

MPI-IO

- **Overview**

- Offers an alternative interface to the HPSS Client API library for applications written for a distributed memory programming model using message passing.
- Coordinates and simplifies parallel access to HPSS files from multiple processes.
- Utilizes the MPI Datatype abstraction paradigm to facilitate distributed data accesses to a file from multiple processes, allowing each process to view only the data distributed to it.
- Provides functionality to merge multiple small data accesses into a single large access to improve performance and efficiency.
- Enables nonblocking accesses to hide I/O costs.
- The MPI-IO API is fully documented in the *HPSS Programmer's Reference Manual, Volume 1*.

A Brief History of MPI-IO

- Initially developed by IBM in 1994 as a research project.
- Gained support from NASA Ames, then LLNL.
- Added as a chapter to MPI-2 draft in 1996.
- Published in MPI-2 standard in 1997.
 - Standard is available on line at:

`http://www.mcs.anl.gov/mpi`
- Distributed with HPSS 4v1 release in 1998.

Compiling MPI-IO Applications

- **Include files**

- You must include the following header file in every C module using MPI or the MPI-IO API.

```
#include "mpio.h"
```

- You must specify the path name to the HPSS include directory on the compilation directive.

```
-I/usr/lpp/hpss/include
```

- **Libraries**

- You must link with the MPI-IO and MPI libraries as well as with all libraries needed for the HPSS Client API (or IPI3 equivalent).

```
-lmpioapi
```

```
-lhpssapi
```

```
-lhpss
```

```
-lmpi
```

- You must specify the path name to the HPSS library directory on the load directive.

```
-L/usr/lpp/hpss/lib
```

Sample Makefile

```
CC                = mpcc_r4
COMPFLAGS         = -g
INCLUDE_PATH      = -I. -I/usr/lpp/hpss/include -I/usr/lpp/encina/include \
                  -I/usr/local/mpi/include
CFLAGS           = $(INCLUDE_PATH) $(COMPFLAGS)

.c.o:;            @echo "Compiling $<"...
                  @$ (CC) $(CFLAGS) -c $<

LIBS              = -L/usr/lpp/hpss/lib \
                  -lmpioapi -lhpsapi_ipi -lhps_ipi -lhpsapi3 -lipi3 \
                  -L/usr/lpp/encina \
                  -lEncina -lEncClient \
                  -ldce \
                  -L/usr/local/mpi/lib \
                  -lmpi \
                  -ldcepthreads -lpthreads

PROG              = sample_mpio

all:              $(PROG)

sample_mpio:      sample_mpio.o
                  @echo "Linking $@"...
                  @$ (CC) $@.o -o $@ $(CFLAGS) $(LIBS)
```

Notes on compilation and linking environment

- Your system administrator will need to set up an `mpcc_r4` entry. This interface to the `xlc` compiler needs to merge the DCE and MPI options together to get the appropriate DCE and MPI libraries, along with the appropriate threads libraries and thread-safe C libraries.
- The sample Makefile on the previous slide shows the set up for MPI-IO with MPICH, from Argonne National Laboratory. HPSS MPI-IO can also be linked to the proprietary MPI from IBM (`/usr/lpp/ppe.poe/lib/libmpi_r`), with appropriate changes to the Makefile for the include and library paths.
- The order of libraries is important! Although `xlc` versions may automatically load the same libraries specified, they do not always get them in the correct order. Explicitly ordering them in the load directive will assure that the libraries are correct.
- The Makefile shows that the IPI3 libraries can be loaded even when the `HPSS_TRANSFER_TYPE` will be `TCP`, provided the underlying HPSS system was built with IPI3 support enabled. If only `TCP` transfers will be used, the IPI3 libraries need not be used.

Environment Variables

- **The following environment variables can be used to control the execution of an MPI-IO program:**
 - **MPIO_LOGIN_NAME** DCE principal name under which to execute the application.
 - **MPIO_KEYTAB_PATH** Path name to keytab file for the given principal.
 - **MPIO_DEBUG** Flag (0 or nonzero) disabling or enabling debugging messages (to stderr) from the MPI-IO library.
- **You must/may use the following HPSS environment variables to control the execution of your MPI-IO program:**
 - **HPSS_LS_NAME** DCE CDS name of HPSS Location Server.
 - **HPSS_TRANSFER_TYPE** TCP or IPI3.
 - **HPSS_DEBUG** Flag disabling/enabling debugging messages.

Error Codes

- MPI-IO is part of the MPI (MPI-2) Standard, which dictates that each API will filter its return status through an error handler.
- The default error handler for most MPI APIs is `MPI_ERRORS_ARE_FATAL`, which causes a program to abort if an error is detected and returned by an MPI API.
- For MPI-IO APIs the default error handler is `MPI_ERRORS_RETURN`, which allows the return value of the API to carry an error code.
- Error return codes must be checked by the application. If an MPI-IO function does not return `MPI_SUCCESS`, the application must handle the error appropriately.
- `MPI_Error_string` can be used to translate an MPI-IO error code into a printable string message.
- User-defined error handlers may be used to alter the default error handling on a per-file basis.

DCE Login Context

- Prior to executing an MPI-IO application, a user should create a keytab file for the DCE principal under which the application will be run.
- This allows authentication for multiple, distributed processes from a single location (e.g., a globally accessible keytab file).
- Create a keytab file using `rgy_edit` interactively:

```
%rgy_edit
rgy_edit=>kta -p login_name -f keytab_path
... prompt for password for login_name
rgy_edit=>quit
%setenv MPIIO_LOGIN_NAME login_name
%setenv MPIIO_KEYTAB_PATH keytab_path
```

- If `MPIO_KEYTAB_PATH` is not specified, MPI-IO will attempt to use the current DCE login context in each process environment; if no such logincontext exists in any process environment, MPI-IO will fail to be able to initialize the HPSS Client API, and `MPI_Init` will fail.

