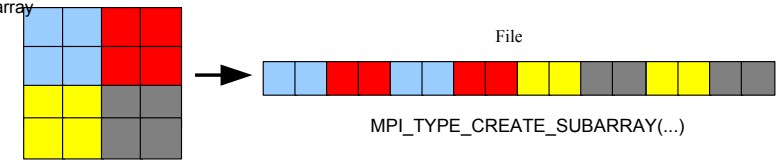
2D-array distributed column-wise




```
...
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)
...
```

Storing multidimensional arrays in files

Domain decomposition for 2D-array

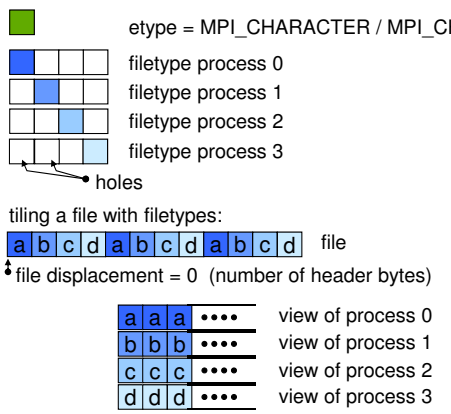



```
...
 INTEGER :: sizes = (/4, 4/)
 INTEGER :: subsizes = (/2, 2/)
 INTEGER, DIMENSION(2,2) :: buf
 ...
 MPI_CART_COORDS(MPI_COMM_WORLD, myid, 2, starts, err)
 CALL MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts,
      MPI_INTEGER, MPI_ORDER_C, filetype, err)
 CALL MPI_TYPE_COMMIT(filetype)
 CALL CALL MPI_FILE_SET_VIEW(file, 0, MPI_INTEGER, filetype, "native", &
      MPI_INFO_NULL, err)
 CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)
...
```

Exercise: MPI-IO exa2: Using fileviews and individual filepointers

- Copy to your local directory:
  cp ~/MPI/course/**C**/mpi_io/mpi_io_exa2_skel.c **my_exa2.c**
  cp ~/MPI/course/**F**/mpi_io/mpi_io_exa2_skel.f **my_exa2.f**
- Tasks:
  - Each MPI-process of `my_exa2` should write one character to a file:
    - **process "rank=0" should write an 'a'**
    - **process "rank=1" should write an 'b'**
    - **...**
  - Use a 1-dimensional fileview with MPI_TYPE_CREATE_SUBARRAY
  - The pattern should be repeated 3 times, i.e., four processes should write: "`abcdabcdabcd`"
  - Please, substitute "____" in your `my_exa2.c / .f`
  - Compile and run your `my_exa2.c / .f`

Exercise: MPI-IO exa2: Using fileviews and individual filepointers



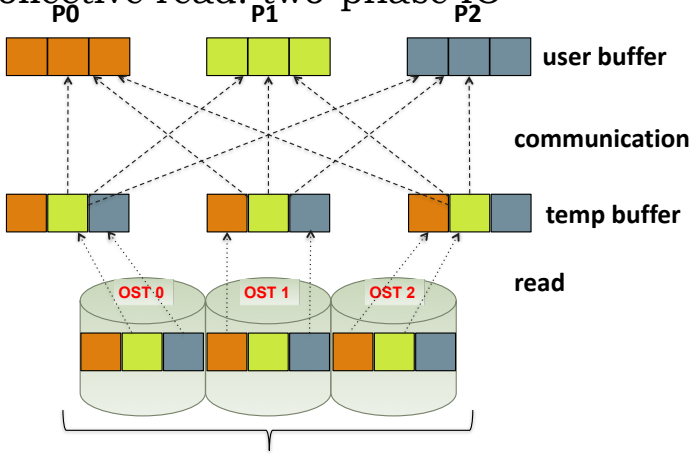
MPI IO

- The MPI interface support two types of IO:
- **Independent**
  - each process handling its own I/O independently
  - supports derived data types (unlike POSIX IO)
- **Collective**
  - I/O calls must be made by **all** processes participating in a particular I/O sequence
  - Used the "shared file, all write" strategy are optimized dynamically by the MPI library.

