Parallel I/O

SciNet
www.scinet.utoronto.ca
University of Toronto
Toronto, Canada

October 19, 2010

Outline / Schedule

1. Introduction
2. File Systems and I/O
3. Data Management
4. Break
5. Parallel I/O
6. MPI-IO
7. HDF5/NETCDF

Disk I/O

Common Uses

- Checkpoint/Restart Files
- Data Analysis
- Data Organization
- Time accurate and/or Optimization Runs
- Batch and Data processing
- Database

Disk I/O

Common Bottlenecks

- Mechanical disks are slow!
- System call overhead (open, close, read, write)
- Shared file system (nfs, lustre, gpfs, etc)
- HPC systems typically designed for high bandwidth (GB/s) not IOPs
- Uncoordinated independent accesses

Disk Access Rates over Time

Average Internal Drive Access Rate (MBytes/sec)

1000
100
10
1
0.1
0.01
0.001

1950 1960 1970 1980 1990 2000 2010

RAMAC
2311
2314
3330
3350
3370
3380
3390
Spitfire
Ultrastar 18ES
Ultrastar 18ZX
Ultrastar 73LZX
Savvio 15K.1
Cheetah 15K.6
Ultrastar A7K1000

In 1956 IBM produced the first computer to include a disk drive.

The rate of performance improvement in supercomputing systems, as measured by Linpack, since 1993.

Figure by R. Ross, Argonne National Laboratory, CScADS09

Memory/Storage Latency

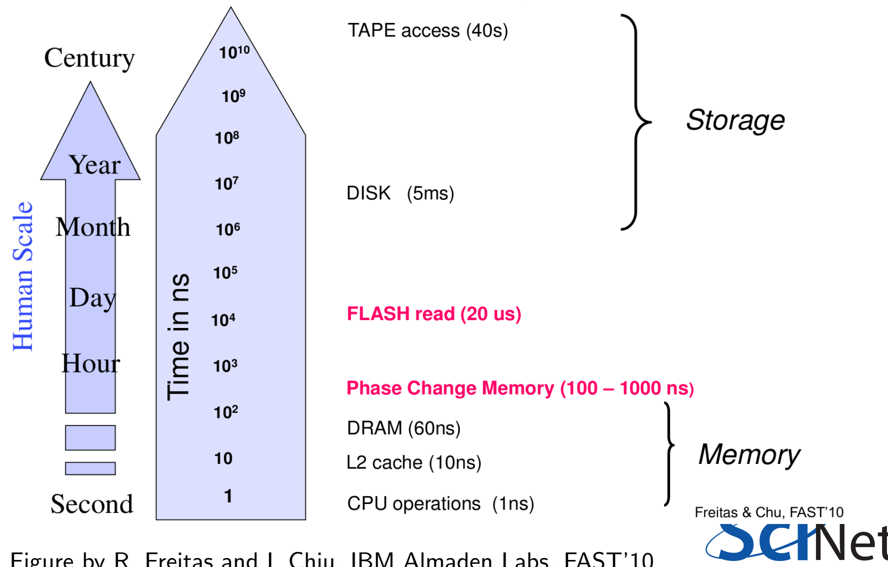


Figure by R. Freitas and L Chiu, IBM Almaden Labs, FAST'10

Definitions

IOPs

Input/Output Operations Per Second (read,write,open,close,seek)

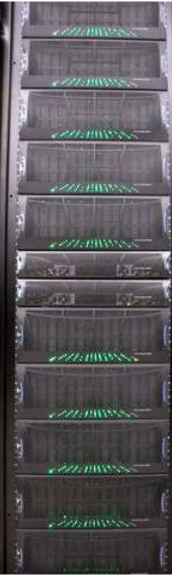
I/O Bandwidth

Quantity you read/write (think network bandwidth)

Comparisons

Device	Bandwidth (MB/s)	per-node	IOPs	per-node
SATA HDD	100	100	100	100
SSD HDD	250	250	4000	4000
SciNet	5000	1.25	30000	7.5

SciNet Filesystem



File System

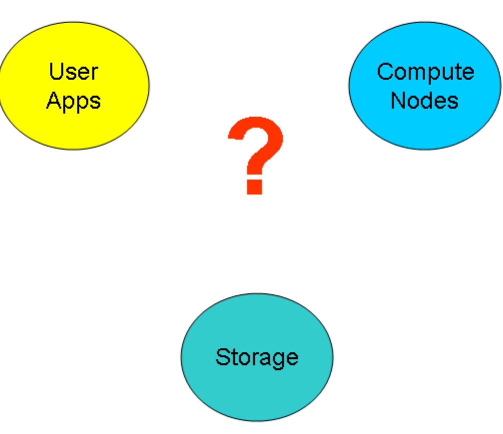
- 1,790 1TB SATA disk drives, for a total of 1.4PB
- Two DCS9900 couplets, each delivering:
  - 4-5 GB/s read/write access (bandwidth)
  - 30,000 IOPs max (open, close, seek, ...)
- Single *GPFS* file system on TCS and GPC
- I/O goes over Gb ethernet network on GPC (infiniband on TCS)
- File system is parallel!