API Overview

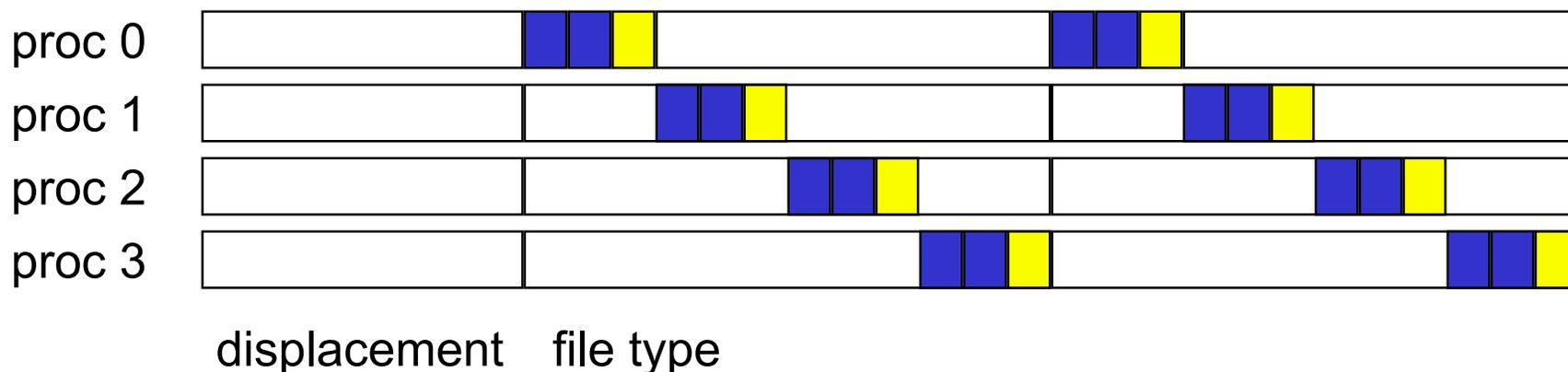
- **The MPI-IO library functions can be divided into the following categories:**
 - MPI-IO environment management
 - **initialization, finalization**
 - File creation and manipulation
 - **open, close, delete, attribute setting and retrieval**
 - File views
 - **MPI_Datatypes used for etypes, filetypes, and buftypes**
 - File accesses
 - **read, write, seek; nonblocking and collective accesses**
 - File interoperability
 - **data conversions**
 - File consistency
 - **atomicity, sync**
 - Error handling facilities
 - **creating error handlers, retrieving error messages**

Preliminary concepts

- **The MPI-IO APIs are defined in terms of the following concepts:**
 - A *file* is an ordered collection of typed data items.
 - A *file displacement* is an absolute byte position relative to the beginning of a file.
 - An *etype* is the unit of data access and positioning used within a file.
 - A *filetype* is the basis for partitioning a file among processes and defines a template for accessing the file.
 - A *view* defines the current set of data visible and accessible from an open file as an ordered set of etypes.
 - An *offset* is a position in the file relative to the current view, expressed as a count of etypes.
 - A *file pointer* is an implicit offset maintained by MPI-IO. Individual file pointers are local to each process that opened the file; a shared file pointer is shared by the group of processes that opened the file.
 - A *file handle* is an opaque object created by `MPI_File_open` and freed by `MPI_File_close`.

MPI Datatypes to describe file data distribution

- *etypes* specify the smallest unit of data accessed in the file; it can be any MPI Datatype. In the example below, the etype is a struct of three elements.
- A *filetype* is composed of repeated tilings of etypes and describes a pattern of accessible and inaccessible data. The collection of filetypes for all processes accessing a file defines the data distribution across processes. In the diagram below, each process has access to the shaded portions of the file, and not to the rest of the file.
- The file *displacement* allows skipping over labelling information in a file.



Establishing the MPI-IO Environment

- MPI-IO is initialized by initializing the MPI environment. By including the “mpio.h” header file, a call to MPI_Init will also invoke an MPI-IO-specific initialization. MPI_Init should be invoked from main, and should pass the addresses of main’s arguments as follows:

```
MPI_Init(&argc, &argv);
```

- MPI_Init for MPI-IO
 - Initializes MPI-IO data structures such as file tables and message types
 - Initializes error handlers
 - Establishes login context from environment variables
 - Initializes HPSS Client API interface
 - Spawns MPI-IO server thread in each process
- MPI-IO, like MPI, needs a finalization step, invoked before exiting main:

```
MPI_Finalize();
```

- MPI_Finalize for MPI-IO
 - Deallocates resources used
 - Terminates server threads

API for creating a file or opening an existing file

- When opening a file you specify an HPSS path name and a group of participating processes through an MPI communicator. The mode of the file must be specified and other hints can be provided as well. If successful, a file handle is returned; otherwise an error code is returned.
- If the file does not exist, `MPI_MODE_CREATE` can be used with the `amode` argument to create the file.

```
#include "mpio.h"
```

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

Creating a file - example

```
amode = MPI_MODE_CREATE | MPI_MODE_RDWR;

rc = MPI_File_open(MPI_COMM_WORLD,
                  "/users/u28/linda/testfile",
                  amode, MPI_INFO_NULL, &fh);

if (rc != MPI_SUCCESS)
    /* Could not open file */ . . .
```

- MPI_File_open is a collective call; all processes in the communicator group must invoke MPI_File_open. The communicator may be MPI_COMM_SELF.
- The full HPSS file path name must be specified.
- The access mode of the file is specified at the MPI-IO interface level; file permissions are specified as a file hint.
- For complete details on how to specify the mode and hints, see the *HPSS Programmer's Reference Manual*.

