Part I: Introduction to parallel I/O

By Mary Thomas
Department of Computer Science, SDSU

- Lecture based on
  - CSC Center for Science workshop on Advanced Parallel Programming (2008)
    - https://www.csc.fi/web/training
  - C code examples can be found on tuckoo in /COMP696:
    - gropp/www.mcs.anl.gov/usingmpi/examples-advmpi/parallelio
Parallel I/O software stack

- Application/Library
  - NetCDF
- Middle-ware
  - HDF5
  - MPI I/O
  - POSIX I/O
- Kernel
  - GPFS
  - Lustre
  - NFS
Motivation

• Study by LLNL (2005):
  – 1 GB/s I/O bandwidth required per Teraflop compute capability
  – Write to the files system dominates reading from it by a factor of 5

• Current High-End Systems:
  – K Computer: ~11 PFLOPS, ~96 GB/s I/O bandwidth using 864 OSTs
  – Jaguar (2010): ~1 PFLOPS, ~90 GB/s I/O bandwidth using 672 OSTs

• Key issue: what is the local file system architecture and what are its networks and bandwidths?

➤ Gap between available I/O performance and required I/O performance.
Parallel I/O

• I/O (Input/output) is needed in all programs but is often overlooked
Parallel I/O

• Mapping problem: how to convert internal structures and domains to files which are a streams of bytes

• Transport problem: how to get the data efficiently from hundreds to thousands of nodes on the supercomputer to physical disks

...11011010101011
01110110010101010
101001010101...

...
Parallel I/O

• Good I/O is non-trivial
  – Performance, scalability, reliability
  – Ease of use of output (number of files, format)

• Portability

• One cannot achieve all of the above - one needs to decide what is most important
Parallel I/O

- New challenges
  - Number of tasks is rising rapidly
  - The size of the data is also rapidly increasing
- The need for I/O tuning is algorithm & problem specific
- Without parallelization, I/O will become scalability bottleneck for practically every application!
I/O layers

- High-level
  - application: need to write or read data from disk
- Intermediate
  - libraries or system tools for I/O
  - high-level libraries (HDF5, NetCDF,...)
  - MPI-I/O
  - POSIX system calls (fwrite / WRITE)
- Low-level
  - parallel filesystem enables the actual parallel I/O
  - Lustre, GPFS, PVFS, dCache
Part II: Parallel I/O with Posix calls

See my lecture on Pthreads:
POSIX: Portable Operating System Interface

• standardized user command line and scripting interface
• based on the Bourne Shell
• user-level programs, services, and utilities including awk, echo, ed
• program-level services including basic I/O (file, terminal, and network)
• defines standard for threading lib & API
POSIX I/O

• Built in language constructs for performing I/O
  – WRITE/READ/OPEN/CLOSE in Fortran
  – stdio.h routines in C (fopen, fread, fwrite, ...)
• No parallel ability built in - all parallel I/O schemes have to be programmed manually
• Binary output not necessarily portable
POSIX I/O

- C and Fortran binary output not necessarily compatible
- Non-contiguous access difficult to implement efficiently
- Contiguous access can be very fast
Part II: Parallel I/O with Posix calls

- POSIX I/O
  - Built in language constructs for performing I/O: WRITE/READ/OPEN/CLOSE in Fortran
  - stdio.h routines in C (fopen, fread, fwrite, ...)
  - No parallel ability built in - all parallel I/O schemes have to be programmed manually
  - Binary output not necessarily portable

- POSIX I/O
  - C and Fortran binary output not necessarily compatible
  - Non-contiguous access difficult to implement efficiently
  - Contiguous access can be very fast

Parallel POSIX I/O

- Spokesman strategy
  - One process takes care of all I/O using normal (POSIX) routines
  - Requires a lot of communication
  - Writing/reading slow, single writer not able to fully utilize filesystem
  - Does not scale, single writer is a bottleneck
  - Can be good option when the amount of data is small (e.g. input files)

Parallel POSIX I/O

- Every man for himself
  - Each process writes its local results to a separate file
  - Difficult to handle a huge number of files in later analysis
  - Can overwhelm filesystem

Parallel POSIX I/O

- Subset of writers/readers
  - Good compromise
  - Most difficult to implement
  - Number of readers/writers often chosen to be sqrt(N)
  - If readers/writers are dedicated then some computational resources are wasted

Part III: Parallel I/O with MPI

- MPI I/O
  - MPI I/O was introduced in MPI-2
  - Defines parallel operations for reading and writing files
    - I/O to only one file and/or to many files
    - Contiguous and non-contiguous I/O
    - Individual and collective I/O
    - Asynchronous I/O
  - Portable programming interface
Parallel POSIX I/O

• Every man for himself
  - Each process writes its local results to a separate file
  - Good bandwidth
  - Difficult to handle a huge number of files in later analysis
  - Can overwhelm filesystem
Parallel POSIX I/O