Collective IO with MPI-IO

- MPI_File_read_**all**, MPI_File_read_at_**all**, …
- **_all** indicates that all processes in the group specified by the communicator passed to MPI_File_open will call this function
- Each process specifies only its own access information – the argument list is the same as for the non-collective functions
- MPI-IO library is given a lot of information in this case:
  - Collection of processes reading or writing data
  - Structured description of the regions
- The library has some options for how to use this data
  - Noncontiguous data access optimizations
  - Collective I/O optimizations

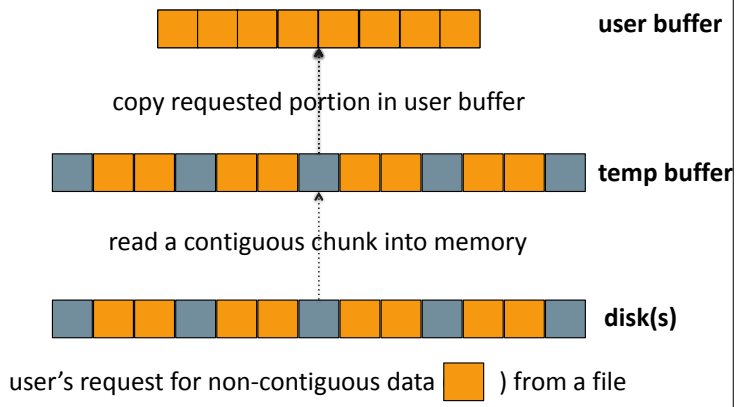
Collective read: two-phase IO



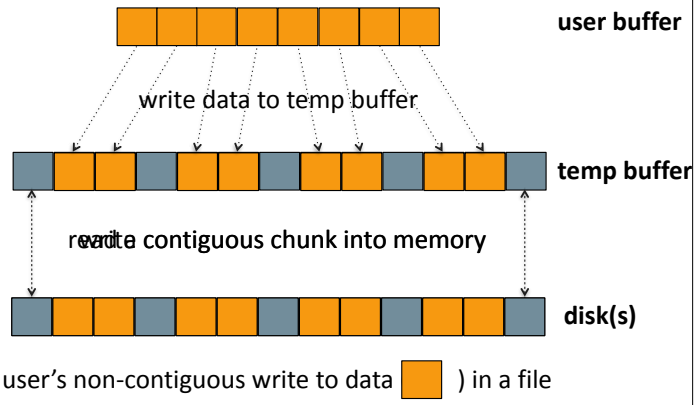
Two techniques : Data sieving and Aggregation

- Data sieving is used to combine lots of small accesses into a single larger one
  - Reducing number of operations important (latency)
  - A system buffer/cache is one example
- Aggregation refers to the concept of moving data through intermediate nodes
  - Different numbers of nodes performing I/O (transparent to the user)
- Both techniques are used by MPI-IO and triggered with HINTS.

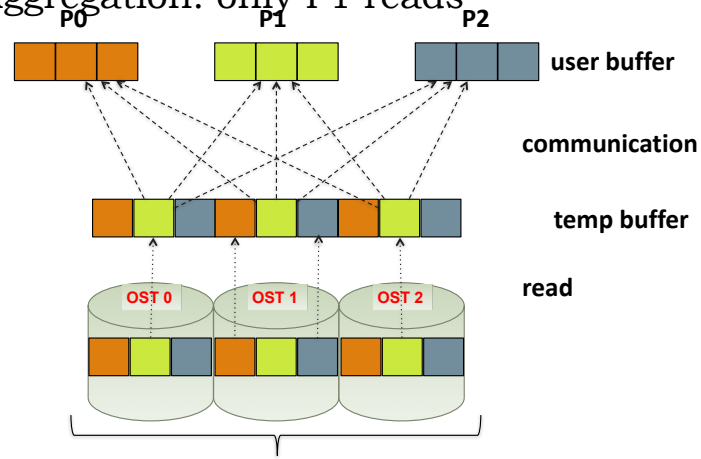
Data Sieving read



Data Sieving write



Aggregation: only P1 reads



Collective IO

- I/O can be performed collectively by all processes in a communicator
  - `MPI_File_read_all`
  - `MPI_File_write_all`
  - `MPI_File_read_at_all`
  - `MPI_File_write_at_all`
- Same parameters as in independent I/O functions
  - `MPI_File_read, MPI_File_write, MPI_File_read_at, MPI_File_write_at`
- All processes in communicator that opened file must call function
- Performance potentially better than for individual functions
  - Even if each processor reads a non-contiguous segment, in total the read is contiguous

