MPI-2 Parallel I/O

Based on notes by Sathish Vadhiyar, Rob Thacker, and
David Cronk

Motivation

- High level parallel I/O interface
- Supports file partitioning among processes
- Transfer of data structures between process memories and files
- Asynchronous/non-blocking I/O
- Strided / Non-contiguous access
- Collective I/O

- Far too many issues to put into one lecture – plenty of web resources provide more details if you need them
- Draws heavily from Using MPI-2

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
  - 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

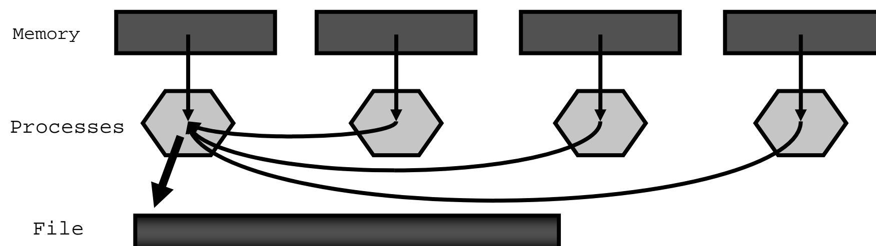
INTRODUCTION (cont)

- 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

- Makes extensive use of derived datatypes
- Allows non-contiguous memory and or disk data to be read/written with a single I/O call
- Can simplify programming and maintenance
- May allow for overlap of computation and I/O
- Collective I/O allows for performance optimization through marshaling
- Significant performance improvement is possible

Non-parallel I/O

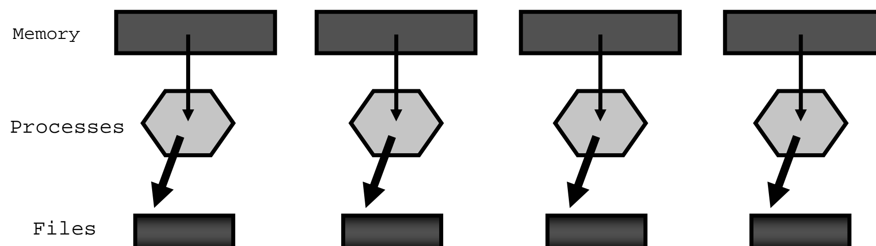


- Simplest way to do I/O – number of factors may contribute to this kind of model:
  - May have to implement this way because only one process is capable of I/O
  - May have to use serial I/O library
  - I/O may be enhanced for large writes
  - Easiest file handling – only one file to keep track of
- Arguments against:
  - Strongly limiting in terms of throughput if the underlying file system does permit parallel I/O

Additional argument for parallel I/O standard

- Standard UNIX I/O is not portable
  - Endianess becomes serious problem across different machines
    - Writing wrappers to perform byte conversion is tedious

Simple parallel I/O



- Multiple independent files are written
  - Same as parallel I/O allowed under OpenMP
  - May still be able to use sequential I/O libraries
  - Significant increases in throughput are possible
- Drawbacks are potentially serious
  - Now have multiple files to keep track of (concatenation may not be an option)
  - May be non-portable if application reading these files must have the same number of processes

Simple parallel I/O (no MPI calls)

- I/O operations are completely independent across the processes
- Append a rank to each file name to specify the different files
- Need to be very careful about performance – individual files may be too small for good throughput

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100

Int main(int argc, char *argv[])
{
   int i,myrank,buf[BUFSIZE];
   char filename[128];
   FILE *myfile;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   for (i=0; i< BUFSIZE; i++)
      buf[i] = myrank * BUFSIZE + I;
   sprintf(filename,"testfile.%d",myrank);
   myfile= fopen(filename, "w");
   fwrite(buf,sizeof(int), BUFSIZE, myfile);
   fclose(myfile);
   MPI_Finalize();
   return 0;
}
```


Simple parallel I/O (using MPI calls)

- Rework previous example to use MPI calls
- Note the file pointer has been replaced by a variable of type MPI_File
- Under MPI I/O open, write (read) and close statements are provided
- Note MPI_COMM_SELF denotes a communicator over local process

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100

Int main(int argc, char *argv[])
{
   int i,myrank,buf[BUFSIZE];
   char filename[128];
   MPI_File myfile;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   for (i=0; i< BUFSIZE; i++)
      buf[i] = myrank * BUFSIZE + i;
   sprintf(filename,"testfile.%d",myrank);
   MPI_File_open(MPI_COMM_SELF, filename,
                 MPI_MODE_WRONLY | MPI_MODE_CREATE,
                 MPI_INFO_NULL, &myfile);
   MPI_File_write(myfile, buf, BUFSIZE, MPI_INT,
                  MPI_STATUS_IGNORE);
   MPI_File_close(&myfile);
   MPI_Finalize();
   return 0;
}
```

