MPI I/O

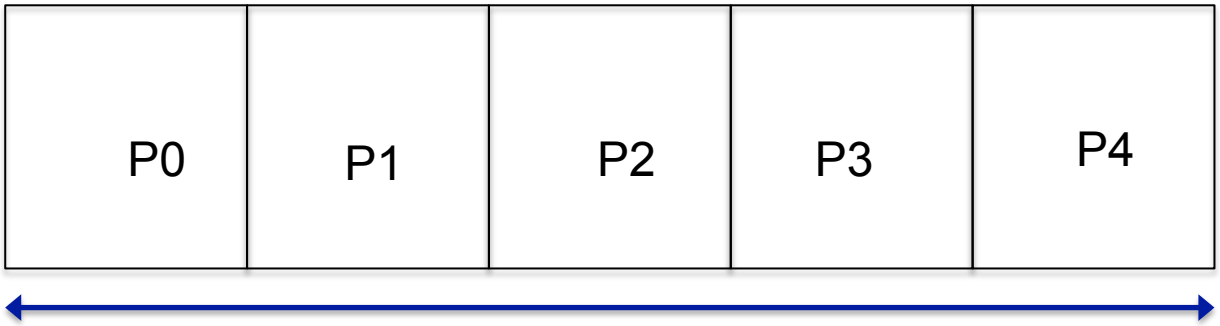
Be sure to look at Cray document
S–2490–40 “Getting Started with MPI-IO”

MPI-I/O

- Defined by the MPI specification
- Allows an application to write into both
  - distinct files
  - or the same file from multiple MPI processes
- Uses MPI datatypes to describe both the file and the process data
- Supports collective operations

Example

Suppose we have a file containing integers, total size FILESIZE bytes



Each processor wants to read an equal portion of the file.....
What does it need to know?

(1) How many integers (bytes) do I want to read?
(2) How far into the file do I need to start reading those integers?

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);
```

And now in Fortran using explicit offsets

```
use mpi ! or include 'mpif.h'
integer status(MPI_STATUS_SIZE)
integer (kind=MPI_OFFSET_KIND) offset ! Note, might be
                                      ! integer*8

call MPI_FILE_OPEN(MPI_COMM_WORLD, 'FILE', &
    MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)
nints = FILESIZE / (nprocs*INTSIZE)
offset = rank * nints * INTSIZE
call MPI_FILE_READ_AT(fh, offset, buf, nints,
MPI_INTEGER, status, ierr)
call MPI_GET_COUNT(status, MPI_INTEGER, count, ierr)
print *, 'process ', rank, 'read ', count, 'integers'
call MPI_FILE_CLOSE(fh, ierr)
```

- The *_AT routines are thread safe (seek+I/O operation in one call)

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.

Collective I/O 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

MPI-IO Case Study

Executive Summary

Benchmark with original Fortran I/O (192 steps)

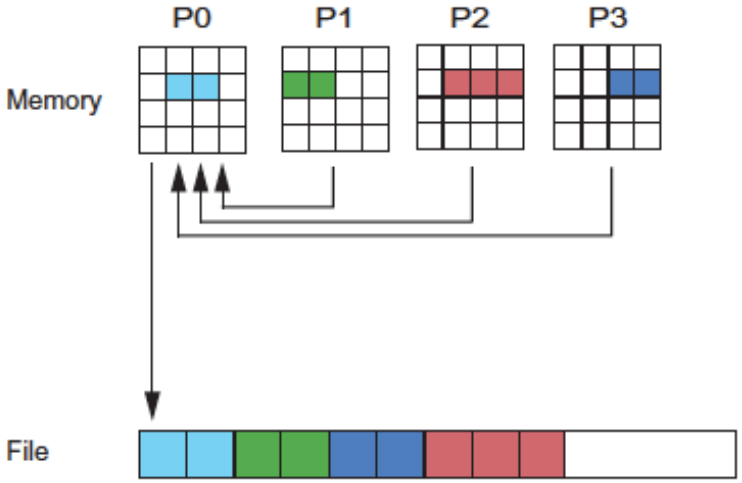
Cores	Walltime	Mean time per step (omit first step)	Time for first time step
960	1459	6.0	26
1920	741	3.3	19
3840	601	1.9	67
4800	628	1.6	104
5760	786	1.5	156

Walltime includes initialization and final I/O phase (of similar order to first time step)

Original code – serial Fortran I/O – 2GB written per output step

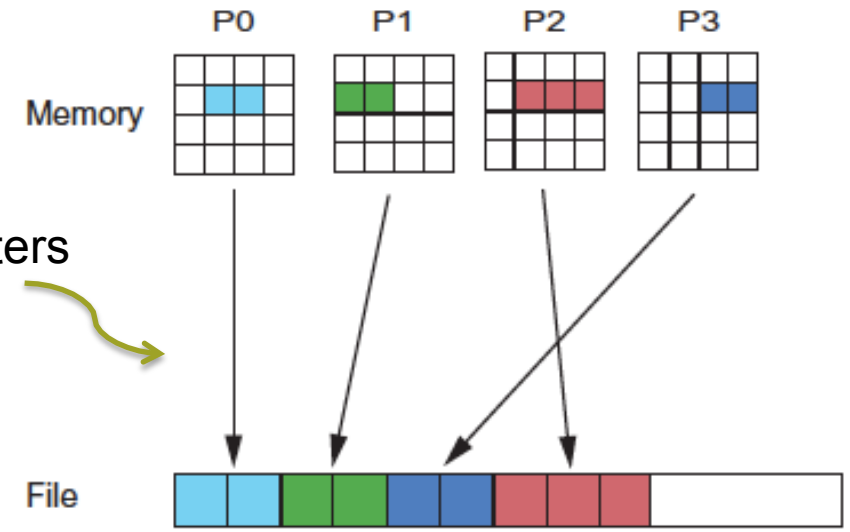
```
ALLOCATE(globbuf3d(ids:ide,1:nzp,jds:jde))
ALLOCATE(globbuf(ids:ide,jds:jde))