I/O Software Stack

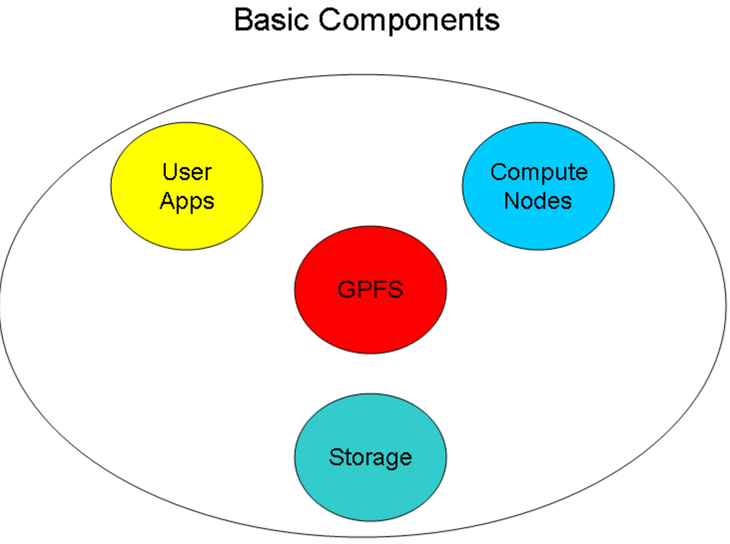
I/O Software Stack

Layer	Description
Application	
High Level I/O Library	Maps application abstractions to storage abstractions and provides data portability. (HDF5, Parallel netCDf, ADIOS)
I/O Middleware	Organizes accesses from many processes, especially collective I/O. (MPI-IO)
Parallel File System	Maintains logical space and provides efficient access to data. (GPFS, Lustre, PVFS)
I/O Hardware	

