Parallel I/O Techniques

Science & Technology Support
High Performance Computing

Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212-1163

OSC
Innovations in computing, networking, and education

Workshop Overview

- Introduction to Parallel I/O
  - RAID
  - Distributed File Systems
  - Parallel File Systems
- Simplistic Parallel I/O
  - Single Process I/O
  - Post-Mortem Reassembly
  - Fortran Direct Unformatted I/O
- MPI-2 I/O Interface
  - Basics
  - File Views
  - Non-blocking Operations
  - Collective Operations
  - File System Hints

Workshop Overview (con't)

- P2NM (MPI-IO example)
  - Concepts
  - Metadata Operations
  - Raster Data I/O Operations
- HDF5
  - Concepts
  - Basics
  - Parallel Interface
  - High-Level Interfaces
- Parallel NetCDF
  - Concepts
  - Parallel Interface
  - Differences from Serial NetCDF
- References

Introduction to Parallel I/O

- Motivations for Parallel I/O
- RAID Technology
- Distributed File Systems
- Parallel File Systems
- Cluster File Systems

Why Think About Parallel I/O?

- Supercomputer(n): A computer which turns a CPU-bound problem into an I/O-bound problem.
- As computers become faster and more parallel, the (often serialized) I/O bus can often become the bottleneck for large computations:
  - Checkpoint/restart files
  - Plot files
  - Scratch files for out-of-core computations
- In some ways, I/O is a lot like a form of one-sided message passing to a file:
  - “Get” (read) and “Put” (write) operations
  - Can be shared between many processes/threads
  - Can use data abstractions (eg. MPI derived datatypes)

I/O Needs on Parallel Computers

- High Performance
  - Take advantage of parallel I/O paths (when available).
  - Support for application-level tuning parameters.
- Data Integrity
  - Deal with hardware and power failures sanely.
- Single Namespace
  - All nodes “see” the same file systems.
  - Equal access from anywhere on the machine.
- Ease of Use
  - Where possible, a parallel file system should to be accessible in exactly the same ways as a traditional UNIX-style file system.

RAID Technology

- RAID == “Redundant Array of Inexpensive Disks”
- Developed in late ’80s in response to rapidly increasing storage capacity and performance needs.
- Basic idea is to increase capacity and performance by splitting a logical device (an “array”) across N disks, with redundancy so that the entire array is not lost if one disk fails
- Commonly available types of RAID:
  - RAID 0 – striping only; a portion of the data is written to each disk -- very fast for writes, but no data redundancy!
  - RAID 1 – mirroring; writes same data to N disks.
  - RAID 5 – striping plus parity; N-1 data stripes plus an XOR parity stripe written to disks – total usable capacity is (N-1)*disk_size. The parity stripes are written to different disks in a round-robin fashion.
  - RAID 10 – striping plus mirroring; a RAID 0 array in which each “disk” is a RAID 1 array – total usable capacity is (N/2)*disk_size

RAID Technology (con't)

- RAID can be implemented in either hardware or software:
  - Hardware
    - RAID controller card
    - RAID disk enclosure (appears as a single drive to controller)
    - Storage Area Network (SAN)
  - Software
    - OS level (eg. Linux software RAID, Irix XLV)
    - User-space (eg. PVFS)
- Almost any disk technology can be used for RAID, including Fibre Channel, SCSI, and even EIDE/ATA!

Distributed File Systems

- A distributed file system (DFS) is a file system that is stored locally on one system (the server) but is accessible by processes on many systems (clients).
- Other attributes of a DFS may include:
  - Access control lists (ACLs)
  - Client-side file replication
  - Server- and client-side caching
- Some examples of DFSes:
  - NFS (Sun)
  - AFS (CMU)
  - DCE/DFS (Transarc/IBM)
  - CIFS (Microsoft)

Distributed File Systems (con't)

- Distributed file systems can be used by parallel programs, but they have significant disadvantages:
  - The network bandwidth of the server system is a limiting factor on performance.
  - To retain UNIX-style file consistency semantics, the DFS software must implement some form of locking (another performance drain).

NFS: Example of a Distributed File System

- Single server, multiple clients
  - Same server can export several file systems
  - Clients can mount file systems from multiple servers
- RPC over UDP/IP is transport
  - Stateless
  - Have to have a transmit retry mechanism
- NFS v3 adds several things:
  - Asynchronous reads and writes
  - Locking (`rcp.lockd` and `rpc.statd`) that mostly works
  - LFS support for >2GB files on 32-bit platforms
  - Option of RPC over TCP/IP as transport
- NFS currently does **NOT** have
  - Client-side file replication
  - ACLs

Parallel File Systems

- A parallel file system is one in which there are multiple servers as well as clients for a given file system – the moral equivalent of RAID across several file servers.
- Parallel file systems are often optimized for high performance rather than general purpose use (i.e. you may not want your home directory on one!):
  - Very large block sizes (=>64kB)
  - Relatively slow metadata operations (eg. `fstat()`) compared to reads and writes
  - Special APIs for direct access
- Examples of parallel file systems:
  - GPFS (IBM)
  - Lustre (Cluster File Systems)
  - PVFS2 (Clemson/ANL)

PVFS2: Example of a Parallel File System

- Two-piece architecture
  - Multiple data servers, each using a local UNIX file system for data stripe storage; one or more of these also act as metadata servers.
  - Multiple clients.
  - Same host can be both data server and client.
- Default behavior is the moral equivalent of RAID 0 across data servers
  - Large block size (default 256kB).
  - No data redundancy by default; however; other data placement algorithms are possible.
- Multiple transports
  - TCP/IP
  - GM
  - InfiniBand
- Multiple APIs
  - PVFS2 library interface
  - UNIX file semantics using Linux kernel driver

Cluster File Systems

- A cluster file system is a file system in which one or more clients have direct block-level access to shared storage devices.
- "Server-less"
  - Clients access storage devices directly.
  - There may be one or more metadata servers, but they handle only file metadata, not data.
- Popular in the enterprise database market
  - Often optimized for direct, unbuffered I/O.
  - Buffered I/O typically has heavy-duty locking.
- Examples of cluster file systems:
  - CXFS (SGI)
  - GFS (Red Hat)
  - OCFS2 (Oracle)
  - QFS/SAMFS (Sun)
  - SANFS (IBM)

SANFS: Example of a Cluster File System

- Direct block-level access to storage from one or more clients
  - AIX, Linux, Solaris, Win2k, others.
  - Extremely large stripe size (256MB!).
  - Clients can re-export SANFS using NFS or CIFS.
- Clustered metadata servers
  - File system broken up into file sets (basically directory trees).
  - Every file set is “owned” by one metadata server.
  - In the case of a failed metadata server, a working server takes over the file sets of the failed server.
- Locking scheme for client-side caching
  - Always on for buffered I/O; not the same as flock() or fcntl().
  - Read locks can be shared, allowing multiple clients to cache file contents in memory.
  - Write locks cannot be shared and invalidate read locks.
  - Direct I/O can be used to bypass locks.

Simplistic Parallel I/O

There are several ways that people have attempted to address I/O in parallel programs without using a parallel I/O interface:

- Single I/O Process
- Post-Mortem Reassembly
- Fortran Direct Unformatted I/O

Unfortunately, none of these scale well to large parallel applications.

Single Process I/O

- In single process I/O, only one process or thread in the parallel application (usually rank==0) does any I/O.
  - Global data sent to other ranks using broadcasts (eg. `MPI_Bcast()`).
  - Local data for each process distributed using other message passing calls (eg. `MPI_Send()` and `MPI_Recv()`, or `MPI_Scatter()` and `MPI_Gather()`).
- This can be extremely painful to implement and has several significant disadvantages with respect to scalability:
  - The maximum I/O bandwidth available to the application is that available to a single process – not as much chance for parallelism in I/O.
  - Scattering and gathering data for checkpoints or plot file generation is very expensive in terms of message passing bandwidth and requires extra memory on rank==0 process.

Post-Mortem Reassembly

- In post-mortem reassembly, each process or thread in the parallel application reads and writes its own independent files. The output files from each process are then reassembled after the job completes using a separate program.
  - Does allow for up to N times single process I/O bandwidth in most cases.
  - Requires development of a reassembly tool.
  - Reassembly of hundreds or thousands of output file could take as long as the job in some cases!
  - Not much possibility for coordinated collective operations.

Example of Post-Mortem Reassembly

```
! Post mortem reassembly
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tstart=MPI_WTIME()
      open(unit=rank+10,form='unformatted')
      if (jstart.eq.2) then jstart=1
      if (jend.eq.(jmax-1)) then jend=jmax
      write(rank+10) 1,imax,jstart,jend
      write(rank+10) ((u(i,j),i=1,imax),j=jstart,jend)
      close(unit=rank+10)
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tend=MPI_WTIME()
      if (rank.eq.0) then
         write(*,*) 'Using post mortem reassembly'
         write(*,*) 'Wrote ',8*imax*jmax,' bytes in ',tend-tstart,
     +              ' seconds.'
         write(*,*) 'Transfer rate = ',
     +              (8*imax*jmax)/(tend-tstart)/(1024.**2),
     +              ' MB/s'
       endif
```

OSC
Innovations in computing, networking, and education

Fortran Direct Unformatted I/O

- The Fortran language has support for doing direct, record-oriented (i.e. fixed length) unformatted I/O to files:

```
OPEN(unit=99, form='unformatted', access='direct',&
  recl=imax*real_size)
DO j=jstart,jend
  WRITE(99, rec=j) u(1:imax,j)
END DO
CLOSE(unit=99)
```

- This can be used to write files in parallel if each process or thread picks non-overlapping regions of the files to write. There are drawbacks to this approach, however:
  - No support for collective operations.
  - Efficiency and consistency depend on how well the Fortran runtime library and the underlying file system handles concurrent read and write accesses to files.
  - Direct unformatted files are rarely portable between systems.

Example of Direct Unformatted I/O

```
! Direct, unformatted I/O
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tstart=MPI_WTIME()
      open(unit=99,form='unformatted',access='direct',
     +     recl=(8*imax))
      if (jstart.eq.2) then jstart=1
      if (jend.eq.(jmax-1)) then jend=jmax
      do j=jstart,jend
         write(99,rec=j) (u(i,j),i=1,imax)
      enddo
      close(unit=99)
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tend=MPI_WTIME()
      if (rank.eq.0) then
         write(*,*) 'Using direct unformatted I/O'
         write(*,*) 'Wrote ',8*imax*jmax,' bytes in ',tend-tstart,
     +              ' seconds.'
         write(*,*) 'Transfer rate = ',
     +              (8*imax*jmax)/(tend-tstart)/(1024.**2),
     +              ' MB/s'
      endif
```

MPI-2 I/O Interface

- Basics
- File Views
- Non-blocking Operations
- Shared File Pointer Operations
- Collective Operations
- File System Hints

MPI-IO Basics

- MPI-IO borrows two abstractions from MPI-1
  - Communicators
  - Data types
- File handles are defined in the context of an MPI communicator.
  - File `open()`s and `close()`s are collective operations.
  - MPI processes not in a communcator will not have access to that communicator's file handles.
- All I/O operations (`read()`, `write()`, `seek()`, etc.) have an MPI data type associated with them.
  - Can be a basic or derived data type.
  - Derived data types may not be contiguous in memory.
  - Data layout used in memory may not be the same as the data layout used on disk!

MPI-IO: The "Big Six"

While there are a large number of MPI-IO calls, most basic I/O can be handled with just six routines:

- `MPI_File_open()` – associate a file with a file handle.
- `MPI_File_seek()` – move the current file position to a given location in the file.
- `MPI_File_read()` – read some fixed amount of data out of the file beginning at the current file position.
- `MPI_File_write()` – write some fixed amount of data into the file beginning at the current file position.
- `MPI_File_sync()` -- flush any caches associated with the file handle.
- `MPI_File_close()` – close the file handle.

Most of the other MPI-IO routines are variations or optimizations on these basic calls.

MPI_File_open()

- C syntax:

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

- Fortran syntax:

```
subroutine MPI_File_open(comm, filename, amode,
info, fh, ierr)
character*(*) filename
integer comm, amode, info, fh, ierr
```

- `fh` is the file handle, which will be used by all other MPI-IO routines to refer to the file. In C, this must be passed as an address (i.e. `&fh`).
- `filename` is the name of the file to open. It may use a relative or absolute path. It may also contain a file system identifier prefix, eg.:
  - `ufs:` -- normal UNIX-style file system
  - `nfs:` -- NFS file system
  - `pvfs:` -- PVFS file system

MPI_File_open() (con't)

- amode is the access mode to open the file with. It should be a bitwise OR-ing (C) or sum (Fortran) of the following:
  - MPI_MODE_RDONLY (read-only)
  - MPI_MODE_RDWR (read/write)
  - MPI_MODE_WRONLY (write-only)
  - MPI_MODE_CREATE (create file if it doesn't exist)
  - MPI_MODE_EXCL (error if file does already exist)
  - MPI_MODE_DELETE_ON_CLOSE (delete file when closed)
  - MPI_MODE_UNIQUE_OPEN (file cannot be opened by other processes)
  - MPI_MODE_SEQUENTIAL (file can only be accessed sequentially)
  - MPI_MODE_APPEND (set initial file position to the end of the file)
- info is a set of file hints, the creation of which will be discussed later. When in doubt, use MPI_INFO_NULL.
- Note the each MPI process which opens a file will have its own individual file pointer; we will discuss shared file pointers later in the workshop.