! 2D data: one Fortran write per 2D array

 CALL patch2global(my2darray,globbuf,mgrid)          ! All PEs send their data to PE 0 in an
                                                     ! mpi_send/mpi_recv loop

 IF (PE0) WRITE(fid) ((globbuf(i,j),i=ids,ide),j=jds,jdg)      <-- PE0 writes data to file

!Repeat for a dozen or so 2D arrays

! 3D data: extra loop over k dimension - nzp Fortran writes per 3D array
 CALL patch2global(my3darray,globbuf3d,local_grid)
   IF ( PE0 ) THEN
     DO k= 1,nzp
       WRITE(fid) ((globbuf3d(i,k,j),i=ids,ide),j=jds,jdg)
     ENDDO
   ENDIF

! And repeat for many 3D input arrays
! Lots of individual writes, all from rank 0!
```

We need parallel I/O!



From: Single file, single writer

To: Single file, many writers



Taken from "Getting Started on MPI-IO"
S-2490-40

MPI-IO: basic concepts

- **Improved output speed in large parallel environments; many processors can output to single file**
- **Focus on collective I/O for this case (_all)**
  - MPI-IO library optimizes I/O pattern
- **Needed commands to open and close files, read and write data, and move file pointers**
- **“File views” describe the layout of data across processors**
  - MPI-IO makes heavy use of MPI types to describe file views
  - Each processor has its own view of the file

MPI-IO: file view

- Give more control over where a given processor will write its data
- Use MPI types to describe the section of the global data that the current processor has, and its location in the final file on disk. Makes it easier to write multidimensional arrays into a single file
- A big advantage of MPI I/O over Unix I/O is the ability to specify noncontiguous accesses in memory and file within a single function call by using derived datatypes

MPI_File_Set_View syntax

CALL MPI_FILE_SET_VIEW (fid, offset, etype, filetype, datarep, info, ierr)

- Sets the file view on file handle fid – assigns region of file to each process
  - Defines which portion of file is visible to each process
- Specified by a triplet (offset, etype, filetype)
  - etype and filetype are mpi_datatypes
    - etype is the basic unit being written to the file (e.g. mpi_real8)
    - filetype is normally a derived datatype that describes the layout (often noncontiguous) of the data in the current process in the file