MPI_File_open arguments

- MPI_File_open(comm,filename,accessmode,info,filehandle,ierr)
  - Comm – can choose immediately whether file access is collective or local
    - MPI_COMM_WORLD or MPI_COMM_SELF
  - Access mode is specified by MPI_MODE_CREATE and MPI_MODE_WRONLY are or'd together
    - Same as Unix open
  - The file handle is passed back from this call to be used later in MPI_File_write

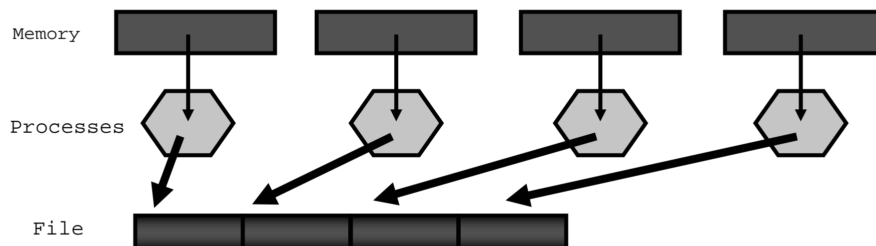
MPI_File_write

- MPI_File_write(filehandler,buff,count,datatype,status,ierr)
  - Very similar to message passing interface
    - Imagine file handler as providing destination
    - "Address,count,datatype" interface
  - Specification of non-contiguous writes would be done via a user-defined datatype
  - MPI_STATUS_IGNORE can be passed in the status field
    - Informs system not to fill field since user will ignore it
      - May slightly improve I/O performance when not needed

ROMIO ("romeo")

- Publically available implementation of the MPI-I/O instruction set
  - http://www-unix.mcs.anl.gov/romio
- Runs on multiple platforms
- Optimized for non-contiguous access patterns
  - Common for parallel applications
- Optimized for collective operations

True parallel MPI I/O



- Processes must all now agree on opening a single file
  - Each process must have its own pointer within the file
  - File clearly must reside on a single file system
  - Can read the file with a different number of processes as compared to what it was written with

Parallel I/O to a single file using MPI calls

- Rework previous example to write to a single file
- Write is now *collective*
  - MPI_COMM_SELF has been replaced by MPI_COMM_WORLD
- All files agree on a collective name for the file “testfile”
- Access to given parts of the file is specifically controlled
  - MPI_File_set_view
  - Shift displacements according to local rank

```
#include "mpi.h"
#include <stdio.h>
#define BUFSIZE 100

Int main(int argc, char *argv[])
{
   int i,myrank,buf[BUFSIZE];
   MPI_File thefile;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   for (i=0; i< BUFSIZE; i++)
      buf[i] = myrank * BUFSIZE + i;
   MPI_File_open(MPI_COMM_WORLD, "testfile",
                 MPI_MODE_WRONLY | MPI_MODE_CREATE,
                 MPI_INFO_NULL, &thefile);
   MPI_File_set_view(thefile,
                     myrank*BUFSIZE*sizeof(int),
                     MPI_INT,MPI_INT, "native",
                     MPI_INFO_NULL);
   MPI_File_write(thefile,buf, BUFSIZE, MPI_INT,
                  MPI_STATUS_IGNORE);
   MPI_File_close(&thefile);
   MPI_Finalize();
   return 0;
}
```


File view

- *File view* defines which portion of a file is “visible” to a given process
  - On first opening a file the entire file is visible
  - Data is described at the byte level initially
- MPI_File_set_view provides information to
  - enable reading of the data (datatypes)
  - Specify which parts of the file should be skipped