MPI_File_seek()

- C syntax:

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

- Fortran syntax:

```
subroutine MPI_File_seek(fh, offset, whence, ierr)
integer fh, offset, whence, ierr
```

- `offset` determines how far from the current file position to move the current file position; this can be negative, although seeking beyond the beginning of the file (or the current view if one is in use) is an error.
- `whence` determines to where the seek offset is relative:
  - `MPI_SEEK_SET` (relative to the beginning of the file)
  - `MPI_SEEK_CUR` (relative to the current file position)
  - `MPI_SEEK_END` (relative to the end of the file)
- Seeks are done in terms of the current record type (`MPI_BYTE` by default, although this can be changed by setting a file view).

MPI_File_read()

- C syntax:

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

- Fortran syntax:

```
subroutine MPI_File_read(fh, buf, count, type,
status, ierr)
<type> BUF(*)
integer fh, count, type, status(MPI_STATUS_SIZE),
ierr
```

- This reads `count` values of datatype `type` from the file into `buf`. `buf` must be at least as big as `count*sizeof(type)`.
- `status` can be used to query for information such as how many bytes were actually read.

OSC
Innovations in computing, networking, and education

MPI_File_write()

- C syntax:

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

- Fortran syntax:

```
subroutine MPI_File_write(fh, buf, count, type,
status, ierr)
<type> BUF(*)
integer fh, count, type, status(MPI_STATUS_SIZE),
ierr
```

- This reads `count` values of datatype `type` from `buf` into the file. `buf` must be at least as big as `count*sizeof(type)`.
- `status` can be used to query for information such as how many bytes were actually written.

MPI_File_sync()

- C syntax:

```
int MPI_File_sync(MPI_File fh);
```

- Fortran syntax:

```
subroutine MPI_File_sync(fh, ierr)
integer fh, ierr
```

- Forces any caches associated with a file handle to be flushed to disk; maybe be very expensive for large files on slow file systems.
- Provides a way to force written data to be committed to disk.
- Collective; must be called by all processes.

MPI_File_close()

- C syntax:

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

- Fortran syntax:

```
subroutine MPI_File_close(fh, ierr)
integer fh, ierr
```

- This closes access to the file associated with the file handle `fh`.

A First MPI-IO Example

```
! Basic MPI-I/O
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tstart=MPI_WTIME()
      call MPI_FILE_OPEN(MPI_COMM_WORLD,'u.dat',MPI_MODE_WRONLY,
     +                   MPI_INFO_NULL,outfile,ierr)
      if (jstart.eq.2) then jstart=1
      if (jend.eq.(jmax-1)) then jend=jmax
      call MPI_FILE_SEEK(outfile,(jstart-1)
    *imax*8,MPI_SEEK_SET,ierr)
      do j=jstart,jend
         call MPI_FILE_WRITE(outfile,u(1,j),imax,MPI_REAL8,istat,
     +        ierr)
      enddo
      call MPI_FILE_SYNC(outfile,ierr)
      call MPI_FILE_CLOSE(outfile,ierr)
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tend=MPI_WTIME()
      if (rank.eq.0) then
         write(*,*) 'Using basic MPI-I/O'
```

A First MPI-IO Example (con't)

```
      write(*,*) 'Wrote ',8*imax*jmax,' bytes in ',tend-tstart,
     +           ' seconds.'
      write(*,*) 'Transfer rate = ',
     +           (8*imax*jmax)/(tend-tstart)/(1024.**2),
     +           ' MB/s'
      endif
```

Reading and Writing at a Specified Location

- It may be desirable in certain circumstances to write to an explicit offset into a file rather than to whatever location is pointed to be the current file pointer.
- In those circumstances, you can use `MPI_File_read_at()` or `MPI_File_write_at()` instead to their non-`_at` equivalents.

MPI_File_read_at()

- C syntax:

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

- Fortran syntax:

```
subroutine MPI_File_read_at(fh, offset, buf, count,
type, status, ierr)
<type> BUF(*)
integer fh, offset, count, type, status
(MPI_STATUS_SIZE), ierr
```

- This reads `count` values of datatype `type` from the file at location `offset` into `buf`. `buf` must be at least as big as `count*sizeof(type)`.

OSC
Innovations in computing, networking, and education

MPI_File_write_at()

- C syntax:

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

- Fortran syntax:

```
subroutine MPI_File_write(fh, offset, buf, count,
type, status, ierr)
<type> BUF(*)
integer fh, offset, count, type
integer status(MPI_STATUS_SIZE), ierr
```

- This writes `count` values of datatype `type` from `buf` into the file at location `offset`. `buf` must be at least as big as `count*sizeof(type)`.

OSC
Innovations in computing, networking, and education

MPI-IO File Views

- An MPI file view is analogous to an MPI data type, except that it applies to data layout on disk rather than in memory.
  - Describes how data should be layed out on disk
  - Three components
    - Initial displacement
    - Record type (basic or derived) -- a single "record" in the file; seeks are done in terms of this type
    - File type -- how multiple records are layed out relative to each other
  - May not be contiguous
    - Sparse files
    - Interleaved file access
    - Storage of multidimensional data
- The default file view is
  - Initial displacement == 0
  - Record type == `MPI_BYTE`
  - File type == `MPI_BYTE`

Creating an MPI File Type

- An MPI file type is simply an MPI derived data type which describes how records are placed in the file relative to each other.
- There are three parts to any MPI derived data type:
  - An array of MPI data types; this can include `MPI_LB` (a lower bound) and `MPI_UB` (an upper bound).
  - An array of block lengths which represents the repetition count of the corresponding MPI data type.
  - An array of displacements in bytes which represents the starting positions of the corresponding MPI data types relative to a common starting point.
- MPI file types must be created with `MPI_Type_struct()` and `MPI_Type_commit()`.

Setting an MPI File View

- Once record and file types and an initial displacement are known, an MPI file view can be set on the file handle with `MPI_File_set_view()`.
- The file view does not have to be the same on all MPI processes; in the case of interleaved file accesses, the initial displacement will likely be different on each MPI process.
- `MPI_File_set_view()` also takes a data representation argument, which can be used to control how data is written to disk. It can have the following values:
  - `"native"` -- data is written to disk exactly as it is stored in memory.
  - `"internal"` -- data is written to disk in a manner determined by the MPI implementation, which may or not not include conversion to a portable format.
  - `"external"` -- data is converted to a portable representation based on the XDR standard.

MPI_File_set_view()

- C syntax:

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

- Fortran syntax:

```
subroutine MPI_File_set_view(fh, disp, rectype,
filetype, datarep, info, ierr)
integer fh, rectype, filetype, info, ierr
integer(KIND=MPI_OFFSET_KIND) disp
character*(*) datarep
```

- This associates a file view with a file handle.

Example: Creating and Using an MPI File View

```
! MPI-IO using a file view
      types(1)=MPI_LB
      types(2)=MPI_REAL8
      types(3)=MPI_UB
      disp(1)=0
      disp(2)=0
      disp(3)=imax*8
      counts(1)=1
      counts(2)=imax
      counts(3)=1
      call MPI_TYPE_STRUCT(3,counts,disp,types,filetype,ierr)
      call MPI_TYPE_COMMIT(filetype,ierr)
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tstart=MPI_WTIME()
      call MPI_FILE_OPEN(MPI_COMM_WORLD,'u.dat',MPI_MODE_WRONLY,
     +                   MPI_INFO_NULL,outfile,ierr)
      if (jstart.eq.2) then jstart=1
      if (jend.eq.(jmax-1)) then jend=jmax
```

Example: Creating and Using an MPI File View (con't)

```
    call MPI_FILE_SET_VIEW(outfile,(jstart-1)*imax*8,MPI_REAL8,
   +                       filetype,'native',MPI_INFO_NULL,ierr)
    do j=jstart,jend
      call MPI_FILE_WRITE(outfile,u(1,j),
imax,MPI_REAL8,istat,ierr)
    enddo
    call MPI_FILE_SYNC(outfile,ierr)
    call MPI_FILE_CLOSE(outfile,ierr)
    call MPI_BARRIER(MPI_COMM_WORLD,ierr)
    tend=MPI_WTIME()
    if (rank.eq.0) then
       write(*,*) 'Using MPI-I/O with a file view'
       write(*,*) 'Wrote ',8*imax*jmax,' bytes in ',tend-tstart,
   +              ' seconds.'
       write(*,*) 'Transfer rate = ',
   +              (8*imax*jmax)/(tend-tstart)/(1024.**2),
   +              ' MB/s'
    endif
```

MPI-IO Non-blocking Operations

- Most of the MPI-IO routines block; in other words, they do not return until it is safe to reuse the buffer passed to them, either because the I/O operation has completed or because the data has been buffered. (Which of these is the case depends on the underlying filesystem.)
- MPI-IO does provide non-blocking forms of the basic read and write operations to allow for asynchronous I/O (the I/O analogue of non-blocking message passing):
  - `MPI_File_iread()`
  - `MPI_File_iread_at()`
  - `MPI_File_iwrite()`
  - `MPI_File_iwrite_at()`
- All of these return a `request` structure which can be checked with `MPI_Test()` or `MPI_Wait()`, just as with non-blocking message passing calls.

MPI_File_iread()

- C syntax:

```
int MPI_File_iread(MPI_File fh, void *buf, int
count, MPI_Datatype type, MPI_Request *request);
```

- Fortran syntax:

```
subroutine MPI_File_iread(fh, buf, count, type,
request, ierr)
<type> BUF(*)
integer fh, count, type, request, ierr
```

- This reads `count` values of datatype `type` from the file into `buf`. `buf` must be at least as big as `count*sizeof(type)`.

MPI_File_iread_at()

- C syntax:

```
int MPI_File_iread_at(MPI_File fh, MPI_Offset
offset, void *buf, int count, MPI_Datatype type,
MPI_Request *request);
```

- Fortran syntax:

```
subroutine MPI_File_iread_at(fh, offset, buf, count,
type, request, ierr)
<type> BUF(*)
integer fh, offset, count, type, request, ierr
```

- This reads `count` values of datatype `type` from the file at location `offset` into `buf`. `buf` must be at least as big as `count*sizeof(type)`.

OSC
Innovations in computing, networking, and education

MPI_File_iwrite()

- C syntax:

```
int MPI_File_iwrite(MPI_File fh, void *buf, int
count, MPI_Datatype type, MPI_Request *request);
```

- Fortran syntax:

```
subroutine MPI_File_iwrite(fh, buf, count, type,
request, ierr)
<type> BUF(*)
integer fh, count, type, request, ierr
```

- This writes `count` values of datatype `type` from `buf` into the file. `buf` must be at least as big as `count*sizeof(type)`.

MPI_File_iwrite_at()

- C syntax:

```
int MPI_File_iwrite_at(MPI_File fh, MPI_Offset
offset, void *buf, int count, MPI_Datatype type,
MPI_Request *request);
```

- Fortran syntax:

```
subroutine MPI_File_iwrite_at(fh, offset, buf,
count, type, request, ierr)
<type> BUF(*)
integer fh, offset, count, type, request, ierr
```

- This writes `count` values of datatype `type` from `buf` into the file at location `offset`. `buf` must be at least as big as `count*sizeof(type)`.

Example: Non-blocking I/O

```
! Non-blocking MPI-I/O
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tstart=MPI_WTIME()
      CALL MPI_File_open(MPI_COMM_WORLD,"u.dat",
     +     MPI_MODE_CREATE+MPI_MODE_WRONLY,MPI_INFO_NULL,
     +     outfile,ierr)
      if (jstart.eq.2) jstart=1
      if (jend.eq.(jmax-1)) jend=jmax
      call MPI_FILE_SEEK(outfile,(jstart-1)
   *imax*8,MPI_SEEK_SET,ierr)
      call MPI_FILE_IWRITE(outfile,u(1,jstart),(jend-jstart+1)
   *imax,
     +     MPI_REAL8,req,ierr)
! Go off and do other stuff not using u(:,:)...
      call MPI_WAIT(req,istat,ierr)
      call MPI_FILE_SYNC(outfile,ierr)
      call MPI_FILE_CLOSE(outfile,ierr)
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tend=MPI_WTIME()
```

Example: Non-blocking I/O (con't)

```
      if (rank.eq.0) then
        write(*,*) 'Using non-blocking MPI-I/O'
        write(*,*) 'Wrote ',8*imax*jmax,' bytes in ',tend-tstart,
     +             ' seconds.'
        write(*,*) 'Transfer rate = ',
     +             (8*imax*jmax)/(tend-tstart)/(1024.**2),
     +             ' MB/s'
       endif
```

OSC
Innovations in computing, networking, and education

MPI-IO Shared File Pointer Operations

- On top of the individual file pointer that each process receives when a file is opened, an `MPI_File` handle also contains a shared file pointer that can be used to coordinate access between all the processes that have a file open.
- This is **not** collective in the usual MPI sense, but it does imply a certain amount of shared state
- This can be very convenient in some situations, but it also has significant drawbacks:
  - Shared file pointers are generally less efficient than local file pointers due to the extra coordination required.
  - Some cluster parallel file systems (eg. PVFS) do not support shared file pointers.