  - offset is the number of bytes from the start of the file at which to apply the file view
- datarep is a string describing the data representation, usually (always for Cray) "native"

MPI_file_set_view

- For individual file pointer each process can specify its own view
- View can be changed any number of times during a program
- All file access done in units of etype
- filetype must be equal to or be derived from etype:
  - Elementary: mpi_int, mpi_real etc
  - Contiguous data – sequences of elementary types
  - Vector – sequences separated by constant stride
  - Indexed – does not assume constant stride
  - Struct – general mixed types

Think about the data that each processor holds and where it is in the full array (3D real*4 array)

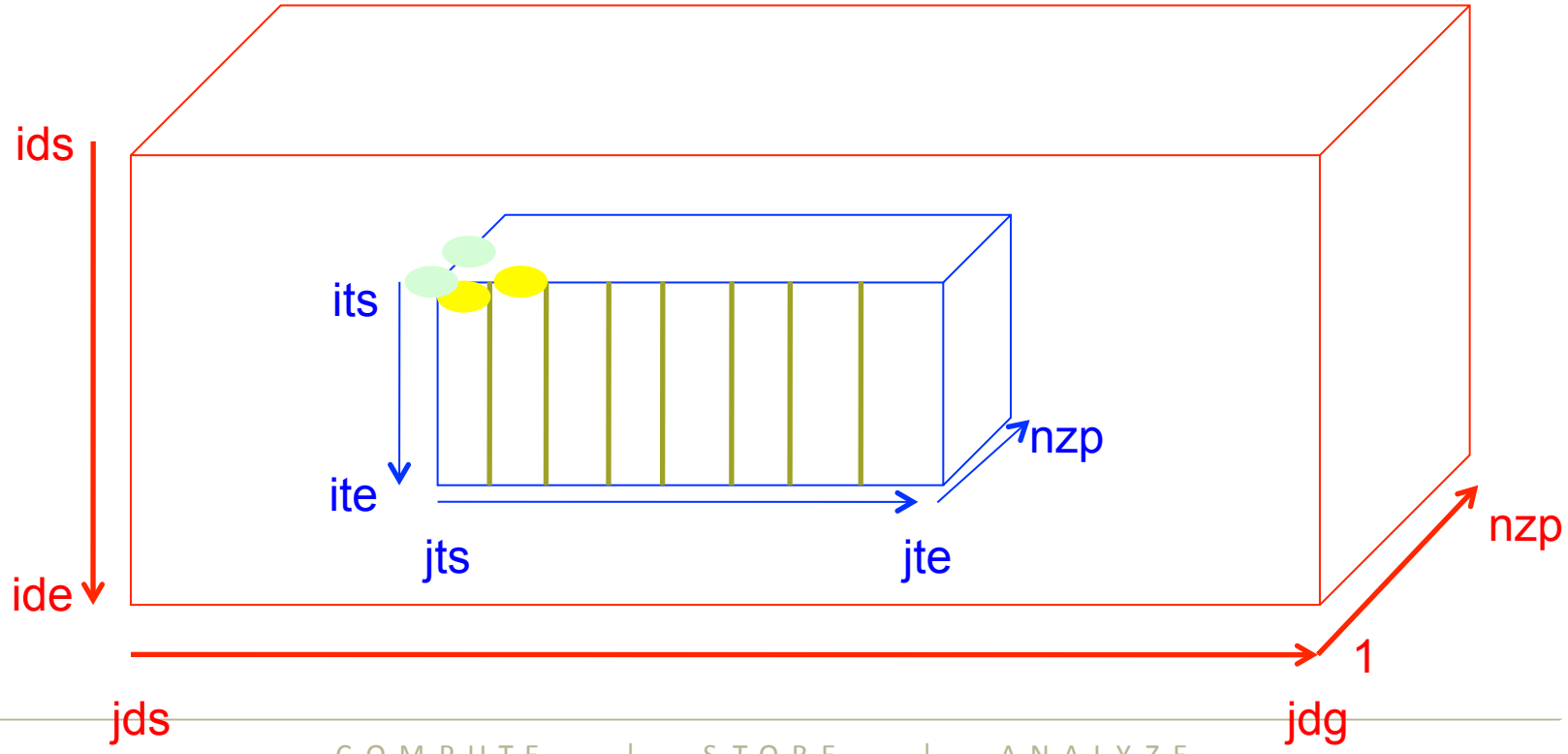
```
CALL patch2global(my3darray,globbuf3d,local_grid)
      IF ( PE0 ) THEN
            DO k= 1,nzp
                  WRITE(fid) ((globbuf3d(i,k,j),i=ids,ide),j=jds,jdg)
            ENDDO
      ENDIF
```

- Each processor holds a "cube" of data within the 3D array to be written to file
  - i: its -> ite (rows)
  - j: jts -> jte (cols)
  - k: 1 -> nzp (levels)
- Remember this is Fortran, so columns written first
- One column holds (jte-jts+1) contiguous (in file) reals
  - One stride between start of each column
  - Different stride between start of each level
- Processor grid does not exactly divide physical grid, so number of rows and cols held by each processor can vary

A few important quantities…

Global quantities:
stridev = (ide-ids+1) #rows or stride between cols
sizevar = stridev * (jdg-jds+1) #elements in full 2D slice or
stride between each k layer

Processor quantities
countv = min(jdg,jte)-jts+1 #cols held by PE
blengthv = (ite-its)+1 #rows held by PE
n_write = count*blengthv*nzp #elements held by PE



Final code for 3D arrays (take into account Fortran style of output file)

```
! Number of cols held by each PE
      count=min(jdg,jte)-jts+1
! Stride between start of each column held by PE
      stride=ide-ids+1
! Number of rows held by each PE
      blength=ite-its+1

! Amount of data this processor is writing (2D block)
      n_write=count*blength
! Size of global block
      sizeglobal = int(stride,8) * int((jdg-jds+1),8) * realsize8