- Subset of writers/readers
  - Good compromise
  - Most difficult to implement
  - Number of readers/writers often chosen to be sqrt(N)
  - If readers/writers are dedicated then some computational resources are wasted
Part II: Parallel I/O with Posix calls

POSIX I/O

- Built in language constructs for performing I/O
  - WRITE/READ/OPEN/CLOSE in Fortran
  - stdio.h routines in C (fopen, fread, fwrite, ...)
- No parallel ability built in - all parallel I/O schemes have to be programmed manually
- Binary output not necessarily portable

POSIX I/O

- C and Fortran binary output not necessarily compatible
- Non-contiguous access difficult to implement efficiently
- Contiguous access can be very fast

Parallel POSIX I/O

- Spokesman strategy
  - One process takes care of all I/O using normal (POSIX) routines
  - Writing/reading slow, single writer not able to fully utilize filesystem
  - Does not scale, single writer is a bottleneck
  - Can be good option when the amount of data is small (e.g. input files)

Parallel POSIX I/O

- Every man for himself
  - Each process writes its local results to a separate file
  - Good compromise
  - Most difficult to implement
  - Number of readers/writers often chosen to be sqrt(N)
  - If readers/writers are dedicated then some computational resources are wasted

Part III: Parallel I/O with MPI

MPI I/O

- MPI I/O was introduced in MPI-2
- Defines parallel operations for reading and writing files
  - I/O to only one file and/or to many files
  - Contiguous and non-contiguous I/O
  - Individual and collective I/O
  - Asynchronous I/O
- Portable programming interface
MPI I/O

• MPI I/O was introduced in MPI-2
• Defines parallel operations for reading and writing files
  – I/O to only one file and/or to many files
  – Contiguous and non-contiguous I/O
  – Individual and collective I/O
  – Asynchronous I/O
• Portable programming interface
MPI I/O

- Potentially good performance
- Easy to use (compared with implementing the same algorithms on your own)
- Used as the backbone of many parallel I/O libraries such as parallel NetCDF and parallel HDF5
- By default, binary files are not necessarily portable
Basic concepts in MPI I/O

• File handle
  – data structure which is used for accessing the file

• File pointer
  – position in the file where to read or write
  – can be individual for all processes or shared between the processes
  – accessed through file handle
Basic concepts in MPI I/O

• File view
  – part of a file which is visible to process
  – enables efficient noncontiguous access to file

• Collective and independent I/O
  – collective: MPI coordinates the reads and writes of processes
  – independent: no coordination by MPI
 MPI I/O: Open/Close file

- All processes in a communicator open a file using
  \texttt{MPI\_File\_open(comm, filename, mode, info, fhandle)}

  - \texttt{comm}: communicator that performs parallel I/O
  - \texttt{mode}: \texttt{MPI\_MODE\_RDONLY}, \texttt{MPI\_MODE\_WRONLY}, \texttt{MPI\_MODE\_CREATE}, \texttt{MPI\_MODE\_RDWR}, ...
    Can be combined with + in Fortran, | in C/C++
  - \texttt{info}: Hints to implementation for optimal performance (No hints: \texttt{MPI\_INFO\_NULL})
  - \texttt{fhandle}: parallel file handle

**Basic concepts in MPI I/O**

- **File handle**: data structure used for accessing the file
- **File pointer**: position in the file where to read or write
  - Individual for all processes or shared
  - Accessed through file handle

- **File view**: part of a file visible to process
  - Enables efficient noncontiguous access to file

- **Collective and independent I/O**
  - **Collective**: MPI coordinates the reads and writes of processes
  - **Independent**: no coordination by MPI

**MPI I/O: Open/Close file**

- File is closed using \texttt{MPI\_File\_close(fhandle)}

**MPI I/O: Read file**

- File opened with \texttt{MPI\_MODE\_RDONLY} or \texttt{MPI\_MODE\_RDWR}
- Each process moves its local file pointer (individual file pointer) with \texttt{MPI\_File\_seek(fhandle, disp, whence)}

  - \texttt{disp}: Displacement in bytes (with default file view)
  - \texttt{whence}: \texttt{MPI\_SEEK\_SET}, \texttt{MPI\_SEEK\_CUR}, \texttt{MPI\_SEEK\_END}

- Read file at individual file pointer
  \texttt{MPI\_File\_read(fhandle, buf, count, datatype, status)}

  - \texttt{buf}: Buffer in memory where to read the data
  - \texttt{count}: number of elements to read
  - \texttt{datatype}: datatype of elements to read
  - \texttt{status}: similar to status in \texttt{MPI\_Recv}

- The location to read can be determined also within the read statement (explicit offset)
  \texttt{MPI\_File\_read\_at(fhandle, disp, buf, count, datatype, status)}

  - \texttt{disp}: displacement in bytes (with the default file view)
    from the beginning of file

