Message Passing with MPI Parallel IO

Graham E Fagg
ARC Georgetown University
August 2003

MPI-I/O

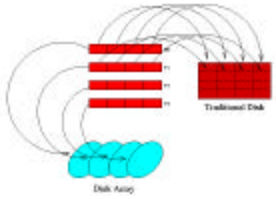
- Introduction
  - What is parallel I/O
  - Why do we need parallel I/O
  - What is MPI-I/O
- MPI-I/O
  - Terms and definitions
  - File manipulation
  - Derived data types and file views

OUTLINE (cont)

- MPI-I/O (cont)
  - Data access
    - Non-collective access
    - Collective access
    - Split collective access
  - File interoperability
  - Gotchas - Consistency and semantics

INTRODUCTION

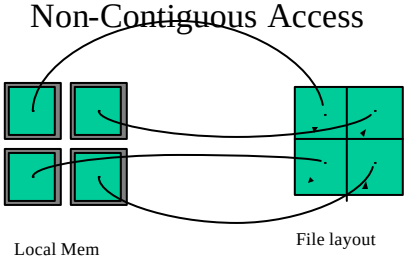
- What is parallel I/O?
  - Multiple processes accessing a single file



INTRODUCTION

- What is parallel I/O?
  - Multiple processes accessing a single file
  - Often, both data and file access is non-contiguous
    - Ghost cells cause non-contiguous data access
    - Block or cyclic distributions cause non-contiguous file access

Non-Contiguous Access



INTRODUCTION

- What is parallel I/O?
  - Multiple processes accessing a single file
  - Often, both data and file access is non-contiguous
    - Ghost cells cause non-contiguous data access
    - Block or cyclic distributions cause non-contiguous file access
  - Want to access data and files with as few I/O calls as possible

INTRODUCTION (cont)

- Why use parallel I/O?
  - Many users do not have time to learn the complexities of I/O optimization

INTRODUCTION (cont)

```
Integer dim
parameter (dim=10000)
Integer*4 out_array(dim)

OPEN (fh,filename,UNFORMATTED)
WRITE(fh) (out_array(I), I=1,dim)

rl = 4*dim
OPEN (fh, filename, DIRECT, RECL=rl)
WRITE (fh, REC=1) out_array
```

INTRODUCTION (cont)

- Why use parallel I/O?
  - Many users do not have time to learn the complexities of I/O optimization
  - Use of parallel I/O can simplify coding
    - Single read/write operation vs. multiple read/write operations

INTRODUCTION (cont)

- Why use parallel I/O?
  - Many users do not have time to learn the complexities of I/O optimization
  - Use of parallel I/O can simplify coding
    - Single read/write operation vs. multiple read/write operations
  - Parallel I/O potentially offers significant performance improvement over traditional approaches

INTRODUCTION (cont)

- Traditional approaches
  - Each process writes to a separate file
    - Often requires an additional post-processing step
    - Without post-processing, restarts must use same number of processor
  - Result sent to a master processor, which collects results and writes out to disk
  - Each processor calculates position in file and writes individually

INTRODUCTION (cont)

- What is MPI-I/O?
  - MPI-I/O is a set of extensions to the original MPI standard
  - This is an interface specification: It does NOT give implementation specifics
  - It provides routines for file manipulation and data access
  - Calls to MPI-I/O routines are portable across a large number of architectures

MPI-I/O

- Terms and Definitions
  - Displacement - Number of bytes from the beginning of a file
  - etype - unit of data access within a file
  - filetype - datatype used to express access patterns of a file
  - file view - definition of access patterns of a file
    - Defines what parts of a file are visible to a process

MPI-I/O

- Terms and Definitions
  - Offset - Position in the file, relative to the current view, expressed in terms of number of etypes
  - file pointers - offsets into the file maintained by MPI
    - Individual file pointer - local to the process that opened the file
    - Shared file pointer - shared (and manipulated) by the group of processes that opened the file

FILE MANIPULATION

- MPI_FILE_OPEN(Comm, filename, mode, info, fh, ierr)
  - Opens the file identified by *filename* on each processor in communicator *Comm*
  - Collective over this group of processors
  - Each processor must use same value for *mode* and reference the same file
  - *info* is used to give hints about access patterns