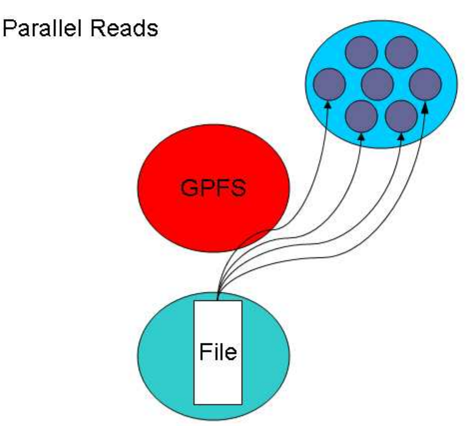
Parallel File System



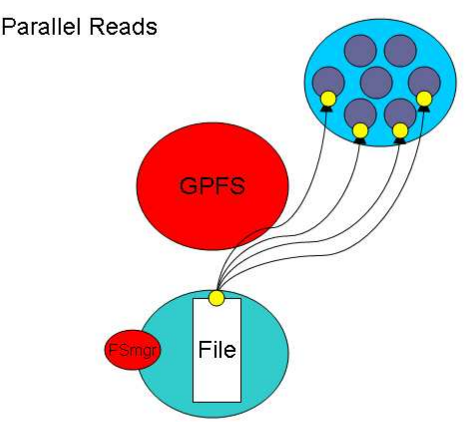
Parallel File System



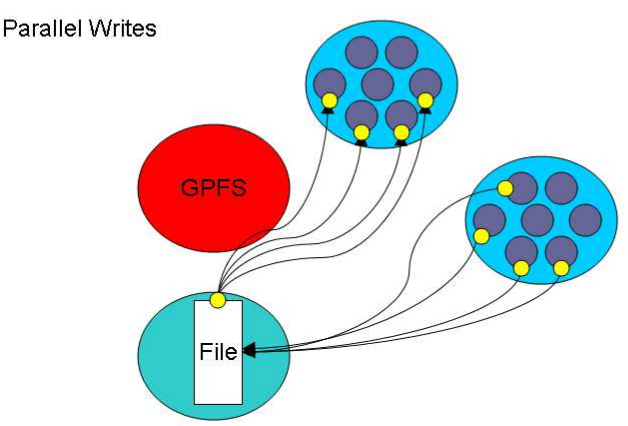
Parallel File System



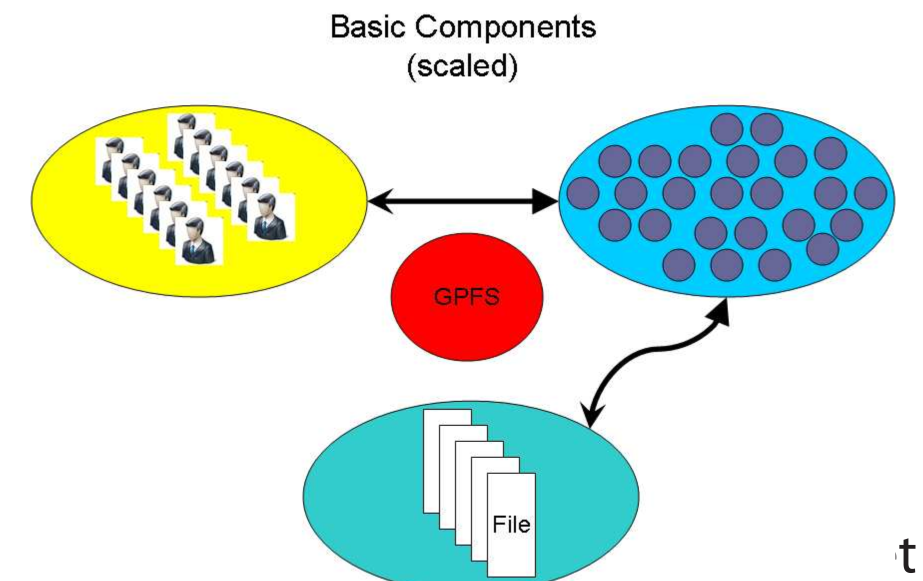
Parallel File System



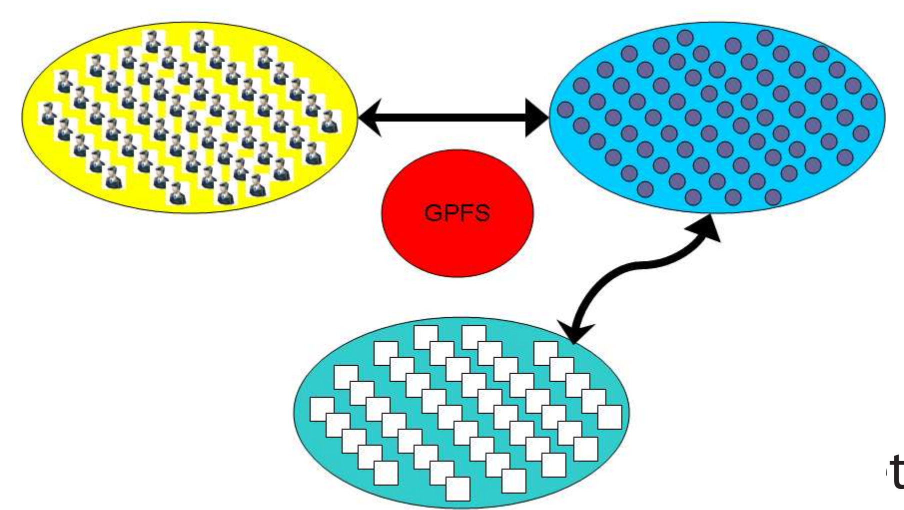
Parallel File System



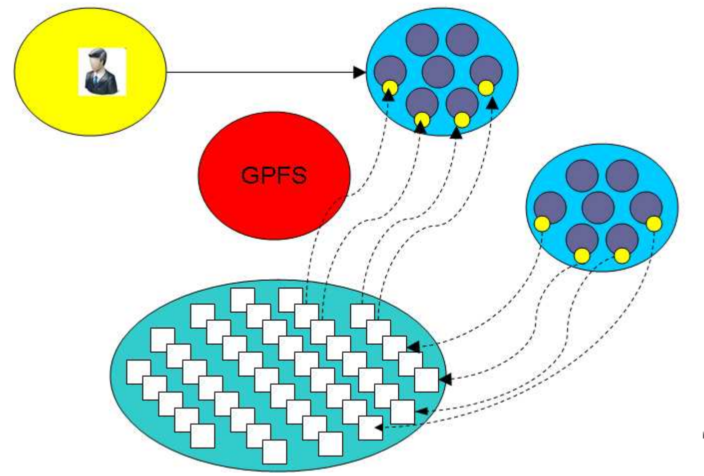
Parallel File System



Parallel File System



Parallel File System



Parallel File System

File Locks

Most parallel file systems use locks to manage concurrent file access

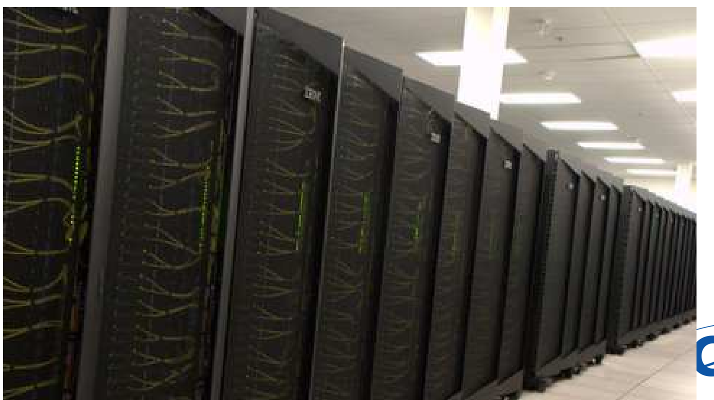
- Files are broken up into lock units
- Clients obtain locks on units that they will access before I/O occurs
- Enables caching on clients as well (as long as client has a lock, it knows its cached data is valid)
- Locks are reclaimed from clients when others desire access

Parallel File System

- Optimal for large shared files.
- Behaves poorly under many small reads and writes, high IOPs
- Your use of it affects everybody!
  (Different from case with CPU and RAM which are not shared.)
- How you read and write, your file format, the number of files in a directory, and how often you `ls`, affects every user!
- The file system is shared over the ethernet network on GPC: Hammering the file system can hurt process communications.
- File systems are not infinite!
  Bandwidth, metadata, IOPs, number of files, space, . . .

Parallel File System

- 2 jobs doing simultaneous I/O can take much longer than twice a single job duration due to disk contention and directory locking.
- SciNet: 500+ users doing I/O from 4000 nodes.
  That's a lot of sharing and contention!



I/O Best Practices

Make a plan

- 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 3 months or less.

Options?

1. Save on your departmental/local server/workstation (it is possible to transfer TBs per day on a gigabit link);
2. Apply for a project space allocation at next RAC call (but space is very limited);
3. Buy tapes through us ($100/TB) and we can archive your data to tape; HSM possibility within next 6 months;
4. Change storage format.

et

I/O Best Practices

Monitor and control usage

- Minimize use of filesystem commands like `ls` and `du`.
- Regularly check your disk usage using /scinet/gpc/bin/diskUsage.
- Warning signs which should prompt careful consideration:
  - More than 100,000 files in your space
  - Average file size less than 100 MB
- Monitor disk actions with `top` and `strace`

- RAM is always faster than disk; think about using ramdisk.
- Use `gzip` and `tar` to compress files to bundle many files into one
- Try gziping your *data* files. 30% not atypical!
- Delete files that are no longer needed
- Do "housekeeping" (`gzip`, `tar`, delete) regularly.

et

I/O Best Practices

Do's

- Write binary format files
  Faster I/O and less space than ASCII files.
- Use parallel I/O if writing from many nodes
- Maximize size of files. Large block I/O optimal!
- Minimize number of files. Makes filesystem more responsive!