MPI_File_set_view

- MPI_File_set_view(filehandler,displ,etype,filedatatype,datarep,info,ierr)
  - Displ controls the BYTE offset from the beginning of the file
  - The displacement is of a type “MPI_Offset” larger than normal MPI_INT to allow for 64 bit addressing (byte offsets can easily exceed 32 bit limit)
  - etype is the datatype of the buffer
  - filedatatype is the corresponding datatype in the file, which must be either etype, or derived from it
  - Datarep
    - native: data is stored in the file as in memory
    - internal: implementation specific format that may provide a level or portability
    - external32: 32bit big endian IEEE format (defined for all MPI implementations)
      - Only use if portability is required (conversion may be necessary)

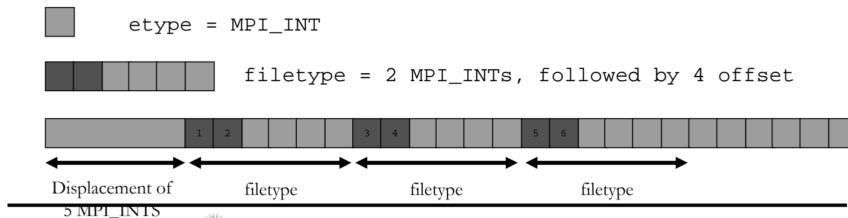
Definitions

- Displacement – file position from the beginning of a file
- etype – unit of data access
- filetype – template for accessing the file
- view – current set of data accessible by a process. Repetitions of filetype pattern define a view
- offset – position relative to the current view

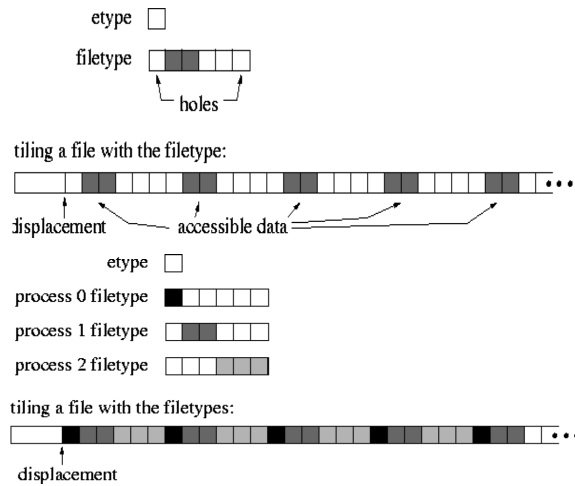
etype and filetype in action

- Suppose we have a buffer with an etype of MPI_INT
- Filetype is defined to be 2 MPI_INTs followed by an offset of 4 MPI_INTS (extent=6)

```
MPI_File_set_view(fh,5*sizeof(int),etype,filetype,"native",MPI_INFO_NULL)
MPI_File_write(fh,buf,1000,MPI_INT,MPI_STATUS_IGNORE)
```



Examples



Reading a single file with an unknown number of processors

- New function:
  - MPI_File_get_size
  - Returns size of open file, need to use 64 bit int (MPI_Offset)
- Check how much data has been read by using status handler
- Pass to MPI_Get_count
  - Determines number of datatypes that were read
- If number of read items is less than expected then (hopefully) EOF is reached

```
#include “mpi.h”
#include <stdio.h>

int main(int argc, char *argv[])
{
   int myrank,numprocs,bufsize,*buf,count;
   MPI_File thefile;
   MPI_Status status;
   MPI_Offset filesize;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_File_open(MPI_COMM_WORLD, “testfile”,
                 MPI_MODE_RDONLY,MPI_INFO_NULL,
                 &thefile);
   MPI_File_get_size(thefile,&filezize);
   filesize=filesize/sizeof(int);
   bufsize=filesize/(numprocs+1);
   buf=(int *) malloc(bufsize*sizeof(int));
   MPI_File_set_view(thefile,
                     myrank*bufsize*sizeof(int),
                     MPI_INT,MPI_INT,”native”,
                     MPI_INFO_NULL);
   MPI_File_read(thefile,buf, bufsize, MPI_INT,
                  &status);
   MPI_Get_count(&status, MPI_INT, &count);
   printf(“process %d read %d
           ints\n”,myrank,count);
   MPI_File_close(&thefile);
   MPI_Finalize();
   return 0;
}
```