FILE MANIPULATION

- MODES
  - MPI_MODE_CREATE
    - Must be used if file does not exist
  - MPI_MODE_RDONLY
  - MPI_MODE_RDWR
  - MPI_MODE_WRONLY
  - MPI_MODE_EXCL
    - Error if creating file that already exists
  - MPI_MODE_DELETE_ON_CLOSE
  - MPI_MODE_UNIQUE_OPEN
  - MPI_MODE_SEQUENTIAL
  - MPI_MODE_APPEND

Hints

- Hints can be passed to the I/O implementation via the info argument
- MPI_Info info
- MPI_Info_create (&info)
- MPI_Info_set (info, key, value)
  - key is a string specifying the hint to be applied
  - value is a string specifying the value key is to be set to
- There are 4 pre-defined keys
- The implementation may or may not make use of hints

Hints

- striping_factor
  - The number of I/O devices to be used
- striping_unit
  - The number of bytes per block
- collective_buffering
  - true or false: whether collective buffering should be performed
- cb_block_size
  - Block size to be used for buffering (nodes access data in chunks this size
- cb_buffer_size
  - The total buffer size that should be used for buffering (often block size times # nodes)

FILE MANIPULATION (cont)

- MPI_FILE_CLOSE (fh)
  - This routine synchronizes the file state and then closes the file
  - The user must ensure all I/O routines have completed before closing the file
  - This is a collective routine (but not synchronizing)

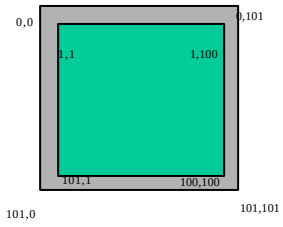
DERIVED DATATYPES & VIEWS

- Derived datatypes are not part of MPI-I/O
- They are used extensively in conjunction with MPI-I/O
- A filetype is really a datatype expressing the access pattern of a file
- Filetypes are used to set file views

DERIVED DATATYPES & VIEWS

- Non-contiguous memory access
- MPI_TYPE_CREATE_SUBARRAY
  - NDIMS - number of dimensions
  - ARRAY_OF_SIZES - number of elements in each dimension of full array
  - ARRAY_OF_SUBSIZES - number of elements in each dimension of sub-array
  - ARRAY_OF_STARTS - starting position in full array of sub-array in each dimension
  - ORDER - MPI_ORDER_(C or FORTRAN)
  - OLDTYPE - datatype stored in full array
  - NEWTYPE - handle to new datatype

NONCONTIGUOUS MEMORY ACCESS

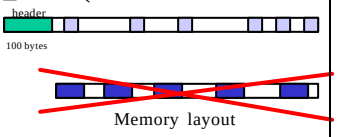


NONCONTIGUOUS MEMORY ACCESS

- INTEGER sizes(2), subsizes(2), starts(2), dtype, ierr
- sizes(1) = 102
- sizes(2) = 102
- subsizes(1) = 100
- subsizes(2) = 100
- starts(1) = 1
- starts(2)= 1
- CALL MPI_TYPE_CREATE_SUBARRAY(2,sizes,subsizes,starts, MPI_ORDER_FORTRAN,MPI_REAL8,dtype,ierr)

NONCONTIGUOUS FILE ACCESS

- MPI_FILE_SET_VIEW(
  - FH,
  - DISP,
  - ETYPE,
  - FILETYPE,
  - DATAREP,
  - INFO,
  - IERROR)

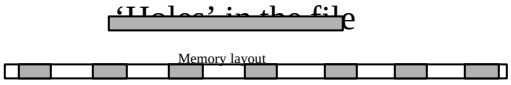


NONCONTIGUOUS FILE ACCESS

- The file has ‘holes’ in it from the processor’s perspective

NONCONTIGUOUS FILE ACCESS

- The file has ‘holes’ in it from the processor’s perspective
  - MPI_TYPE_CONTIGUOUS(NUM,OLD,NEW,IERR)
    - NUM - Number of contiguous elements
    - OLD - Old data type
    - NEW - New data type
  - MPI_TYPE_CREATE_RESIZED(OLD,LB,EXTENT, NEW, IERR)
    - OLD - Old data type
    - LB - Lower Bound
    - EXTENT - New size
    - NEW - New data type



File layout (2 ints followed by 3 ints)

```
CALL MPI_TYPE_CONTIGUOUS(2, MPI_INT, CTYPE, IERR)
DISP = 4
LB = 0
EXTENT=5*4
CALL MPI_TYPE_CREATE_RESIZED(CTYPE,LB,EXTENT,FTYPE,IERR)
CALL MPI_TYPE_COMMIT(FTYPE, IERR)
CALL MPI_FILE_SET_VIEW(FH,DISP,MPI_INT,FTYPE,’native’,MPI_INFO_NULL, IERR)
```