APIs for closing and deleting files

- Closing a file is done with `MPI_File_close`.
- If the file was opened with `MPI_MODE_DELETE_ON_CLOSE` included in the `amode` argument, the file will automatically be deleted.
- Otherwise, a file may be deleted with `MPI_File_delete`.

```
#include "mpio.h"
```

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

```
int MPI_File_delete(char * filename,      /* IN */  
                    MPI_Info info);     /* IN */
```

APIs for changing or retrieving file size

- To set the size of the file without preallocating, use `MPI_File_set_size`.
- To set the size of the file AND preallocate space for the file as specified, use `MPI_File_preallocate`.
- To get the current size of the file, use `MPI_File_get_size`. (Note: this size does not necessarily indicate the amount of space that has been allocated; only segments to which some data has been written have been allocated.)

```
#include "mpio.h"
```

```
int MPI_File_set_size(MPI_File      fh,          /* IN */
                     MPI_Offset    size);      /* IN */
```

```
int MPI_File_preallocate(MPI_File    fh,          /* IN */
                         MPI_Offset  size);      /* IN */
```

```
int MPI_File_get_size(MPI_File      fh,          /* IN */
                      MPI_Offset *  size);      /* OUT */
```

MPI-IO File hints

- **MPI_Info type used to provide file hints**

- MPI-2 introduced a new predefined type, MPI_Info, that associates a key with a value, where both the key and value are strings.
- The MPI-2 standard reserves some info keys for MPI-IO. An implementation may or may not interpret these reserved keys, and is free to interpret others.
- The HPSS MPI-IO interprets the following info keys and translates them into appropriate HPSS hint values at open time. Although other MPI-IO APIs allow file hint arguments, only the hints given at open time are currently used.

- | | |
|------------------------|--------------------------------------------|
| • “file_perm” | Mode |
| • “striping_factor” | StripeWidth |
| • “striping_unit” | StripeLength |
| • “hpss_cos” | COSId |
| • “hpss_sclasstype” | AvgLatency (0 for DISK, 1 for TAPE) |
| • “hpss_max_file_size” | MaxFileSize |
| • “hpss_min_file_size” | MinFileSize |
| • “hpss_access_size” | OptimumAccessSize |

File Views

- When a file is initially opened, the file view for each process allows access to every byte of the file. That is, the default etype and filetype are MPI_BYTE. Use MPI_File_set_view to change the view for each process.
- Use MPI_File_get_view to retrieve characteristics of the current view.

```
#include "mpio.h"
```

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

```
int MPI_File_get_view(MPI_File      fh,                /* IN */
                     MPI_Offset *   disp,              /* OUT */
                     MPI_Datatype * etype,            /* OUT */
                     MPI_Datatype * filetype,         /* OUT */
                     char *         datarep);          /* OUT */
```

File view example

- Example:

```
MPI_Datatype filetype;  
...  
MPI_Type_contiguous(100, MPI_FLOAT, &filetype);  
MPI_Type_commit(&filetype);  
  
rc = MPI_File_set_view(fh, (MPI_Offset) 0, MPI_FLOAT,  
                      filetype, "native");
```

- Data representation used is native (default).
- Initial displacement of 0 must be cast to MPI_Offset.
- Example datatype does not contain 'holes'; if all processes used this datatype for setting the file view, each process would have access to all elements in the file.
- Positioning and read/write counts are in terms of MPI_FLOAT units; no smaller unit of data can be accessed using this view.
- The filetype will be repeatedly tiled over the file (i.e., there can be more than 100 floats in the file).

Data Accesses

- **Three flavors of offset specification supported**
 - Explicit offset specification
 - **Read/Write API requires an explicit nonnegative offset in etype units.**
 - Individual pointer offset specification
 - **Read/Write API implicitly uses a per-process file pointer, which indicates the offset in etype units, according to each process' view.**
 - Shared pointer offset specification
 - **Read/Write API implicitly uses a per-file pointer, which indicates the offset in etype units; requires that all processes specify the same view (i.e., all have access to the same data in the file).**
- **Collective operations**
 - Participation of all processes that opened the file is required.
 - Merges multiple requests into a single request to HPSS.
- **Nonblocking operations**
 - Can be split collective or noncollective.
 - Allows overlapping of I/O with computations.

Explicit offset APIs

```
#include "mpio.h"

int MPI_File_read_at(MPI_File fh, /* IN */
                    MPI_Offset offset, /* IN */
                    void * buf, /* OUT */
                    int count, /* IN */
                    MPI_Datatype datatype, /* IN */
                    MPI_Status * status); /* OUT */

int MPI_File_write_at(MPI_File fh, /* IN/OUT */
                    MPI_Offset offset, /* IN */
                    void * buf, /* IN */
                    int count, /* IN */
                    MPI_Datatype datatype, /* IN */
                    MPI_Status * status); /* OUT */
```

- *offset* is the position in # etype units from the beginning of the file view.
- *buf* is the address of the application's buffer to read or write.
- *count* is the number of datatype units to read from or write to the file.
- *status* object can be queried to get count of datatype units read or written.

