SHARCNET

General Interest Seminar Series

- Introduction to Parallel I/O

Isaac Ye

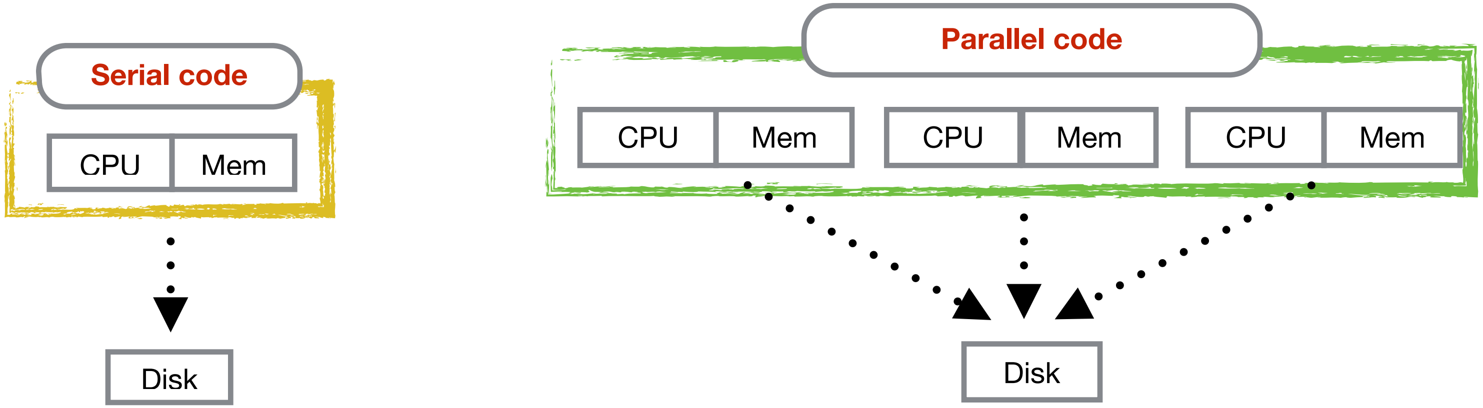
HPTC @York University

Outline

- I/O Issues in large-scale computation
- Disk I/O problems
- Definition of I/O speed
- SHARCNET filesystems
- Overview of I/O software and hardware
- Parallel filesystem
- Best Practices for I/O
- Data formats
- I/O strategies (serial/parallel)
- MPI-IO
- Introduction to Parallel I/O libraries (NetCDF/HDF5/ADIOS)

Issue - HPC I/O

- High Performance Computing (HPC) application requires Input/Output (I/O) activities for
  - Reading initial conditions or datasets for processing
  - Writing numerical data from simulations for later analysis
  - Checkpointing to files
- For many parallel programs, Input and Output (I/O) become a major bottleneck.



Issue - Goal

- Efficient I/O without stressing out the HPC system is challenging
  - Load and store operations are more time-consuming than multiply operations
  - **Total Execution Time**
    = Computation Time + Communication Time + I/O time
  - Optimize all the components of the equation above to get best performance!!