Don'ts

- Don't write lots of ASCII files. Lazy, slow, and wastes space!
- Don't write many hundreds of files in a 1 directory. (File Locks)
- Don't write many small files ($<$ 10MB).
  System is optimized for large-block I/O.

1. Introduction
2. File Systems and I/O
3. Data Management
4. Break
5. Parallel I/O
6. MPI-IO
7. HDF5/NETCDF

Data Management

Formats

- ASCII
- Binary
- MetaData (XML)
- Databases
- Standard Library's (HDF5,NetCDF)

ASCII

American Standard Code for Information Interchange

Pros

- Human Readable
- Portable (architecture independent)

Cons

- Inefficient Storage
- Expensive for Read/Write (conversions)

Native Binary

100100100

Pros

- Efficient Storage (256 x floats @4bytes takes 1024 bytes)
- Efficient Read/Write (native)

Cons

- Have to know the format to read
- Portability (Endianness)

ASCII vs. binary

Writing 128M doubles

Format	/scratch (GPCS)	/dev/shm (RAM)	/tmp (disk)
ASCII	173s	174s	260s
Binary	6s	1s	20s

Syntax

Format	C	FORTRAN
ASCII	`fprintf()`	open(6,file='test',form='formatted') write(6,*)
Binary	`fwrite()`	open(6,file='test',form='unformatted') write(6)

Metadata

What is Metadata?

Data about Data

- File System: size, location, date, owner, etc.
- App Data: File format, version, iteration, etc.