! Size in bytes of global file (for header)
! Each 3D array is of same size
      lenrec = sizeglobal
      call byteswap(lenrec)

! Create MPI datatype for contiguous column data with
! constant stride
      call mpi_type_vector(count,blength,stride,mpi_real4,filetype,ierr)
      call mpi_type_commit(filetype,ierr)

      offset= offset_save +  ((jts-1)*int(stride,8) + (its-1))*realsize8


! Loop over all 3D arrays.
      do nvar=1,nvars_3d

! Loop over all k levels.
      do k=1,nzp
```

```
! First write the header.  etype and filetype are the same
      call mpi_file_set_view(nvel,offset,mpi_integer4,mpi_integer4, &
           'native',mpi_info_null,ierr)

    if (PE0) call mpi_file_write(nvel,lenrec,1,mpi_integer4,status,ierr)
 ! Update offset
      offset = offset + intsize8

! Then write the data (all 3D arrays stored in one array "temp")
      call mpi_file_set_view(nvel,offset,mpi_real4,filetype,'native', &
                mpi_info_null,ierr)
      off=0
      call mpi_file_write_at_all(nvel,off,temp(1,1,k,nvar),n_write,&
           mpi_real4,status,ierr)
      offset=offset + sizeglobal

! Then write the tail
     call mpi_file_set_view(nvel,offset,mpi_integer4,mpi_integer4,&
           'native',mpi_info_null,ierr)

     if (PE0)  call mpi_file_write(nvel,lenrec,1,mpi_integer4,status,ierr)

      offset = offset + intsize8

      enddo              ! k levels
enddo                    ! 3D arrays
```

And finally…

Cores	Walltime (orig)	Time for 1st time step	Walltime (new)	Time for 1st time step
960	1459	26	1233	21
1920	741	19	687	14
3840	601	67	418	15
4800	628	104	379	17
5760	786	156	384	19
6000			357	23

Walltime includes initialization and final I/O phase (of similar order to first time step), all of which have been optimized.

NetCDF, parallel NetCDF, HDF5...

The Purpose of NetCDF

- The purpose of the Network Common Data Form (NetCDF) interface is to allow you to create, access, and share array-oriented scientific data in a form that is self-describing and portable.
  - **Self-describing** means that a dataset includes information defining the data it contains (cf. utilities such as ncdump).
  - **Portable** means that the data in a dataset is represented in a form that can be accessed by computers with different ways of storing integers, characters, and floating-point numbers.
- The NetCDF software includes C, Fortran 77, Fortran 90, and C++ interfaces for accessing netCDF data.
- These libraries are available for many common computing platforms.

Format description

- The NetCDF libraries support three different binary formats for NetCDF files:
  - The classic format was used in the first NetCDF release, and is still the default format for file creation.
  - The 64-bit offset format was introduced in version 3.6.0, and it supports larger variable and file sizes.
  - The NetCDF-4/HDF5 format was introduced in version 4.0; it is the HDF5 data format, with some restrictions.
- All formats are "self-describing"
- Starting with version 4.0, the netCDF API allows the use of the HDF5 data format.
  - NetCDF users can create HDF5 files with benefits not available with the NetCDF format, such as much larger files and multiple unlimited dimensions.
- Full backward compatibility in accessing old NetCDF files and using previous versions of the C and Fortran APIs is supported.

Parallelism and NetCDF

- **Prior to version 4, NetCDF APIs did not support parallel I/O.**
  - For parallel programs, write operations must be done by shipping data to a single process which then writes to the file. Thus, the communication contention on the writing process can make the I/O performance considerably slow.
- **Starting from version 4, NetCDF supports parallel I/O either through PNetCDF or HDF5.**
  - Through PNetCDF, NetCDF-4 can access files in CDF formats in parallel.
  - Through HDF5, NetCDF-4 can access files in HDF5 format.
- **Modules on XC:** cray-netcdf and cray-parallel-netcdf

NetCDF HDF5 data format

- NetCDF-4 files are created with the HDF5 library, and they are HDF5 files that can be read without the NetCDF-4 interface.
  - Note that modifying these files with HDF5 will almost certainly make them unreadable to NetCDF-4
- Groups in a NetCDF-4 file correspond with HDF5 groups
- Variables in NetCDF correspond with identically named datasets in HDF5.
  - And similarly for attributes
- Since there is more metadata in a NetCDF file than in an HDF5 file, special datasets are used to hold NetCDF metadata.
  - The _netcdf_dim_info dataset (in group _netCDF) contains the ids of the shared dimensions, and their length (0 for unlimited dimensions).
  - The _netcdf_var_info dataset (in group _netCDF) holds an array of compound types which contain the variable ID, and the associated dimension ids.
- Backward compatibility to the classical format is preserved
- Support for parallel I/O
- http://www.unidata.ucar.edu/netcdf/netcdf-4.

Using Parallel I/O in NetCDF4

- Special nc_create_par and nc_open_par functions are used to create/open a NetCDF file. The files they open are normal NetCDF-4/HDF5 files, but these functions also take MPI parameters.

- The parallel access associated with these functions is not a characteristic of the data file, but the way it was opened. The data file is the same, but using the parallel open/create function allows parallel I/O to take place.

  - EXTERNAL int nc_create_par(const char *path, int cmode, MPI_Comm comm, MPI_Info info, int *ncidp);
  - EXTERNAL int nc_open_par(const char *path, int mode, MPI_Comm comm, MPI_Info info, int *ncidp);

Fortran 90 example