MPI_File_seek_shared()

- C syntax:

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

- Fortran syntax:

```
subroutine MPI_File_seek_shared(fh, offset, whence,
ierr)
integer fh, offset, whence, ierr
```

- This moves the location of the shared file pointer by `offset`.
- In all other ways it behaves like `MPI_File_seek()`, including the legal values for `whence`.

MPI_File_read_shared()

- C syntax:

```
int MPI_File_read_shared(MPI_File fh, void *buf, int
count, MPI_Datatype type, MPI_Status *status);
```

- Fortran syntax:

```
subroutine MPI_File_read_shared(fh, buf, count,
type, status, ierr)
<type> BUF(*)
integer fh, count, type, ierr
integer status(MPI_STATUS_SIZE)
```

- This is the shared file pointer version of `MPI_File_read()`.

MPI_File_write_shared()

- C syntax:

```
int MPI_File_write_shared(MPI_File fh, void *buf,
int count, MPI_Datatype type, MPI_Status *status);
```

- Fortran syntax:

```
subroutine MPI_File_write_shared(fh, buf, count,
type, status, ierr)
<type> BUF(*)
integer fh, count, type, ierr
integer status(MPI_STATUS_SIZE)
```

- This is the shared file pointer version of `MPI_File_write()`.

MPI_File_iread_shared()

- C syntax:

```
int MPI_File_iread_shared(MPI_File fh, void *buf,
int count, MPI_Datatype type, MPI_Request *request);
```

- Fortran syntax:

```
subroutine MPI_File_iread_shared(fh, buf, count,
type, request, ierr)
<type> BUF(*)
integer fh, count, type, request, ierr
```

- This is the shared file pointer version of `MPI_File_iread()`.

OSC
Innovations in computing, networking, and education

MPI_File_iwrite_shared()

- C syntax:

```
int MPI_File_iwrite_shared(MPI_File fh, void *buf,
int count, MPI_Datatype type, MPI_Request *request);
```

- Fortran syntax:

```
subroutine MPI_File_iwrite_shared(fh, buf, count,
type, request, ierr)
<type> BUF(*)
integer fh, count, type, request, ierr
```

- This is the shared file pointer version of `MPI_File_iwrite()`.

MPI-IO Collective Operations

In addition to single-process reads and writes, MPI-IO supports two types of collective I/O operations:

- Blocking Collective I/O Operations
- Split (Non-Blocking) Collective I/O Operations

Blocking Collective I/O Operations

- Collective I/O operations allow for a greater degree of coordination in I/O than non-collective I/O operations.
- As with collective message passing operations, collective I/O operations have to be called by all MPI processes in a given communicator.
- There is no requirement that each process in a collective I/O operation read or write the same amount of information, or that their local file pointers point to the same location. In fact, in most cases one would want just the opposite -- a collective I/O operation where each process is reading or writing a different segment of the file, possibly with different amounts of data.

MPI_File_read_all()

- C syntax:

```
int MPI_File_read_all(MPI_File fh, void *buf, int
count, MPI_Datatype type, MPI_Status *status);
```

- Fortran syntax:

```
subroutine MPI_File_read_all(fh, buf, count,
type, status, ierr)
<type> BUF(*)
integer fh, count, type, ierr
integer status(MPI_STATUS_SIZE)
```

- This is the collective form of `MPI_File_read()`.

MPI_File_read_at_all()

- C syntax:

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

- Fortran syntax:

```
subroutine MPI_File_read_at_all(fh, offset, buf,
count, type, status, ierr)
<type> BUF(*)
integer fh, offset, count, type
integer status(MPI_STATUS_SIZE), ierr
```

- This is the collective form of `MPI_File_read_at()`.

OSC
Innovations in computing, networking, and education

MPI_File_read_ordered()

- C syntax:

```
int MPI_File_read_ordered(MPI_File fh, void *buf,
int count, MPI_Datatype type, MPI_Status *status);
```

- Fortran syntax:

```
subroutine MPI_File_read_ordered(fh, buf, count,
type, status, ierr)
<type> BUF(*)
integer fh, offset, count, type, status
(MPI_STATUS_SIZE), ierr
```

- This is the collective form of `MPI_File_read_shared()`.

MPI_File_write_all()

- C syntax:

```
int MPI_File_write_all(MPI_File fh, void *buf,
int count, MPI_Datatype type, MPI_Status
*status);
```

- Fortran syntax:

```
subroutine MPI_File_write_all(fh, buf, count,
type, status, ierr)
<type> BUF(*)
integer fh, count, type, status(MPI_STATUS_SIZE),
ierr
```

- This is the collective form of `MPI_File_write()`.

MPI_File_write_at_all()

- C syntax:

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

- Fortran syntax:

```
subroutine MPI_File_write_at_all(fh, offset, buf,
count, type, status, ierr)
<type> BUF(*)
integer fh, offset, count, type
integer status(MPI_STATUS_SIZE), ierr
```

- This is the collective form of `MPI_File_write_at()`.

MPI_File_write_ordered()

- C syntax:

```
int MPI_File_write_ordered(MPI_File fh, void *buf,
int count, MPI_Datatype type, MPI_Status *status);
```

- Fortran syntax:

```
subroutine MPI_File_write_ordered(fh, buf, count,
type, status, ierr)
<type> BUF(*)
integer fh, offset, count, type, status
(MPI_STATUS_SIZE), ierr
```

- This is the collective form of `MPI_File_write_shared()`.

Example: Using Collective I/O

```
! Collective MPI-I/O
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tstart=MPI_WTIME()
      call MPI_FILE_OPEN(MPI_COMM_WORLD,"u.dat",
     +     MPI_MODE_CREATE+MPI_MODE_WRONLY,MPI_INFO_NULL,
     +     outfile,ierr)
      if (jstart.eq.2) jstart=1
      if (jend.eq.(jmax-1)) jend=jmax
      call MPI_FILE_SEEK(outfile,(jstart-1)
   *imax*8,MPI_SEEK_SET,ierr)
      call MPI_FILE_WRITE_ALL(outfile,u(1,jstart),
     +     (jend-jstart+1)*imax,MPI_REAL8,istat,ierr)
      call MPI_FILE_SYNC(outfile,ierr)
      call MPI_FILE_CLOSE(outfile,ierr)
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tend=MPI_WTIME()
      if (rank.eq.0) then
         write(*,*) 'Using collective MPI-I/O'
         write(*,*) 'Wrote ',8*imax*jmax,' bytes in ',tend-tstart,
     +              ' seconds.'
```

OSC
Innovations in computing, networking, and education

Example: Collective I/O (con't)

```
          write(*,*) 'Transfer rate = ',
     +               (8*imax*jmax)/(tend-tstart)/(1024.**2),
     +               ' MB/s'
      endif
```

Split (Non-blocking) Collective I/O

- Unlike collective message passing operations, there can also be non-blocking collective I/O operations. However, they don't work like non-blocking message passing or non-collective I/O
  - Most non-blocking MPI calls generate a request object which can be tested for completion.
  - However, there is no collective equivalent to `MPI_Wait()` or `MPI_Test()`.
- Instead, the non-blocking collective I/O operations are split across two calls:
  - An `MPI_File_*_begin()` call initiates the non-blocking collective I/O operation and returns immediately.
  - Other work not involving the buffer being read or written can then take place.
  - An `MPI_File_*_end()` call then blocks the MPI processes until the operation completes.

MPI_File_read_all_begin()

- C syntax:

```
int MPI_File_read_all_begin(MPI_File fh, void *buf,
int count, MPI_Datatype type);
```

- Fortran syntax:

```
subroutine MPI_File_read_all_begin(fh, buf, count,
type, ierr)
<type> BUF(*)
integer fh, count, type, ierr
```

- This initiates a non-blocking version of `MPI_File_read_all()`.

MPI_File_read_all_end()

- C syntax:

```
int MPI_File_read_all_end(MPI_File fh, void *buf,
MPI_Status *status);
```

- Fortran syntax:

```
subroutine MPI_File_read_all_end(fh, buf, status,
ierr)
<type> BUF(*)
integer fh, ierr
integer status(MPI_STATUS_SIZE)
```

- This completes a non-blocking version of `MPI_File_read_all()`.

OSC
Innovations in computing, networking, and education

MPI_File_read_at_all_begin()

- C syntax:

```
int MPI_File_read_all_begin(MPI_File fh, MPI_Offset
offset, void *buf, int count, MPI_Datatype type);
```

- Fortran syntax:

```
subroutine MPI_File_read_all_begin(fh, offset, buf,
count, type, ierr)
<type> BUF(*)
integer fh, offset, count, type, ierr
```

- This initiates a non-blocking version of `MPI_File_read_at_all()`.

MPI_File_read_at_all_end()

- C syntax:

```
int MPI_File_read_at_all_end(MPI_File fh, void *buf,
MPI_Status *status);
```

- Fortran syntax:

```
subroutine MPI_File_read_at_all_end(fh, buf, status,
ierr)
<type> BUF(*)
integer fh, ierr
integer status(MPI_STATUS_SIZE)
```

- This completes a non-blocking version of `MPI_File_read_at_all()`.

MPI_File_read_ordered_begin()

- C syntax:

```
int MPI_File_read_ordered_begin(MPI_File fh, void
*buf, int count, MPI_Datatype type);
```

- Fortran syntax:

```
subroutine MPI_File_read_ordered_begin(fh, buf,
count, type, ierr)
<type> BUF(*)
integer fh, count, type, ierr
```

- This initiates a non-blocking version of `MPI_File_read_ordered()`.

MPI_File_read_ordered_end()

- C syntax:

```
int MPI_File_read_ordered_end(MPI_File fh, void
*buf, MPI_Status *status);
```

- Fortran syntax:

```
subroutine MPI_File_read_ordered_end(fh, buf,
status, ierr)
<type> BUF(*)
integer fh, ierr
integer status(MPI_STATUS_SIZE)
```

- This completes a non-blocking version of `MPI_File_read_ordered()`.

MPI_File_write_all_begin()

- C syntax:

```
int MPI_File_write_all_begin(MPI_File fh, void *buf,
int count, MPI_Datatype type);
```

- Fortran syntax:

```
subroutine MPI_File_write_all_begin(fh, buf, count,
type, ierr)
<type> BUF(*)
integer fh, count, type, ierr
```

- This initiates a non-blocking version of `MPI_File_write_all()`.

MPI_File_write_all_end()

- C syntax:

```
int MPI_File_read_write_end(MPI_File fh, void *buf,
MPI_Status *status);
```

- Fortran syntax:

```
subroutine MPI_File_write_all_end(fh, buf, status,
ierr)
<type> BUF(*)
integer fh, ierr
integer status(MPI_STATUS_SIZE)
```

- This completes a non-blocking version of `MPI_File_write_all()`.

MPI_File_write_at_all_begin()

- C syntax:

```
int MPI_File_write_all_begin(MPI_File fh, MPI_Offset
offset, void *buf, int count, MPI_Datatype type);
```

- Fortran syntax:

```
subroutine MPI_File_write_all_begin(fh, offset, buf,
count, type, ierr)
<type> BUF(*)
integer fh, offset, count, type, ierr
```

- This initiates a non-blocking version of `MPI_File_write_at_all()`.

MPI_File_write_at_all_end()

- C syntax:

```
int MPI_File_write_at_all_end(MPI_File fh, void
*buf, MPI_Status *status);
```

- Fortran syntax:

```
subroutine MPI_File_write_at_all_end(fh, buf,
status, ierr)
<type> BUF(*)
integer fh, ierr
integer status(MPI_STATUS_SIZE)
```

- This completes a non-blocking version of `MPI_File_write_at_all()`.

MPI_File_write_ordered_begin()

- C syntax:

```
int MPI_File_write_ordered_begin(MPI_File fh, void
*buf, int count, MPI_Datatype type);
```

- Fortran syntax:

```
subroutine MPI_File_write_ordered_begin(fh, buf,
count, type, ierr)
<type> BUF(*)
integer fh, count, type, ierr
```

- This initiates a non-blocking version of `MPI_File_write_ordered()`.