Using explicit offsets

- MPI_File_read/write are *individual-file-pointer functions*
  - Both use current location of pointer to determine where to read/write
  - Must perform a seek to arrive at the correct region of data
- *Explicit offset functions* don’t use an individual file pointer
  - File offset is passed directly as an argument to the function
  - Seek is not required
  - MPI_File_read/write_at
  - Must use this version in multithreaded environment

Revised code for explicit offsets

- No need to apply seeks
- Specific movement of file pointer is not applied, instead offset is passed as argument
- Remember offsets must be of kind MPI_OFFSET_KIND
- Same issue here with precalculating offset and resolving into a variable with the appropriate typing

```
PROGRAM main
   include 'mpif.h

   integer FILESIZE,MAX_BUFSIZE,INTSIZE
   parameter (FILESIZE=1048576, MAX_BUFSIZE=1048576)
   parameter (INTSIZE=4)
   integer buf(MAX_BUFSIZE),rank,ierr,fh,nprocs,nints
   integer status(MPI_STATUS_SIZE), count
   integer(kind=MPI_OFFSET_KIND) offset

   call MPI_Init(ierr)
   call MPI_Comm_rank(MPI_COMM_WORLD,rank,ierr)
   call MPI_Comm_size(MPI_COMM_WORLD,nprocs,ierr)
   call MPI_File_open(MPI_COMM_WORLD, 'testfile',&
               MPI_MODE_RDONLY,&
               MPI_INFO_NULL,fh,ierr)
   nints=FILESIZE/(nprocs*INTSIZE)
   offset=rank*nints*INTSIZE
   call MPI_File_read_at(fh,offset, buf, nints, &
             MPI_INTEGER,status,ierr)
   call MPI_Get_count(status,MPI_INTEGER,count,ierr)
   print*, 'process ',rank,'read ',count,'ints'
   call MPI_File_close(fh,ierr)
   call MPI_Finalize(ierr);

END PROGRAM main
```


Dealing with multidimensional arrays

- Storage formats differ for C versus fortran
  - row major versus column major
- MPI_Type_create_darray, and also MPI_Type_create_subarray are used to specify derived datatypes
  - These datatypes can then be used to resolve local regions within a linearized global array
  - Specifically they deal with the noncontiguous nature of the domain decomposition
  - See Using MPI-2 book for more details

Summary

- MPI I/O provides a straightforward interface for performing parallel I/O
  - Single file output is supported via multiple file pointers into the same file
  - Multiple output files may also be written
- Can use explicit pointer based schemes, or alternatively use file views to specify local access

File Info Object

- Can provide hints to the implementation to improve I/O performance
- MPI_FILE_SET_INFO(fh, info)
- MPI_FILE_GET_INFO(fh, info_used)
- Default hints for file access patterns and data layout

Default Hints (Info values)

- **{** access_style} (comma separated list of strings)
  - read_once, write_once, read_mostly, write_mostly, sequential, reverse_sequential, and random
- **{** collective_buffering} (boolean)
- **{** cb_buffer_size} (integer)
- **{** cb_nodes} (integer)
- **{** filename} (string)
- **{** file_perm} (string)
- **{** io_node_list} (comma separated list of strings)
- **{** nb_proc} (integer)
- **{** num_io_nodes} (integer)
- **{** striping_factor} (integer)
- **{** striping_unit} (integer)

API

Positioning	Synchronism	Coordination Non-Collective	Coordination Collective
Explicit offsets	Blocking	MPI_FILE_READ_AT	MPI_FILE_READ_AT_ALL
	Non-blocking	MPI_FILE_IREAD_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END
Individual file pointers	Blocking	MPI_FILE_READ	MPI_FILE_READ_ALL
	Non-blocking	MPI_FILE_IREAD	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END
Shared file pointers	Blocking	MPI_FILE_READ_SHARED	MPI_FILE_READ_ORDERED
	Non-blocking	MPI_FILE_IREAD_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END