```
!    This is a very simple example which writes a 2D array of sample
!    data. To handle this in netCDF we create two shared dimensions,
!    "x" and "y", and a netCDF variable, called "data". It uses
!    parallel I/O to write the file from all processors at the same
!    time.
```

See “simple_xy_par_wr.F90 in NetCDF documentation along with loads of other examples....

In next slide, all NetCDF functions are called from within this “check” subroutine:

```
subroutine check(status)
    integer, intent ( in) :: status
    if(status /= nf90_noerr) then
      print *, trim(nf90_strerror(status))
      stop 2
    end if
  end subroutine check
```

Fortran 90 example (cont.)

```
program simple_xy_par_wr
    use netcdf
    implicit none
    include 'mpif.h'

    ! This is the name of the data file we will create.
    character (len = *), parameter :: FILE_NAME = "simple_xy_par.nc"
    ! We are writing 2D data.
    integer, parameter :: NDIMS = 2
    ! When we create netCDF files, variables and dimensions, we get back an ID
    !for each one.
    integer :: ncid, varid, dimids(NDIMS), x_dimid, y_dimid
    ! These will tell where in the data file this processor should write.
    integer :: start(NDIMS), count(NDIMS)
    ! This is the data array we will write. It will just be filled with the rank of
    ! this processor.
    integer, allocatable :: data_out(:)
    ! MPI stuff: number of processors, rank of this processor, and error code.
    integer :: p, my_rank, ierr
    ! Loop indexes, and error handling.
    integer :: x, stat

    ! Initialize MPI, learn local rank and total number of processors.
    call MPI_Init(ierr)
    call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierr)
    call MPI_Comm_size(MPI_COMM_WORLD, p, ierr)

    ! Create some pretend data. We just need one row.
    allocate(data_out(p), stat = stat)
   do x = 1, p
      data_out(x) = my_rank
    end do
    ! Create the netCDF file. The NF90_NETCDF4 flag causes a HDF5/netCDF-4 file to be
    ! created. The comm and info parameters cause parallel I/O to be enabled. Use either
    ! NF90_MPIIO or NF90_MPIPOSIX to select between  MPI/IO and MPI/POSIX.
    call check(nf90_create(FILE_NAME, IOR(NF90_NETCDF4, NF90_MPIIO), ncid, &
        comm = MPI_COMM_WORLD, info = MPI_INFO_NULL))

    ! Define the dimensions. NetCDF will hand back an ID for each. Metadata
    !operations must take place on all processors.
    call check(nf90_def_dim(ncid, "x", p, x_dimid))
    call check(nf90_def_dim(ncid, "y", p, y_dimid))

    ! The dimids array is used to pass the IDs of the dimensions of the variables. Note
    ! that in fortran arrays are stored in column-major format.
    dimids = (/ y_dimid, x_dimid /)

    ! Define the variable. The type of the variable in this case is NF90_INT (4-byte
    ! integer).
    call check(nf90_def_var(ncid, "data", NF90_INT, dimids, varid))

    ! End define mode. This tells netCDF we are done defining metadata. This operation
    !  Is collective and all processors will write their metadata to disk.
    call check(nf90_enddef(ncid))

    ! Write the pretend data to the file. Each processor writes one row.
    start = (/ 1, my_rank + 1/)
    count = (/ p, 1 /)
    call check(nf90_put_var(ncid, varid, data_out, start = start, count = count))

    ! Close the file. This frees up any internal netCDF resources associated with the file,
    ! and  flushes any buffers.
    call check( nf90_close(ncid) )
    ! Free my local memory.
    deallocate(data_out)
    ! MPI library must be shut down.
    call MPI_Finalize(ierr)
    if (my_rank .eq. 0) print *, "*** SUCCESS writing example file ", FILE_NAME, "! "

end program simple_xy_par_wr
```

Similar c example of parallel I/O

```
/* Create a parallel netcdf-4 file. */
if ((res = nc_create_par(FILE, NC_NETCDF4|NC_MPIIO, MPI_COMM_WORLD, info, &ncid))) ERR;