NONCONTIGUOUS FILE ACCESS

- The file has ‘holes’ in it from the processor’s perspective
- A block-cyclic data distribution

NONCONTIGUOUS FILE ACCESS

- The file has ‘holes’ in it from the processor’s perspective
- A block-cyclic data distribution
  - MPI_TYPE_VECTOR(
    - COUNT - Number of blocks
    - BLOCKLENGTH - Number of elements per block
    - STRIDE - Elements between start of each block
    - OLDTYPE - Old datatype
    - NEWTYPE - New datatype)

Block-cyclic distribution

File layout (blocks of 4 ints)

CALL MPI_TYPE_VECTOR(3, 4, 16, MPI_INT, FILETYPE, IERR)

CALL MPI_TYPE_COMMIT (FILETYPE, IERR)

DISP = 4 * 4 * MYRANK

CALL MPI_FILE_SET_VIEW (FH, DISP, MPI_INT, FILETYPE, 'native', MPI_INFO_NULL, IERR)

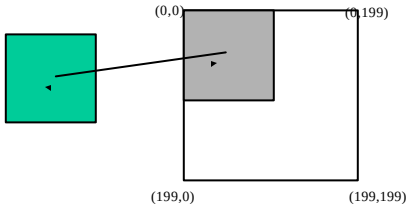
NONCONTIGUOUS FILE ACCESS

- The file has 'holes' in it from the processor's perspective
- A block-cyclic data distribution
- multi-dimensional array access

NONCONTIGUOUS FILE ACCESS

- The file has 'holes' in it from the processor's perspective
- A block-cyclic data distribution
- multi-dimensional array access
  - MPI_TYPE_CREATE_SUBARRAY()

Distributed array access



Distributed array access

Sizes(1) = 200

sizes(2) = 200

subsizes(1) = 100

subsizes(2) = 100

starts(1) = 0

starts(2) = 0

CALL MPI_TYPE_CREATE_SUBARRAY(2, SIZES, SUBSIZES, STARTS, MPI_ORDER_FORTRAN, MPI_INT, FILETYPE, IERR)

CALL MPI_TYPE_COMMIT(FILETYPE, IERR)

CALL MPI_FILE_SET_VIEW(FH, 0, MPI_INT, FILETYPE, 'NATIVE', MPI_INFO_NULL, IERR)

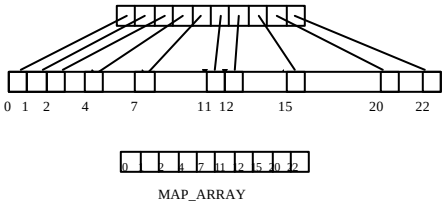
NONCONTIGUOUS FILE ACCESS

- The file has 'holes' in it from the processor's perspective
- A block-cyclic data distribution
- multi-dimensional array distributed with a block distribution
- Irregularly distributed arrays

Irregularly distributed arrays

- MPI_TYPE_CREATE_INDEXED_BLOCK
  - COUNT - Number of blocks
  - LENGTH - Elements per block
  - MAP - Array of displacements
  - OLD - Old datatype
  - NEW - New datatype

Irregularly distributed arrays



Irregularly distributed arrays

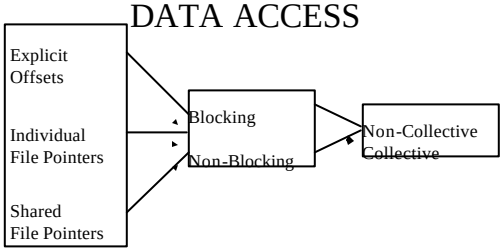
CALL MPI_TYPE_CREATE_INDEXED_BLOCK (10, 1, FILE_MAP, MPI_INT, FILETYPE, IERR)

CALL MPI_TYPE_COMMIT (FILETYPE, IERR)

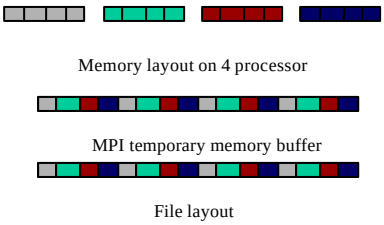
DISP = 0

CALL MPI_FILE_SET_VIEW (FH, DISP, MPI_INT, FILETYPE, 'native', MPI_INFO_NULL, IERR)