Example: XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<slice_data>
  <format>UTF1000</format>
  <verstion>6.8</version>
  <img src="slice1_2010.img" alt='Slice 1 of Data'/>
  <date> January 15th, 2010 </date>
  <loc> 47 23.516 -122 02.625 </loc>
</slice_data>
```

Databases

Beyond flat files

- Very powerful and flexible storage approach
- Data organization and analysis can be greatly simplified
- Enhanced performance over seek/sort depending on usage
- Open Source Software
  - SQLite (serverless)
  - PostgreSQL
  - mySQL

“Standard” Formats

- CGNS (CFD General Notation System)
- IGES/STEP (CAD Geometry)
- HDF5 (Hierarchical Data Format)
- NetCDF (Network Common Data Format)
- disciplineX version

HDF5/NetCDF - Serial

Jonathan

1 Introduction

2 File Systems and I/O

3 Data Management

4 Break

5 Parallel I/O

6 MPI-IO

7 HDF5/NETCDF

1 Introduction

2 File Systems and I/O

3 Data Management

4 Break

5 Parallel I/O

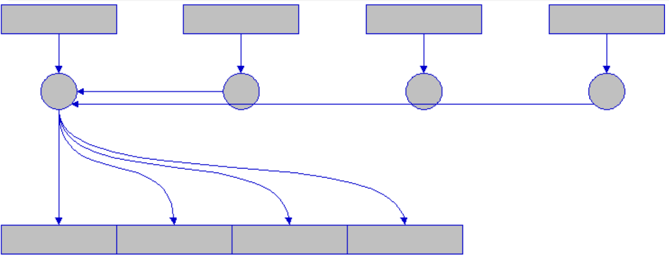
6 MPI-IO

7 HDF5/NETCDF

Common Ways of Doing Parallel I/O

Sequential I/O (only proc 0 Writes/Reads)

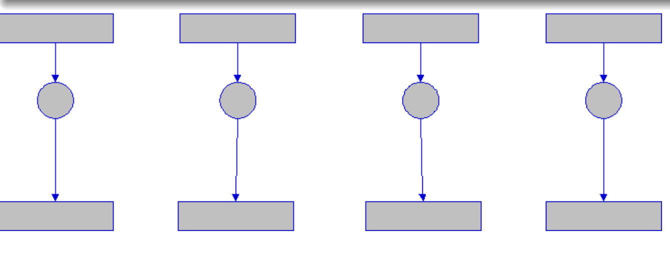
- Pro
  - Trivially simple for small I/O
  - Some I/O libraries not parallel
- Con
  - Bandwidth limited by rate one client can sustain
  - May not have enough memory on node to hold all data
  - Won't scale (built in bottleneck)



Common Ways of Doing Parallel I/O

N files for N Processes

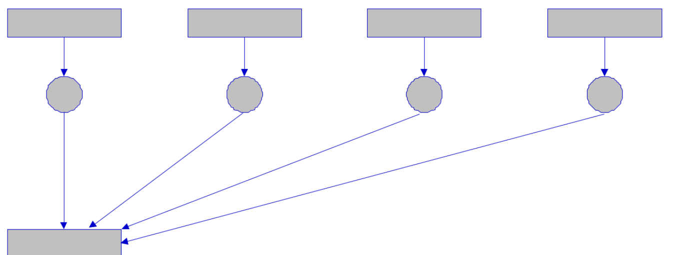
- Pro
  - No interprocess communication or coordination necessary
  - Possibly better scaling than single sequential I/O
- Con
  - As process counts increase, lots of (small) files, won't scale
  - Data often must be post-processed into one file
  - Uncoordinated I/O may swamp file system (File LOCKS!)



Common Ways of Doing Parallel I/O

All Processes Access One File

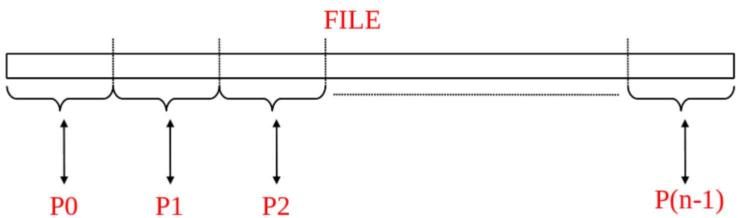
- Pro
  - Only one file
  - Data can be stored canonically, avoiding post-processing
  - Will scale if done correctly
- Con
  - Uncoordinated I/O WILL swamp file system (File LOCKS!)
  - Requires more design and thought



Parallel I/O

What is Parallel I/O?

Multiple processes of a parallel program accessing data (reading or writing) from a common file.

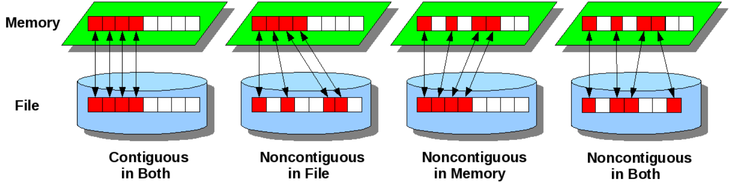


Parallel I/O

Why Parallel I/O?

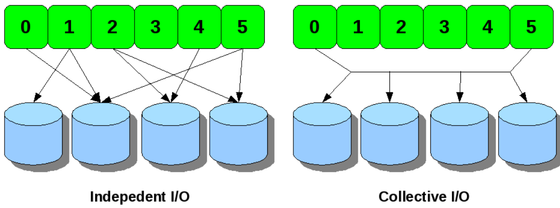
- Non-parallel I/O is simple but:
  - Poor performance (single process writes to one file)
  - Awkward and not interoperable with other tools (each process writes a separate file)
- Parallel I/O
  - Higher performance through collective and contiguous I/O
  - Single file (visualization, data management, storage, etc)
  - Works with file system not against it

Contiguous and Noncontiguous I/O



- Contiguous I/O move from a single memory block into a single file block
- Noncontiguous I/O has three forms:
  - Noncontiguous in memory, in file, or in both
- Structured data leads naturally to noncontiguous I/O (e.g. block decomposition)
- Describing noncontiguous accesses with a single operation passes more knowledge to I/O system

Independent and Collective I/O



- Independent I/O operations specify only what a single process will do
  - calls obscure relationships between I/O on other processes
- Many applications have phases of computation and I/O
  - During I/O phases, all processes read/write data
  - We can say they are collectively accessing storage
- Collective I/O is coordinated access to storage by a group of processes
  - functions are called by all processes participating in I/O
  - Allows file system to know more about access as a whole, more optimization in lower software layers, better performance

et

Parallel I/O

Available Approaches

- MPI-IO: MPI-2 Language Standard
- HDF (Hierarchical Data Format)
- NetCDF (Network Common Data Format)
- Adaptable IO System (ADIOS)
  - Actively developed (OLCF,SandiaNL,GeorgiaTech) and used on largest HPC systems (Jaguar,Blue Gene/P)
  - External to the code XML file describing the various elements
  - Uses MPI-IO, can work with HDF/NetCDF

MPI-IO

MPI-IO

MPI-IO

MPI

- MPI: Message Passing Interface
- Language-independent communications protocol used to program parallel computers.

MPI-IO: Parallel file access protocol

- MPI-IO: The parallel I/O part of the MPI-2 standard (1996).
- Many other parallel I/O solutions are built upon it.
- Versatile and better performance than standard unix I/O.
- Usually collective I/O is the most efficient.

MPI-IO

Advantages MPI-IO

- noncontiguous access of files and memory
- collective I/O
- individual and shared file pointers
- explicit offsets
- portable data representation
- can give hints to implementation/file system

- no text/formatted output!

MPI concepts

- Process: An instance of your program, often 1 per core.
- Communicator: Groups of processes and their topology. Standard communicators:
  - MPI_COMM_WORLD: all processes launched by mpirun.
  - MPI_COMM_SELF: just this process.
- Size: the number of processes in the communicator.
- Rank: a unique number assigned to each process in the communicator group.

When using MPI, each process always call MPI_INIT at the beginning and MPI_FINALIZE at the end of your program.

MPI-IO

Basic MPI code example

in C:

```
#include <mpi.h>
int main(int argc,char**argv)
{
 int rank,nprocs;
 MPI_Init(&argc,&argv);
 MPI_Comm_size
   (MPI_COMM_WORLD,&nprocs);
 MPI_Comm_rank
   (MPI_COMM_WORLD,&rank);
 ...
 MPI_Finalize();
 return 0;
}
```

in Fortran:

```
program main
include 'mpif.h'
integer rank,nprocs
integer ierr
call MPI_INIT(ierr)
call MPI_COMM_SIZE &
  (MPI_COMM_WORLD,nprocs,ierr)
call MPI_COMM_RANK &
  (MPI_COMM_WORLD,rank,ierr)
...
call MPI_FINALIZE(ierr)
return
end
```

MPI-IO

MPI-IO exploits analogies with MPI

- Writing $\leftrightarrow$ Sending message
- Reading $\leftrightarrow$ Receiving message
- File access grouped via communicator: collective operations
- User defined MPI datatypes for e.g. noncontiguous data layout
- IO latency hiding much like communication latency hiding (IO may even share network with communication)
- All functionality through function calls.

MPI-IO

Basic I/O Operations - 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)

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(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)
```