- Thread-safe
- The file pointer is neither used or incremented

**Updates position of file pointer after reading**

- Not thread-safe

**Updates position of file pointer after reading**
MPI I/O: Open/Close file

• File is closed using
  MPI_File_close(fhandle)
MPI I/O: Read file

• File opened with MPI_MODE_RDONLY or MPI_MODE_RDWR
• Each process moves its local file pointer (individual file pointer) with

 MPI_File_seek(fhandle, disp, whence)
  disp  Displacement in bytes (with default file view)
  whence MPISEEK_SET, MPISEEK_CUR, MPISEEK_END
MPI I/O: Read file

• Read file at individual file pointer

```c
MPI_File_read(fhandle, buf, count, datatype, status)
```

- `buf` Buffer in memory where to read the data
- `count` number of elements to read
- `datatype` datatype of elements to read
- `status` similar to status in MPI_Recv, amount of data read can be determined by MPI_Get_count

- Updates position of file pointer after reading
- Not thread safe
MPI I/O: Read file

• The location to read can be determined also within the read statement (explicit offset)

\[
\text{MPI\_File\_read\_at}(\text{fhandle, disp, buf, count, datatype, status})
\]

- disp displacement in bytes (with the default file view) from the beginning of file

- Thread-safe

- The file pointer is neither used or incremented
MPI I/O: Write file

- Similar to reading
  - File opened with MPI_MODE_WRONLY or MPI_MODE_CREATE

- Write file at individual file pointer
  
  ```
  MPI_File_write(fhandle, buf, count, datatype, status)
  ```

  - Updates position of file pointer after writing
  - Not thread safe
MPI I/O: Write file

• Determine location within the write statement (explicit offset)
  MPI_File_write_at(fhandle, disp, buf, count, datatype, status)
  - Thread-safe
  - The file pointer is neither used or incremented
Example: parallel write & read

PROGRAM Output
USE MPI
IMPLICIT NONE
INTEGER :: err, i, myid, file, intsize
INTEGER :: status(MPI_STATUS_SIZE)
INTEGER, PARAMETER :: count=100
INTEGER, DIMENSION(count) :: buf
INTEGER(KIND=MPI_OFFSET_KIND) :: disp
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid,&
   err)
DO i = 1, count
   buf(i) = myid * count + i
END DO
...
MPI I/O: Write file

- Similar to reading
  ```
  FILE opened with MPI_MODE_WRONLY or MPI_MODE_CREATE
  ```
- Write file at individual file pointer:
  ```
  MPI_File_write(fhandle, buf, count, datatype, status)
  ```
  - Updates position of file pointer after writing
  - Not thread safe

- Determine location within the write statement (explicit offset)
  ```
  MPI_File_write_at(fhandle, disp, buf, count, datatype, status)
  ```
  - Thread-safe
  - The file pointer is neither used or incremented

Example: parallel write

```fortran
PROGRAM Output
  USE MPI
  IMPLICIT NONE
  INTEGER :: err, i, myid, file, intsize
  INTEGER :: status(MPI_STATUS_SIZE)
  INTEGER, PARAMETER :: count=100
  INTEGER DIMENSION(count) :: buf
  INTEGER(KIND=MPI_OFFSET_KIND) :: disp
  CALL MPI_INIT(err)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid,&
  CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test', &
  CALL MPI_TYPE_SIZE(MPI_INTEGER, intsize, err)
  disp = myid * count * intsize
  CALL MPI_FILE_SEEK(file, disp,&
  CALL MPI_FILE_WRITE(file, buf, count, &
  CALL MPI_FILE_CLOSE(file, err)
END PROGRAM Output
```

Note: File (and total data) size depends on number of processes in this example

File offset determined by MPI_File_seek
Example: parallel read

Note: Same number of processes for reading and writing is assumed in this example.

... 
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test', & 
MPI_MODE_RDONLY, & 
MPI_INFO_NULL, file, err) 
CALL MPI_TYPE_SIZE(MPI_INTEGER, intsize,err) 
disp = myid * count * intsize 
CALL MPI_FILE_READ_AT(file, disp, buf, & 
count, MPI_INTEGER, status, err) 
CALL MPI_FILE_CLOSE(file, err) 
CALL MPI_FINALIZE(err) 
END PROGRAM Output
/* read from a common file using individual file pointers */
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define FILESIZE (1024 * 1024)

int main(int argc, char **argv) {
 int *buf, rank, nprocs, nints, bufsize;
 MPI_File fh;
 MPI_Status status;

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

 bufsize = FILESIZE/nprocs;
 buf = (int *) malloc(bufsize);
 nints = bufsize/sizeof(int);

 MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", MPI_MODE_RDONLY,MPI_INFO_NULL, &fh);
 MPI_File_seek(fh, rank*bufsize, MPI_SEEK_SET);
 MPI_File_read(fh, buf, nints, MPI_INT, &status);
 MPI_File_close(&fh);

 free(buf);
 MPI_Finalize();
 return 0;  }
Collective operations

• I/O can be performed collectively by all processes in a communicator
  – MPI_File_read_all
  – MPI_File_write_all
  – MPI_File_read_at_all
  – MPI_File_write_at_all

• Same parameters as in independent I/O functions (MPI_File_read etc)
Collective operations

• All processes in communicator that opened file must call function

• Performance potentially better than for individual functions
  – Even if each processor reads a non-contiguous segment, in total the read is contiguous
/ * noncontiguous access with a single collective I/O function *
#include "mpi.h"
#include <stdlib.h>
#define FILESIZE 1048576
#define INTS_PER_BLK 16

int main(int argc, char **argv) {
  int *buf, rank, nprocs, nints, bufsize;
  MPI_File fh;
  MPI_Datatype filetype;

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

  bufsize = FILESIZE/nprocs;
  buf = (int *) malloc(bufsize);
  nints = bufsize/sizeof(int);

  MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
  MPI_Type_vector(nints/INTS_PER_BLK, INTS_PER_BLK, INTS_PER_BLK*nprocs, MPI_INT, &filetype);
  MPI_Type_commit(&filetype);
  MPI_File_set_view(fh, INTS_PER_BLK*sizeof(int)*rank, MPI_INT, filetype, "native", MPI_INFO_NULL);

  MPI_File_read_all(fh, buf, nints, MPI_INT, MPI_STATUS_IGNORE);
  MPI_File_close(&fh);

  MPI_Type_free(&filetype);
  free(buf);
  MPI_Finalize();
  return 0;
}
File view

- A file view defines which portion of a file is “visible” to a process
- File view defines also the type of the data in the file (byte, integer, float, ...)

```
1  2  3  4  5  6  7  8
```

<table>
<thead>
<tr>
<th>Default file view</th>
</tr>
</thead>
<tbody>
<tr>
<td>etype=MPI_INT</td>
</tr>
<tr>
<td>filetype=MPI_INT</td>
</tr>
</tbody>
</table>

```
1  3  5  7
```

| etype=MPI_INT      |
| filetype=MPI_Type_vector(4, 1, 2, MPI_INT, &filetype); |

Example: parallel write & read

Example: parallel write

```cpp
PROGRAM Output
USE MPI
IMPLICIT NONE
INTEGER :: err, i, myid, file, intsize
INTEGER :: status(MPI_STATUS_SIZE)
INTEGER, PARAMETER :: count=100
INTEGER, DIMENSION(count) :: buf
INTEGER(KIND=MPI_OFFSET_KIND) :: disp
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, myid, err)
DO i = 1, count
  buf(i) = myid * count + i
END DO
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test', MPI_MODE_WRONLY, MPI_MODE_CREATE, MPI_INFO_NULL, file, err)
CALL MPI_TYPE_SIZE(MPI_INTEGER, intsize, err)
disp = myid * count * intsize
CALL MPI_FILE_SEEK(file, disp, MPI_SEEK_SET, err)
CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)
CALL MPI_FILE_CLOSE(file, err)
CALL MPI_FINALIZE(err)
END PROGRAM Output
```

File offset determined by `MPI_File_seek`...

Example: parallel read

```cpp
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test', MPI_MODE_RDONLY, MPI_INFO_NULL, file, err)
CALL MPI_TYPE_SIZE(MPI_INTEGER, intsize, err)
disp = myid * count * intsize
CALL MPI_FILE_READ_AT(file, disp, buf, count, MPI_INTEGER, status, err)
CALL MPI_FILE_CLOSE(file, err)
CALL MPI_FINALIZE(err)
END PROGRAM Output
```

Collective operations

- I/O can be performed collectively by all processes in a communicator
  - `MPI_File_read_all`
  - `MPI_File_write_all`
  - `MPI_File_read_at_all`
  - `MPI_File_write_at_all`

- Same parameters as in independent I/O functions (`MPI_File_read` etc)

Collective operations

- All processes in communicator that opened file must call function
- Performance potentially better than for individual functions
  - Even if each processor reads a non-contiguous segment, in total the read is contiguous
File view