MPI_File_write_ordered_end()

- C syntax:

```
int MPI_File_write_ordered_end(MPI_File fh, void
*buf, MPI_Status *status);
```

- Fortran syntax:

```
subroutine MPI_File_write_ordered_end(fh, buf,
status, ierr)
<type> BUF(*)
integer fh, ierr
integer status(MPI_STATUS_SIZE)
```

- This completes a non-blocking version of `MPI_File_write_ordered()`.

OSC
Innovations in computing, networking, and education

Example: Non-blocking Collective I/O

```
! Non-blocking collective MPI-I/O
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tstart=MPI_WTIME()
      CALL MPI_FILE_OPEN(MPI_COMM_WORLD,"u.dat",
     +     MPI_MODE_CREATE+MPI_MODE_WRONLY,MPI_INFO_NULL,
     +     outfile,ierr)
      if (jstart.eq.2) jstart=1
      if (jend.eq.(jmax-1)) jend=jmax
      call MPI_FILE_SEEK(outfile,(jstart-1)
   *imax*8,MPI_SEEK_SET,ierr)
      call MPI_FILE_WRITE_ALL_BEGIN(outfile,u(1,jstart),
     +     (jend-jstart+1)*imax,MPI_REAL8,ierr)
! Go off and do other stuff not using u(:,:)...
      call MPI_FILE_WRITE_ALL_END(outfile,u(1,jstart),istat,ierr)
      call MPI_FILE_SYNC(outfile,ierr)
      call MPI_FILE_CLOSE(outfile,ierr)
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tend=MPI_WTIME()
      if (rank.eq.0) then
         write(*,*) 'Using non-blocking collective MPI-I/O'
```

OSC
Innovations in computing, networking, and education

Example: Non-blocking Collective I/O (con't)

```
      write(*,*) 'Wrote ',8*imax*jmax,' bytes in ',tend-tstart,
     +           ' seconds.'
      write(*,*) 'Transfer rate = ',
     +           (8*imax*jmax)/(tend-tstart)/(1024.**2),
     +           ' MB/s'
      endif
```

OSC
Innovations in computing, networking, and education

MPI-IO File System Hints

- MPI-2 allows applications to describe their access patterns and preferences to the underlying file system through *file system hints*. These are (keyword,value) pairs stored in an `MPI_Info` object.
- File system hints can include the following:
  - File stripe size
  - Number of I/O nodes used
  - Planned access patterns
  - File system specific hints
- Hints not supported by the MPI implementation or the file system are ignored.
- Alternatively, the null info object (`MPI_INFO_NULL`) can be passed to request the default configuration.

MPI_Info_create()

- C syntax:

```
int MPI_Info_create(MPI_Info *info);
```

- Fortran syntax:

```
subroutine MPI_Info_create(info, ierr)
integer info, ierr
```

- This creates a new `MPI_Info` object containing no (`key`,`value`) pairs.

MPI_Info_set()

- C syntax:

```
int MPI_Info_set(MPI_Info *info, char *key, char
*value);
```

- Fortran syntax:

```
subroutine MPI_Info_set(info, key, value, ierr)
integer info, ierr
character*(*) key, value
```

- This adds (`key`, `value`) to the `MPI_Info` object.
- If `key` had already been added, the previous value is overwritten.

Valid Keys for MPI-IO File Hints

Generic file hints	
Key:	**Value:**

`access_style`	A comma separated list of one or more of:
	`read_once`
	`write_once`
	`read_mostly`
	`write_mostly`
	`sequential`
	`reverse_sequential`
	`random`

`collective_buffering`	`true` or `false`; enables/disables buffering of collective operations
`cb_block_size`	integer block size for collective buffering
`cb_buffer_size`	integer per-node collective buffer size
`cb_nodes`	integer number of nodes for collective buffering

Valid Keys for MPI-IO File Hints (con't)

Generic file hints	
`chunked`	a comma-separated list of integers describing the dimensions of a multi-dimensional array accessed using subarrays, starting with the most significant dimension (the 1st in C, the last in Fortran)
`chunked_item`	a comma-separated list of integers describing the size of each dimension in bytes
`chunked_size`	a comma-separated list of integers describing the dimensions of the subarrays, starting with the most significant dimension (the 1st in C, the last in Fortran)
`file_perm`	file permissions at time of creation
`io_node_list`	a comma-separated list of I/O nodes to use
`nb_proc`	integer number of processes expected to access the file simultaneously

Valid Keys for MPI-IO File Hints (con't)

Generic file hints	
`num_io_nodes`	integer number of I/O nodes/devices on the system
`striping_factor`	integer number of I/O nodes/devices across which the file should be striped
`striping_unit`	integer size of the stripe size used to each I/O node/device
ROMIO file hints	
`ind_rd_buffer_size`	integer size of independent read buffers for data sieving in bytes
`ind_wr_buffer_size`	integer size of independent write buffer for data sieving in bytes
PVFS file hints	
`start_io_device`	integer number of starting I/O node

Valid Keys for MPI-IO File Hints (con't)

SGI XFS file hints	
`direct_read`	`true` or `false`; enables/disables direct I/O for reads
`direct_write`	`true` or `false`; enables/disables direct I/O for writes

OSC
Innovations in computing, networking, and education

MPI_Info_delete()

- C syntax:

```
int MPI_Info_delete(MPI_Info *info, char *key);
```

- Fortran syntax:

```
subroutine MPI_Info_delete(info, key, ierr)
integer info, ierr
character*(*) key
```

- This deletes the (`key`, `value`) pair from the `MPI_Info` object.

MPI_Info_free()

- C syntax:

```
int MPI_Info_free(MPI_Info *info);
```

- Fortran syntax:

```
subroutine MPI_Info_freee(info, ierr)
integer info, ierr
```

- This removes all (`key`, `value`) pairs from the `MPI_Info` object and sets it to `MPI_INFO_NULL`.

MPI_File_set_info()

- C syntax:

```
int MPI_File_set_info(MPI_File fh, MPI_Info info);
```

- Fortran syntax:

```
subroutine MPI_File_set_info(fh, info, ierr)
integer fh, info, ierr
```

- This associates the file system hints in the `MPI_Info` object with the file handle `fh`.
- Note that not all hints can be applied to a file file handle once a file has been created and opened. In most cases it is better to set file hints with either `MPI_File_open()` or `MPI_File_set_view()`.

Querying `MPI_Info` Objects

- In addition to using an `MPI_Info` object to configure the attributes of an `MPI_File` handle, they can also be used to inquire about the current attributes of a file.
- Every `MPI_File` handle has an `MPI_Info` object associated with it. This object can be extracted using `MPI_File_get_info()`, and human-readable information can be gathered from it using a a number of `MPI_Info_*()` routines.

MPI_File_get_info()

- C syntax:

```
int MPI_File_get_info(MPI_File fh, MPI_Info *info);
```

- Fortran syntax:

```
subroutine MPI_File_get_info(fh, info, ierr)
integer fh, info, ierr
```

- This returns a new `MPI_Info` object containing the current file system hints associated with the file handle `fh`.

MPI_Info_get_valuelen()

- C syntax:

```
int MPI_Info_get_valuelen(MPI_Info info, char *key,
int *valuelen, int *flag);
```

- Fortran syntax:

```
subroutine MPI_Info_get_valuelen(info, key,
valuelen, flag, ierr)
integer info, valuelen, ierr
character*(*) key
logical flag
```

- The sets `valuelen` to the length of the value string associated with `key` in the `MPI_Info` object.
- If no such key exists in the `MPI_Info` object, `flag` is set to FALSE; otherwise `flag` is set to TRUE.

MPI_Info_get()

- C syntax:

```
int MPI_Info_get_value(MPI_Info info, char *key, int
valuelen, char *value, int *flag);
```

- Fortran syntax:

```
subroutine MPI_Info_get_valuelen(info, key,
valuelen, value, flag, ierr)
integer info, valuelen, ierr
character*(*) key, value
logical flag
```

- This places the first `valuelen` characters of the value associated with `key` in the `MPI_Info` object into `value`.
- If no such key exists in the `MPI_Info` object, `flag` is set to FALSE; otherwise `flag` is set to TRUE.

MPI_Info_get_nkeys()

- C syntax:

```
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys);
```

- Fortran syntax:

```
subroutine MPI_Info_get_nkeys(info, nkeys, ierr)
integer info, nkeys, ierr
```

- This sets `nkeys` to the number of keys held in the `MPI_Info` object.

MPI_Info_get_nthkey()

- C syntax:

```
int MPI_Info_get_nthkey(MPI_Info info, int n, char
*key);
```

- Fortran syntax:

```
subroutine MPI_Info_get_nthkey(info, n, key, ierr)
integer info, n, ierr
character*(*) key
```

- This sets key to the nth key in the MPI_Info object.
- n can be in the range 0 to nkeys-1.

OSC
Innovations in computing, networking, and education

Example: Getting the Hints from an `MPI_File` Handle

```
MPI_Info_create(&info_out);
MPI_File_get_info(fh,&info_out);
MPI_Info_get_nkeys(info_out,&nkeys);
for (i=0;i<nkeys;i++)
  {
    MPI_Info_get_nthkey(info_out,i,key);
    MPI_Info_get_valuelen(info_out,key,&valuelen,&flag);
    if (flag)
      {
        MPI_Info_get(info_out,key,valuelen,value,&flag);
        printf("[%d/%d]: (%s,%s)\n",
         rank,size,key,value);
      }
    else
      {
        printf("[%d/%d]: MPI_Info_get_valuelen() failed for key
 [%d]=%s\n",
         rank,size,i,key);
      }
  }
```

OSC Innovations in computing, networking, and education

Example: Tying It All Together

```
! Collective MPI-I/O with a file view and hints
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      call MPI_INFO_CREATE(outinfo,ierr)
      call MPI_INFO_SET(outinfo,'access_style','write_once',ierr)
      call MPI_INFO_SET(outinfo,'collective_buffering','true',
     +     ierr)
      call MPI_INFO_SET(outinfo,'file_perm','0600',ierr)
      call MPI_INFO_SET(outinfo,'striping_unit','262144',ierr)
      tstart=MPI_WTIME()
      call MPI_FILE_OPEN(MPI_COMM_WORLD,'u.dat',
     +     MPI_MODE_CREATE+MPI_MODE_WRONLY,outinfo,outfile,ierr)
      if (jstart.eq.2) jstart=1
      if (jend.eq.(jmax-1)) jend=jmax
      call MPI_FILE_SET_VIEW(outfile,(jstart-1)*imax*8,MPI_REAL8,
     +                       MPI_REAL8,'native',outinfo,ierr)
      call MPI_FILE_WRITE_ALL(outfile,u(1,jstart),
     +     (jend-jstart+1)*imax,MPI_REAL8,istat,ierr)
      call MPI_FILE_SYNC(outfile,ierr)
      call MPI_FILE_CLOSE(outfile,ierr)
```

Example: Tying It All Together (con't)

```
      call MPI_BARRIER(MPI_COMM_WORLD,ierr)
      tend=MPI_WTIME()
      if (rank.eq.0) then
         write(*,*) 'Using collective MPI-I/O with a file view and
   ',
     +              'hints'
         write(*,*) 'Wrote ',8*imax*jmax,' bytes in ',tend-tstart,
     +              ' seconds.'
         write(*,*) 'Transfer rate = ',
     +              (8*imax*jmax)/(tend-tstart)/(1024.**2),
     +              ' MB/s'
      endif
```

Comparing the MPI-IO Approaches

Using 8 dual-processor Itanium nodes with Myrinet, writing to a PVFS file system served by 16 dual-processor Pentium IIIs over Gigabit Ethernet

`REAL*8 u(8001,8001)` == approximately 0.5 GB of data

Approach	w/ `MPI_File_sync()`	w/o `MPI_File_sync()`
Basic	53.849 MB/s	73.228 MB/s
With file view	52.048 MB/s	71.883 MB/s
Non-blocking	65.730 MB/s	100-250 MB/s (varies)
Collective	122.306 MB/s	241.583 MB/s
Non-blocking Collective	83.802 MB/s	246.183 MB/s
Collective with hints	122.448 MB/s	258.165 MB/s

OSC
Innovations in computing, networking, and education

Comparing the MPI-IO Approaches

Using 16 processors on a 24-CPU Sun E6800 to local `/tmp`

`REAL*8 u(8001,8001)` == approximately 0.5 GB of data

Approach	w/ `MPI_File_sync()`	w/o `MPI_File_sync()`
Basic	654.859 MB/s	724.524 MB/s
With file view	237.975 MB/s	236.183 MB/s
Non-blocking	2111.533 MB/s	2174.507 MB/s
Collective	Hung	Hung
Non-blocking Collective	2278.810 MB/s	2175.289 MB/s
Collective with hints	Hung	Hung

OSC
Innovations in computing, networking, and education

P2NM (MPI-IO Example)

- Motivation
- PGM and PPM Image Formats
- Data Model
- Header/Metadata Operations
- Raster Data Operations
  - Independent
  - Collective
- Code Recipes
- Performance Results
- Limitations of Current Implementation

P2NM -- Parallel Portable aNyMaps

- Modern digital imaging technologies can easily generate images in the tens or hundreds of megabytes (eg. http://photojournal.jpl.nasa.gov/).
- Processing these very large images can be done in parallel, iff the data can be read and written in parallel.
- Need an API which can be used to read and write image files in some widely-used format serially or in parallel
  - Written in terms of MPI-I/O
  - Independent or collective I/O operations
  - Uses MPI abstractions (data types, file views, etc.) wherever possible

- Available at http://www.osc.edu/~troy/p2nm/.