Explicit offset example

```
float      array[50];
MPI_Status status;
int        count;
...
... /* code to assign array values, etc. */
...
rc = MPI_File_write_at(fh, (MPI_Offset)100, array, 50,
                      MPI_FLOAT, &status);

if (rc != MPI_SUCCESS)
    ... /* Handle error */

MPI_Get_count(status, MPI_FLOAT, &count);
if (count != 50)
    ... /* Handle error */

rc = MPI_File_read_at(fh, (MPI_Offset)0, array, 50,
                     MPI_FLOAT, &status);
...
```

Nonblocking explicit offset APIs

```
#include "mpio.h"
```

```
MPI_File_iread_at(MPI_File      fh,          /* IN */
                  MPI_Offset    offset,     /* IN */
                  void *        buf,        /* OUT */
                  int           count,      /* IN */
                  MPI_Datatype  datatype,  /* IN */
                  MPI_Request *  request);  /* OUT */
```

```
MPI_File_fwrite_at(MPI_File      fh,          /* IN/OUT */
                   MPI_Offset    offset,     /* IN */
                   void *        buf,        /* IN */
                   int           count,      /* IN */
                   MPI_Datatype  datatype,  /* IN */
                   MPI_Request *  request);  /* OUT */
```

- *request* is returned when I/O is initiated; it is queried later by an `MPI_Wait` or `MPI_Test` to determine if I/O is complete.

Collective explicit offset APIs

```
#include "mpio.h"
```

```
MPI_File_read_at_all(MPI_File      fh,          /* IN */
                    MPI_Offset     offset,      /* IN */
                    void *         buf,        /* OUT */
                    int             count,      /* IN */
                    MPI_Datatype   datatype,   /* IN */
                    MPI_Status *   status);    /* OUT */
```

```
MPI_File_write_at_all(MPI_File      fh,          /* IN/OUT */
                    MPI_Offset     offset,      /* IN */
                    void *         buf,        /* IN */
                    int             count,      /* IN */
                    MPI_Datatype   datatype,   /* IN */
                    MPI_Status *   status);    /* OUT */
```

- All processes that opened the file must participate in the read or write.
- A single (logical) transfer is described to HPSS from the collective requests, and parallel transfer of data to all processes is attempted.

Individual File Pointers

- Individual file pointers are set to 0 when a file is opened and when a view for a file is set.
- `MPI_File_seek` can be used to set an individual file pointer to a new position in the file. This is always relative to the current process' view.
- Recall that the position in the file kept by the pointer is in terms of number of etype units from the beginning of the file in the current view. This is a byte offset only if the etype in the current view is `MPI_BYTE`.
- `MPI_File_get_position` can be used to retrieve the current position of an individual file pointer for a given process.
- `MPI_File_get_byte_offset` can be used to convert an etype-unit position into a byte offset.

Individual file pointer APIs

```
#include "mpio.h"

int MPI_File_read(MPI_File fh, /* IN */
                 void * buf, /* OUT */
                 int count, /* IN */
                 MPI_Datatype datatype, /* IN */
                 MPI_Status * status); /* OUT */

int MPI_File_write(MPI_File fh, /* IN/OUT */
                  void * buf, /* IN */
                  int count, /* IN */
                  MPI_Datatype datatype, /* IN */
                  MPI_Status * status); /* OUT */
```

- *buf* is the address of the application's buffer to read or write.
- *count* is the number of datatype units to read from or write to the file.
- *status* object can be queried to get count of datatype units read or written.
- Each process reads or writes at the etype position indicated by that process' individual file pointer, according to the current process' view.

Nonblocking individual file pointer APIs

```
#include "mpio.h"
```

```
int MPI_File_iread(MPI_File      fh,          /* IN */
                  void *        buf,        /* OUT */
                  int           count,      /* IN */
                  MPI_Datatype  datatype,  /* IN */
                  MPI_Request *  request);  /* OUT */
```

```
int MPI_File_fwrite(MPI_File      fh,          /* IN/OUT */
                   void *        buf,        /* IN */
                   int           count,      /* IN */
                   MPI_Datatype  datatype,  /* IN */
                   MPI_Request *  request);  /* OUT */
```

- *request* is returned when I/O is initiated; it is queried later by an `MPI_Wait` or `MPI_Test` to determine if I/O is complete.
- Each process reads or writes at the etype position indicated by that process' individual file pointer, according to the process' current view.

Nonblocking individual file pointer example

```
float buffer1[BUFSIZE], buffer2[BUFSIZE];
MPI_Request request;
MPI_Status status;

... /* Compute buffer1 elements */

rc = MPI_File_iread(fh, buffer1, BUFSIZE, MPI_FLOAT,
                  &request);

if (rc != MPI_SUCCESS)
    ... /* Handle error */

... /* Compute buffer2 elements while I/O completes */

MPI_Wait(request, &status);

rc = MPI_File_iread(fh, buffer2, BUFSIZE, MPI_FLOAT,
                  &request);
```

Collective individual file pointer APIs

```
#include "mpio.h"
```

```
MPI_File_read_all(MPI_File      fh,          /* IN */
                  void *       buf,         /* OUT */
                  int          count,       /* IN */
                  MPI_Datatype datatype,   /* IN */
                  MPI_Status *  status);    /* OUT */
```

```
MPI_File_write_all(MPI_File      fh,         /* IN/OUT */
                   void *       buf,        /* IN */
                   int          count,     /* IN */
                   MPI_Datatype datatype,  /* IN */
                   MPI_Status *  status);  /* OUT */
```

- All processes that opened the file must participate in the read or write.
- A single (logical) transfer is described to HPSS from the collective requests, and parallel transfer of data to all processes is attempted.
- Each process reads or writes at the etype position indicated by that process' individual file pointer, according to the process' current view.