- By default, file is treated as consisting of bytes, and process can access (read or write) any byte in the file
- A file view consists of three components
  - displacement: number of bytes to skip from the beginning of file
  - etype: type of data accessed, defines unit for offsets
  - filetype: portion of file visible to process
    - same as etype or MPI derived type consisting of etypes
### File view

```c
MPI_File_set_view(fhandle, disp, etype, filetype, datarep, info)
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disp</td>
<td>Offset from beginning of file. Always in bytes</td>
</tr>
<tr>
<td>etype</td>
<td>Basic MPI type or user defined type</td>
</tr>
<tr>
<td></td>
<td>Basic unit of data access</td>
</tr>
<tr>
<td></td>
<td>Offsets in I/O commands in units of etype</td>
</tr>
<tr>
<td>filetype</td>
<td>Same type as etype or user defined type</td>
</tr>
<tr>
<td></td>
<td>constructed of etype</td>
</tr>
<tr>
<td></td>
<td>Specifies which part of the file is visible</td>
</tr>
<tr>
<td>datarep</td>
<td>Data representation, sometimes useful for portability</td>
</tr>
<tr>
<td></td>
<td>“native”: store in same format as in memory</td>
</tr>
<tr>
<td>info</td>
<td>Hints for implementation that can improve performance</td>
</tr>
<tr>
<td></td>
<td>MPI_INFO_NULL: No hints</td>
</tr>
</tbody>
</table>

--

Location to write with default file view

```c
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test', MPI_MODE_WRONLY + MPI_MODE_CREATE, MPI_INFO_NULL, file, err)
CALL MPI_TYPE_SIZE(MPI_INTEGER, intsize, err)
disp = myid * count * intsize
CALL MPI_FILE_SEEK(file, disp, &MPI_SEEK_SET, err)
CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)
CALL MPI_FILE_CLOSE(file, err)
CALL MPI_FINALIZE(err)
END PROGRAM Output
```

Using file view with displacement

```c
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test', MPI_MODE_WRONLY + MPI_MODE_CREATE, MPI_INFO_NULL, file, err)
CALL MPI_TYPE_SIZE(MPI_INTEGER, intsize, err)
disp = myid * count * intsize + 100
CALL MPI_FILE_SEEK(file, disp, &MPI_SEEK_SET, err)
CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)
CALL MPI_FILE_CLOSE(file, err)
```

Using file view with displacement and etype

```c
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test', MPI_MODE_WRONLY + MPI_MODE_CREATE, MPI_INFO_NULL, file, err)
CALL MPI_TYPE_SIZE(MPI_INTEGER, intsize, err)
disp = myid * count + 100
CALL MPI_FILE_SET_VIEW(file, disp, MPI_BYTE, MPI_BYTE, "native", MPI_INFO_NULL, err)
CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)
CALL MPI_FILE_CLOSE(file, err)
```

--

File view for non-contiguous data: filetype

```c
INTEGER :: count = 4
INTEGER, DIMENSION(count) :: buf
CALL MPI_TYPE_VECTOR(4, 1, 4, MPI_INTEGER, filetype, err)
CALL MPI_TYPE_COMMIT(filetype, err)
disp = myid * intsize
CALL MPI_FILE_SET_VIEW(file, disp, MPI_INTEGER, filetype, "native", MPI_INFO_NULL, err)
CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)
```

...
mthomas@tuckoo parallelio]$ cat view.c
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;
MPI_File fh;
int buf[1000];

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);

MPI_Type_contiguous(2, MPI_INT, &contig);
lb = 0;
extent = 6 * sizeof(int);
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);

disp = 5 * sizeof(int); /* assume displacement in this file
   view is of size equal to 5 integers */
etype = MPI_INT;

MPI_File_set_view(fh, disp, etype, filetype, "native", MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
File view: displacement

... 
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test', &
    MPI_MODE_WRONLY + MPI_MODE_CREATE, &
    MPI_INFO_NULL, file, err)
CALL MPI_TYPE_SIZE(MPI_INTEGER, intsize,err)
disp = myid * count * intsize
CALL CALL MPI_FILE_SEEK(file,disp,&
    MPI_SEEK_SET, err)
CALL MPI_FILE_WRITE(file, buf, count, &
    MPI_INTEGER, status, err)
CALL MPI_FILE_CLOSE(file, err)
CALL MPI_FINALIZE(err)
END PROGRAM Output

Location to write with default file view
File view: displacement

Using file view with displacement

```fortran
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test', &
                 MPI_MODE_WRONLY + MPI_MODE_CREATE, &
                 MPI_INFO_NULL, file, err)
CALL MPI_TYPE_SIZE(MPI_INTEGER, intsize, err)
disp = myid * count * intsize
CALL CALL MPI_FILE_SET_VIEW(file, disp,&
                              MPI_BYTE, MPI_BYTE, "native", &
                              MPI_INFO_NULL, err)
CALL MPI_FILE_WRITE(file, buf, count, &
                     MPI_INTEGER, status, err)
CALL MPI_FILE_CLOSE(file, err)
CALL MPI_FINALIZE(err)
END PROGRAM Output
```
File view: displacement and etype

Skip always 100 byte header and write integers

...  
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test', &
  MPI_MODE_WRONLY + MPI_MODE_CREATE, &
  MPI_INFO_NULL, file, err)
CALL MPI_TYPE_SIZE(MPI_INTEGER, intsize,err)
disp = myid * count * intsize + 100
CALL MPI_FILE_SEEK(file, disp, &
  MPI_SEEK_SET, err)
CALL MPI_FILE_WRITE(file, buf, count, &
  MPI_INTEGER, status, err)
CALL MPI_FILE_CLOSE(file, err)
...

File view for non-contiguous data

- File view for non-contiguous data consists of etype or MPI derived type consisting of etypes.
- File view: displacement
- File view: etype

Using file view with displacement

- Offsets in I/O commands in units of etype
- Offsets from beginning of file. Always in bytes
- Location to write with default file view

File type to implement the non-contiguous reads/writes

- Very expensive if implemented with separate scattered throughout a file
- Each process has to access small pieces of data
- wcs:: File consists of etype
- wcs:: File type to implement the non-contiguous reads/writes
- Fileview
- Filetype
- Offset from beginning of file. Always in bytes
- Basic MPI type or user defined type
File view: displacement and etype

Using file view with displacement and etype

