MPI-IO

Ramses van Zon

SciNet HPC Consortium
Toronto, Canada

February 27, 2013

MPI-IO

MPI

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

MPI-IO: Parallel file access protocol

- MPI-IO: The parallel I/O part of the MPI-2 standard (1996).
- Started at IBM Watson
- Maps I/O reads and write to message passing
- 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-IO

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 boiler-plate code:

in C:

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

in Fortran:

```
program main
 use mpi
 integer :: rank,nprocs,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)
end program main
```

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

Get examples and setup environment

```
$ ssh -X <user>@login.scinet.utoronto.ca
$ ssh -X gpc04
$ cp -r /scinet/course/parIO .
$ cd parIO
$ source parallellibs
$ cd samples/mpiio
$ make
...
$ mpirun -np 4 ./helloworldc
Rank 0 has message <Hello >
Rank 1 has message <World!>
Rank 2 has message <Hello >
Rank 3 has message <World!>
$ cat helloworld.txt
Hello World!Hello World!$
```

MPI-IO

helloworldc.c

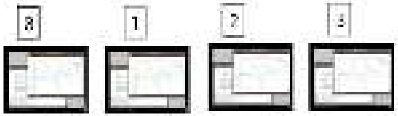
```c
#include <string.h>
#include <mpi.h>
int main(int argc,char**argv) {
 int rank,size;
 MPI_Offset offset;
 MPI_File file;
 MPI_Status status;
 const int msgsize=6;
 char message[msgsize+1];
 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&size);
 MPI_Comm_rank(MPI_COMM_WORLD,&rank);
 if(rank%2)strcpy(message,"World!");else strcpy(message,"Hello ");
 offset=msgsize*rank;
 MPI_File_open(MPI_COMM_WORLD,"helloworld.txt",
              MPI_MODE_CREATE|MPI_MODE_WRONLY,
              MPI_INFO_NULL,&file);
 MPI_File_seek(file,offset,MPI_SEEK_SET);
 MPI_File_write(file,message,msgsize,MPI_CHAR,&status);
 MPI_File_close(&file);
 MPI_Finalize();
}
```

helloworldf.f90

```
program MPIIO_helloworld
 use mpi
 implicit none
 integer(mpi_offset_kind) :: offset
 integer,dimension(mpi_status_size) :: wstatus
 integer,parameter :: msgsize=6
 character(msgsize):: message
 integer :: ierr,rank,comsize,fileno
 call MPI_Init(ierr)
 call MPI_Comm_size(MPI_COMM_WORLD,comsize,ierr)
 call MPI_Comm_rank(MPI_COMM_WORLD,rank,ierr)
 if (mod(rank,2) == 0) then
      message = "Hello "
 else
      message = "World!"
 endif
 offset = rank*msgsize
 call MPI_File_open(MPI_COMM_WORLD,"helloworld.txt",&
   ior(MPI_MODE_CREATE,MPI_MODE_WRONLY),MPI_INFO_NULL,fileno,ierr)
 call MPI_File_seek(fileno,offset,MPI_SEEK_SET,ierr)
 call MPI_File_write(fileno,message,msgsize,MPI_CHARACTER,wstatus,
 call MPI_File_close(fileno,ierr)
 call MPI_Finalize(ierr)
end program MPIIO_helloworld
```

MPI-IO

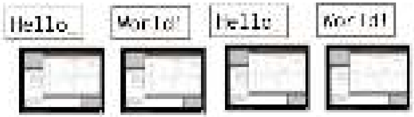
MPI-IO
Hello
World



```
mpirun -np 4 ./helloworldc
```

MPI-IO

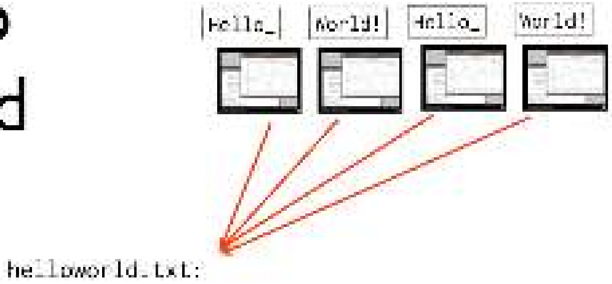
MPI-IO
Hello
World



```
if ((rank % 2) == 0)
   strcpy (message, "Hello ");
else
   strcpy (message, "World!");
```

