Introduction to Parallel I/O

Si Liu

Robert McLay, Ritu Arora, John Cazes

Texas Advanced Computing Center
June, 2014

Email: {siliu, mclay, rauta, cazes}@tacc.utexas.edu
Outline

• Introduction to parallel I/O and parallel file system
• Parallel I/O pattern
• Introduction to MPI I/O
• Introduction to HDF5
• Introduction to T3PIO (If time allows)
• I/O strategies
I/O in HPC Applications

- High Performance Computing (HPC) applications often
  - Read initial conditions or datasets for processing
  - Write numerical data from simulations
    - Parallel applications commonly need to write distributed arrays to disk
    - Saving application-level checkpoints

- Total Execution Time = Computation Time
  + Communication Time
  + I/O time
  - Optimize all the components of the equation above for best performance

- However, doing efficient I/O without stressing out the HPC system is challenging and often an afterthought.
  - Every week 3-5 user accounts are temporarily shut-off for abusing Stampede’s file system --- motivation to introduce parallel I/O
Modern HPC Cluster
Modern HPC Cluster

...we need some magic to make the collection of spinning disks act like a single disk for the user...
Lustre File System Provides the Magic

Lustre Clients
Application processes running on compute nodes

Memory
RAM of compute nodes

Network
Ethernet or InfiniBand

Lustre Servers
Metadata and Object storage Servers

Lustre Targets
Metadata and Object storage Targets

Hundreds of thousands of processors

A few hundred spinning disks => OST

Source: Reference 2, 4
Lustre File System at TACC

- Each Lustre file system has a different number of OSTs.
- The greater the number of OSTs the better the I/O capability.

<table>
<thead>
<tr>
<th></th>
<th>$HOME</th>
<th>$WORK</th>
<th>$SCRATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stampede</td>
<td>24</td>
<td>672</td>
<td>348</td>
</tr>
<tr>
<td>Lonestar</td>
<td>N/A (NFS)</td>
<td>30</td>
<td>90</td>
</tr>
</tbody>
</table>
Taxing the file system without really trying

- Open and close the same file every few milliseconds
  - Stresses the MDS
- Too often, too many
  - Stresses the MDS and OSTs
- Write large files to $HOME
  - $SCRATCH and $WORK have more OSTs
- `ls` in a crowded directory
  - `ls` is aliased to “`ls --color=ttv” by default
  - Every “stat" needs the MDS
  - Use `/bin/ls` in a crowded directory
- Create thousands of files in the same directory
  - A directory too is a file managed by the MDS
Parallel I/O – Why & How?

• Goal of Parallel I/O is to use parallelism to avoid I/O bottlenecks.

• Parallel I/O can be hard to coordinate and optimize if working directly at the level of Lustre API or POSIX I/O Interface (not discussed in this tutorial).

• Therefore, specialists implement a number of intermediate layers for coordination of data access and mapping from application layer to I/O layer.

• Hence, application developers only have to deal with a high-level interface built on top of a software stack, which in turn sits on top of the underlying hardware.

   MPI-I/O, parallel HDF5, parallel NetCDF, T3PIO,...
Outline

• Introduction to parallel I/O and parallel file system
• Parallel I/O pattern
• Introduction to MPI I/O
• Introduction to HDF5
• Introduction to T3PIO
• I/O strategies
Typical Pattern: Parallel Programs Doing Sequential I/O

- All processes send data to master process, and then the process designated as master writes the collected data to the file.
- This sequential nature of I/O can limit performance and scalability of many applications.
Typical Pattern: Each Process Reads/Writes to a Separate File

- This is totally parallel.
- This pattern may generate a huge number of small files.
- Number of CPUs and data files are fixed.
Desired Pattern: Parallel Programs Doing Parallel I/O

- Multiple processes participating in reading data from or writing data to a common file in parallel.
- This strategy improves performance and provides a single file for storage and transfer purposes.
Balance Parallel I/O (1)

1024 nodes x 16 tasks = 16384 I/O clients

Terribly oversubscribed
Balance Parallel I/O (2)

1024 nodes x 1 task = 1024 I/O clients

Much better
160 nodes x 1 task = 160 I/O clients

Best for I/O bandwidth

But, reality sets in

• I/O servers are shared
• Still have to move the data to other tasks
Lustre File System - Striping

• Lustre supports the “striping” of files across several I/O servers (similar to RAID).