```fortran
... CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test', &
  MPI_MODE_WRONLY + MPI_MODE_CREATE, &
  MPI_INFO_NULL, file, err)
  disp = myid * count
  CALL MPI_FILE_SET_VIEW(file, 100, &
    MPI_INTEGER, MPI_INTEGER, "native", &
    MPI_INFO_NULL, err)
  CALL MPI_FILE_WRITE_AT(file, disp, buf, &
    count, MPI_INTEGER, status, err)
  CALL MPI_FILE_CLOSE(file, err)
...```

File view: displacement and etype

Using file view with displacement and etype

```fortran
... CALL MPI_FILE_OPEN(MPI_COMM_WORLD, 'test', &
  MPI_MODE_WRONLY + MPI_MODE_CREATE, &
  MPI_INFO_NULL, file, err)
  disp = myid * count
  CALL MPI_FILE_SET_VIEW(file, 100, &
    MPI_INTEGER, MPI_INTEGER, "native", &
    MPI_INFO_NULL, err)
  CALL MPI_FILE_WRITE_AT(file, disp, buf, &
    count, MPI_INTEGER, status, err)
  CALL MPI_FILE_CLOSE(file, err)
...```
File view for non-contiguous data: filetype

2D-array distributed column-wise

- Each process has to access small pieces of data scattered throughout a file
- Very expensive if implemented with separate reads/writes
- Use file type to implement the non-contiguous access
File view for non-contiguous data

```
INTEGER :: count = 4
INTEGER, DIMENSION(count) :: buf

CALL MPI_TYPE_VECTOR(4, 1, 4, MPI_INTEGER, filetype, err)
CALL MPI_TYPE_COMMIT(filetype, err)
disp = myid * intsize
CALL MPI_FILE_SET_VIEW(file, disp, MPI_INTEGER, filetype, “native”, &
                         MPI_INFO_NULL, err)
CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)
```

2D-array distributed column-wise
Storing multidimensional arrays

Domain decomposition for 2D-array

```
INTEGER :: sizes = (/4, 4/)
INTEGER :: subsizes = (/2, 2/)
INTEGER, DIMENSION(2,2) :: buf