DATA ACCESS



COLLECTIVE I/O



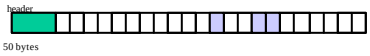
EXPLICIT OFFSETS

- Parameters
  - FH - File handle
  - OFFSET - Location in file to start
  - BUF - Buffer to write from/read to
  - COUNT - Number of elements
  - DATATYPE - Type of each element
  - STATUS - Return status (blocking)
  - REQUEST - Request handle (non-blocking,non-collective)

EXPLICIT OFFSETS (cont)

- I/O Routines
  - MPI_FILE_(READ/WRITE)_AT ()
  - MPI_FILE_(READ/WRITE)_AT_ALL ()
  - MPI_FILE_I(READ/WRITE)_AT ()
  - MPI_FILE_(READ/WRITE)_AT_ALL_BEGIN ()
  - MPI_FILE_(READ/WRITE)_AT_ALL_END (FH, BUF, STATUS)

EXPLICIT OFFSETS



50 bytes

```
int buff[3];

count = 5;
blocklen = 2;
stride = 4

MPI_Type_vector (count, blocklen,
    stride, MPI_INT, &ftype);
MPI_Type_commit (ftype);
```

```
disp = 58;
MPI_File_open (MPI_COMM_WORLD, filename,
    MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_set_view (fh, disp, MPI_INT, ftype, "native",
    MPI_INFO_NULL);
MPI_File_write_at (fh, 5, buff, 3, MPI_INT, &status);
```

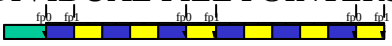
IDIVIDUAL FILE POINTERS

- Parameters
  - FH - File handle
  - BUF - Buffer to write to/read from
  - COUNT - number of elements to be read/written
  - DATATYPE - Type of each element
  - STATUS - Return status (blocking)
  - REQUEST - Request handle (non-blocking, non-collective)

INDIVIDUAL FILE POINTERS

- I/O Routines
  - MPI_FILE_(READ/WRITE) ()
  - MPI_FILE_(READ/WRITE)_ALL ()
  - MPI_FILE_I(READ/WRITE) ()
  - MPI_FILE_(READ/WRITE)_ALL_BEGIN()
  - MPI_FILE_(READ/WRITE)_ALL_END (FH, BUF, STATUS)

INDIVIDUAL FILE POINTERS




```
int buff[12];

count = 6;
blocklen = 2;
stride = 4

MPI_Type_vector (count, blocklen,
    stride, MPI_INT, &ftype);
MPI_Type_commit (ftype);
```

```
disp = 50 + myrank*8;
MPI_File_open (MPI_COMM_WORLD, filename,
    MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
MPI_File_set_view (fh, disp, MPI_INT, ftype, "native",
    MPI_INFO_NULL);
MPI_File_write(fh, buff, 6, MPI_INT, &status);
MPI_File_write(fh, buff, 6, MPI_INT, &status);
```

SHARED FILE POINTERS

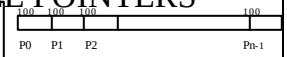
- All processes must have the same view
- Parameters
  - FH - File handle
  - BUF - Buffer
  - COUNT - Number of elements
  - DATATYPE - Type of the elements
  - STATUS - Return status (blocking)
  - REQUEST - Request handle (Non-blocking, non-collective

SHARED FILE POINTERS

- I/O Routines
  - MPI_FILE_(READ/WRITE)_SHARED ()
  - MPI_FILE_I(READ/WRITE)_SHARED ()
  - MPI_FILE_(READ/WRITE)_ORDERED ()
  - MPI_FILE_(READ/WRITE)_ORDERED_BEGIN ()
  - MPI_FILE_(READ/WRITE)_ORDERED_END (FH, BUF, STATUS)

SHARED FILE POINTERS

```
comm = MPI_COMM_WORLD;
MPI_Comm_rank (comm, &rank);
amode = MPI_MODE_CREATE |
  MPI_MODE_WRONLY;
.....
MPI_File_open (comm, logfile, amode,
  MPI_INFO_NULL, &fh);
.....
do some computing
if (some event occurred) {
 sprintf(buf, "Process %d: %s\n", rank, event);
 size = strlen(buf);
 MPI_File_write_shared (fh, buf, size
   MPI_CHAR, &status);
}
.....
```




```
int buff[100];

MPI_File_open (comm, logfile, amode,
   MPI_INFO_NULL, &fh);
MPI_File_write_ordered (fh, buf, 100,
   MPI_INT, &status);
```