PGM and PPM File Formats

- PGM (Portable Gray Map) and PPM (Portable PixMap) are relatively simple image formats:
  - Human-readable text header
    - Image type
    - Dimensions
    - Maximum intensity value per sample (determines depth)
  - Uncompressed raster data
- A PGM or PPM file usually looks like the following:
  - P# (5=PGM/grayscale, 6=PPM/RGB
  - Width Height
  - MaxVal (<256 means 8 bits/sample, >256 means 16 bits/sample)
  - Raster data in row-major order

Example PPM File Header

```
amd084% head -4 /pvfs/troy/yucatan.ppm
P6
18001 11438
255
[…raster data…]
```

- This is an 8-bit RGB file with dimensions 18001x11438.
- Run `man pgm` or `man ppm` on oscbw for more information on the file formats.

OSC
Innovations in computing, networking, and education

P2NM Data Model

- Because the header of a PNM file is variable length, it must be treated separately from and before the raster data.
- The handle used to manipulate an image is an `image_t` object. This object includes
  - The file name
  - Overall image dimensions, depth
  - Offsets and extents of the local MPI process' subset of the image, if any
  - A pointer to a buffer for the raster data
- Like MPI file handles, an `image_t` object is defined in the context of an MPI communicator.
- MPI datatypes and file views are use to map image subsets onto processes.

Using P2NM

- P2NM is currently callable from MPI programs written in C or C++.
- Must have `#include <p2nm.h>`
- Must call `p2nm_register_types()` after `MPI_Init()` but before calling any other P2NM routines.
- File handles in P2NM are of type `MPI_File`.
- There are two types of calls in P2NM:
  - Metadata operations (reading and writing headers, memory allocation, and other housekeeping tasks, most of which involve the image header)
  - Raster data operations (reading and writing image data)

Header/Metadata Operations

- `p2nm_register_types()`
- `p2nm_new_image()`
- `p2nm_name_image()`
- `p2nm_read_header()`
- `p2nm_write_header()`
- `p2nm_dup_image()`
- `p2nm_create_subimage()`
- `p2nm_allocate pixbuf()`
- `p2nm_deallocate_pixbuf()`

p2nm_register_types()

```
void p2nm_register_types(void);
```

- This routine must be called after `MPI_Init()` but before any other P2NM routines.
- It registers several MPI data types used internally by the P2NM library.

p2nm_new_image()

```
int p2nm_new_image(image_t *image, int type, int
   width, int height, int maxval);
```

- This routine initializes a new `image_t` object for a previously nonexistent image file. It does not need to be used to access already existing files.
- The `type` argument may have the following values:
  - `PGM_TYPE` (image is grayscale)
  - `PPM_TYPE` (image is RGB color)
- The `maxval` argument may have any integer value between 1 and 65536, but the following are convenient values to use
  - `PNM_MAXVAL_8BIT` (8 bits per sample)
  - `PNM_MAXVAL_16BIT` (16 bits per sample)

p2nm_name_image()

```
int p2nm_name_image(image_t *image, const char
   *name);
```

- This sets the file name associated with `image` to the string `name`.

p2nm_read_header()

```
int p2nm_read_header(image_t *image, MPI_Comm
   comm);
```

- This reads in the header of the file associated with `image`.
- This routine is collective and must be called by all MPI processes belonging to `comm`.

p2nm_write_header()

```
int p2nm_write_header(image_t *image, MPI_Comm
   comm, MPI_Info info);
```

- This writes out the header of the file associated with `image`.
- This routine is collective and must be called by all MPI processes belonging to `comm`.
- The `info` argument may contain MPI file hints for file creation, or simply be `MPI_INFO_NULL`.

p2nm_dup_image()

```
int p2nm_dup_image(image_t image, image_t
   *newimage, int flag);
```

- This creates a duplicate of image in newimage.
- The `flag` argument may have the following values:
  - `P2NM_BUFFER_SHARE` (`image` and `newimage` share the same pixel buffer)
  - P2NM_BUFFER_DUP (`newimage` has a duplicate of `image`'s pixel buffer)

p2nm_create_subimage()

```
int p2nm_create_subimage(image_t image, int xstart,
   int ystart, int width, int height, image_t
   *newimage);
```

- This creates an new `image_t` object `newimage` whose dimensions and name are that of `image` but whose subimage is an area `width` by `height` pixels starting at (`xstart`,`ystart`).
- The `newimage` object can be use to subsample an image or to access portions of the larger image in parallel.
- Note that `newimage` is **NOT** allocated a pixel buffer -- one must be allocated for it using `p2nm_allocate_pixbuf()`.

p2nm_allocate_pixbuf()

```
int p2nm_allocate_pixbuf(image_t *image);
```

- This allocates a pixel buffer large enough to hold the current subimage area associated with `image`.
- By default, the current subimage is the entirety of the image data unless `image` is the result of a call to `p2nm_create_subimage()`.

p2nm_deallocate_pixbuf()

```
int p2nm_deallocate_pixbuf(image_t *image);
```

- This frees the pixel buffer associated with `image`.

Raster Data Operations

- `p2nm_open_image()`
- `p2nm_read_image()`
- `p2nm_write_image()`
- `p2nm_read_image_all()`
- `p2nm_write_image_all()`

p2nm_open_image()

```
int p2nm_open_image(image_t *image, MPI_Comm comm,
   int amode, MPI_Info info, MPI_File *fd);
```

- This attaches an MPI file handle `fd` to the file associated with `image`.
- A file view is also set up on this file handle such as that the local file pointer points to the beginning of the file's raster data and `MPI_File_seek()`s are in terms of the file's pixel type.
- This routine may not be called until **after** a call to either `p2nm_read_header()` or `p2nm_write_header()`.
- This routine is collective and must be called by all MPI processes belonging to `comm`.
- The MPI file handle `fd` may be closed using `MPI_File_close()`.

p2nm_read_image()

```
int p2nm_read_image(image_t *image, MPI_File fd,
   MPI_Status *status);
```

- This reads the pixel data from a file's subimage area into `image`'s pixel buffer.

p2nm_write_image()

```
int p2nm_write_image(image_t *image, MPI_File fd,
   MPI_Status *status);
```

- This writes the pixel data in `image`'s pixel buffer to the subimage area in a file.

p2nm_read_image_all()

```
int p2nm_read_image_all(image_t *image, MPI_File
   fd, MPI_Status *status);
```

- This reads the pixel data from a file's subimage area into `image`'s pixel buffer.
- This routine is collective and must be called by all MPI processes in the communicator where `fd` was opened.
- This routine resets the local file pointer back to the beginning of the file's raster data.

p2nm_write_image_all()

```
int p2nm_write_image_all(image_t *image, MPI_File
   fd, MPI_Status *status);
```

- This writes the pixel data in `image`'s pixel buffer to the subimage area in a file.
- This routine is collective and must be called by all MPI processes in the communicator where `fd` was opened.
- This routine resets the local file pointer back to the beginning of the file's raster data.

Code Recipes

- To read an existing PNM file serially:

```
image_t image;
MPI_File fd;
MPI_Status status;
...
MPI_Init(&argc,&argv);
p2nm_register_types();
...
p2nm_name_image(&image,"foo.ppm");
p2nm_read_header(&image,MPI_COMM_SELF);
p2nm_allocate_pixbuf(&image);
p2nm_open_image(&image,MPI_COMM_SELF,MPI_MODE_RDONLY,
                MPI_INFO_NULL,&fd);
p2nm_read_image(&image,fd,&status);
MPI_File_close(&fd);
```

Code Recipes (con't.)

- To write out an image to a file serially:

```
image_t image;
MPI_File fd;
MPI_Status status;
...
MPI_Init(&argc,&argv);
p2nm_register_types();
...
p2nm_create_image(&image,PPM_TYPE,width,height,maxval);
/* or dup another image object */
p2nm_name_image(&image,"foo.ppm");
/* code to fill in image's pixel buffer */
p2nm_write_header(&image,MPI_COMM_SELF);
p2nm_open_image(&image,MPI_COMM_SELF,MPI_MODE_WRONLY,
                MPI_INFO_NULL,&fd);
p2nm_write_image(&image,fd,&status);
MPI_File_close(&fd);
```

Code Recipes (con't.)

- To read segments of an existing PNM file in parallel:

```
int xstart,ystart,subwidth,subheight;
image_t image,parimage;
MPI_File fd;
MPI_Status status;
...
MPI_Init(&argc,&argv);
p2nm_register_types();
...
p2nm_name_image(&image,"foo.ppm");
p2nm_read_header(&image,MPI_COMM_WORLD);
/* code to compute my values for xstart, ystart, subwidth,
subheight */
p2nm_create_subimage(image,xstart,ystart,subwidth,subheight,
                     &parimage);
p2nm_allocate_pixbuf(&parimage);
p2nm_open_image(&parimage,MPI_COMM_WORLD,
                MPI_MODE_RDONLY,MPI_INFO_NULL,&fd);
p2nm_read_image_all(&parimage,fd,&status);
MPI_File_close(&fd);
```

Code Recipes (con't.)

- To write segments of an image to a PNM file in parallel:

```
int xstart,ystart,subwidth,subheight;
image_t image,parimage;
MPI_File fd;
MPI_Status status;
...
MPI_Init(&argc,&argv);
p2nm_register_types();
...
p2nm_create_image(&image,PPM_TYPE,width,height,maxval);
/* or dup another image object */
p2nm_name_image(&image,"foo.ppm");
p2nm_write_header(&image,MPI_COMM_WORLD);
/* code to compute my values for xstart,ystart,subwidth,subheight */
/* NOTE -- MUST NOT OVERLAP!!!!! */
p2nm_create_subimage(image,xstart,ystart,subwidth,subheight,
                     &parimage);
p2nm_allocate_pixbuf(&parimage);
/* code to fill in pixel buffer */
p2nm_open_image(&parimage,MPI_COMM_WORLD,MPI_MODE_WRONLY,
                MPI_INFO_NULL,&fd);
p2nm_write_image_all(&parimage,fd,&status);
MPI_File_close(&fd);
```

Performance Results

Using 16 dual-processor Athlon nodes with Myrinet, reading and writing a 589MB 8-bit PPM file on a PVFS filesystem served by 16 dual-processor Pentium III I/O servers over Myrinet

Operation	Aggregate Read Performance	Aggregate Write Performance
Simultaneous Individual Access (1 proc/node)	955 MB/s	90 MB/s
Collective Parallel Access (1 proc/node)	1056 MB/s	96 MB/s
Simultaneous Individual Access (2 procs/node)	1088 MB/s	57 MB/s
Collective Parallel Access (2 procs/node)	944 MB/s	111 MB/s

Performance Results (con't)

- In the simultaneous individual access case, each MPI process reads all of a shared image file and then writes out its own copy of it.
- In the collective parallel access case, each MPI process reads a section of a shared image file and then writes that section back to another shared image file.
- Hints required for best collective parallel write performance
  - "`access_style`" set to "`write_once`"
  - "`collective_buffering`" set to "`false`"
  - "`striping_unit`" set to "`262144`" or greater

Limitation of Current Implementation

- The P2NM interface currently has not handle PNM files which have comments in the header.
- The P2NM interface also does not handle PBM (Portable BitMap) images.

Parallel NetCDF

- Concepts
- Parallel Interface
- Differences with original "serial" NetCDF

Introduction to NetCDF

- NetCDF is a file format and support library developed at the National Center for Atmospheric Research (NCAR).
- NetCDF is self-describing and portable across platforms:
  - XDR conversions for data portability
  - Labeling conventions
  - Documented best practices for usage
- NetCDF is not hierarchical, unlike some of its competitors (eg. HDF5):
  - Multiple variables per file, but not directory structures
  - Datasets may be multi-dimensional

- More information is available from http://www.unidata.ucar.edu/packages/netcdf/.

NetCDF in Parallel

- The original NetCDF API from NCAR was not designed with parallel write access in mind and predates the MPI-2 specification which introduced MPI-I/O.
- However, a collaboration between Northwestern University and Argonne National Lab has produced a parallel NetCDF API
  - Significantly different from the serial NetCDF API
  - Produces compatible files
  - Available from http://www-unix.mcs.anl.gov/parallel-netcdf/

Differences with Serial NetCDF

- The serial NetCDF API allows parallel read access to a file, but not parallel write access. Parallel NetCDF allows parallel write access as well.
- Parallel NetCDF uses MPI datatype abstractions in several places (eg. the flexible data mode interface), where serial NetCDF does not. This allows for the use of more complex data structures in memory than serial NetCDF would permit.
- Parallel NetCDF has collective parallel I/O operations, where serial NetCDF does not.