• Each stripe is a fixed size block.

myfile : 8 MB file
4 stripes
1 MB stripe size
Balance Parallel I/O (4)

– Match the I/O bandwidth with the switch bandwidth

– This can be done by setting the stripe count to a higher number to saturate the IB bandwidth (Lustre only):

```bash
lfs setstripe -c 4 input_dir
```
Lustre File System

• Administrators set a default stripe count and stripe size that apply to all newly created files:
  – Stampede:  $SCRATCH$: 2 stripes/1MB
    $WORK$: 1 stripe /1MB
  – Lonestar:  $SCRATCH$: 2 stripes/1MB
    $WORK$: 1 stripe /1MB

• However, users can reset the default stripe count or stripe size using the `lfs setstripe` command.
Lustre Commands

• Get stripe count

  % lfs getstripe ./testfile
  ./testfile
  lmm_stripe_count: 2
  lmm_stripe_size: 1048576
  lmm_stripe_offset: 50

<table>
<thead>
<tr>
<th>obdidx</th>
<th>objid</th>
<th>objid</th>
<th>group</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>8916056</td>
<td>0x880c58</td>
<td>0</td>
</tr>
<tr>
<td>38</td>
<td>8952827</td>
<td>0x889bfb</td>
<td>0</td>
</tr>
</tbody>
</table>

• Set stripe count

  % lfs setstripe -c 4 -s 4M testfile2
  % lfs getstripe ./testfile2
  ./testfile2
  lmm_stripe_count: 4
  lmm_stripe_size: 4194304
  lmm_stripe_offset: 21

<table>
<thead>
<tr>
<th>obdidx</th>
<th>objid</th>
<th>objid</th>
<th>group</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>8891547</td>
<td>0x87ac9b</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>8946053</td>
<td>0x888185</td>
<td>0</td>
</tr>
<tr>
<td>57</td>
<td>8906813</td>
<td>0x87e83d</td>
<td>0</td>
</tr>
<tr>
<td>44</td>
<td>8945736</td>
<td>0x888048</td>
<td>0</td>
</tr>
</tbody>
</table>
Outline

• Introduction to parallel I/O and parallel file system
• Parallel I/O pattern
• Introduction to MPI I/O
• Introduction to HDF5
• Introduction to T3PIO
• I/O strategies
MPI for Parallel I/O

• A parallel I/O system for distributed memory architectures will need a mechanism to specify collective operations and specify noncontiguous data layout in memory and file.

• Reading and writing in parallel is like receiving and sending messages.

• An MPI-like machinery is a good setting for Parallel I/O (think MPI communicators and MPI datatypes).

• MPI-I/O featured in MPI-2 which was released in 1997, and it interoperates with the file system to enhance I/O performance for distributed-memory applications.
Using MPI-I/O

• Given N number of processes, each process participates in reading or writing a portion of a common file.

• There are three ways of positioning where the read or write takes place for each process:
  – Use individual file pointers (e.g., `MPI_File_seek`/`MPI_File_read`)
  – Calculate byte offsets (e.g., `MPI_File_read_at`)
  – Access a shared file pointer (e.g., `MPI_File_seek_shared`, `MPI_File_read_shared`)

![Diagram of file access by processes]

Source: Reference 3
MPI-I/O API Opening and Closing a File

• Calls to the MPI functions for reading or writing must be preceded by a call to `MPI_File_open`:
  - `int MPI_File_open(MPI_Comm comm, char *filename, int a_mode, MPI_Info info, MPI_File *fh)`