MPI_CART_COORDS(MPI_COMM_WORLD, myid, 2, starts, err)
CALL MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, MPI_INTEGER, MPI_ORDER_C, filetype, err)
CALL MPI_TYPE_COMMIT(filetype)
CALL MPI_FILE_SET_VIEW(file, 0, MPI_INTEGER, filetype, "native", MPI_INFO_NULL, err)
CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)
```
Storing multidimensional arrays: Collective I/O

Collective write can be over hundred times faster than the individual for large arrays!

... INTEGER :: sizes = (/4, 4/) INTEGER :: subsizes = (/2, 2/) INTEGER, DIMENSION(2,2) :: buf ... CALL MPI_CART_COORDS(MPI_COMM_WORLD, myid, 2, starts, err) CALL MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, MPI_INTEGER, MPI_ORDER_C, filetype, err) CALL MPI_TYPE_COMMIT(filetype) CALL MPI_FILE_SET_VIEW(file, 0, MPI_INTEGER, filetype, "native", MPI_INFO_NULL, err) CALL MPI_FILE_WRITE_ALL(file, buf, count, MPI_INTEGER, status, err) ...
gsizes[0] = m;  gsizes[1] = n;
/* no. of rows and columns in global array*/
psizes[0] = 2;  psizes[1] = 3;
/* no. of processes in vertical and horizontal dimensions of process grid */
lsizes[0] = m/psizes[0];  /* no. of rows in local array */
lsizes[1] = n/psizes[1];  /* no. of columns in local array */
dims[0] = 2;  dims[1] = 3;
periods[0] = periods[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
MPI_Comm_rank(comm, &rank);
MPI_Cart_coords(comm, rank, 2, coords);

/* global indices of the first element of the local array */
start_indices[0] = coords[0] * lsizes[0];
MPI_Type_create_subarray(2, gsizes, lsizes, start_indices, MPI_ORDER_C, MPI_FLOAT, &filetype);
MPI_Type_commit(&filetype);
MPI_File_open(comm, "/pfs/datafile", MPI_MODE_CREATE | MPI_MODE_WTRONLY, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, 0, MPI_FLOAT, filetype, "native", MPI_INFO_NULL);

/* create a derived datatype that describes the layout of the local array in the memory buffer that includes the ghost area. This is another subarray datatype! */
memsizes[0] = lsizes[0] + 8;  /* no. of rows in allocated array */
memsizes[1] = lsizes[1] + 8;  /* no. of cols in allocated array */
start_indices[0] = start_indices[1] = 4;
/* indices of the first element of the local array in the allocated array */
MPI_Type_create_subarray(2, memsizes, lsizes, start_indices, MPI_ORDER_C, MPI_FLOAT, &memtype);
MPI_Type_commit(&memtype);
MPI_File_write_all(fh, local_array, 1, memtype, &status);
MPI_File_close(&fh);
Shared file, independent access

- All processes independently access exclusive regions of a single file
  - Coordination can take place at the parallel file system
  - Significant overhead for some access patterns where the file system has to serialize access

- Advantage of shared file lies in data management and portability
Shared file, collective access

- Improves performance of shared-file access by offloading some of the coordination work from the file system to the application
  - A subset of processes is assigned to do I/O and buffers access to the file
  - Available in all modern MPI libraries (called “collective buffering”)
/* writing to a common file using the shared file pointer */
#include "mpi.h"

int main(int argc, char *argv[])
{
    int buf[1000];
    MPI_File fh;

    MPI_Init(&argc, &argv);

    MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", 
                  MPI_MODE_RDONLY, MPI_INFO_NULL, &fh);
    MPI_File_write_shared(fh, buf, 1000, MPI_INT, 
                           MPI_STATUS_IGNORE);
    MPI_File_close(&fh);

    MPI_Finalize();
    return 0;
}
Non-blocking MPI I/O

- Non-blocking independent I/O is similar to non-blocking send/recv routines
  
  `MPI_File_iread`
  `MPI_File_iwrite`
  `MPI_File_iread_at`
  `MPI_File_iwrite_at`

- Wait for completion using `MPI_Test`, `MPI_Wait`, etc.

- Can be used to overlap I/O with computation
Giving hints to MPI I/O

- Hints may enable the implementation to optimize performance
- MPI 2 standard defines several hints via MPI_Info object
  - MPI_INFO_NULL: no info
  - Functions MPI_Info_create and MPI_Info_set allow one to create and set hints
- Effect of hints on performance is implementation and application dependent
Giving hints to MPI I/O

• For example, Cray XT systems support the following hints
  striping_factor, striping_unit, direct_io,
  romio_cb_read, romio_cb_write, cb_buffer_size,
  cb_nodes, cb_config_list, romio_no_indep_rw,
  romio_ds_read, ind_rd_buffer_size, ind_wr_buffer_size

• Consult "man mpi" for their meaning and default values
Giving hints to MPI I/O

• Some implementations allow setting of hints via environment variables
  – e.g. MPICH_MPIIO_HINTS
  – Example: for file “test.dat”, in collective I/O aggregate data to 32 nodes
    export MPICH_MPIIO_HINTS="test.dat:cb_nodes=32"
Storing multidimensional arrays:

Collective I/O

Collective write can be over hundred times faster than the individual for large arrays!