SciNet

MPI-IO

Basic I/O Operations - Fortran

```
MPI_FILE_OPEN(comm,filename,amode,info,fh,ierr)
character*(*) filename
integer comm,amode,info,fh,ierr
MPI_FILE_SEEK(fh,offset,whence,ierr)
integer(kind=MPI_OFFSET_KIND) offset
integer fh,whence,ierr
MPI_FILE_SET_VIEW(fh,disp,etype,filetype,datarep,info,ierr)
integer(kind=MPI_OFFSET_KIND) disp
integer fh,etype,filetype,info,ierr
character*(*) datarep
MPI_FILE_READ(fh,buf,count,datatype,status,ierr)
⟨type⟩ buf(*)
integer fh,count,datatype,status(MPI_STATUS_SIZE),ierr
MPI_FILE_WRITE(fh,buf,count,datatype,status,ierr)
⟨type⟩ buf(*)
integer fh,count,datatype,status(MPI_STATUS_SIZE),ierr
MPI_FILE_CLOSE(fh)
integer fh
```

SciNet

MPI-IO
Opening and closing a file

Files are maintained via file handles. Open files with `MPI_File_open`. The following codes open a file for reading, and close it right away:

in C:

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

in Fortran:

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

MPI-IO

Opening a file requires...

- communicator,
- file name,
- file handle, for all future reference to file,
- file mode, 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

- info structure, or MPI_INFO_NULL,
- In Fortran, error code is the function's last argument
  In C, the function returns the error code.

et

MPI-IO

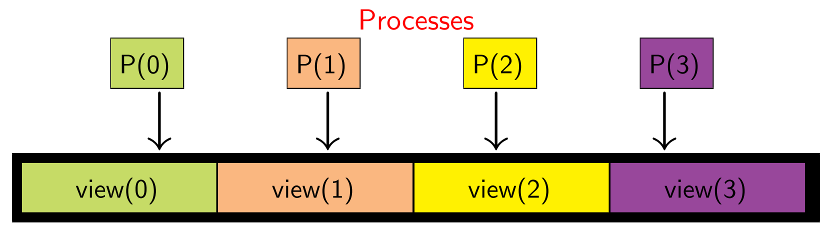
etypes, filetypes, file views

To make binary access a bit more natural for many applications, MPI-IO defines file access through the following concepts:

1. etype: Allows to access the file in units other than bytes. *Some parameters have to be given in bytes.*
2. filetype: Each process defines what part of a shared file it uses.
   - Filetypes specify a pattern which gets repeated in the file.
   - Useful for noncontiguous access.
   - For contiguous access, often etype=filetype.
3. displacement: Where to start in the file, in bytes.

Together, these specify the file view, set by **`MPI_File_set_view`**. Default view has etype=filetype=`MPI_BYTE` and displacement 0.

MPI-IO
Contiguous Data




```
int buf[...];
MPI_Offset bufsize=...;
MPI_File_open(MPI_COMM_WORLD,"file",MPI_MODE_WRONLY,
              MPI_INFO_NULL,&fh);
MPI_Offset disp=rank*bufsize*sizeof(int);
MPI_File_set_view(fh,disp,MPI_INT,MPI_INT,"native",
                  MPI_INFO_NULL);
MPI_File_write(fh,buf,bufsize,MPI_INT,MPI_STATUS_IGNORE);
MPI_File_close(&fh);
```

et

MPI-IO

Overview of all read functions

	Single task	Collective
Individual file pointer		
blocking	`MPI_File_read`	`MPI_File_read_all`
nonblocking	`MPI_File_iread` +(`MPI_Wait`)	`MPI_File_read_all_begin` `MPI_File_read_all_end`
Explicit offset		
blocking	`MPI_File_read_at`	`MPI_File_read_at_all`
nonblocking	`MPI_File_iread_at` +(`MPI_Wait`)	`MPI_File_read_at_all_begin` `MPI_File_read_at_all_end`
Shared file pointer		
blocking	`MPI_File_read_shared`	`MPI_File_read_ordered`
nonblocking	`MPI_File_iread_shared` +(`MPI_Wait`)	`MPI_File_read_ordered_begin` `MPI_File_read_ordered_end`

MPI-IO

Overview of all write functions

	Single task	Collective
Individual file pointer		
blocking	**MPI_File_write**	**MPI_File_write_all**
nonblocking	**MPI_File_iwrite**	**MPI_File_write_all_begin**
	+(MPI_Wait)	**MPI_File_write_all_end**
Explicit offset		
blocking	**MPI_File_write_at**	**MPI_File_write_at_all**
nonblocking	**MPI_File_iwrite_at**	**MPI_File_write_at_all_begin**
	+(MPI_Wait)	**MPI_File_write_at_all_end**
Shared file pointer		
blocking	**MPI_File_write_shared**	**MPI_File_write_ordered**
nonblocking	**MPI_File_iwrite_shared**	**MPI_File_write_ordered_begin**
	+(MPI_Wait)	**MPI_File_write_ordered_end**

MPI-IO

Collective vs. single task

After a file has been opened and a fileview is defined, processes can independently read and write to their part of the file.

If the IO occurs at regular spots in the program, which different processes reach the same time, it will be better to use collective I/O: These are the `_all` versions of the MPI-IO routines.

Two file pointers

An MPI-IO file has two different file pointers:

1. individual file pointer: one per process.
2. shared file pointer: one per file: `_shared`/`_ordered`

'Shared' doesn't mean 'collective', but does imply synchronization!