/* Create two dimensions. */
if ((res = nc_def_dim(ncid, "d1", DIMSIZE, dimids))) ERR;
if ((res = nc_def_dim(ncid, "d2", DIMSIZE, &dimids[1]))) ERR;

/* Create one var. */
if ((res = nc_def_var(ncid, "v1", NC_INT, NDIMS, dimids, &v1id))) ERR;

if ((res = nc_enddef(ncid))) ERR;

/* Set up slab for this process. */
start[0] = mpi_rank * DIMSIZE/mpi_size; start[1] = 0;
count[0] = DIMSIZE/mpi_size; count[1] = DIMSIZE;

/* Create phoney data.
for (i=mpi_rank*QTR_DATA; i < (mpi_rank+1)*QTR_DATA; i++) data[i] = mpi_rank;

if ((res = nc_var_par_access(ncid, v1id, NC_INDEPENDENT))) ERR;
if ((res = nc_put_vara_int(ncid, v1id, start, count, &data[mpi_rank*QTR_DATA]))) ERR;

if ((res = nc_var_par_access(ncid, v1id, NC_COLLECTIVE))) ERR;
if ((res = nc_put_vara_int(ncid, v1id, start, count, &data[mpi_rank*QTR_DATA]))) ERR;

if ((res = nc_close(ncid))) ERR;
```

Parallel NetCDF

- **Parallel NetCDF (PNetCDF)**
  - Collaborative work of Northwestern University and Argonne National Laboratory, extension of Unidata's netCDF
  - Parallel I/O library for accessing NetCDF files in CDF and CDF-2 (and CDF-5) formats.
- **NetCDF supports parallel I/O starting from version 4, but the file format is restricted to HDF5.**
  - PNetCDF is currently the only choice to perform parallel I/O on files in classic formats (CDF-1 and 2).
- **Goal of PNetCDF:provide high-performance parallel I/O to the applications by enabling all client processes to access a shared file in parallel.**
- **PNetCDF is built on top of MPI-IO.**
  - For both performance and portability
- **PNetCDF APIs mimic the syntax of the NetCDF APIs with only minor changes to add parallel I/O concept.**
  - Ease of code migration

Parallel NetCDF

- **Unfortunately, the PnetCDF package implements a different API from the netCDF API, making portability with other NetCDF code a problem.**
- **But! Starting from version 4.2.1.1, netCDF-4 programs can perform parallel I/O on the classic CDF-1 and CDF-2 files through PNetCDF. This is done by passing file create flag NC_PNETCDF** to nc_create_par(), for instance:
  - nc_create_par(filename, NC_PNETCDF, MPI_COMM_WORLD, info, &ncid);
- **Great guide with lots of links: http://cucis.ece.northwestern.edu/projects/PnetCDF/**
- **Also see http://trac.mcs.anl.gov/projects/parallel-netcdf**

Features

- PnetCDF contains a set of new APIs for accessing netCDF files in parallel
- The new APIs incorporate the parallel semantics following the Message Passing Interfaces (MPI) and provide backward compatibility with the netCDF file formats
- Minimize the changes to the netCDF API syntax (see next slide)

Features 2

- PNetCDF APIs mimic the syntax of the netCDF APIs with only a few changes to add parallel I/O concept.
  - All parallel APIs are named after originals with prefix of
    - "ncmpi_" for C/C++,
    - "nfmpi_" for Fortran 77
    - "nf90mpi_" for Fortran 90