```fortran
INTEGER :: sizes = (/4, 4/)
INTEGER :: subsizes = (/2, 2/)
INTEGER, DIMENSION(2,2) :: buf
...
CALL MPI_CART_COORDS(MPI_COMM_WORLD, myid, 2, starts, err)
CALL MPI_TYPE_CREATE_SUBARRAY(2, sizes, subsizes, starts, MPI_INTEGER, MPI_ORDER_C, filetype, err)
CALL MPI_TYPE_COMMIT(filetype)
CALL MPI_FILE_SET_VIEW(file, 0, MPI_INTEGER, filetype, "native", MPI_INFO_NULL, err)
CALL MPI_FILE_WRITE(file, buf, count, MPI_INTEGER, status, err)
...
```

Non-blocking MPI I/O

- Non-blocking independent I/O is similar to non-blocking send/recv routines
- MPI_File_iread
- MPI_File_iwrite
- MPI_File_iread_at
- MPI_File_iwrite_at

- Wait for completion using MPI_Test, MPI_Wait, etc.
- Can be used to overlap I/O with computation

Giving hints to MPI I/O

- Hints may enable the implementation to optimize performance
- MPI 2 standard defines several hints via MPI_Info object
  - MPI_INFO_NULL: no info
  - Functions MPI_Info_create and MPI_Info_set allow one to create and set hints
- Effect of hints on performance is implementation and application dependent

Example, Cray XT systems support the following hints
- striping_factor, striping_unit, direct_io, romio_cb_read, romio_cb_write, cb_buffer_size, cb_nodes, cb_config_list, romio_no_indep_rw, romio_ds_read, ind_rd_buffer_size, ind_wr_buffer_size
- Consult "man mpi" for their meaning and default values

Achieving good I/O performance

- Do not perform I/O from one process only, use parallel I/O!
- Make large requests wherever possible
- For non-contiguous requests, use derived datatypes and a single collective I/O call
- Experiment with MPI I/O hints

Performance

Parallel write with 64 cores including file open/close

Parallel write with 1024 cores including file open/close
Common mistakes with MPI I/O

✗ Not defining file offsets as MPI_Offset in C and integer (kind=MPI_OFFSET_KIND) in Fortran

✗ In Fortran, passing the offset or displacement directly as a constant (e.g., 0)

✗ Filetype defined using offsets that are not monotonically nondecreasing
Parallel I/O - summary

• POSIX
  – single reader/writer, all read/write, subset read/write
  – user is responsible for communication

• MPI I/O
  – MPI library is responsible for communication
  – file views enable non-contiguous access patterns
  – collective I/O can enable the actual disk access to remain contiguous
Web resources

- William Gropp’s ”Advanced MPI” tutorial in PRACE Summer School 2011, including very in-depth discussion about MPI I/O
  http://www.csc.fi/courses/archive/material/prace-summer-school-material/MPI-tutorial
C interfaces to MPI I/O routines

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

int MPI_File_close(MPI_File *fh)

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

int MPI_File_read(MPI_File fh, void *buf, int count,
                   MPI_Datatype datatype, MPI_Status *status)

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

int MPI_File_write(MPI_File fh, void *buf, int count,
                   MPI_Datatype datatype, MPI_Status *status)

int MPI_File_write_at(MPI_File fh, MPI_Offset offset, void *buf,
                       int count, MPI_Datatype datatype, MPI_Status *status)
```
C interfaces to MPI I/O routines

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

int MPI_File_read_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

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

int MPI_File_write_all(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)

int MPI_File_write_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
```
Fortran interfaces for MPI I/O routines

MPI_FILE_OPEN(comm, filename, amode, info, fh, ierr)
   INTEGER :: comm, amode, info, fh, ierr
   CHARACTER*(*) :: filename

MPI_FILE_CLOSE(fh, ierr)

MPI_FILE_SEEK(fh, offset, whence)
   INTEGER :: fh, offset, whence

MPI_FILE_READ(fh, buf, count, datatype, status)
   INTEGER :: fh, buf, count, datatype, status(MPI_STATUS_SIZE)

MPI_FILE_READ_AT(fh, offset, buf, count, datatype, status)
   INTEGER :: fh, offset, buf, count, datatype, status(MPI_STATUS_SIZE)

MPI_FILE_WRITE(fh, buf, count, datatype, status)

MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)
Common mistakes with MPI I/O

Not defining file offsets as `MPI_Offset` in C and `integer (kind=MPI_OFFSET_KIND)` in Fortran.

In Fortran, passing the offset or displacement directly as a constant (e.g., 0).

Filetype defined using offsets that are not monotonically nondecreasing.

Parallel I/O - summary

- POSIX – single reader/writer, all read/write, subset read/write
- user is responsible for communication

- MPI I/O – MPI library is responsible for communication
- file views enable non-contiguous access patterns
- collective I/O can enable the actual disk access to remain contiguous

Web resources

- William Gropp’s Advanced MPI tutorials in PRACE Summer School 2011, including very in-depth discussion about MPI I/O
  
  [link](http://www.csc.fi/courses/archive/material/prace-summer-school-material/MPI-tutorial)

C interfaces to MPI I/O routines

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

int MPI_File_close(MPI_File *fh)

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

int MPI_File_read(MPI_File fh, void *buf, int count,
                  MPI_Datatype datatype, MPI_Status *status)

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

int MPI_File_write(MPI_File fh, void *buf, int count,
                    MPI_Datatype datatype, MPI_Status *status)

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

Fortran interfaces for MPI I/O routines

```fortran
MPI_FILE_OPEN(comm, filename, amode, info, fh, ierr)

INTEGER       ::  comm, amode, info, fh, ierr
CHARACTER*(*) ::  filename

MPI_FILE_CLOSE(fh, ierr)

MPI_FILE_SEEK(fh, offset, whence)

INTEGER :: fh, offset, whence

MPI_FILE_READ(fh, buf, count, datatype, status)

INTEGER :: fh, buf, count, datatype, status(MPI_STATUS_SIZE)

MPI_FILE_READ_AT(fh, offset, buf, count, datatype, status)

INTEGER :: fh, offset, buf, count, datatype, status(MPI_STATUS_SIZE)

MPI_FILE_WRITE(fh, buf, count, datatype, status)

MPI_FILE_WRITE_AT(fh, offset, buf, count, datatype, status)
```