Shared File Pointers

- The shared file pointer is set to 0 when a file is opened and when a view for a file is set.
- All processes using the shared file pointer must have the same file view.
- `MPI_File_seek_shared` can be used to set a shared file pointer to a new position in the file. This is always relative to the current view.
- Recall that the position in the file kept by the pointer is in terms of number of etype units from the beginning of the file in the current view. This is a byte offset only if the etype in the current view is `MPI_BYTE`.
- `MPI_File_get_shared_position` can be used to retrieve the current position of a shared file pointer.
- `MPI_File_get_byte_offset` can be used to convert an etype-unit position into a byte offset.

Shared file pointer APIs

```
#include "mpio.h"

int MPI_File_read_shared(MPI_File fh, /* IN/OUT */
                        void * buf, /* OUT */
                        int count, /* IN */
                        MPI_Datatype datatype, /* IN */
                        MPI_Status * status); /* OUT */

int MPI_File_write_shared(MPI_File fh, /* IN/OUT */
                        void * buf, /* IN */
                        int count, /* IN */
                        MPI_Datatype datatype, /* IN */
                        MPI_Status * status); /* OUT */
```

- *buf* is the address of the application's buffer to read or write.
- *count* is the number of datatype units to read from or write to the file.
- *status* object can be queried to get count of datatype units read or written.
- Each process reads or writes at the etype position indicated by the shared file pointer, according to the current file view.

Nonblocking shared file pointer APIs

```
#include "mpio.h"
```

```
int MPI_File_iread_shared(MPI_File fh, /* IN/OUT */  
                          void * buf, /* OUT */  
                          int count, /* IN */  
                          MPI_Datatype datatype, /* IN */  
                          MPI_Request * request); /* OUT */
```

```
int MPI_File_fwrite_shared(MPI_File fh, /* IN/OUT */  
                           void * buf, /* IN */  
                           int count, /* IN */  
                           MPI_Datatype datatype, /* IN */  
                           MPI_Request * request); /* OUT */
```

- *request* is returned when I/O is initiated; it is queried later by an `MPI_Wait` or `MPI_Test` to determine if I/O is complete.
- Each process reads or writes at the etype position indicated by the shared file pointer, according to the current file view.

Collective shared file pointer APIs

```
#include "mpio.h"

int MPI_File_read_ordered(MPI_File fh, /* IN/OUT */
                          void * buf, /* OUT */
                          int count, /* IN */
                          MPI_Datatype datatype, /* IN */
                          MPI_Status * status); /* OUT */

int MPI_File_write_ordered(MPI_File fh, /* IN/OUT */
                           void * buf, /* IN */
                           int count, /* IN */
                           MPI_Datatype datatype, /* IN */
                           MPI_Status * status); /* OUT */
```

- *buf* is the address of the application's buffer to read or write.
- *count* is the number of datatype units to read from or write to the file.
- *status* object can be queried to get count of datatype units read or written.
- Each process reads or writes at the etype position at which the shared pointer would be after all processes whose ranks are less than that of this process had accessed their data.

Collective shared file pointer example

```
integer buffer[SLICESIZE];
MPI_Status status;
...
... /* Compute elements of buffer */
...
rc = MPI_File_seek_shared(fh, (MPI_Offset)0,
                          MPI_SEEK_END);

if (rc != MPI_SUCCESS)
    ... /* Handle error */

rc = MPI_File_write_ordered(fh, buffer, SLICESIZE,
                            MPI_INT, &status);

if (rc != MPI_SUCCESS)
    ... /* Handle error */
```

- Not all processes need write the same amount (i.e., SLICESIZE can vary per process). The write position for each process depends on the rank of each process and the amount of data written by all processes of lesser rank.

Split Collective Accesses

- The APIs in this category are intended to allow a restricted form of nonblocking collective accesses. (Note that all the previously described nonblocking APIs have been noncollective.)
- Each API is split into two interfaces, one to begin the I/O and one to end (complete) the I/O.
- At most one split collective call is allowed per file handle at any one time. This is consistent with MPI's paradigm of allowing only one collective communication per communicator, which is extended to allowing only one collective operation per file handle at a time.
- Although an implementation is allowed to implement split collective calls using the corresponding blocking collective routine, at either the begin or end point of the collective call, HPSS MPI-IO split collective calls are implemented as nonblocking calls. Like other nonblocking APIs, a thread is spawned to perform the I/O at the begin call, allowing concurrency with the calling thread. Unlike other nonblocking APIs, the corresponding end call is used instead of MPI_Wait or MPI_Test when the I/O is complete.