Disk access rates over time

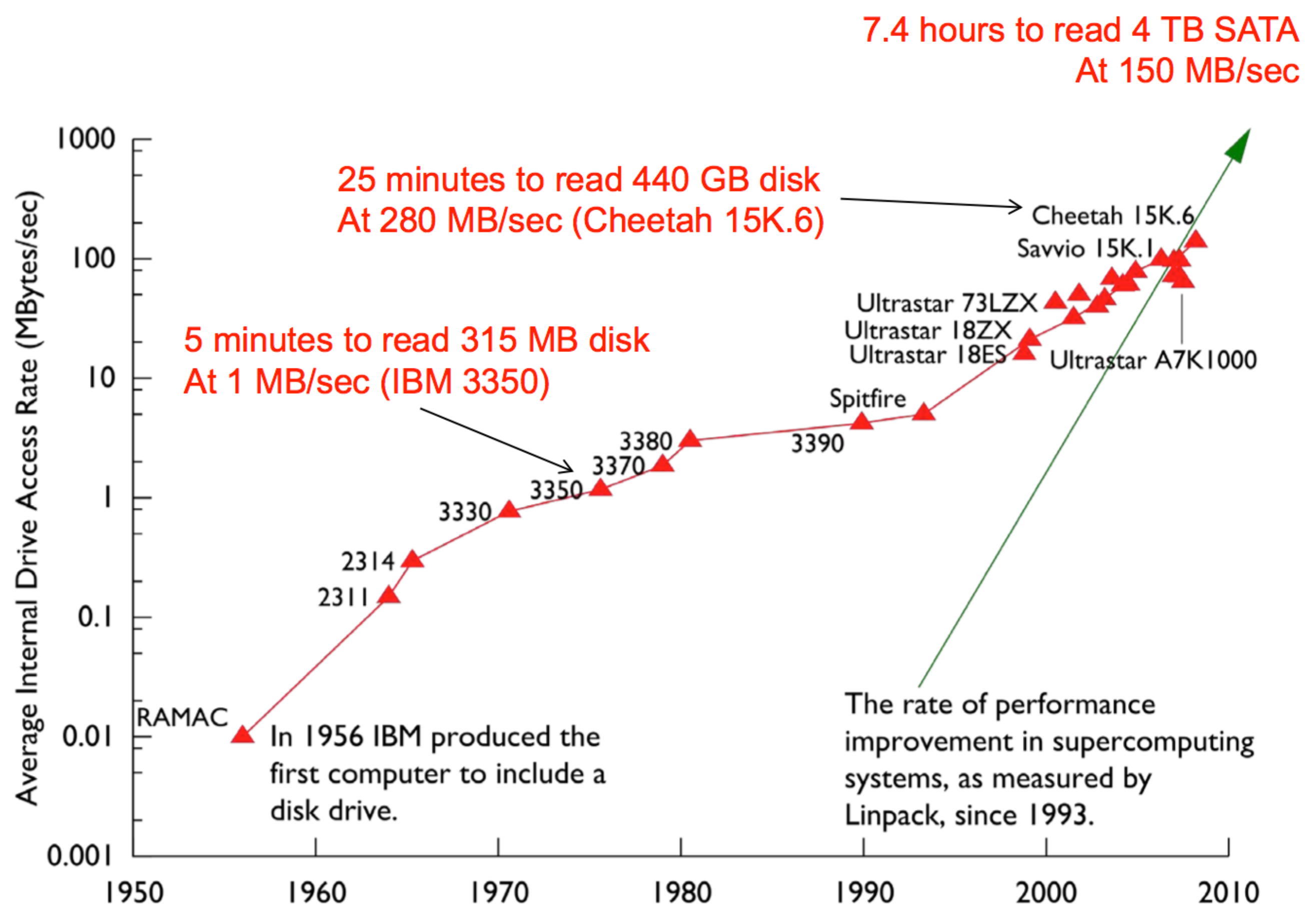


Figure by Rob Ross, Argonne National Laboratory

Memory / storage latency

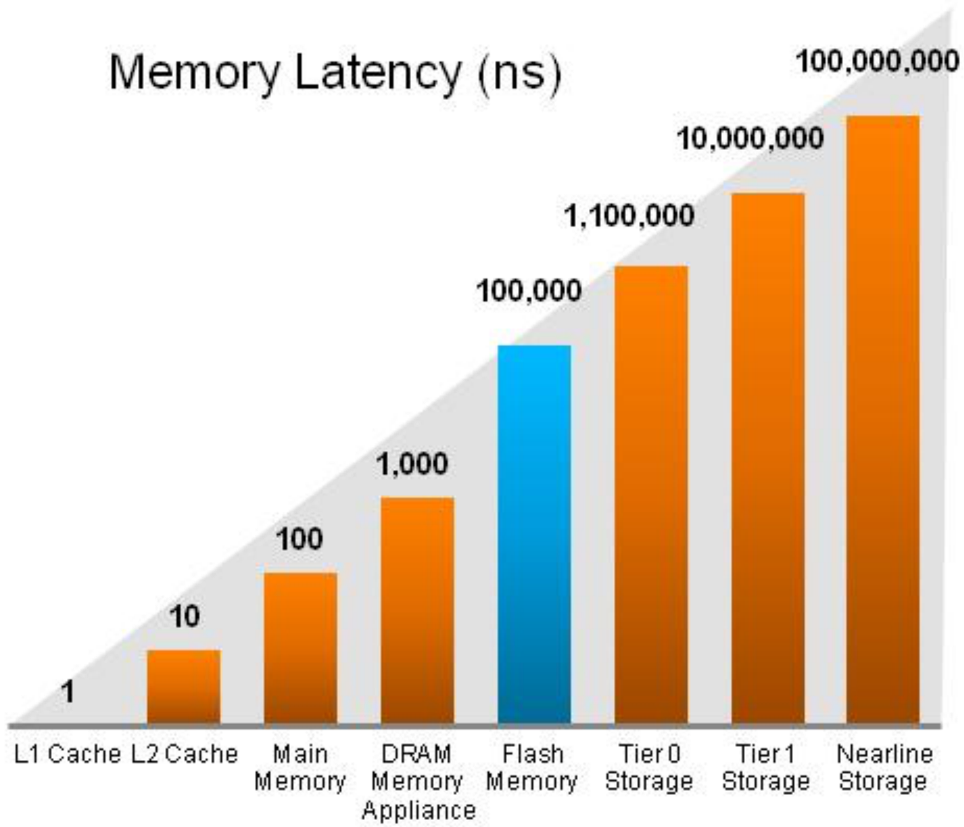


Figure by Jeff Richardson, datacenterjournal.com

How to calculate I/O speed

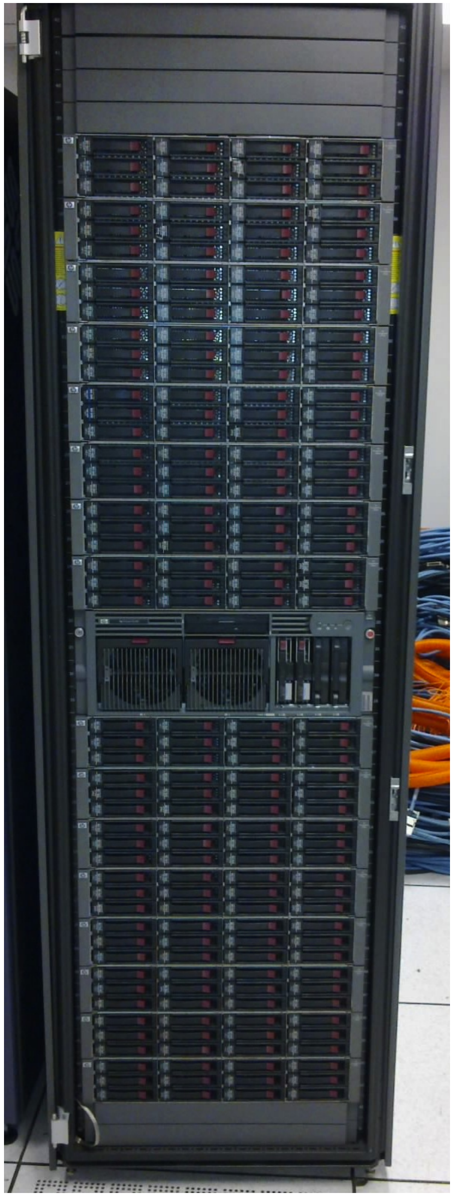
- **IOPs** = Input / Output operations per second (read/write/open/close/seek) ; essentially an inverse of latency
- **I/O Bandwidth** = quantity you read / write

Device	Bandwidth(MB/s)	IOPs
7200 rpm SATA HDD	100	100-300
SSD drive	250-500	< 4000
SHARCNET global /work	100 /stream	700

- Parallel (distributed) filesystems are optimized for efficient I/O by multiple users on multiple machines/nodes, do not result in “supercomputing” performance
  - disk-access time + communication over the network (limited bandwidth, many users)

SHARCNET filesystems

- We have a hierarchy of parallel (/scratch, /work, /home, /archive) and serial (/tmp) filesystems.
- Mostly based on Lustre
- All large filesystems feature many servers and disks+large number of compute node
- Shown in the right: one of our smaller local scratch filesystems with 192 disks
  - local to a cluster, I/O data travel over the network
- Global /work and /home are mounted on all clusters through a wider-area network (slower access than /scratch)



I/O Software + Hardware stack

Application

High-end I/O library

HDF5, Parallel NetCDF, ADIOS
- maps application abstractions to storage abstractions I/O in terms of the data structures of the code not bytes and blocks
- provides data portability

I/O Middleware

MPI-IO
- organizes access from many processes, especially collective I/O
- provides data sieving