MPI-IO

Strategic considerations

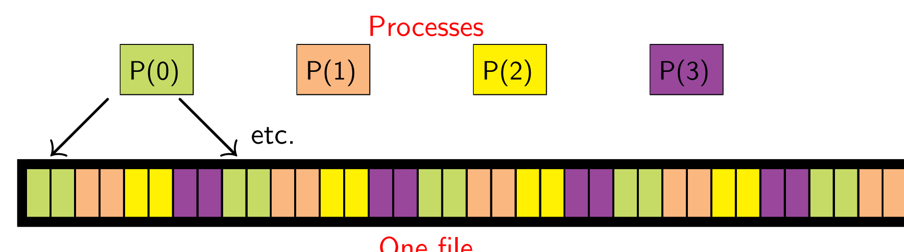
Pros for single task I/O

- One can virtually always use only indivivual file pointers,
- If timings variable, no need to wait for other processes

Cons

- If there are interdependences between how processes write, there may be collective I/O operations may be faster.
- Collective I/O can collect data before doing the write or read.

True speed depends on file system, size of data to write and implementation.



Filetypes to the rescue!

- Define a 2-etype basic MPI_Datatype.
- Increase its size to 8 etypes.
- Shift according to rank to pick out the right 2 etypes.
- Use the result as the filetype in the file view.
- Then gaps are automatically skipped.

et

MPI-IO

Overview of data/filetype constructors

Function	Creates a...
MPI_Type_contiguous	contiguous datatype
MPI_Type_vector	vector (strided) datatype
MPI_Type_indexed	indexed datatype
MPI_Type_indexed_block	indexed datatype w/uniform block length
MPI_Type_create_struct	structured datatype
MPI_Type_create_resized	type with new extent and bounds
MPI_Type_create_darray	distributed array datatype
MPI_Type_create_subarray	n-dim subarray of an n-dim array
...	

Before using the create type, you have to do **MPI_Commit**.

MPI-IO

Accessing a noncontiguous file type

in C:

```
MPI_Datatype contig, ftype;
MPI_Datatype etype=MPI_INT;
MPI_Aint extent=sizeof(int)*8; /* in bytes! */
MPI_Offset d=2*sizeof(int)*rank; /* in bytes! */
MPI_Type_contiguous(2,etype,&contig);
MPI_Type_create_resized(contig,0,extent,&ftype);
MPI_Type_commit(&ftype);
MPI_File_set_view(fh,d,etype,ftype,"native",
                  MPI_INFO_NULL);
```

MPI-IO

Accessing a noncontiguous file type

in Fortran:

```
integer :: etype,extent,contig,ftype,ierr
integer(kind=MPI_OFFSET_KIND) :: d
etype=MPI_INT
extent=4*8
d=4*rank
call MPI_TYPE_CONTIGUOUS(2,etype,contig,ierr)
call MPI_TYPE_CREATE_RESIZED(contig,0,extent,ftype,ierr)
call MPI_TYPE_COMMIT(ftype,ierr)
call MPI_FILE_SET_VIEW(fh,d,etype,ftype,"native",
                       MPI_INFO_NULL,ierr)
```

MPI-IO

File data representation

native: *Data is stored in the file as it is in memory: no conversion is performed. No loss in performance, but not portable.*

internal: *Implementation dependent conversion. Portable across machines with the same MPI implementation, but not across different implementations.*

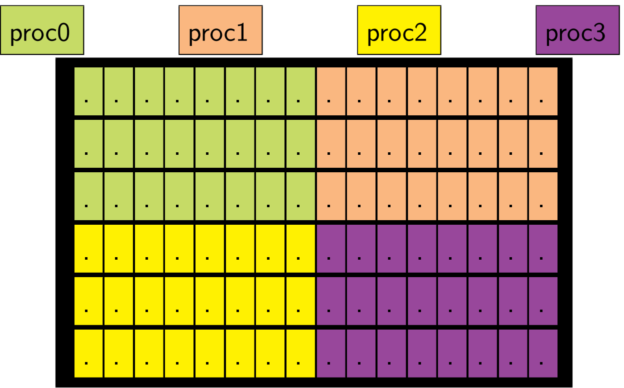
external32: *Specific data representation, basically 32-bit big-endian IEEE format. See MPI Standard for more info. Completely portable, but not the best performance.*

These have to be given to `MPI_File_set_view` as strings.

MPI-IO

More noncontiguous data: subarrays

What if there's a 2d matrix that is distributed across processes?



Common cases of noncontiguous access → specialized functions:
`MPI_File_create_subarray` & **`MPI_File_create_darray`**.

et

MPI-IO

More noncontiguous data: subarrays

```
int gsizes[2]={16,6};
int lsizes[2]={8,3};
int psizes[2]={2,2};
int coords[2]={rank%psizes[0],rank/psizes[0]};
int starts[2]={coords[0]*lsizes[0],coords[1]*lsizes[1]};
MPI_Type_create_subarray(2,gsizes,lsizes,starts,,
                         MPI_ORDER_C,MPI_INT,&filetype);
MPI_Type_commit(&filetype);
MPI_File_set_view(fh,0,MPI_INT,filetype,"native",
                  MPI_INFO_NULL);
MPI_File_write_all(fh,local_array,local_array_size,MPI_INT,
                  MPI_STATUS_IGNORE);
```

Tip

`MPI_Cart_create` can be useful to compute coords for a proc.

et

MPI-IO

Example: N-body MD checkpointing

Challenge

- Simulate n-body molecular dynamics, Lennard-Jones potential.
- Parallel MPI run: atoms distributed.
- At intervals, checkpoint the state of system to 1 file.
- Restart should be allowed to use more or less mpi processes.
- Restart should be efficient.