Collective IO example

```
...
INTEGER :: sizes = (/4, 4/)
INTEGER :: subsizes = (/2, 2/)
INTEGER, DIMENSION(2,2) :: buf
...
MPI_CART_COORDS(MPI_COMM_WORLD, myid, 2, starts, err)
CALL MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts,
     MPI_INTEGER, MPI_ORDER_C, filetype, err)
CALL MPI_TYPE_COMMIT(filetype)
CALL CALL MPI_FILE_SET_VIEW(file, 0, MPI_INTEGER, filetype, "native", &
     MPI_INFO_NULL, err)
CALL MPI_FILE_WRITE_ALL(file, buf, count, MPI_INTEGER, status, err)
...
```

- Collective write can be over hundred times faster than the individual for large arrays!

MPIIO hints

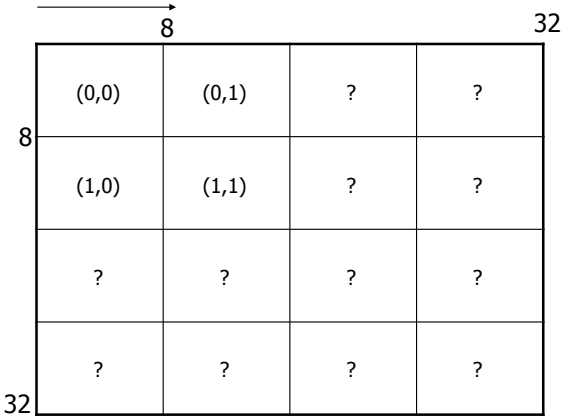
- Hints may enable the implementation to optimize performance
- MPI 2 standard defines several hints via MPI_Info object
  - MPI_INFO_NULL : no info
  - Functions **MPI_Info_create**, **MPI_Info_set** allow one to create and set hint
- Some implementations allow setting of hints via environment variat
  - e.g. MPICH_MPIIO_HINTS
  - Example: for file “test.dat”, in collective I/O aggregate data to 32 nodes
    export MPICH_MPIIO_HINTS=”test.dat:cb_nodes=32”
- Effect of hints on performance is implementation and application dependent

Exercise: MPI-IO mpiio_2D_r/w.c: collective operations

- Use number of MPI-tasks N, such that sqrt(N) = integer
- review MPI_Cart and MPI_Type_create_subarray
- MPI-IO for writing 2D array (32x32) to disk
- Without running the program draw a picture how the output will look like.


```
nx=32;
mpi_dims[0] = mpi_dims[1] = sqrt(nproc);
int ndims = 2;
int dimsf[2] = {nx, nx};
  for (i=0; i < 2; i++) {
      count[i] = nx / mpi_dims[i];
      offset[i] = coords[i] * count[i];
      bufsize *= count[i];
    }
MPI_Type_create_subarray(ndims, dimsf, count, offset,
MPI_ORDER_C,MPI_INT, &filetype);
```

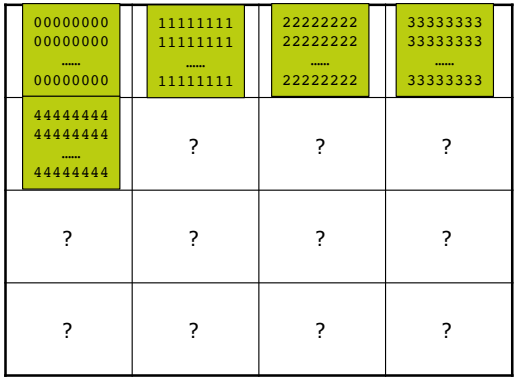

Picture of mpiio_2D_r output N=16



N=16 nx=32

- count[i]=nx/dims[i] => count[0]=8; count[1]=8
- offset[i]=coords[i]*count[i] => offset[0]=x*8; count[1]=y*8
- bufsize *= count[i] => 8*8