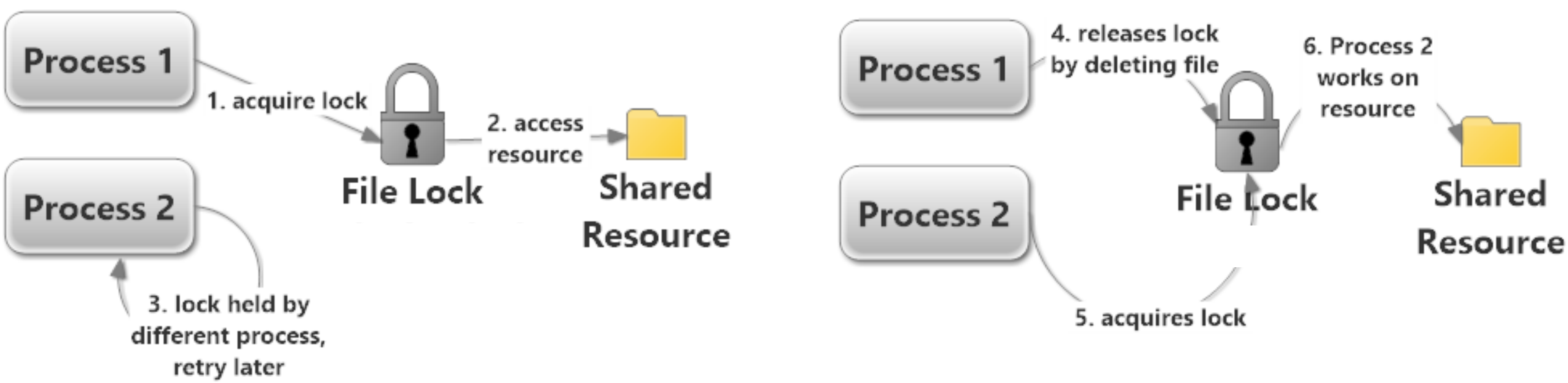
Parallel filesystem

GPFS, Lustre, PVFS
- maintains logical space and provides efficient access to data

I/O Hardware

Parallel filesystem - I

- Files can be striped across multiple drives for better performance
- ‘Lock’s used to manage concurrent file across in most parallel file system
  - Files are pieced into ‘lock’ units (scattered across many drives)
  - Client nodes obtain locks on units that they access before I/O occurs
  - Enables caching on clients
  - Locks are reclaimed from clients when others desire access



Parallel filesystem - II

- Optimized for large shared files
- Poor performance under many small reads/writes (high IOPs)
  - Do not store millions of small files
- Your use of it affects everybody!
  (Different from case with CPU and RAM which are not shared)
- Critical factors: how you read / write, file format, # of files in a directory and how often per sec
- File system is shared over the ethernet network on a cluster: heavy I/O can prevent the processes from communication
- File systems are LIMITED: bandwidth, IOPs, # of files, space and etc.

Best Practices for I/O - I

- Make a plan for your data needs:
  - How much will you generate
  - How much do you need to save
  - And where will you keep it?
    - Note that /scratch is temporary storage for 4 months or less
- Monitor and control usage
  - Minimize use of filesystem commands like ‘`ls`’ and ‘`du`’ in large directories
- Check your disk usage regularly with ‘`quota`’
- Warning!!
  - more than 100K files in your space
  - average data file size less than 100 MB for large output
- Do ‘housekeeping’ (gzip, tar, delete) regularly

Best Practices for I/O - II

Do

- Write binary format files ==> faster I/O and less space than ASCII format
- Use parallel I/O if writing form many nodes
- Maximize size of files: large block I/O optimal
- Minimize number of files ==> more responsive filesystem

Don't

- Write lots of ASCII files
- Many hundreds of files in a single directory
- Many small files (< 10MB). System is optimized for large-block I/O

Data Formats - ASCII

(1) ASCII = American Standard Code for Information Interchange

- pros
  - human readable, portable (architecture independent)
- cons
  - inefficient storage
    (13 bytes per single precision float,
    22 bytes per double precision,
    plus delimiters), **expensive** for read/write
- `fprintf()` in C
- `open(6,file='test',form='formatted');write(6,*)` in F90

Data Formats - Binary

(2) Binary

- pros
  - efficient storage
    (4 bytes per single precision float,
    8 bytes per double precision, no delimiters), efficient read / write
- cons
  - have to know the format to read, portability (endians)
- `fwrite()` in C
- `open(6,file='test',form='unformatted'); write(6)` in F90

Format	/scratch	/tmp (disk)
ASCII	173 s	260 s
Binary	6 s	20 s

Table. Writing 128M doubles on GPCS in SciNet

Data Format - XML, Databases

(3) MetaData (XML) – can be wrapped around text or binary data

- encodes data about data: number and names of variables, their dimensions and sizes, endians, owner, date, links, comments, etc.

(4) Databases – good for many small records

- very powerful and flexible storage approach
- data organization and analysis can be greatly simplified
- enhanced performance over seek / sort depending on usage
- open-sourcesoftware: SQLite(serverless), PostgreSQL, mySQL