Split collective APIs with explicit offsets

```
#include "mpio.h"
```

```
int MPI_File_read_at_all_begin(MPI_File      fh,          /* IN */
                               MPI_Offset    offset,      /* IN */
                               void *        buf,         /* OUT */
                               int           count,       /* IN */
                               MPI_Datatype  datatype);   /* IN */

int MPI_File_read_at_all_end(MPI_File      fh,          /* IN */
                              void *        buf,         /* OUT */
                              MPI_Status *  status);     /* OUT */

int MPI_File_write_at_all_begin(MPI_File      fh,          /* IN/OUT */
                                MPI_Offset    offset,      /* IN */
                                void *        buf,         /* IN */
                                int           count,       /* IN */
                                MPI_Datatype  datatype);   /* IN */

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

Split collective APIs with individual file pointers

```
#include "mpio.h"

int MPI_File_read_all_begin(MPI_File      fh,          /* IN */
                           void *       buf,         /* OUT */
                           int           count,      /* IN */
                           MPI_Datatype datatype);   /* IN */

int MPI_File_read_all_end(MPI_File      fh,          /* IN */
                          void *       buf,         /* OUT */
                          MPI_Status * status);      /* OUT */

int MPI_File_write_all_begin(MPI_File      fh,        /* IN/OUT */
                             void *       buf,       /* IN */
                             int           count,    /* IN */
                             MPI_Datatype datatype); /* IN */

int MPI_File_write_all_end(MPI_File      fh,          /* IN */
                          void *       buf,         /* OUT */
                          MPI_Status * status);      /* OUT */
```

Split collective APIs with shared file pointers

```
#include "mpio.h"

int MPI_File_read_ordered_begin(MPI_File      fh,          /* IN */
                               void *        buf,          /* OUT */
                               int            count,        /* IN */
                               MPI_Datatype  datatype);    /* IN */

int MPI_File_read_ordered_end(MPI_File      fh,          /* IN */
                              void *        buf,          /* OUT */
                              MPI_Status *  status);      /* OUT */

int MPI_File_write_ordered_begin(MPI_File      fh,          /* IN/OUT */
                                 void *        buf,          /* IN */
                                 int            count,        /* IN */
                                 MPI_Datatype  datatype);    /* IN */

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

Split collective example

```
char text[MY_NLINES * 512];
MPI_Status status;

...
rc = MPI_File_read_ordered_begin(fh, text,
                                MY_NLINES * 512,
                                MPI_CHAR);

if (rc != MPI_SUCCESS)
    ... /* Handle error */

... /* Do some other computational work */

rc = MPI_File_read_ordered_end(fh, text, &status);

if (rc != MPI_SUCCESS)
    ... /* Handle error */
```

File Interoperability

- File interoperability is the ability to read a file previously written.
- The file may or may not have been written using the same data representation as is used by the current application.
- File views define the data representation to use for interpreting the bits of data written to a file. This representation may or may not be the native representation on the current platform.
- MPI-IO supports three predefined data representations: “native”, “internal”, and “external32”.
- For HPSS MPI-IO, “native” and “internal” are equivalent on AIX systems, but “external32” representation differs from AIX representation for long doubles (MPI_LONG_DOUBLE).
- MPI-IO also supports user-defined data representations, where the user is responsible for providing functions that can be used to convert to/from a file representation to the native representation of the platform.

APIs for file interoperability

- Use `MPI_Register_datarep` to create a user-defined data representation. A registered datarep can be used in a set view operation to indicate that the file data read or written must be converted accordingly.
- Use `MPI_File_get_type_extent` to determine the extent of an MPI datatype in a given file. The data representation in the current view of the file will be used to determine the datatype extent.

```
#include "mpio.h"
```

```
int MPI_Register_datarep(char *      datarep,          /* IN */
    MPI_Datarep_conversion_function * read_conv_fn,    /* IN */
    MPI_Datarep_conversion_function * write_conv_fn,  /* IN */
    MPI_Datarep_extent_function *   dtype_ext_fn,    /* IN */
    void *                          extra_state);     /* IN */

int MPI_File_get_type_extent(MPI_File fh,             /* IN */
                             MPI_Datatype datatype, /* IN */
                             MPI_Aint *  extent);    /* OUT */
```

File interoperability example

```
int my_read_fn(void *, MPI_Datatype, int, void *,
               MPI_Offset, void *);
int my_write_fn(void *, MPI_Datatype, int, void *,
                MPI_Offset, void *);
int my_extent_fn(MPI_Datatype, MPI_Aint *, void *);
MPI_Aint my_extent, mpi_extent;

rc = MPI_Register_datarep("my_own_rep", my_read_fn,
                          my_write_fn, my_extent_fn,
                          NULL);

if (rc != MPI_SUCCESS) ... /* Handle error */

rc = MPI_File_set_view(fh, (MPI_Offset)0,
                       MPI_LONG_DOUBLE, MPI_LONG_DOUBLE,
                       "my_own_rep", MPI_INFO_NULL);

if (rc != MPI_SUCCESS) ... /* Handle error */

MPI_File_get_type_extent(fh, MPI_DOUBLE, &my_extent);
MPI_Type_extent(MPI_DOUBLE, &mpi_extent);
```

File Consistency

- **MPI-IO provides three levels of consistency semantics to deal with multiple accesses to a single file:**
 - sequential consistency among all accesses through a single file handle;
 - sequential consistency among all accesses using file handles created from a single collective open with atomic mode enabled; and
 - user-imposed consistency among accesses other than the above.
- **Maintaining file consistency means guaranteeing that concurrent accesses are sequentialized so that data is not being modified while being accessed.**
 - Multiple read accesses can take place concurrently with no loss of consistency, provided no write access is also concurrent with the reads.
 - Multiple write accesses must be ordered so that no two writes of the same data positions are attempted concurrently.
- **Setting the MPI-IO atomicity mode of an MPI-IO file handle to *true* requires sequentialization of concurrent accesses. Since HPSS sequentializes all accesses through a given HPSS file handle, MPI-IO atomic mode is enforced by default. (However, if MPI-IO atomic mode is enabled, errors will be reported if concurrent access attempts are detected.)**

File consistency (continued)

- **Single file handle consistency**

- If a file is opened by a single process, all accesses are sequentialized by the HPSS BFS and Client API, regardless of the MPI-IO atomicity setting, and regardless of any concurrency (e.g., multithreading) in the process.