Parallel NetCDF Programming Basics

- Must include appropriate header files
  - C: `#include <pnetcdf.h>`
  - Fortran: `include 'pnetcdf.inc'`
- All routine names begin with `ncmpi_` for the C API or `nfmpi_` for the Fortran API
- Like serial NetCDF, parallel NetCDF has two access "modes"
  - Define mode
    - Metadata and data description operations
    - Always collective
  - Data mode
    - Data transfer (`get`/`put`) operations
    - Block (`vara`) or strided (`vars`) array access
    - Function named by data type transferred in high-level data mode (eg. `ncmpi_get_vara_int()`), or specified as argument in flexible data mode (eg. `ncmpi_get_vara()`)
    - May be independent or collective (`_all` function name suffix for the latter)
    - Non-blocking versions in development

Parallel NetCDF Programming Basics (con't)

- Inquiry and attribute functions
  - May be called during either mode
  - Always collective
- C **and** Fortran data types
  - `short`
  - `int`
  - `float`
  - `double`

Creating a New NetCDF File

- C syntax:

```
int ncmpi_create(MPI_Comm comm, const char *path,
   int cmode, MPI_Info info, int *ncid);
```

- Fortran syntax:

```
subroutine nfmpi_create(comm, path, cmode, info,
   ncid)
integer comm,cmode,info,ncid
character*(*) path
```

- `ncid` is a NetCDF file handle which will be used by other parallel NetCDF routines at reference a particular file.

Opening an Existing NetCDF File

- C syntax:

```
int ncmpi_open(MPI_Comm comm, const char *path, int
   omode, MPI_Info info, int *ncidp);
```

- Fortran syntax:

```
subroutine nfmpi_open(comm, path, cmode, info,
   ncid)
integer comm,omode,info,ncid
character*(*) path
```

File Mode Flags for Creating or Opening Files

The `cmode` and `omode` arguments to the file create and open functions can be the sum of one or more of the following:

- `NC_NOWRITE/NF_NOWRITE` (default)
- `NC_WRITE/NF_WRITE`
- `NC_CLOBBER/NF_WRITE` (default)
- `NC_NOCLOBBER/NF_NOCLOBBER`
- `NC_FILL/NF_FILL` (default)
- `NC_NOFILL/NF_NOFILL`
- `NC_LOCK/NF_LOCK`
- `NC_SHARE/NF_SHARE`

Define Mode

- When a NetCDF file is first created or opened, the handle associated with it is in *define mode*.
- In define mode, the number and names of dimensions and variables in a NetCDF file can be set or reconfigured.
- Define mode ends with a call to `ncmpi_enddef()`, but can be reentered with a call to `ncmpi_redef()`.
- Define mode routines include:
  - `ncmpi_def_dim()`
  - `ncmpi_def_var()`

Defining a NetCDF Dimension

- C syntax

```
int ncmpi_def_dim(int ncid, const char *name,
   MPI_Offset len, int *dimid);
```

- Fortran syntax

```
subroutine nfmpi_def_dim(ncid, name, len, dimid)
integer ncid,len,dimid
char*(*) name
```

- `dimid` is a handle to the dimension created. In particular, it is used when creating a variable in a NetCDF file.

Defining a NetCDF Variable

- C syntax

```
int ncmpi_def_var(int ncid, const char *name,
   nc_type xtype, int ndims, const int *dimid, int
   *varid);
```

- Fortran syntax

```
subroutine nfmpi_def_var(ncid, name, xtype, ndims,
   dimid, varid)
integer ncid,xtype,ndims,varid
integer dimid(ndims)
char*(*) name
```

- `varid` is a handle to the variable created.
- `dimid` is an array of dimension handles.

Defining a Variable (con't)

- `xtype` can be one of the following:
  - `NC_BYTE/NF_BYTE` (signed 8-bit integer)
  - `NC_CHAR/NF_BYTE` (ASCII character)
  - `NC_SHORT/NF_SHORT` (signed 16-bit integer)
  - `NC_INT/NF_INT` (signed 32-bit integer)
  - `NC_FLOAT/NF_FLOAT` (32-bit IEEE floating point number)
  - `NC_DOUBLE/NF_DOUBLE` (64-bit IEEE floaring point number)

Exiting Define Mode

- C syntax:

```
int ncmpi_enddef(int ncid);
```

- Fortran syntax:

```
subroutune nfmpi_enddef(ncid)
integer ncid
```

Data Mode

- Once `ncmpi_enddef()` is called on a NetCDF handle, that handle is considered to be in *data mode*. This means that data can be read from or written to the file associated with that handle.
- There are two ways of how the data is stored within the file:
  - Block (`vara`), where the stride between elements along each dimension is 1.
  - Strided (`vars`), where the stride between elements along each dimension is specified.
- There are also two different interfaces within data mode:
  - High level, where the type of data being transferred is encoded in the function name (eg. `ncmpi_put_vara_int()`)
  - Flexible, where the type of data being transferred is passed as an argument of type `MPI_Datatype` (eg. `ncmpi_put_vara()`)
- For the sake of brevity, we will look only at the block and strided access routines of the flexible data interface.

Block Read Access in Flexible Data Mode

- C syntax:

```
int ncmpi_get_vara(int ncid, int varid, const
   MPI_Offset start[], const MPI_Offset count[],
   void *buf, int bufcount, MPI_Datatype datatype);
```

- Fortran syntax:

```
subroutine nfmpi_get_vara(ncid, varid, start,
   count, buf, bufcount, datatype)
integer ncid, varid, bufcount, datatype
integer start(*), count(*)
<type> buf(*)
```

- `start` and `count` are arrays of length equal to the number of dimensions in the variable referred to by `varid`.

Block Read Access in Flexible Data Mode (con't)

- `start` represents the starting indices along each dimension for the data block being read from the file.
- `count` is the number of elements along each dimension for the data block being read from the file.
- `bufcount` is the number of elements to be read into `buf`, which is typically the product of the elements in `count`.
- `datatype` is the MPI datatype of `buf`.

Block Write Access in Flexible Data Mode

- C syntax:

```
int ncmpi_put_vara(int ncid, int varid, const
   MPI_Offset start[], const MPI_Offset count[], void
   *buf, int bufcount, MPI_Datatype datatype);
```

- Fortran syntax:

```
subroutine nfmpi_put_vara(ncid, varid, start, count,
   buf, bufcount, datatype)
integer ncid, varid, bufcount, datatype
integer start(*), count(*)
<type> buf(*)
```

- The arguments here are identical in meaning to those in `ncmpi_geta_vara()`, except that data is being written instead of read.

Strided Read Access in Flexible Data Mode

- C syntax:

```
int ncmpi_get_vars(int ncid, int varid, const
   MPI_Offset start[], const MPI_Offset count[], const
   MPI_Offset stride[], void *buf, int bufcount,
   MPI_Datatype datatype);
```

- Fortran syntax:

```
subroutine nfmpi_get_vars(ncid, varid, start, count,
   stride, buf, bufcount, datatype)
integer ncid, varid, bufcount, datatype
integer start(*), count(*), stride(*)
<type> buf(*)
```

- `stride` is an array of length equal to the number of dimensions in the variable referred to by `varid`. It is the number of elements to skip between elements read along each dimension.

Strided Write Access in Flexible Data Mode

- C syntax:

```
int ncmpi_put_vars(int ncid, int varid, const
   MPI_Offset start[], const MPI_Offset count[], const
   MPI_Offset stride[], void *buf, int bufcount,
   MPI_Datatype datatype);
```

- Fortran syntax:

```
subroutine nfmpi_put_vars(ncid, varid, start, count,
   stride, buf, bufcount, datatype)
integer ncid, varid, bufcount, datatype
integer start(*), count(*), stride(*)
<type> buf(*)
```

- The arguments here are identical in meaning to those in `ncmpi_geta_vara()`, except that data is being written instead of read.

Collective I/O in Flexible Data Mode

- The preceeding flexible data mode routines are implemented in terms of independent MPI-I/O routines.
- The collective versions of these routines are very similar
  - Routine names end with `_all`
  - Same argument lists.
  - Must be called as collective operations.

Example: Using Parallel NetCDF

```
! Parallel NetCDF
      call MPI_Info_create(outinfo,ierr)
      call MPI_Info_set(outinfo,'access_style',
     +                  'write_once',ierr)
      call MPI_Info_set(outinfo,'collective_buffering',
     +                  'true',ierr)
      call MPI_Info_set(outinfo,'file_perm',
     +                  '0600',ierr)
      call MPI_Info_set(outinfo,'striping_unit',
     +                  '262144',ierr)
      call MPI_Type_contiguous(imax,MPI_REAL8,
     +                         coltype,ierr)
      call MPI_Type_commit(coltype,ierr)
      call MPI_Barrier(MPI_COMM_WORLD,ierr)
      tstart=MPI_Wtime()
```

Example: Paralle NetCDF (con't)

```
 call NFMPI_CREATE(MPI_COMM_WORLD,"laplace.nc",
+                  NF_WRITE+NF_CLOBBER,outinfo,
+                  ncid)
 call NFMPI_DEF_DIM(ncid,"i",imax,dimid(1))
 call NFMPI_DEF_DIM(ncid,"j",jmax,dimid(2))
 call NFMPI_DEF_VAR(ncid,"u",NF_DOUBLE,2,dimid,
+                  varid)
 call NFMPI_ENDDEF(ncid)
 if (jstart.eq.2) jstart=1
 if (jend.eq.(jmax-1)) jend=jmax
 start(1)=1
 start(2)=jstart
 count(1)=imax
 count(2)=jend-jstart+1
```

Example: Parallel NetCDF (con't)

```
 call NFMPI_PUT_VARA_ALL(ncid,varid,start,count,
+                        u(1,jstart),
+                        (jend-jstart+1),
+                        coltype)
 call NFMPI_CLOSE(ncid)
 call MPI_Barrier(MPI_COMM_WORLD,ierr)
 tend=MPI_Wtime()
 if (rank.eq.0) then
    write(*,*) 'Using collective parallel NetCDF'
    write(*,*) 'Wrote ',8*imax*jmax,' bytes of data in ',
+               tend-tstart,' seconds.'
    write(*,*) 'Transfer rate = ',
+               (8*imax*jmax)/(tend-tstart)/(1024.**2),
+               ' MB/s'
 endif
```

OSC
Innovations in computing, networking, and education

Inquiry and Attribute Routines

- It is possible to inquire the state of most of the metadata associated with a NetCDF file (dimensions, variables, etc.) during either define or data mode.
- It is also possible to assign or view named *attributes* of a variable (eg. annotations) during either define or data mode.
  - Attributes are (keyword,value) pairs
  - Values can be in any valid NetCDF data type
- Inquiry and attribute routines are always collective!

Inquiring on a NetCDF File

- C syntax:

```
int ncmpi_inq(int ncid, int *ndims, int *nvars,
   natts, int *unlimdimid)
```

- Fortran syntax:

```
subroutine nfmpi_inq(ncid, ndims, nvars, natts,
   unlimdimid)
integer ncid, ndims, nvars, natts, unlimdimid
```

- `ndims` is the number of dimensions defined for `ncid`
- `nvars` is the number of variables defined for `ncid`
- `natts` is the number of attributes defined for `ncid`
- `unlimdimid` is the dimension id for the "unlimited" dimension for `ncid`.

Finding the ID of a Named Dimension

- C syntax:

```
int ncmpi_inq_dimid(int ncid, const char *name, int
   *dimid);
```

- Fortran syntax:

```
subroutine nfmpi_inq_dimid(ncid, name, dimid)
integer ncid,dimid
character*(*) name
```

- `dimid` is returned as the dimension ID in `ncid` which is named `name`.

Inquiring on a Dimension

- C syntax:

```
int ncmpi_inq_dim(int ncid, int dimid, char *name,
   MPI_Offset *len);
```

- Fortran syntax:

```
subroutine nfmpi_inq_dim(ncid, dimid, name, len)
integer ncid, dimid, len
character*(*) name
```

- The name of the dimension associated with `dimid` is returned in `name`.
- The length of the dimension associated with `dimid` is returned in `len`.

Finding the ID of a Named Variable

- C syntax:

```
int ncmpi_inq_varid(int ncid, const char *name, int
   *varid);
```

- Fortran syntax:

```
subroutine nfmpi_inq_varid(ncid, name, varid)
integer ncid, varid
character*(*) name
```

- `varid` is returned as the variable ID in `ncid` which is named `name`.

Inquiring on a Variable

- C syntax:

```
int ncmpi_inq_var(int ncid, int varid, char *name,
   nc_type xtype, int *ndims, int *dimid, int
   *natts);
```

- Fortran syntax:

```
subroutine nfmpi_inq_var(ncid, varid, name, xtype,
   ndims, dimid, natts)
integer ncid, varid, xtype, ndims, dimid(*), natts
character*(*) name
```

- `name` is the name string for `varid`.
- `xtype` is the NetCDF datatype for `varid`.