FILE INTEROPERABILITY

- MPI puts no constraints on how an implementation should store files
- If a file is not stored as a linear byte stream, there must be a utility for converting the file into a linear byte stream
- Data representation aids interoperability

FILE INTEROPERABILITY (cont)

- Data Representation
  - Native - Data stored exactly as it is in memory.
  - Internal - Data may be converted, but can always be read by the same MPI implementation, even on different architectures
  - external32 - This representation is defined by MPI. Files written in external32 format can be read by any MPI on any machine

FILE INTEROPERABILITY (cont)

- Some MPI-I/O implementations (Romio), created files are no different than those created by the underlying file system.
- This means normal Posix commands (cp, rm, etc) work with files created by these implementations
- Non-MPI programs can read these files

GOTCHAS - Consistency & Semantics

- Collective routines are NOT synchronizing
- Output data may be buffered
  - Just because a process has completed a write does not mean the data is available to other processes
- Three ways to ensure file consistency:
  - MPI_FILE_SET_ATOMICITY ()
  - MPI_FILE_SYNC ()
  - MPI_FILE_CLOSE ()

CONSISTENCY & SEMANTICS

- MPI_FILE_SET_ATOMICITY ()
  - Causes all writes to be immediately written to disk. This is a collective operation
- MPI_FILE_SYNC ()
  - Collective operation which forces buffered data to be written to disk
- MPI_FILE_CLOSE ()
  - Writes any buffered data to disk before closing the file

GOTCHA!!!

CALL MPI_FILE_OPEN (…, FH)	CALL MPI_FILE_OPEN (…, FH)
CALL MPI_FILE_SET_ATOMICITY (FH)	CALL MPI_FILE_SET_ATOMICITY (FH)
CALL MPI_FILE_WRITE_AT (FH, 0, …)	CALL MPI_FILE_WRITE_AT (FH, 100, …)
CALL MPI_FILE_READ_AT (FH, 100, …)	CALL MPI_FILE_READ_AT (FH, 0, …)

GOTCHA!!!

CALL MPI_FILE_OPEN (…, FH)	CALL MPI_FILE_OPEN (…, FH)
CALL MPI_FILE_SET_ATOMICITY (FH)	CALL MPI_FILE_SET_ATOMICITY (FH)
CALL MPI_FILE_WRITE_AT (FH, 0, …)	CALL MPI_FILE_WRITE_AT (FH, 100, …)
CALL MPI_BARRIER ()	CALL MPI_BARRIER ()
CALL MPI_FILE_READ_AT (FH, 100, …)	CALL MPI_FILE_READ_AT (FH, 0, …)

GOTCHA!!!

CALL MPI_FILE_OPEN (…, FH)	CALL MPI_FILE_OPEN (…, FH)
CALL MPI_FILE_WRITE_AT (FH, 0, …)	CALL MPI_FILE_WRITE_AT (FH, 100, …)
CALL MPI_FILE_CLOSE (FH)	CALL MPI_FILE_CLOSE (FH)
CALL MPI_FILE_OPEN (…, FH)	CALL MPI_FILE_OPEN (…, FH)
CALL MPI_FILE_READ_AT (FH, 100, …)	CALL MPI_FILE_READ_AT (FH, 0, …)

GOTCHA!!!

CALL MPI_FILE_OPEN (…, FH)	CALL MPI_FILE_OPEN (…, FH)
CALL MPI_FILE_WRITE_AT (FH, 0, …)	CALL MPI_FILE_WRITE_AT (FH, 100, …)
CALL MPI_FILE_CLOSE (FH)	CALL MPI_FILE_CLOSE (FH)
CALL MPI_BARRIER ()	CALL MPI_BARRIER ()
CALL MPI_FILE_OPEN (…, FH)	CALL MPI_FILE_OPEN (…, FH)
CALL MPI_FILE_READ_AT (FH, 100, …)	CALL MPI_FILE_READ_AT (FH, 0, …)

CONCLUSIONS

- MPI-I/O potentially offers significant improvement in I/O performance
- This improvement can be attained with minimal effort on part of the user
  - Simpler programming with fewer calls to I/O routines
  - Easier program maintenance due to simple API

Recommended references

- MPI - The Complete Reference Volume 1, The MPI Core
- MPI - The Complete Reference Volume 2, The MPI Extensions
- USING MPI: Portable Parallel Programming with the Message-Passing Interface
- Using MPI-2: Advanced Features of the Message-Passing Interface