- **Multiple file handle consistency**

- If a file is opened by multiple processes with a single open, each process has its own MPI-IO file handle for the file, but these all refer to the same HPSS file handle and hence to the same BFS file handle. Multiple accesses on this file handle are still sequentialized by HPSS, regardless of atomicity setting and amount of concurrency in the processes.
- If a file is opened by multiple processes with multiple opens, each open gets a new BFS file handle. Although each file handle refers to the same file, there is no sequentialization enforced by HPSS with multiples handles, so the user must take precautions to protect the consistency of the file. Since each file open also generates its own MPI-IO file handle per process, atomicity and file sync operations are not sufficient to synchronize accesses to the file.

File consistency APIs

- Use `MPI_File_set_atomicity` to toggle atomic mode on or off.
- Use `MPI_File_get_atomicity` to retrieve the current atomicity mode.
- Use `MPI_File_sync` to synchronize multiple processes accessing file handles returned from a single open file operation.

```
#include "mpio.h"
```

```
int MPI_File_set_atomicity(MPI_File fh,      /* IN/OUT */  
                           int      flag);  /* IN */
```

```
int MPI_File_get_atomicity(MPI_File fh,      /* IN */  
                           int *    flag);  /* OUT */
```

```
int MPI_File_sync(MPI_File fh);             /* IN/OUT */
```

File Error Handlers

- MPI-IO associates an error handler with a file handle (i.e., an MPI_File object) just as MPI allows an error handler to be associated with an MPI_Comm handle.
- For each MPI-IO API that takes a file handle as an argument, when an error is detected, the error code is passed to the error handler associated with the file handle. If the error handler returns, so does the MPI-IO API, returning the detected error code. Otherwise, the error handler aborts, and the application terminates.
- For each MPI-IO API that does not have a file handle passed to it (e.g., MPI_File_open, MPI_File_delete), the default file error handler (MPI_ERRORS_RETURN) is invoked for the error code.
- The default error handler for a file handle can be changed to another predefined error handler or to a user-defined error handler. Likewise, the default file error handler can be changed.

MPI-IO Error Handler APIs

- Use `MPI_File_create_errhandler` to create an `MPI_Errhandler` that can be used for errors detected with operations on a given file handle.
- Use `MPI_File_set_errhandler` to associate an error handler with a file handle or the default file error handler (file argument of `MPI_FILE_NULL`).
- Use `MPI_File_get_errhandler` to retrieve the error handler currently associated with a file handle or the default file error handler (file argument of `MPI_FILE_NULL`).

```
#include "mpio.h"
```

```
int MPI_File_create_errhandler(  
    MPI_File_errhandler_fn * function, /* IN */  
    MPI_Errhandler * errhandler); /* OUT */  
  
int MPI_File_set_errhandler(MPI_File file, /* IN */  
    MPI_Errhandler errhandler); /* OUT */  
  
int MPI_File_get_errhandler(MPI_File file, /* IN */  
    MPI_Errhandler * errhandler); /* OUT */
```

MPI-2 APIs Included

- In order to implement MPI-IO, many other features of the MPI-2 standard were needed. In addition to the APIs for MPI-IO, the following types and/or APIs were implemented. See the MPI-2 Standard for complete details.

- MPI_Info

- MPI_Info_create
- MPI_Info_set
- MPI_Info_delete
- MPI_Info_get
- MPI_Info_get_valuelen
- MPI_Info_get_nkeys
- MPI_Info_get_nthkey
- MPI_Info_dup
- MPI_Info_free

- MPI error handling

- MPI_Add_error_class
- MPI_Add_error_code
- MPI_Add_error_string

MPI-2 APIs (continued)

- MPI_Datatype attribute caching
 - MPI_Type_create_keyval
 - MPI_Type_free_keyval
 - MPI_Type_set_attr
 - MPI_Type_get_attr
 - MPI_Type_delete_attr
 - MPI_Type_dup
- MPI_Datatype decoding
 - MPI_Type_get_envelope
 - MPI_Type_get_contents

Sample MPI-IO Program

```
#include "mpi.h"
#include "mpio.h"

#define FILE_SIZE (32*32*32*4)

/* This program writes a 3D block-distributed array to a file
 * corresponding to a global array in row-major (C) order, reads
 * it back, and checks that the data read is correct.
 *
 * It uses a 32^3 array. For other array sizes, change FILE_SIZE above and
 * array_of_gsizes below.
 *
 * The file name is taken as a command-line argument.
 *
 * Note that the file access pattern is noncontiguous and collective I/O is used.
 */

main(int argc, char **argv)
{
    int i, ndims, array_of_gsizes[3], array_of_distrib[3];
    int order, filecount, nprocs, j, len, flag, rc, msglen;
    int array_of_dargs[3], array_of_psizes[3];
    int *readbuf, *writebuf, bufcount, mynod, *tmpbuf, array_size;
    char filename[HPSS_MAX_PATH_NAME + 1], errmsg[MPI_MAX_ERROR_STRING + 1];
    MPI_Datatype newtype;
    MPI_File fh;
    MPI_Status status;
    MPI_Request request;
```

Sample MPI-IO Program - continued

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &mynod);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

/* process 0 takes the file name as a command-line argument and
   broadcasts it to other processes */

if (!mynod) {
    i = 1;
    while ((i < argc) && strcmp("-fname", argv[i])) i++;

    if (i >= argc) {
        printf("\n*# Usage: %s -fname filename\n", argv[0]);
        printf("*# The filename must be a full HPSS path name\n");
        printf("*# An example filename is /users/u28/linda/test\n\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    strcpy(filename, argv[i]);
    len = strlen(filename);