• The parameters below are used to indicate how the file is to be opened

<table>
<thead>
<tr>
<th><code>MPI_File_open mode</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MODE_RDONLY</td>
<td>read only</td>
</tr>
<tr>
<td>MPI_MODE_WRONLY</td>
<td>write only</td>
</tr>
<tr>
<td>MPI_MODE_RDWR</td>
<td>read and write</td>
</tr>
<tr>
<td>MPI_MODE_CREATE</td>
<td>create file if it doesn’t exist</td>
</tr>
</tbody>
</table>

• To combine multiple flags, use bitwise-or “|” in C, or addition “+” in Fortran

• Close the file using: `MPI_File_close(MPI_File fh)`
MPI-I/O API for Reading Files

After opening the file, read data from files by either using `MPI_File_seek` & `MPI_File_read` or `MPI_File_read_at`:

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

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

Where `in MPI_File_seek` updates the individual file pointer according to:

- `MPI_SEEK_SET`: the pointer is set to offset
- `MPI_SEEK_CUR`: the pointer is set to the current pointer position plus offset
- `MPI_SEEK_END`: the pointer is set to the end of file plus offset

```c
int MPI_File_read_at( MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
```
MPI-I/O API for Writing Files

• While opening the file in the write mode, use the appropriate flag(s) in
  
  MPI_File_open: MPI_MODE_WRONLY or MPI_MODE_RDWR and if
  needed, MPI_MODE_CREATE.

• For writing, use MPI_File_set_view and MPI_File_write or
  MPI_File_write_at.

```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_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)
```
Writing a File: writeFile.c (1)

1. #include<stdio.h>
2. #include "mpi.h"
3. int main(int argc, char **argv){
4.   int i, rank, size, offset, nints, N=16 ;
5.   MPI_File fhw;
6.   MPI_Status status;
7.   MPI_Init(&argc, &argv);
8.   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9.   MPI_Comm_size(MPI_COMM_WORLD, &size);
10.  int buf[N];
11.   for ( i=0;i<N;i++){
12.      buf[i] = i ;
13.   }
15. offset = rank*(N/size)*sizeof(int);

16. `MPI_File_open` (MPI_COMM_WORLD, "datafile",
    MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &fhw);

17. `printf("\nRank: %d, Offset: %d\n", rank, offset);

18. `MPI_File_write_at` (fhw, offset, buf, (N/size),
    MPI_INT, &status);

19. `MPI_File_close` (&fhw);

20. `MPI_Finalize` ();
21. return 0;
22.}
Note about Atomicity Read/Write

```c
int MPI_File_set_atomicity ( MPI_File mpi_fh, int flag );
```

- Use this API to set the atomicity mode – 1 for true and 0 for false – so that only one process can access the file at a time.

- When atomic mode is enabled, MPI-IO will guarantee sequential consistency and this can result in significant performance drop.

- This is a collective function.
Collective I/O (1)

• Collective I/O is a critical optimization strategy for reading from, and writing to, the parallel file system.

• The collective read and write calls force all processes in the communicator to read/write data simultaneously and to wait for each other.

• The MPI implementation optimizes the read/write request based on the combined requests of all processes and can merge the requests of different processes for efficiently servicing the requests.

• This is particularly effective when the accesses of different processes are noncontiguous.
Collective I/O (2)

• The collective functions for reading and writing are:
  – MPI_File_read_all
  – MPI_File_write_all
  – MPI_File_read_at_all
  – MPI_File_write_at_all

• Their signature is the same as for the non-collective versions.
MPI-I/O Hints

• MPI-IO hints are extra information supplied to the MPI implementation through the following function calls for improving the I/O performance:
  – MPI_File_open
  – MPI_File_set_info
  – MPI_File_set_view

• Hints are optional and implementation-dependent
  – you may specify hints but the implementation can ignore them.