Data Format - Others

(5) Standard scientific dataset libraries – good for large arrays

– HDF5 = Hierarchical Data Format

– NetCDF = Network Common Data Format

– open standards and open-source libraries

– provide data portability across platforms and languages

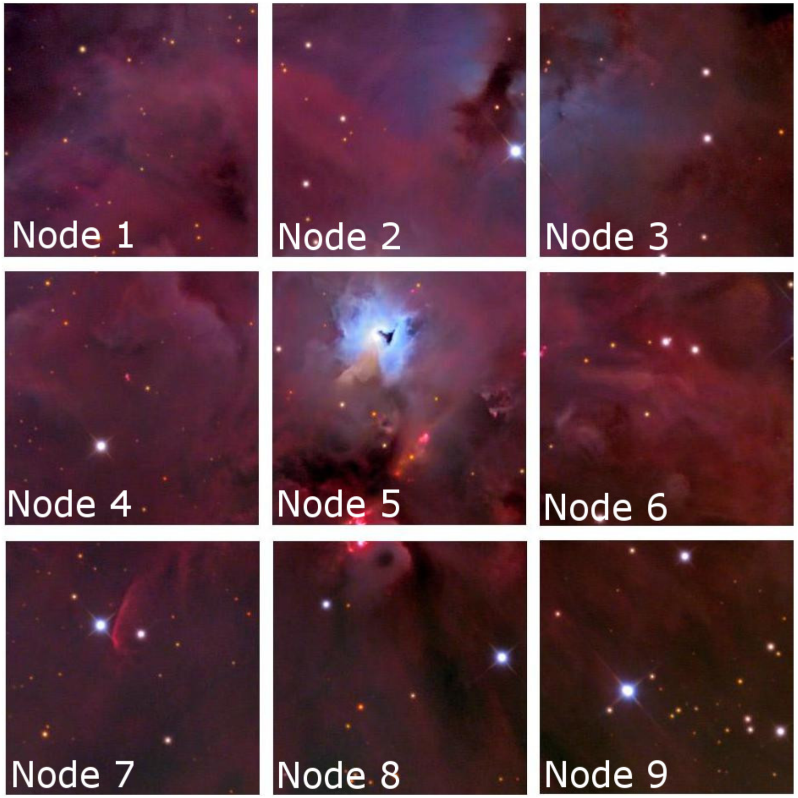
– store data in binary with optional compression

– include data description

– optionally provide parallel I/O

Using parallel I/O

- In large parallel calculations your dataset is distributed across many processors/nodes
- In this case using parallel filesystem isn't enough – you must organize parallel I/O yourself
- Data can be written as raw binary, HDF5 and NetCDF.



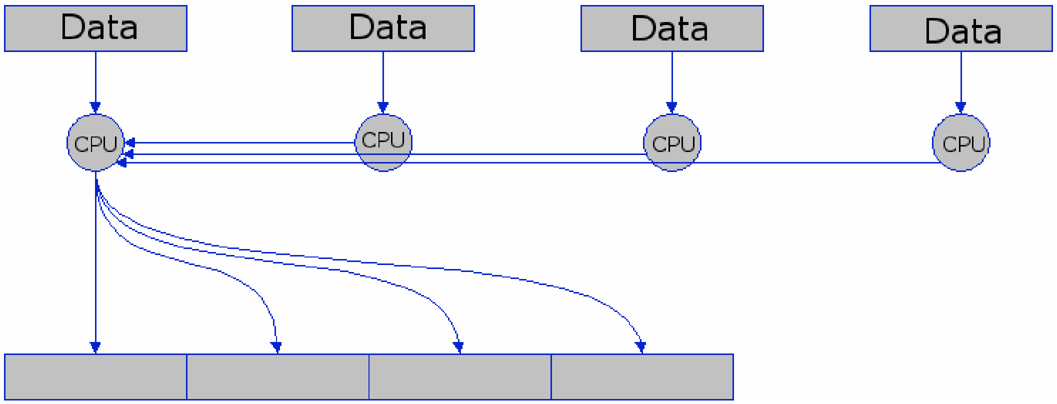
Serial I/O (single cpu)

Pros:

- trivially simple for small I/O
- some I/O libraries not parallel

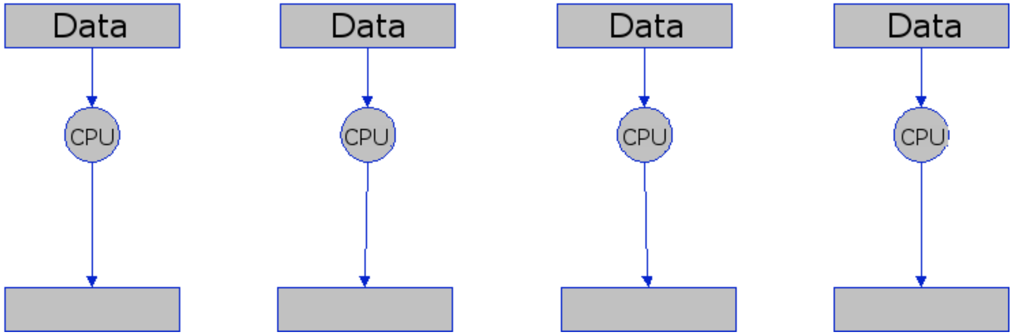
Cons:

- bandwidth limited by the rate one client can sustain
- may not have enough memory on a node to hold all data
- won't scale (built-in bottleneck)



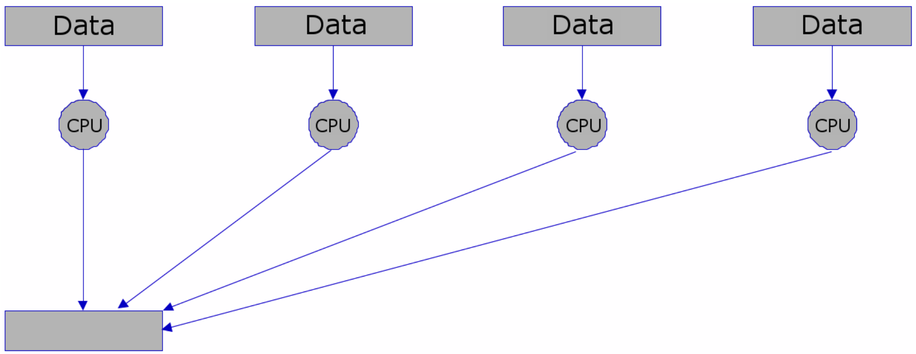
Serial I/O (N processors)

- Pros:
  - no interprocess communication or coordination necessary
  - possibly better scaling than single sequential I/O
- Cons:
  - as process counts increase, lots of (small) files, won't scale
  - data often must be post-processed into one file
  - uncoordinated I/O may swamp the filesystem (file locks!)



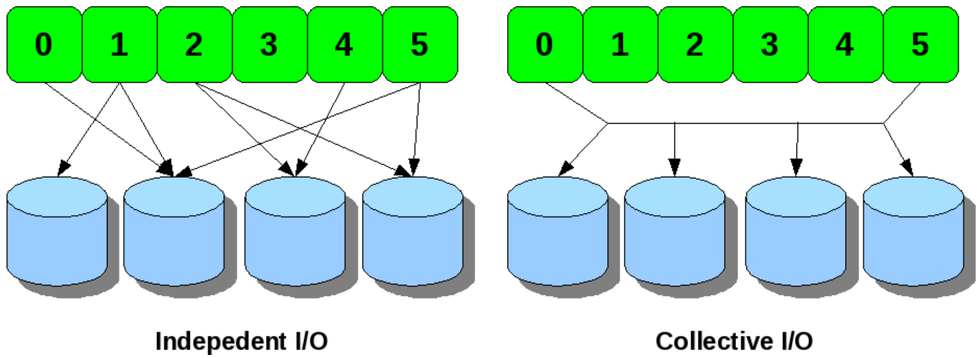
Parallel I/O (N processes to/from one file)

- Pros:
  - only one file (good for visualization, data management, storage)
  - data can be stored canonically
  - avoiding post-processing will scale if done correctly
- Cons:
  - uncoordinated I/O **will** swamp the filesystem (file locks!)
  - requires more design and thought



Parallel I/O should be collective!

- **Independent I/O** operations specify only what a single process will do
  – **Collective I/O** is coordinated access to storage by a group of processes
- functions are called by all processes participating in I/O
- allows filesystem to know more about access as a whole, more optimization in lower software layers, better performance



Parallel I/O techniques

- MPI-IO: parallel I/O part of the MPI-2 standard (1996)
  - basics covered in this webinar
- HDF5 (Hierarchical Data Format), built on top of MPI-IO
- Parallel NetCDF (Network Common Data Format), built on top of MPI-IO
- Adaptable IO System (ADIOS), built on top of MPI-IO
  - actively developed (OLCF, Sandia NL, GeorgiaTech) and used on largest HPC systems (Jaguar, Blue Gene/P)
  - external to the code XML file describing the various elements
  - can work with HDF/NetCDF

MPI-IO

- Part of the MPI-2 standard
- ROMIO is the implementation of MPI-IO in OpenMPI (default in SHARCNET), MPICH2
- Really only widely available scientific computing parallel I/O middleware
- MPI-IO exploits analogies with MPI
  - writing , sending message
  - reading , receiving message
  - file access grouped via communicator: collective operations
  - user defined MPI datatypes, e.g. for noncontiguous data layout
  - all functionality through function calls

Basic MPI-IO operations in C

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

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

- updates individual file pointer

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

- changes process's view of data in file ,

- etype is the elementary datatype

int MPI_File_read (MPI_File fh, 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_close (MPI_File* fh)

Basic MPI-IO operations in F90

MPI_FILE_OPEN (integer comm, character[] filename, integer amode, integer info, integer fh, integer ierr)