State of the system: total of `tot` atoms with properties:

```c
struct Atom {
 double     q[3];
 double     p[3];
 long long tag;
 long long id;
};
```

```fortran
type atom
 double precision :: q(3)
 double precision :: p(3)
 integer(kind=8)  :: tag
 integer(kind=8)  :: id
end type atom
```

SciNet

MPI-IO

Example: N-body MD checkpointing

Issues

- Atom data more than array of doubles: indices etc.
- Writing problem: processes have different # of atoms
  how to define views, subarrays, ... ?
- Reading problem: not known which process gets which atoms.
- Even worse if number of processes is changed in restart.

Approach

- Abstract the atom datatype.
- Compute where in file each proc. should write + how much.
- Store that info in header.
- Restart with same nprocs is then straightforward.
- Different nprocs: MPI exercise outside scope of 1-day class.

.et

MPI-IO

Example: N-body MD checkpointing

Defining the Atom etype

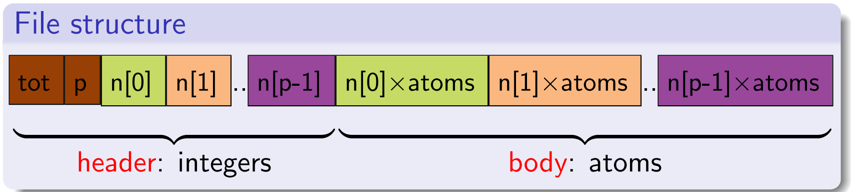
```
struct Atom atom;
int i,len[4]={3,3,1,1};
MPI_Aint addr[5];
MPI_Aint disp[4];
MPI_Get_address(&atom,&(addr[0]));
MPI_Get_address(&(atom.q[0]),&(addr[1]));
MPI_Get_address(&(atom.p[0]),&(addr[2]));
MPI_Get_address(&atom.tag,&(addr[3]));
MPI_Get_address(&atom.id,&(addr[4]));
for (i=0;i<4;i++)
  disp[i]=addr[i]-addr[0];
MPI_Datatype t[4]
  ={MPI_DOUBLE,MPI_DOUBLE,MPI_LONG_LONG,MPI_LONG_LONG};
MPI_Type_create_struct(4,len,disp,t,&MPI_ATOM);
MPI_Type_commit(&MPI_ATOM);
```

et

MPI-IO

Example: N-body MD checkpointing

File structure



Functions

- `void cpwrite(struct Atom* a, int n, MPI_Datatype t, char* f, MPI_Comm c)`
- `void cpread(struct Atom* a, int* n, int tot, MPI_Datatype t, char* f, MPI_Comm c)`
- `void redistribute()` → *consider done*

MPI-IO: Checkpoint writing

```
void cpwrite(struct Atom*a,int n,MPI_Datatype t,char*f,MPI_Comm c){
 int p,r;
 MPI_Comm_size(c,&p);
 MPI_Comm_rank(c,&r);
 int header[p+2];
 MPI_Allgather(&n,1,MPI_INT,&(header[2]),1,MPI_INT,c);
 int i,n_below=0;
 for(i=0;i<r;i++)
  n_below+=header[i+2];
 MPI_File h;
 MPI_File_open(c,f,MPI_MODE_CREATE|MPI_MODE_WRONLY,MPI_INFO_NULL,&h);
 if(r==p-1){
  header[0]=n_below+n;
  header[1]=p;
  MPI_File_write(h,header,p+2,MPI_INT,MPI_STATUS_IGNORE);
 }
 MPI_File_set_view(h,(p+2)*sizeof(int),t,t,"native",MPI_INFO_NULL);
 MPI_File_write_at(h,n_below,a,n,t,MPI_STATUS_IGNORE);
 MPI_File_close(&h);
}
```

SciNet

MPI-IO
Example: N-body MD checkpointing

Reading and redistributing atoms

- Copy example lj code to your own directory.
```
$ cp -r ~rzon/Courses/lj .
$ cd lj
```
- Get your environment setup with
```
$ source ./envsetup
```
- Type 'make' to build.
```
$ make
```
- Run:
```
$ mpirun -np 8 lj run.ini
```
  Creates 70778.cp as a checkpoint (try `ls -l`).
- Rerun to see that it successfully reads the checkpoint.
- Run again with different # of processors. What happens?

MPI-IO

Example: N-body MD checkpointing

Reading and redistributing atoms

- The lj program has an option to produce an xsf file with the trajectory, which can be displayed in vmd.
- Edit `run.ini`, change the line “0” to “5”
- *Careful: xsf is an ascii format, and the file quickly gets huge!*
- Do:

```
mpirun -np 8 lj run.ini
```

- Note difference in speed!
- Open in vmd:

```
$ vmd pos.xsf
```

Checking binary files

Interactive binary file viewer in `~rzon/Courses/lj`: cbin
Useful for quickly checking your binary format, without having to write a test program.

- Start the program with:
  `$ cbin 70778.cp`
- Gets you a prompt, with the file loaded.
- Commands at the prompt are a letter plus optional number.
- E.g. `i2` reads 2 integers, `d6` reads 6 doubles.
- Has support to reverse the endian-ness, switching between loaded files, and moving the file pointer.
- Type '?' for a list of commands.
- Check the format of the file.

Good References on MPI-IO

- W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface* (MIT Press, 1999).
- J. H. May, *Parallel I/O for High Performance Computing* (Morgan Kaufmann, 2000).
- W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, *MPI-2: The Complete Reference: Volume 2, The MPI-2 Extensions* (MIT Press, 1998).

HDF5/NETCDF