MPI_Type_create_subarray()

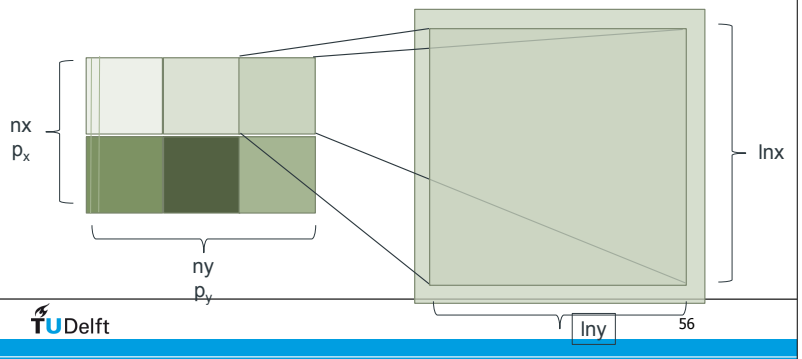


Example: 3D-Finite Difference

- 3D domain decomposition to distribute the work
  - MPI_Cart_create to set up cartesian domain
- Hide communication of halo-areas with computational work
- MPI-IO for writing snapshots to disk
  - use local and global derived types to exclude halo areas write in global snapshot file
- Tutorial about domain decomposition: http://www.nccs.nasa.gov/tutorials/mpi_tutorial2/

Problem we want to solve

- We have 2 dim domain on a 2 dimensional processor grid
- Each local subdomain has a halo (ghost cells).
- The data (without halo) is going to be stored in a single file, which can be re-read by any processor count
- Here an example with 2x3 procesor grid :



Approach for writing the file

- First step is to create the MPI 2 dimensional processor grid
- Second step is to describe the local data layout using a MPI datatype
- Then we create a „global MPI datatype" describing how the data should be stored
- Finally we do the I/O

Basic MPI setup

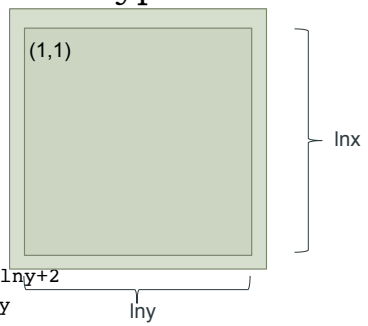
```
nx=512; ny=512 ! Global Domain Size
call MPI_Init(mpierr)
call MPI_Comm_size(MPI_COMM_WORLD, mysize, mpierr)
call MPI_Comm_rank(MPI_COMM_WORLD, myrank, mpierr)

dom_size(1)=2;  dom_size(2)=mysize/dom_size(1)
lnx=nx/dom_size(1) ;  lny=ny/dom_size(2) ! Local Domain size
periods=.false. ; reorder=.false.
call MPI_Cart_create(MPI_COMM_WORLD, dim, dom_size, periods,
reorder,      comm_cart, mpierr)
call MPI_Cart_coords(comm_cart, myrank, dim, my_coords, mpierr)

halo=1
allocate (domain(0:lnx+halo, 0:lny+halo))
```

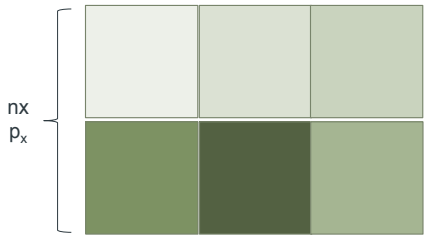

Creating the local data type

- Use a subarray datatype to describe the noncontiguous layout in memory
- Pass this datatype as argument to `MPI_File_write_all`




```
gsize(1)=lnx+2; gsize(2)=lny+2
lsize(1)=lnx; lsize(2)=lny
start(1)=1; start(2)=1
call MPI_Type_create_subarray(dim, gsize, lsize,
start,      MPI_ORDER_FORTRAN, MPI_INTEGER,
type_local, mpierr)
call MPI_Type_commit(type_local, mpierr)
```