MPI_FILE_SEEK (integer fh, integer(kind=MPI_OFFSET_KIND) offset, integer whence, integer ierr)

- updates individual file pointer

MPI_FILE_SET_VIEW (integer fh, integer(kind=MPI_OFFSET_KIND) offset, integer etype, integer filetype, character[] datarep, integer info, integer ierr)

- changes process's view of data in file
- etype is the elementary datatype

MPI_FILE_READ (integer fh, type buf, integer count, integer datatype, integer[MPI_STATUS_SIZE] status, integer ierr)

MPI_FILE_WRITE (integer fh, type buf, integer count, integer datatype, integer[MPI_STATUS_SIZE] status, integer ierr)

MPI_FILE_CLOSE (integer fh)

Opening a file requires a …

- Communicator
- File name
- File handle, for all future reference to file
- File access mode ‘amode’, made up of combinations of:

MPI_MODE_RDONLY	read only
MPI_MODE_RDWR	reading and writing
MPI_MODE_WRONLY	write only
MPI_MODE_CREATE	create file if it does not exist
MPI_MODE_EXCL	error if creating file that exists
MPI_MODE_DELETE_ON_CLOSE	delete file on close
MPI_MODE_UNIQUE_OPEN	file not to be opened elsewhere
MPI_MODE_SEQUENTIAL	file to be accessed sequentially
MPI_MODE_APPEND	position all file pointers to end

- Combine it using bitwise or “|” in C or addition “+” in FORTRAN
- Info argument usually set to ‘MPI_INFO_NULL’
  - http://www.open- mpi.org/doc/v1.4/man3/MPI_File_open.3.php

Opening files

C example

```
MPI_FILE fh ;
MPI_File_open (MPI_COMM_WORLD, "test.dat" ,MPI_MODE_RDONLY,
               MPI_INFO_NULL,&fh );

... read some data here ...

MPI_File_close(&fh ) ;
```

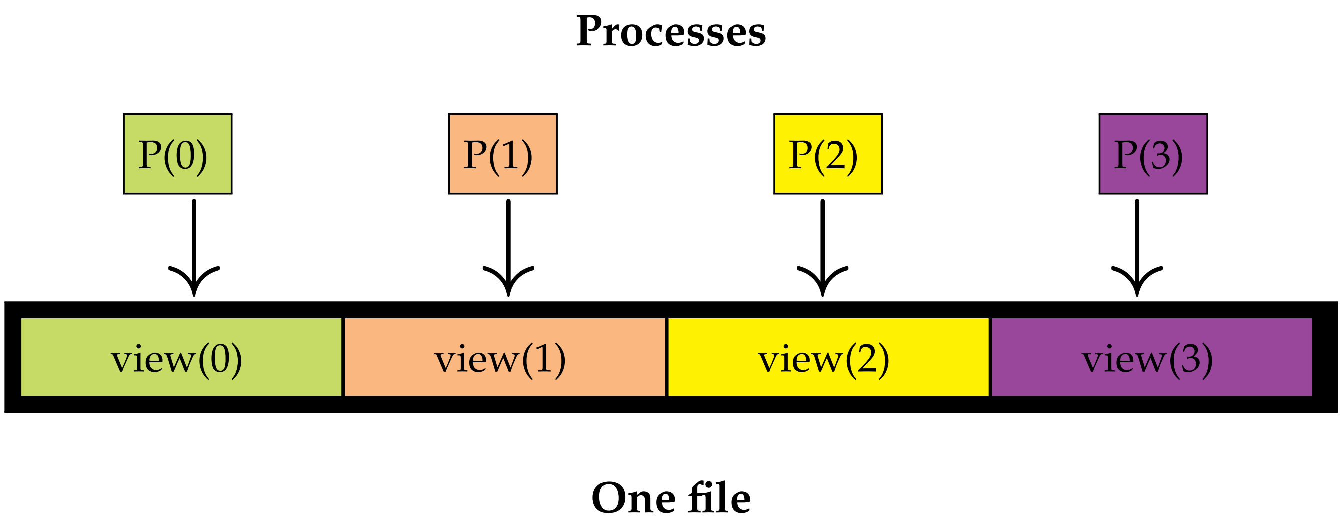
F90 example

```
integer :: fh,ierr
call MPI_FILE_OPEN(MPI_COMM_WORLD,"test.dat",
                   MPI_MODE_RDONLY, MPI_INFO_NULL, fh, ierr)

... read some data here ...

call MPI_FILE_CLOSE(fh, ierr )
```