Inquiring on a Variable (con't)

- `ndims` is the number of dimensions in `varid`.
- `dimid` is an array of dimension IDs for `varid`. It must be at least of length `ndims`.
- `natts` is the number of attributes assigned to `varid`.

Setting a Text Attribute on a Variable

- C syntax:

```
int ncmpi_put_att_text(int ncid, int varid, const
   char *key, MPI_Offset len, const char *value);
```

- Fortran syntax:

```
subroutine ncmpif_put_att_text(ncid, varid, key,
   len, value)
integer ncid, varid, len
character*(*) key, value
```

- This assigns the attribute associated with `key` to the text string `value`.

Setting a Integer Attribute on a Variable

- C syntax:

```
int ncmpi_put_att_int(int ncid, int varid, const
   char *key, nc_type xtype, MPI_Offset len, const
   int *value);
```

- Fortran syntax:

```
subroutine ncmpif_put_att_int(ncid, varid, key,
   xtype, len, value)
integer ncid, varid, len, value(*)
character*(*) key
```

- This assigns the attribute associated with `key` to the integer array `value`, which is of length `len`.
- Note that unlike the previous routine, this routine has an `xtype` argument to specify the NetCDF datatype.

Setting a Floating Point Attribute on a Variable

- C syntax:

```
int ncmpi_put_att_float(int ncid, int varid, const
   char *key, nc_type xtype, MPI_Offset len, const
   float *value);
```

- Fortran syntax:

```
subroutine ncmpif_put_att_float(ncid, varid, key,
   xtype, len, value)
integer ncid, varid, len
real value(*)
character*(*) key
```

- This assigns the attribute associated with `key` to the floating point array `value`, which is of length `len`.
- Note that unlike the text attribute routine, this routine has an `xtype` argument to specify the NetCDF datatype.

Parallel NetCDF Topics Not Covered in This Workshop

- The high level interface in data mode is not covered here; however, the general pattern of usage of it in C is:

```
ncmpi_<op>_vara_<type>[_all](ncid, varid, start[],
   count[], buf); /* Block */
ncmpi_<op>_vars_<type>[_all](ncid, varid, start[],
   count[], stride[], buf); /* Strided */
```

  - <*op*> == `get` or `put`
  - <*type*> == `char`, `double`, `float`, `int`, or `short`
  - The optional `_all` suffix indicates a collective operation
- There are several more inquiry and attribute routines than what are shown here.

Limitations of Current Parallel NetCDF Implementation

- This API is under active development and feedback from real-world users is desired.
- There are currently no non-blocking versions of the data mode operations, but it should be possible to implement them in terms of non-blocking MPI-I/O routines.

HDF5

- Concepts
- HDF5 Basics
- High Level Interface
- Parallel Interface

An Introduction to HDF5

- HDF5 is a file format and support library developed by the National Center for Supercomputing Applications (NCSA).
- HDF5 files are self describing collections of data objects.
- HDF5 is a bit different from previous HDF releases:
  - It supports file greater than 2 GB in size, even on 32-bit platforms (as long as they conform to the LFS conventions).
  - It has a much simpler set of objects, consisting of
    - Multidimensional arrays of data elements
    - Grouping objects
  - It has support for threading and parallel I/O using MPI-IO.

- More information is available at http://hdf.ncsa.uiuc.edu/HDF5/.

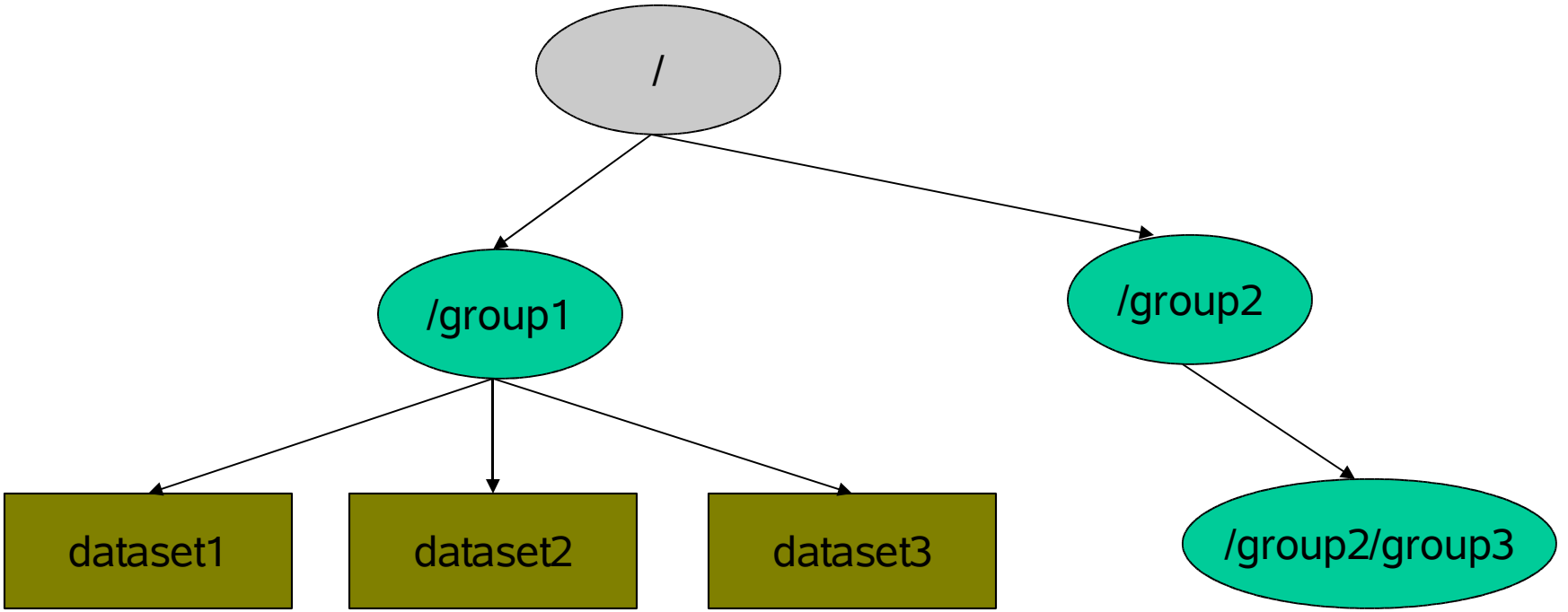
HDF5 Terminology

- An HDF5 file is similar to a hierarchical UNIX-style file system, contained in a single file.
- An *HDF5 group* is an object containing zero or more HDF5 objects. All HDF5 files have a base group called the root (`/`). An HDF5 group object consists of
  - A group header, containing the group name and attributes.
  - A symbol table, a list of object which the group contains.
- An *HDF5 dataset* is an object containing data. It consists of
  - A dataset header, containing metadata describing the data array as well as annoations.
  - A data array, which may be multidimensional.
- Any HDF5 object may also have an *attribute list*. This is a list of `(key,value)` pairs which describes the object to which the attribute list is attached.

HDF5 Terminology (con't)

- When creating new datasets in a file, the layout of the data in the file in terms of dimensionality and size along each dimension is needed. This is represented by a *dataspace* object which can reused by multiple datasets, even if they are of different datatypes.
- Within a dataspace, subsets may be selected when non-contiguous data access is required. These dataspace subsets are referred to as *hyperslabs*.
- An *property list* is an object which contains properties of an HDF5 file, including:
  - On-disk layout.
  - Chunking sizes.
  - Filters.
  - A list of external HDF5 files to be "mounted" as part of the HDF5 file system within a file.
  - Interactions with underlying I/O systems (eg. MPI-IO)

Structure of an HDF5 File



HDF5 Programming Basics

- In the C interface to HDF5, all routines are named `H5Xname()`. `X` can be one of the following:
  - (none) -- general routines
  - `A` -- attribute handling functions
  - `D` -- dataset functions
  - `E` -- error handling
  - `F` -- file-level access routines
  - `G` -- group functions
  - `I` -- identifier functions
  - `P` -- property list functions
  - `R` -- reference routines
  - `S` -- dataspace functions
  - `T` -- data type functions
  - `Z` -- compression routines

HDF5 Programming Basics (con't)

- In the Fortran 90 interface, the routine names **and** symbolic constants are the same except with an added trailing `_F`
  - The F90 interface uses a module interface for argument checking, so the code needed to have a `USE HDF5` statement at the beginning of any routines calling the HDF5 library.
  - As with the Fortran interface to MPI, the HDF5 Fortran interface consists of subroutines rather than functions, and there is an additional `ierr` argument at the end of each routine's argument list which corresponds to the return value of the function in the C interface.
- Many HDF5 routines have optional parameters. In the C interface, `NULL` can be passed as the value to an optional parameter to get the default value. In the F90 interface, the standard rules for optional arguments apply.

HDF5 Elemental Data Types

HDF5 Datatype	C Variable Type	F90 Variable Type
H5T_NATIVE_CHAR	signed char	N/A
H5T_NATIVE_CHARACTER	N/A	CHARACTER
H5T_NATIVE_INT	int	N/A
H5T_NATIVE_INTEGER	N/A	INTEGER
H5T_NATIVE_FLOAT	float	N/A
H5T_NATIVE_REAL	N/A	REAL
H5T_NATIVE_DOUBLE	double	DOUBLE PRECISION
H5T_NATIVE_HSIZE	hsize_t	INTEGER(kind=HSIZE_T)
H5T_NATIVE_HSSIZE	hssize_t	INTEGER(kind=HSSIZE_T)
H5T_NATIVE_HERR	herr_t	INTEGER(kind=HERR_T)
H5T_NATIVE_HBOOL	hbool_t	LOGICAL

HDF5 Data Representations

HDF5 Datatype	Representation
H5T_IEEE_F32BE	IEEE 32-bit floating point, big-endian
H5T_IEEE_F32LE	IEEE 32-bit floating point, little-endian
H5T_IEEE_F64BE	IEEE 64-bit floating point, big-endian
H5T_IEEE_F64LE	IEEE 64-bit floating point, little-endian
H5T_STD_I32BE	32-bit signed integer, big-endian
H5T_STD_I32LE	32-bit signed integer, little-endian
H5T_STD_I64BE	64-bit signed integer, big-endian
H5T_STD_I64LE	64-bit signed integer, little-endian
H5T_C_S1	C-style string
H5T_FORTRAN_S1	Fortran-style string

For more info, see http://hdf.ncsa.uiuc.edu/HDF5/doc/PredefDTypes.html

Creating an HDF5 File

- C syntax:

```
hid_t H5Fcreate(const char *name, uintn flags, hid_t
create_prp, hid_t access_prp);
```

- Fortran syntax:

```
subroutine H5Fcreate_F(name, flags, file_id, ierr,
create_prp, access_prp)
character(len=*) :: name
integer :: flags
integer(kind=HID_T) :: file_id
integer :: ierr
integer(kind=HID_T),optional :: create_prp
integer(kind=HID_T),optional :: access_prp
```

Creating an HDF5 File (con't)

- `flags` can have the following values:
  - `H5F_ACC_RDWR`
  - `H5F_ACC_RDONLY`
  - `H5F_ACC_TRUNC`
  - `H5F_ACC_EXCL`

Opening an Existing HDF5 File

- C syntax:

```
hid_t H5Fopen(const char *name, unsigned flags,
hid_t access_prp);
```

- Fortran syntax:

```
subroutine H5Fopen_F(name, flags, file_id, ierr,
access_prp)
character(len=*) :: name
integer :: flags
integer(kind=HID_T) :: file_id
integer :: ierr
integer(kind=HID_T) :: access_prp
```

- `flags` may be one of the following:
  - `H5F_ACC_RDWR`
  - `H5F_ACC_RDONLY`

Creating a Simple Multidimensional Dataspace

- C syntax:

```
hid_t H5Screate_simple(int rank, const hsize_t
*dims, const hsize_t *maxdims);
```

- Fortran syntax:

```
subroutine H5Screate_simple_F(rank, dims, space_id,
ierr, maxdims)
integer :: rank
integer(kind=HSIZE_T) :: dims(7)
integer(kind=HID_T) :: space_id
integer :: ierr
integer(kind=HSIZE_T),optional :: maxdims(7)
```

- `rank` is the dimensionality of the dataspace.
- `dims` is the size of the dataspace along each dimension.

Creating a Dataset in an HDF5 File

- C syntax:

```
hid_t H5Dcreate(hid_t loc_id, const char *name,
hid_t type_id, hid_t space_id, hid_t create_prp);
```

- Fortran syntax:

```
subroutine H5Dcreate_F(loc_id, name, type_id,
space_id, dset_id, ierr, create_prp)
integer(kind=HID_T) :: loc_id
character(len=*) :: name
integer(kind=HID_T) :: type_id, space_id, dset_id
integer :: ierr
integer(kind=HID_T),optional :: create_prp
```

- `type_id` is the datatype of the data stored in the dataset.
- `space_id` is the dataspace of the data stored in the dataset.
- `loc_id` may be a handle to either a file or a group within a file.