• MPI_File_get_info used to get list of hints, examples of Hints: striping_unit, striping_factor
Lustre – setting stripe count in MPI Code

• MPI may be built with Lustre support
  – MVAPICH2 & OpenMPI support Lustre

• Set stripe count in MPI code
  Use MPI I/O hints to set Lustre stripe count, stripe size, and # of writers

  Fortran:
  call mpi_info_set(myinfo,"striping_factor",stripe_count,mpierr)
call mpi_info_set(myinfo,"striping_unit",stripe_size,mpierr)
call mpi_info_set(myinfo,"cb_nodes",num_writers,mpierr)

  C:
  MPI_Info_set(myinfo,"striping_factor",stripe_count);
  MPI_Info_set(myinfo,"striping_unit",stripe_size);
  MPI_Info_sett(myinfo,"cb_nodes",num_writers);

• Default:
  – # of writers = # Lustre stripes (stripe count)
Outline

• Introduction to parallel I/O and parallel file system
• Parallel I/O pattern
• Introduction to MPI I/O
• Introduction to HDF5
• Introduction to T3PIO
• I/O strategies
HDF5: Hierarchical Data Format

• HDF5 nice features:
  – Interface support for C, C++, Fortran, and Java
  – Supported by data analysis packages
    (Matlab, IDL, Mathematica, Octave, Visit, Paraview, Tekplot, etc.)
  – Machine independent data storage format
  – Supports user defined datatypes and metadata

• HDF5 is a file format
  – An HDF5 file can be viewed as a file system inside a file.
  – It uses a Unix style directory structure.
  – It is a mixture of entities: groups, datasets and attributes.
  – Any entity can have descriptive attributes (metadata), e.g. physical units.

• Read or write to a portion of a dataset (Hyperslab)
HDF5: The Benefits of Metadata

• It is easy to record many metadata items within a solution file.
• Adding attributes later won’t break any program that reads the data.
• With HDF5 it is easy to save with each solution file:
  – Computer Name, OS Version
  – Compiler and MPI name and version
  – Program Version
  – Input file
  – etc.
PHDF5 Overview

• PHDF5 is the Parallel HDF5 library.
  – You can write one file in parallel efficiently!
  – Parallel performance of HDF5 very close to MPI I/O.

• Uses MPI I/O (Don’t reinvent the wheel)

• MPI I/O techniques apply to HDF5.

• Use MPI_Info object to control # writers, # stripes(Lustre), stripe size(Lustre), etc.
Why one HDF5 file is best

• Only 1 file is opened → Efficient interaction with MDS.

• Every task calls HDF5 dataset write routines...

  ... but internally HDF5 and MPI move data to a small number of writer nodes (writers/aggregators)

• PHDF5 uses MPI I/O (ROMIO) to handle the parallel parts.
A Dump of a Simple HDF5 File

$h5dump dset.h5$

HDF5 "dset.h5" {
  GROUP "/" {
    DATASET "T" {
      DATATYPE   H5T IEEE F64LE
      DATASPACE   SIMPLE { ( 10 ) / ( 10 ) }
      DATA {
        (0): 1.5, 1, 1.0625, 1.0625, 2.0625,
        (5): 1.4375, 1.4375, 0.625, 1.625, 1.625
      }
      ATTRIBUTE "Description" {
        DATASPACE   SIMPLE { ( 1 ) / ( 1 ) }
        DATA {
          (0): "thermal soln"
        }
      }
    }
  }
}
Basic HDF5 Structure

• Open HDF5

• Open File
  - Open Group
    • Open Dataset
    • Write Dataset
    • Close Dataset
  - Close Group

• Close File

• Close HDF5
HDF5 write example

// Set up file access property list with parallel I/O access
plist_id = H5Pcreate(H5P_FILE_ACCESS);
H5Pset_fapl_mpio(plist_id, comm, info);

//Create a new file collectively and release property list identifier.
file_id = H5Fcreate(H5FILE_NAME, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
H5Pclose(plist_id);

//Create the dataspace for the dataset.
filespace = H5Screate_simple(RANK, dims, NULL);

//Create the dataset with default properties and close file space.
dset_id = H5Dcreate(file_id, DATASETNAME, H5T_NATIVE_INT, filespace,
                     H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);

//Create property list for collective dataset write.
plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);

//Write the data
status = H5Dwrite(dset_id, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, plist_id, data);
Outline

• Introduction to parallel I/O and parallel file system
• Parallel I/O pattern
• Introduction to MPI I/O
• Introduction to HDF5
• Introduction to T3PIO
• I/O strategies
T3PIO Library

• TACC's Terrific Tool for Parallel I/O

• Lustre parallel I/O performance depends on
  – Number of Writers
  – Number of Stripes
  – Stripe Size

• By default MPI I/O sets
  – Number of Writers = Number of nodes
  – Number of Stripes = directory default (typically 4)
  – Stripe Size = 1 MB

• This library will set these parameters for you.
T3PIO Basic Heuristics

This T3PIO library sets your MPI Info object for you.