    MPI_Bcast(&len, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(filename, len+1, MPI_CHAR, 0, MPI_COMM_WORLD);
}

else {
    MPI_Bcast(&len, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(filename, len+1, MPI_CHAR, 0, MPI_COMM_WORLD);
}
```

Sample MPI-IO Program - continued

```
/* create the distributed array filetype */

ndims = 3;
order = MPI_ORDER_C;

array_of_gsizes[0] = 32;
array_of_gsizes[1] = 32;
array_of_gsizes[2] = 32;

array_of_distrib[0] = MPI_DISTRIBUTE_BLOCK;
array_of_distrib[1] = MPI_DISTRIBUTE_BLOCK;
array_of_distrib[2] = MPI_DISTRIBUTE_BLOCK;

array_of_dargs[0] = MPI_DISTRIBUTE_DFLT_ARG;
array_of_dargs[1] = MPI_DISTRIBUTE_DFLT_ARG;
array_of_dargs[2] = MPI_DISTRIBUTE_DFLT_ARG;

for (i = 0; i < ndims; i++) array_of_psize[i] = 0;

MPI_Dims_create(nprocs, ndims, array_of_psize);

MPI_Type_create_darray(nprocs, mynod, ndims, array_of_gsize,
                      array_of_distrib, array_of_dargs,
                      array_of_psize, order, MPI_INT, &newtype);

MPI_Type_commit(&newtype);
```

Sample MPI-IO Program - continued

```
/* initialize writebuf */

MPI_Type_size(newtype, &bufcount);
bufcount = bufcount / sizeof(int);
writebuf = (int *) malloc(bufcount * sizeof(int));
for (i = 0; i < bufcount; i++) writebuf[i] = 1;

array_size = array_of_gsizes[0] * array_of_gsizes[1] * array_of_gsizes[2];
tmpbuf = (int *) calloc(array_size, sizeof(int));
MPI_Irecv(tmpbuf, 1, newtype, mynod, 10, MPI_COMM_WORLD, &request);
MPI_Send(writebuf, bufcount, MPI_INT, mynod, 10, MPI_COMM_WORLD);
MPI_Wait(&request, &status);

j = 0;
for (i = 0; i < array_size; i++)
    if (tmpbuf[i]) {
        writebuf[j] = i;
        j++;
    }
free(tmpbuf);

if (j != bufcount) {
    printf("Error in initializing writebuf on node %d\n", mynod);
    MPI_Abort(MPI_COMM_WORLD, 1);
}

/* end of initialization */
```

Sample MPI-IO Program - continued

```
/* write the array to the file */

rc = MPI_File_open(MPI_COMM_WORLD, filename,
                  MPI_MODE_CREATE | MPI_MODE_RDWR | MPI_MODE_UNIQUE_OPEN,
                  MPI_INFO_NULL, &fh);
if (rc != MPI_SUCCESS) {
    MPI_Error_string(rc, errmsg, &msglen);
    printf("Error opening file: %s\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
}
rc = MPI_File_set_view(fh, 0, MPI_INT, newtype, "native", MPI_INFO_NULL);
if (rc != MPI_SUCCESS) {
    MPI_Error_string(rc, errmsg, &msglen);
    printf("Error setting file view: %s\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
}
rc = MPI_File_write_all(fh, writebuf, bufcount, MPI_INT, &status);
if (rc != MPI_SUCCESS) {
    MPI_Error_string(rc, errmsg, &msglen);
    printf("Error writing file: %s\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
}
rc = MPI_File_close(&fh);
if (rc != MPI_SUCCESS) {
    MPI_Error_string(rc, errmsg, &msglen);
    printf("Error closing file: %s\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
}
```

Sample MPI-IO Program - continued

```
/* now read it back */

readbuf = (int *) malloc(bufcount * sizeof(int));
MPI_File_open(MPI_COMM_WORLD, filename,
              MPI_MODE_RDWR | MPI_MODE_UNIQUE_OPEN | MPI_MODE_DELETE_ON_CLOSE,
              MPI_INFO_NULL, &fh);
if (rc != MPI_SUCCESS) {
    MPI_Error_string(rc, errmsg, &msglen);
    printf("Error opening file: %s\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
}
MPI_File_set_view(fh, 0, MPI_INT, newtype, "native", MPI_INFO_NULL);
if (rc != MPI_SUCCESS) {
    MPI_Error_string(rc, errmsg, &msglen);
    printf("Error setting file view: %s\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
}
MPI_File_read_all(fh, readbuf, bufcount, MPI_INT, &status);
if (rc != MPI_SUCCESS) {
    MPI_Error_string(rc, errmsg, &msglen);
    printf("Error reading file: %s\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
}
MPI_File_close(&fh);
if (rc != MPI_SUCCESS) {
    MPI_Error_string(rc, errmsg, &msglen);
    printf("Error closing file: %s\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
}
```

Sample MPI-IO Program - continued

```
/* check the data read */

flag = 0;
for (i = 0; i < bufcount; i++)
    if (readbuf[i] != writebuf[i]) {
        printf("Node %d, readbuf %d, writebuf %d, i %d\n",
            mynod, readbuf[i], writebuf[i], i);
        flag = 1;
    }
if (!flag) printf("Node %d: data read back is correct\n", mynod);

/* clean up */

MPI_Type_free(&newtype);
free(readbuf);
free(writebuf);

MPI_Finalize();
}
```