MPI-IO

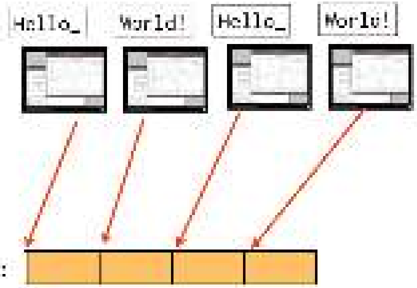
MPI-IO
Hello
World



```
MPI_File_open(MPI_COMM_WORLD, "helloworld.txt", MPI_MODE_CREATE|MPI_MODE_WRONLY,
                  MPI_INFO_NULL, &file);
```

MPI-IO

MPI-IO
Hello
World

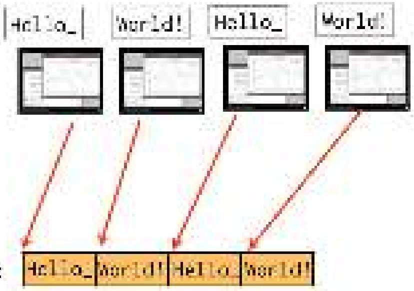


```
offset = (msgsize*rank);

MPI_File_seek(file, offset, MPI_SEEK_SET);
```

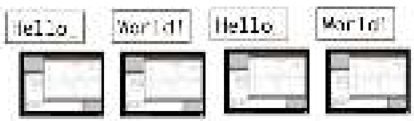
MPI-IO

MPI-IO
Hello
World



```
MPI_File_write(file, message, msgsize, MPI_CHAR, &status);
```

MPI-IO



MPI-IO Hello World

```
helloworld.txt:
```

Hello_	World!	Hello_	World!

```
MPI_File_close(&file);
```

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)
```

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
```

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.

MPI-IO

etypes, filetypes, file views

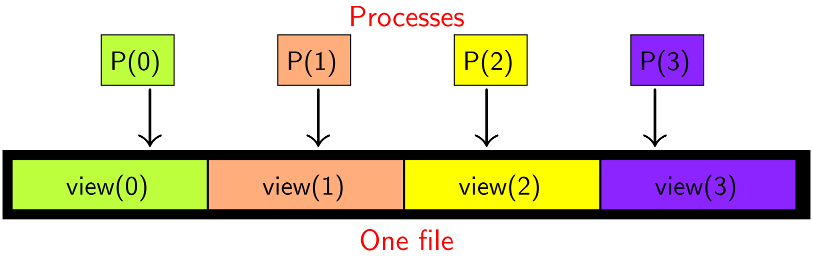
To make binary access a bit more natural for many applications, MPI-IO defines file access through the following concepts:

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);
```

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_File_read_all_begin`
	`+(MPI_Wait)`	`MPI_File_read_all_end`
Explicit offset		
blocking	`MPI_File_read_at`	`MPI_File_read_at_all`
nonblocking	`MPI_File_iread_at`	`MPI_File_read_at_all_begin`
	`+(MPI_Wait)`	`MPI_File_read_at_all_end`
Shared file pointer		
blocking	`MPI_File_read_shared`	`MPI_File_read_ordered`
nonblocking	`MPI_File_iread_shared`	`MPI_File_read_ordered_begin`
	`+(MPI_Wait)`	`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

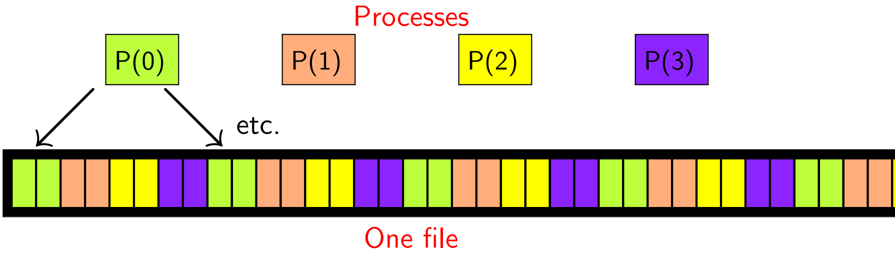
Cons

- 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.

MPI-IO

Noncontiguous Data



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.

MPI-IO

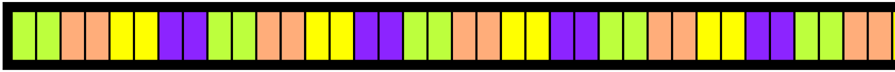
Overview of data/filetype constructors

Function	Creates a...
`MPI_Type_contiguous`	contiguous datatype
`MPI_Type_vector`	vector (strided) datatype
`MPI_Type_create_indexed`	indexed datatype
`MPI_Type_create_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 created type, you have to do **`MPI_Commit`**.

MPI-IO

Accessing a noncontiguous file type

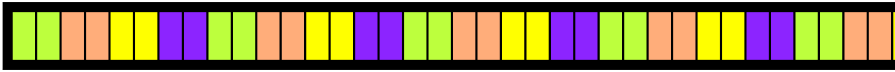


in C:

```
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

More examples

In the samples/mpiio directory:

- fileviewc.c
- fileviewf.f90
- helloworld-noncontigc.c
- helloworld-noncontigf.f90
- writeatc.c
- writeatf.f90
- writeatallc.c
- writeatallf.f90

MPI-IO

File data representation

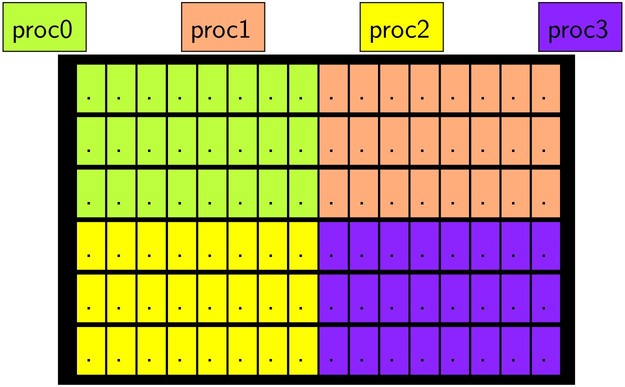
native:	*Data is stored in the file as it is in memory: no conversion is performed. No loss in performance, but not portable.*
internal:	*Implementation dependent conversion. Portable across machines with the same MPI implementation, but not across different implementations.*
external32:	*Specific data 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`**.

MPI-IO

More noncontiguous data: subarrays

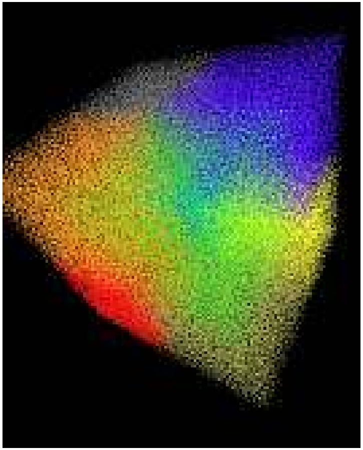
```
int gsizes[2]={16,6};
int lsizes[2]={8,3};
int psizes[2]={2,2};
int coords[2]={rank%psizes[0],rank/psizes[0]};
int starts[2]={coords[0]*lsizes[0],coords[1]*lsizes[1]};
MPI_Type_create_subarray(2,gsizes,lsizes,starts,
                              MPI_ORDER_C,MPI_INT,&filetype);
MPI_Type_commit(&filetype);
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.

MPI-IO

Example: N-body MD checkpointing



Challenge

- Simulating n-body molecular dynamics, interacting through a 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.

MPI-IO

Example: N-body MD checkpointing

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

```
struct Atom {                      type atom
 double    q[3];                    double precision :: q(3)
 double    p[3];                    double precision :: p(3)
 long long tag;                     integer(kind=8)  :: tag
 long long id;                      integer(kind=8)  :: id
};                                 end type atom
```

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.

MPI-IO

Example: N-body MD checkpointing

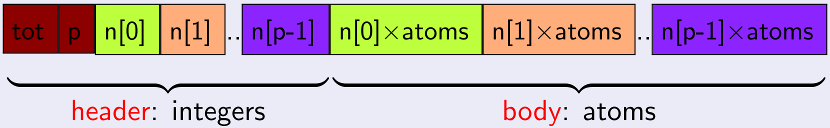
Defining the Atom etype

```
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);
```

MPI-IO

Example: N-body MD checkpointing

File structure



Functions

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

MPI-IO: Checkpoint writing

```
void cp_write(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_all(h,n_below,a,n,t,MPI_STATUS_IGNORE);
 MPI_File_close(&h);
}
```

MPI-IO

Example: N-body MD checkpointing

- Code in the samples directory

```
$ ssh <user>@login.scinet.utoronto.ca
$ ssh gpc04
$ cp -r /scinet/course/parIO .
$ cd parIO
$ source parallellibs
$ cd samples/lj
$ make
...
$ 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

Checking binary files

Interactive binary file viewer in `samples/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: The Complete Reference: Vol. 2, MPI-2 Extensions* (MIT Press, 1998).
- The `man` pages for various MPI commands.
- http://www.mpi-forum.org/docs/