Read / Write contiguous data



Read / Write contiguous data: example

```
#include <stdio .h>
#include <mpi.h>
int main(int argc, char **argv) {

int rank, i; char a[10];
MPI_Offset n = 10; MPI_File fh ; MPI_Status status ;

MPI_Init(&argc, &argv);

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

for (i=0; i<10; i++)
a[i] = (char)( '0' + rank); // e.g. on processor 3 creates a[0:9]='3333333333'

MPI_File_open (MPI_COMM_WORLD, "data.out" , MPI_MODE_CREATE|MPI_MODE_WRONLY,
MPI_INFO_NULL, &fh);

MPI_Offset displace = rank*n*sizeof(char); // start of the view for each processor
MPI_File_set_view (fh , displace , MPI_CHAR, MPI_CHAR, "native" ,MPI_INFO_NULL);
// note that etype and filetype are the same

MPI_File_write(fh, a, n, MPI_CHAR, &status);

MPI_File_close(&fh ) ;

MPI_Finalize ( ) ;

return 0;
```

0000000000111111111122222222223333333333

Summary: MPI-IO

- Requires no additional libraries
- Relatively easy to implement for users with MPI experience
- Writes raw data to file
  - not portable across platforms
  - hard to append new variables
  - does not include data description

NetCDF = Network Common Data Form

- Format for storing large arrays, uses MPI-IO under the hood
- Libraries for C/C++, Fortran 77/90/95/2003, Python, Java, R, Ruby, etc.
- Data stored as binary
- Self-describing, metadata in the header (can be queried by utilities)
- Portable across different architectures
- Optional compression
- Uses MPI-IO, optimized for performance
- parallel NetCDF in SHARCNET http://bit.ly/KL6L5V

Parallel NetCDF example

```
#include <stdlib.h>
#include <stdio.h>
#include <netcdf.h>
#define FILE_NAME "simple_xy.nc" #define NDIMS 2
#define NX 3
#define NY 4
int main() {
int ncid, x_dimid, y_dimid, varid; int dimids[NDIMS];
int data_out[NX][NY];
int x, y, retval;
for (x = 0; x < NX; x++)
     for (y = 0; y < NY; y++)
          data_out[x][y] = x * NY + y;
retval = nc_create(FILE_NAME, NC_CLOBBER, &ncid);
retval = nc_def_dim(ncid, "x", NX, &x_dimid);
retval = nc_def_dim(ncid, "y", NY, &y_dimid);
dimids[0] = x_dimid;
dimids[1] = y_dimid;
retval = nc_def_var(ncid, "data", NC_INT, NDIMS, dimids, &varid);
retval = nc_enddef(ncid);
retval = nc_put_var_int(ncid, varid, &data_out[0][0]);
retval = nc_close(ncid);
return 0;
}
```

Open/Create

Read/define dimensions

Define variables

Read/define attributes

Read/Write data

Close

HDF5 = Hierarchical Data Format

- Self-describing file format for large datasets, uses MPI-IO under the hood
- Libraries for C/C++, Fortran 90, Java, Python, R
- More general than NetCDF, with object-oriented description of datasets, groups, attributes, types, data spaces and property lists
- File content can be arranged into a Unix-like filesystem /path/to/resource
  - data sets containing homogeneous multidimensional images/tables/arrays
  - groups containing structures which can hold datasets and other groups
- Header information can be queried by utilities
- Optional compression (good for arrays with many similar elements)
- In SHARCNET we have both serial and parallel HDF5 (http://bit.ly/JLkKYo)

ADIOS = Adaptable I/O System

- A high-performance library for scientific I/O, also based on MPI-IO Libraries for C/C++, Fortran
- A data file and a separate external XML file describing data layout
- Allows a number of *transport methods*, including raw MPI-IO, POSIX (one-per-process posix files), NetCDF, HDF5, MPI-AIO (asynchronous output = I/O while computing)
- don't need to change the code to switch the transport method, just edit the XML file
- allows you to play with different I/O technologies without rewriting your code
- when using MPI-IO method, packs data into its own binary format
- Slowly gaining popularity, have not had any requests for it in SHARCNET yet

Summary: parallel I/O

- A wide choice of methods for parallel I/O
- The choice of a parallel library is largely dictated by the data storage format
  - raw binary:MPI-IO
  - large multidimensional arrays:NetCDF,HDF5(possiblyADIOS)–data portability, self-description
  - data on unstructured grids,particles,polygons,tetrahedra:pVTK–with extra work can also be stored with any of the above formats
- Pay attention to the disk I/O bandwidth requirements of your code: ←··· 100 200 MB/s rate is still a physical limit
- Use common sense when organizing your data: few files as opposed to many, store as binary with compression, might not need to store everything but only differences, etc.

compute canada | calcul canada

Thank you!