Opening an Existing Dataset in an HDF5 File

- C syntax:

```
hid_t H5Dopen(hid_t loc_id, const char *name );
```

- Fortran syntax:

```
subroutine H5Dopen_F(loc_id, name, dset_id, ierr)
integer(kind=HID_T) :: loc_id
character(len=*) :: name
integer(kind=HID_T) :: dset_id
integer :: ierr
```

- As with `H5Dcreate()`, `loc_id` may be a handle to either a file or a group within a file.

Reading an HDF5 Dataset

- C syntax:

```
herr_t H5Dread(hid_t dset_id, hid_t mem_type_id,
hid_t mem_space_id, hid_t file_space_id, hid_t
xfer_prp, const void *buf);
```

- Fortran syntax:

```
subroutine H5Dread_F(dset_id, mem_type_id, buf, n,
ierr, mem_space_id, file_space_id, xfer_prp)
integer(kind=HID_T) :: dset_id, mem_type_id
<type> :: buf(*)
integer :: n
integer(kind=HID_T),optional :: mem_space_id, &
 file_space_id, xfer_prp
```

Writing an HDF5 Dataset

- C syntax:

```
herr_t H5Dwrite(hid_t dset_id, hid_t mem_type_id,
hid_t mem_space_id, hid_t file_space_id, hid_t
xfer_prp, const void *buf);
```

- Fortran syntax:

```
subroutine H5Dwrite_F(dset_id, mem_type_id, buf,
dims, ierr, mem_space_id, file_space_id, xfer_prp)
integer(kind=HID_T) :: dset_id, mem_type_id
<type> :: buf(*)
integer :: dims(*)
integer(kind=HID_T),optional :: mem_space_id, &
 file_space_id, xfer_prp
```

Dataspace Arguments in `H5Dread()` and `H5Dwrite()`

- `H5Dread()` and `H5Dwrite()` have two optional dataspace arguments:
  - `mem_space_id`
  - `file_space_id`
- If used, these dataspaces describe how the data being written is layed out in memory (`mem_space_id`) versus how it should be layed out on disk (`file_space_id`). These are not required to be the same; for example, consider a distributed array in C, where the index ranges in memory do not correspond to their locations in the overall dataset.
- Care should be taken not to make the memory and file dataspaces too different (eg. transposing columns vs. rows, or rearranging data so that all the data values for a given point are stored together), as this can have a significant impact on performance. See the paper by Ross, et al. from SC2001 cited in the References for an example of this.

Example: Creating and Writing a Dataset

```
      integer(kind=HID_T) h5file,dspace,dset
      integer(kind=HSIZE_T) dims(7)
      …
! HDF5 I/O
      call H5open_F(ierr)
      tstart=MPI_WTIME()
      call H5Fcreate_F('laplace.h5',H5F_ACC_TRUNC_F,h5file,ierr)
      dims(1)=imax
      dims(2)=jmax
      call H5Screate_simple_F(2,dims,dspace,ierr)
      call H5Dcreate_F(h5file,"u",H5T_IEEE_F64BE,dspace,dset,ierr)
      call H5Dwrite_F(dset,H5T_NATIVE_DOUBLE,u,dims,ierr)
      call H5Dclose_F(dset,ierr)
      call H5Sclose_F(dspace,ierr)
      call H5Fclose_F(h5file,ierr)
      tend=MPI_WTIME()
```

Example: Creating and Writing a Dataset (con't)

```
 write(*,*) 'Using serial HDF5'
 write(*,*) 'Wrote ',8*imax*jmax,' bytes in ',tend-tstart,
+     ' seconds.'
 write(*,*) 'Transfer rate = ',
+     (8*imax*jmax)/(tend-tstart)/(1024.**2),
+     ' MB/s'
```

Retrieving a Existing Dataset's Dataspace

- C syntax:

```
hid_t H5Dget_space(hid_t dataset_id );
```

- Fortran syntax:

```
subroutine H5Dget_space_F(dataset_id, dataspace_id,
ierr)
integer(kind=HID_T) :: dataset_id, dataspace_id
integer :: ierr
```

Selecting Points for a Hyperslab

- C syntax:

```
herr_t H5Sselect_hyperslab(hid_t space_id,
H5S_seloper_t operator, const hssize_t *start, const
hsize_t *stride, const hsize_t *count, const hsize_t
*block);
```

- Fortran syntax:

```
subroutine H5Sselect_hyperslab_F(space_id, operator,
start, count, ierr, stride, block)
integer(kind=HID_T) :: space_id
integer :: operator
integer(kind=HSSIZE_T) :: start(*)
integer(kind=HSIZE_T) :: count(*)
integer :: ierr
integer(kind=HSIZE_T),optional :: stride, block
```

Selecting Points for a Hyperslab (con't)

- `operator` determines how the new hyperslab selection is to be combined with any existing selections. The possible values are:
  - `H5S_SELECT_SET` (replace any previous selections)
  - `H5S_SELECT_OR` (add to any previous selections)
- `start`, `count`, `stride` and `block` must all be the same dimensionality as the dataspace.
- `start` determines the starting position of the hyperslab relative to the beginning of the dataset.
- `count` determines how many blocks to select along each dimension of the dataspace.
- `stride` determines the distance between selected blocks along each dimension. The default if none is specified is one.
- `block` specifies how many elements are in each selected block along each dimension. The default if none is specified is one.

Creating a Property List

- C syntax:

```
hid_t H5Pcreate(H5P_class_t classtype);
```

- Fortran syntax:

```
subroutine H5Pcreate_F(classtype, prp_id, ierr)
integer :: classtype
integer(kind=HID_T) :: prp_id
integer :: ierr
```

- `classtype` may be one of the following:
  - `H5P_FILE_CREATE`
  - `H5P_FILE_ACCESS`
  - `H5P_DATASET_CREATE`
  - `H5P_DATASET_XFER`
  - `H5P_MOUNT`

OSC
Innovations in computing, networking, and education

HDF5 Parallel Interface

- The steps required to access a file in parallel through HDF5 are an extension of the serial interface:
  - Create an file creation or access property list and configure it to use MPI-IO.
  - Create or open the HDF5 file using that property list.
  - Create a dataset creation or access property list and set its data transfer mode to independent or collective I/O.
  - Create or open a dataset in the file
  - Create or extract the dataset's dataspace and create hyperslabs as needed
  - Create a dataspace that describes the data's layout in memory.
  - Read or write data.
- HDF5 parallel calls should be treated as collectives, except for `H5Dread()`s and `H5Dwrite()`s to datasets set to independent I/O mode.

Configuring MPI-IO File Access

- C syntax:

```
herr_t H5Pset_fapl_mpio(hid_t plist_id, MPI_Comm
comm, MPI_Info info);
```

- Fortran syntax:

```
subroutine H5Pset_fapl_mpio_F(plist_id, comm, info,
ierr)
integer(kind=HID_T) :: plist_id
integer :: comm, info, ierr
```

- `comm` is an MPI communicator; the current implementation assume this will be `MPI_COMM_WORLD`.
- `info` may be either an `MPI_Info` object containing file hints or `MPI_INFO_NULL`.

Configuring MPI-IO Dataset Access

- C syntax:

```
herr_t H5Pset_dxpl_mpio(hid_t prp_id,
H5FD_mpio_xfer_t xfer_mode);
```

- Fortran syntax:

```
subroutine H5Pset_dxpl_mpio_F(prp_id, xfer_mode,
ierr)
integer(kind=HID_T) :: prp_id
integer :: xfer_mode, ierr
```

- Valid values for `xfer_mode` are:
  - `H5FD_MPIO_INDEPENDENT`
  - `H5FD_MPIO_COLLECTIVE`

Example: Parallel HDF5

```
! Collective PHDF5 with hints
      call MPI_Info_create(outinfo,ierr)
      call MPI_Info_set(outinfo,'access_style','write_once',ierr)
      call MPI_Info_set(outinfo,'collective_buffering','true',
     +     ierr)
      call MPI_Info_set(outinfo,'file_perm','0600',ierr)
      call MPI_Info_set(outinfo,'striping_unit','262144',ierr)
      call H5Pcreate_F(H5P_FILE_ACCESS_F,fprp,ierr)
      call H5Pset_fapl_mpio_F(fprp,MPI_COMM_WORLD,outinfo,ierr)
      call H5Pcreate_F(H5P_DATASET_ACCESS_F,dprp,ierr)
      call H5Pset_dxpl_mpio_F(dprp,H5FD_MPIO_COLLECTIVE_F,ierr)
      if (jstart.eq.2) jstart=1
      if (jend.eq.(jmax-1)) jend=jmax
      call MPI_Barrier(MPI_COMM_WORLD,ierr)
      tstart=MPI_Wtime()
      call H5Fcreate_F('laplace.h5',H5F_ACC_TRUNC_F,h5file,ierr,
     +     access_prp=fprp)
```

Examples: Parallel HDF5 (con't)

```
      dims(1)=imax
      dims(2)=jmax
      call H5Screate_simple_F(2,dims,dspace,ierr)
      call H5Dcreate_F(h5file,"u",H5T_IEEE_F64BE,dspace,dset,ierr)
      call H5Sclose_F(dspace,ierr)
      call H5Dget_space_F(dset,dspace,ierr)
      hstart(1)=1
      hstart(2)=jstart
      hcount(1)=imax
      hcount(2)=jend-jstart+1
      call H5Sselect_hyperslab_F(dspace,H5S_SELECT_SET_F,hstart,
     +     hcount,ierr)
      dims(1)=imax
      dims(2)=jend-jstart+1
      call H5Screate_simple_F(2,dims,memspace,ierr)
      call H5Dwrite_F(dset,H5T_NATIVE_DOUBLE,u(1,jstart),dims,ierr,
     +
  file_space_id=dspace,mem_space_id=memspace,xfer_prp=dprp)
      call H5Dclose_F(dset,ierr)
```

Examples: Parallel HDF5 (con't)

```
      call H5Sclose_F(dspace,ierr)
      call H5Sclose_F(memspace,ierr)
      call H5Fclose_F(h5file,ierr)
      call MPI_Barrier(MPI_COMM_WORLD,ierr)
      tend=MPI_Wtime()
      call H5Pclose_F(dprp,ierr)
      call H5Pclose_F(fprp,ierr)
      if (rank.eq.0) then
         write(*,*) 'Using collective parallel HDF5'
         write(*,*) 'Wrote ',8*imax*jmax,' bytes of data in ',
     +              tend-tstart,' seconds.'
         write(*,*) 'Transfer rate = ',
     +              (8*imax*jmax)/(tend-tstart)/(1024.**2),
     +              ' MB/s'
      endif
```

HDF5 High Level Interfaces

- The HDF5 library currently contains 3 more abstract interfaces:
  - `H5LT` -- an simplified "HDF5 lite" interface which does not support groups or property lists.
  - `H5IM` -- an HDF5 interface for manipulating images, built atop `H5LT`.
  - `H5TB` -- an HDF5 interface for database-like tables of fixed-length records.
- All of these higher-level interfaces currently have C bindings only.

HDF5 Topics Not Covered in the Workshop

There are a number of additional topics on HDF5 not covered in this workshop for the sake of brevity, including:

- Using groups of organize datasets within a file.
- Modifying the attribute lists of groups and datasets.
- Creating and using compound datatypes.
- Identifiers and references.
- Setting compression filters.
- Error handling.

Problems with the Current HDF5 Implementation

- The parallel I/O interface is not supported on some platforms.
- The naming conventions and ordering of argument lists is not consistent between the C and F90 interfaces.

References

- Buyya, ed. *High Performance Cluster Computing: Volume 1, Architectures and Systems*. Prentice Hall, 1999.
- Ellis, Phillips, and Lahey. *Fortran 90 Programming*. Addison Wesley, 1994.
- Gropp, et al. *MPI -- The Complete Reference: Volume 2, the MPI Extensions*. MIT Press, 1998.
- Gropp, Lusk, and Thakur. *Using MPI-2*. MIT Press, 1999.
- Latham, et al. "A Parallel API for Creating and Reading NetCDF Files".
- Lewine. *POSIX Programmer's Guide*. O'Reilly & Associates, 1991.
- May. *Parallel I/O for High Performance Computing*. Morgan Kaufman, 2001.
- Ross, et al. "A Case Study in Application I/O on Linux Clusters", *Proceedings of Supercomputing 2001*. ACM/IEEE, 2001.

References (con't)

- Schoof, et al. "Sharable and Scalable I/O Solutions for High Performance Computing Applications", *Supercomputing 2001 Tutorial S6 Notes*. ACM/IEEE, 2001.
- Sterling, ed. *Beowulf Cluster Computing with Linux*. MIT Press, 2002.