- An MPI communicator and an MPI_Info object are added to the argument list of the open/create APIs. The communicator defines the set of processes accessing the NetCDF file. The info object allows users to specify I/O hints for PnetCDF and MPI-IO to further improve performance
- PNetCDF allows two I/O modes, collective and independent, which correspond to MPI collective and independent I/O operations. Similar to MPI naming convention, all collective APIs carry an extra suffix "_all".
- PnetCDF changes the integer data types for all the API arguments that are defined as size_t in NetCDF to MPI_Offset.

HDF5 – a brief mention

See

http://www.hdfgroup.org/HDF5/

http://www.hdfgroup.org/HDF5/Tutor/introductory.html

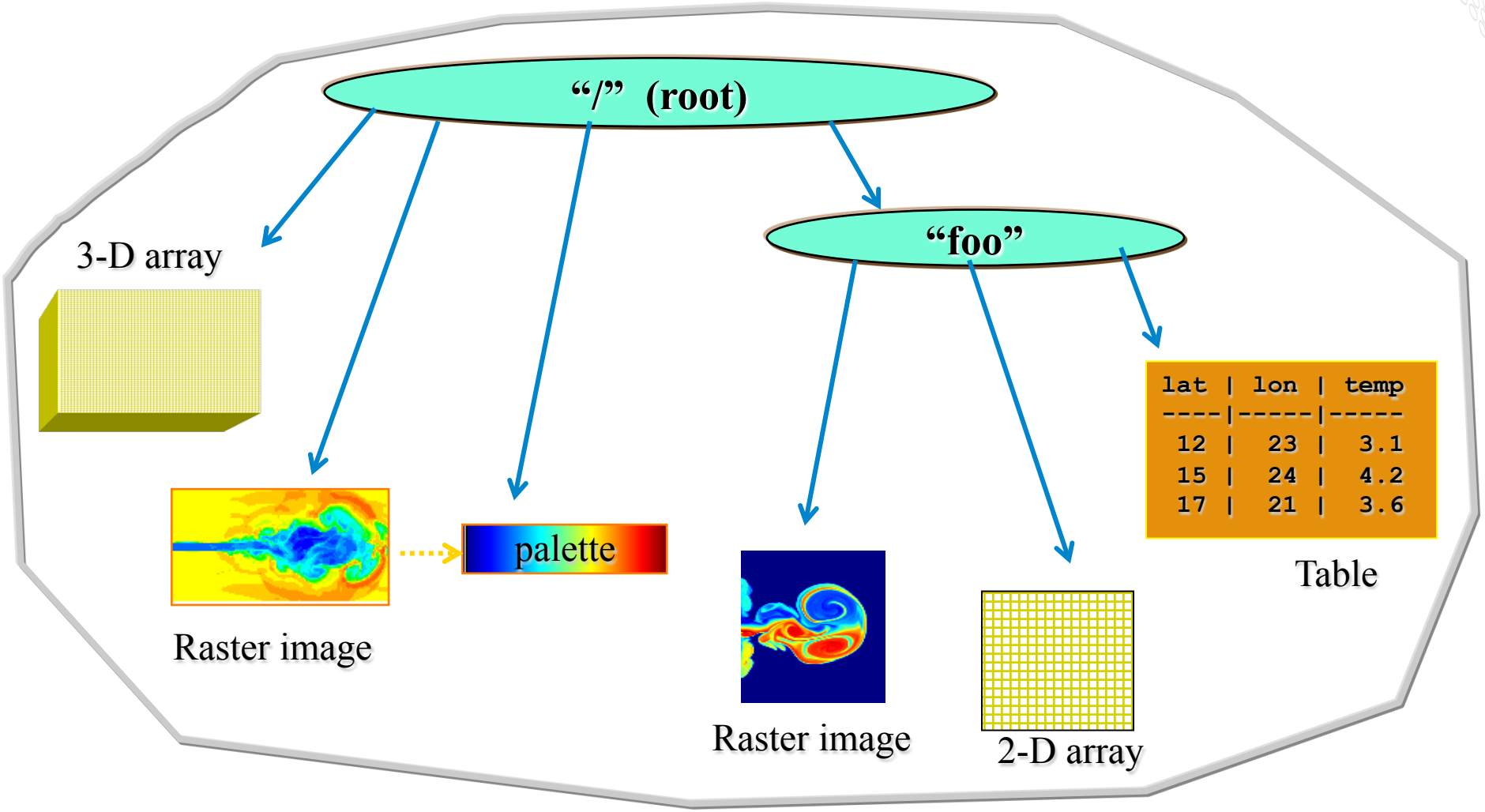
What is HDF5

- A versatile data model that can represent very complex data objects and a wide variety of metadata.
- A completely portable file format with no limit on the number or size of data objects in the collection.
- A software library that runs on a range of computational platforms, from laptops to massively parallel systems, and implements a high-level API with C, C++, Fortran 90, and Java interfaces.
- A rich set of integrated performance features that allow for access time and storage space optimizations.
- Tools and applications for managing, manipulating, viewing, and analyzing the data in the collection.

An HDF5 file is a container…

…into which you can put your data objects.

HDF5 Structures for Organizing Objects



HDF5 Data Model

- The HDF5 data is organized by only 2 primary objects
  - Groups
    a grouping structure containing instances of zero or more groups or datasets, together with supporting metadata.
  - Datasets
    a multidimensional array of data elements, together with supporting metadata

- You can also use additional way to organize and annotate data
  - Attributes
  - Storage and access properties

HDF5 API

- The HDF5 library provides several interfaces, or APIs. These APIs provide routines for creating, accessing, and manipulating HDF5 files and objects.
- The library itself is implemented in C. To facilitate the work of FORTRAN 90, C++ and Java programmers, HDF5 function wrappers have been developed in each of these languages.
- All C routines in the HDF5 library begin with a prefix of the form *H5**, where * is one or two uppercase letters indicating the type of object on which the function operates. (see next slide)
- The FORTRAN wrappers come in the form of subroutines that begin with *h5* and end with *_f*.

What is an HDF5 file?

- An HDF5 file is a binary file containing scientific data and supporting metadata.
- To create an HDF5 file, an application must specify not only a file name, but a file access mode, a file creation property list, and a file access property list.
- The file can be analyzed and used with provided tools like h5dump

Example 1 : Creating a HDF5 file in C