And now the global datatype




```
gsize(1)=nx; gsize=ny
lsize(1)=lnx; lsize(2)=lny
start(1)=lnx*my_coords(1); start(2)=lny*my_coords(2)
call MPI_Type_create_subarray(dim, gsize, lsize,
start,      MPI_ORDER_FORTRAN, MPI_INTEGER,
type_domain, mpierr)
call MPI_Type_commit(type_domain, mpierr)
```

Now we have all together

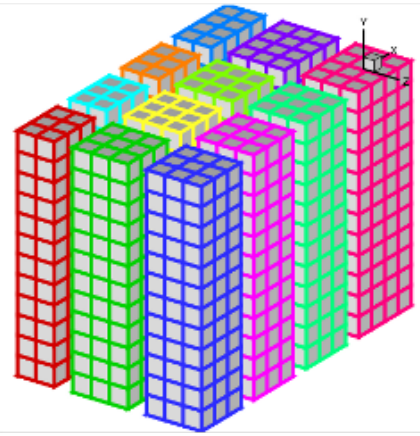
```
call MPI_Info_create(fileinfo, mpierr)
call MPI_File_delete('FILE', MPI_INFO_NULL, mpierr)
call MPI_File_open(MPI_COMM_WORLD,    'FILE',
IOR(MPI_MODE_RDWR,MPI_MODE_CREATE), fileinfo, fh,
mpierr)

disp=0 ! Note : INTEGER(kind=MPI_OFFSET_KIND) :: disp
call MPI_File_set_view(fh, disp, MPI_INTEGER,
type_domain, 'native',   fileinfo, mpierr)
call MPI_File_write_all(fh, domain, 1, type_local,
status, mpierr)
call MPI_File_close(fh, mpierr)
```


Example

- 1024x1024x512 sized snapshots (2.1 GB) are written to disk; 16 in total (each 100 time steps).
- stripe size is 1MB
- stripe count is 4 or 16
- At 1024 cores each MPI task write a 2 MB portion to disk
- Interlagos 32 core nodes at 2.1 GHz

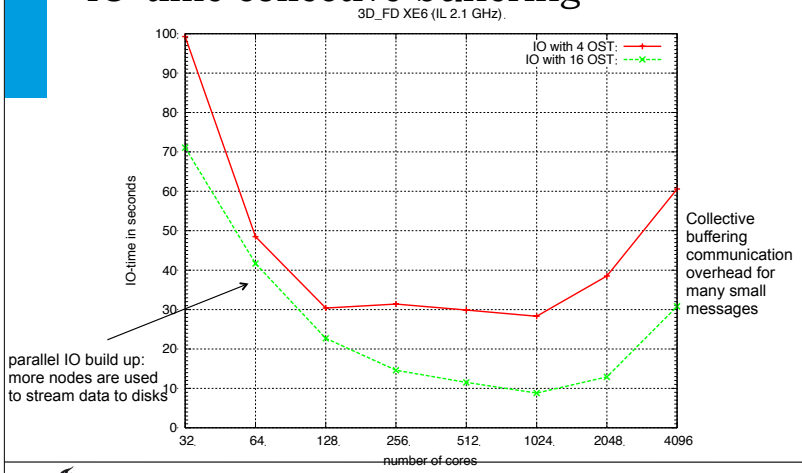
Storage into file per MPI task



Each MPI domain has a non-contiguous storage view into the snapshot file.

This is transparently handled by MPI-IO

IO-time collective buffering



And there is more

- http://docs.cray.com
  - Search for MPI-IO : „Getting started with MPI I/O", „Optimizing MPI-IO for Applications on CRAY XT Systems"
  - Search for lustre (a lot for admins but not only)
  - Message Passing Toolkit
- Man pages (man mpi, man <mpi_routine>, ...)
- mpich2 standard : http://www.mcs.anl.gov/research/projects/mpich2/

Summary

- POSIX
  - single reader/writer, all read/write, subset read/write
  - user is responsible for communication
- MPI I/O
  - MPI library is responsible for communication
  - file views enable non-contiguous access patterns
  - collective I/O can enable the actual disk access to remain contiguous