• Set stripe count to a small multiple of N (nodes)
• Number of Writers <= Number of Tasks
• Stripe count <= Number of available OSTs /2
• No more than 2 writers per node
T3PIO Library: Fortran

Fortran interface:

```fortran
subroutine t3pio_set_info(comm, info, dir, err, &
GLOBAL_SIZE=size, &
MAX_STRIPES=nstripes, &
FACTOR=factor, &
RESULTS=results, &
FILE="file" &
... )
```
T3PIO Library: C/C++

C/C++ interface:

```c
#include <t3pio.h>

int ierr = t3pio_set_info(comm, info, dir,
                          T3PIO_GLOBAL_SIZE, size,
                          T3PIO_MAX_STRIPES, maxStripes,
                          T3PIO_FACTOR, factor,
                          T3PIO_FILE, "file",
                          T3PIO_RESULTS, &results
)
```
Performance Benefit (on Stampede)

![Graph showing performance benefit with and without T3PIO.](image-url)
subroutine hdf5_writer(....)
use hdf5
use t3pio
integer info ! MPI Info object
integer comm ! MPI Communicator
integer(hid_t) :: plist_id ! Property list identifier
...
comm = MPI_COMM_WORLD
! Initialize info object.
call MPI_Info_create(info,ierr)
! use library to fill info with nwriters, stripe
call t3pio_set_info(comm, info, "./", ierr, &
  GLOBAL_SIZE=globalSize)
call H5open_f(ierr)
call H5Pcreate_f(H5P_FILE_ACCESS_F,plist_id,ierr)
call H5Pset_fapl_mpio_f(plist_id, comm, info, ierr)
call H5Fcreate_f(fileName, H5F_ACC_TRUNC_F, file_id, ierr, &
  access_prp = plist_id)
#include "t3pio.h"
#include "hdf5.h"

void hdf5_writer(....)
{
    MPI_Info info = MPI_INFO_NULL;
    hid_t plist_id;
    ...
    MPI_Info create(&info);
    ierr = t3pio_set_info(comm, info, "./",
                           T3PIO_GLOBAL_SIZE, globalSize);
    plist_id id = H5Pcreate(H5P_FILE_ACCESS);
    ierr = H5Pset_fapl_mpio(plist_id, comm, info);
    File_id = H5Fcreate(fileName, H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
    ...
}
Outline

- Introduction to parallel I/O and parallel file system
- Parallel I/O pattern
- Introduction to MPI I/O
- Introduction to HDF5
- Introduction to T3PIO
- I/O strategies
Strategies for I/O on Stampede

• Be gentle with the MDS (Meta Data Server)
  Avoid “Too often, too many”
• Write large files to $SCRATCH$ (not $HOME$)
• Write one global file instead of multiple files
• Use parallel I/O
  – MPI I/O
  – Parallel HDF5, parallel NetCDF
• Access data contiguously in memory and on disk if possible
• Set file attributes (stripe count, stripe size, #writers)
  – T3PIO
Summary

• I/O can impact performance at large scale

• Take advantage of the parallel file system

• Consider using MPI-IO, Parallel HDF5, or Parallel netCDF libraries

• Analyze your code to determine if you may benefit from parallel I/O

• Set stripe count and stripe size for optimal use if on a Lustre file system
References

1. HDF5 Tutorial: www.hdfgroup.org/HDF5/Tutor/introductory.html

2. NICS I/O guide: www.nics.tennessee.edu/io-tips

3. T3PIO: github.com/TACC/t3pio


5. Introduction to Parallel I/O and MPI-IO by Rajeev Thakur