```
#include "hdf5.h"
#define FILE "file.h5"
int main() {
hid_t file_id; /* file identifier */
herr_t status;

/* Create a new file using default properties. */
file_id = H5Fcreate(FILE, H5F_ACC_TRUNC, H5P_DEFAULT,
          H5P_DEFAULT);
/* Terminate access to the file. */
status = H5Fclose(file_id);
}
```

H5dump of the output from example 1

You can use h5dump to see the structure of a HDF5 file

```
>h5dump  file.h5
HDF5 "file.h5" {
GROUP "/" {
}
}
```

The file from last example only contains the group ‘/’

Parallel HDF5

Parallel HDF5 design goals

- There were several requirements that the HDF5 group had for Parallel HDF5 (PHDF5).
  - Parallel HDF5 files had to be compatible with serial HDF5 files and sharable between different serial and parallel platforms.
  - Parallel HDF5 had to be designed to have a single file image to all processes, rather than having one file per process. Having one file per process can cause expensive post processing, and the files are not usable by different processes.
  - A standard parallel I/O interface had to be portable to different platforms.
- With these requirements of HDF5 the initial target was to support MPI programming, but not shared memory programming.

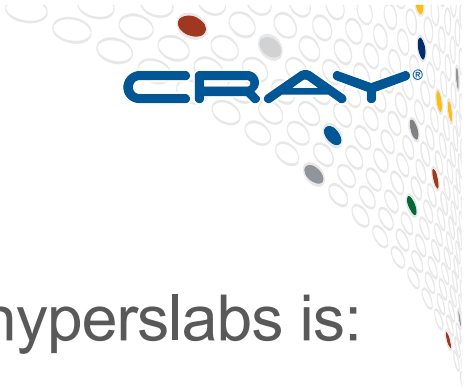
How to use

- An HDF5 file can be opened in parallel from an MPI application by specifying a parallel 'file driver' with an MPI communicator and info structure.

- This information is communicated to HDF5 through a 'property list,' a special HDF5 structure that is used to modify the default behavior of the library. In the following code, a file access property list is created and set to use the MPI-IO file driver:

```
/* create the file in parallel */
fapl_id = H5Pcreate(H5P_FILE_ACCESS);

H5Pset_fapl_mpio(fapl_id, mpi_comm, mpi_info);

file_id = H5Fcreate("myparfile.h5", H5F_ACC_TRUNC, H5P_DEFAULT, fapl_id);
```

Writing and Reading Hyperslabs



- The programming model for writing and reading hyperslabs is:
  - Each process defines the memory and file hyperslabs.
  - Each process executes a partial write/read call which is either collective or independent.
- The memory and file hyperslabs in the first step are defined with H5Sselect_hyperslab.
- The *start* (or *offset*), *count*, *stride*, and *block* parameters define the portion of the dataset to write to. By changing the values of these parameters you can write hyperslabs with Parallel HDF5 by contiguous hyperslab, by regularly spaced data in a column/ row, by patterns, and by chunks

Writing in parallel : code extract

```
// Init MPI
MPI_Init(&argc, &argv); MPI_Comm_size(comm, &mpi_size); MPI_Comm_rank(comm, &mpi_rank);

// Set up file access property list with parallel I/O access
plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist_id, comm, info);

// Create a new file collectively and release property list identifier.
file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
H5Pclose(plist_id);

// Create the dataspace for the dataset.
filespace = H5Screate_simple(RANK, dimsf, NULL);

// Create the dataset with default properties and close filespace.
dset_id = H5Dcreate(file_id, DATASETNAME, H5T_NATIVE_INT, filespace, H5P_DEFAULT, H5P_DEFAULT,
                    H5P_DEFAULT);

// Create property list for collective dataset write.
plist_id = H5Pcreate(H5P_DATASET_XFER);

// To write dataset collective use
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);
// To write dataset independently use
// H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_INDEPENDENT);

// Write the data
status = H5Dwrite(dset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, plist_id, data);
```

There is a lot more to HDF5

Starting points

- http://www.hdfgroup.org/
- http://www.hdfgroup.org/HDF5/Tutor/
- http://www.hdfgroup.org/HDF5/Tutor/parallel.html