Advanced Parallel Programming with MPI

Bruno C. Mundim

SciNet HPC Consortium

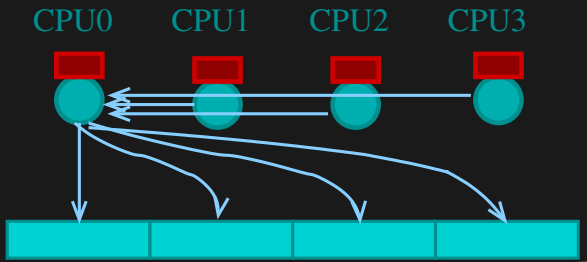
October 29, 2021

MPI-IO

Common Ways of Doing Parallel I/O

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

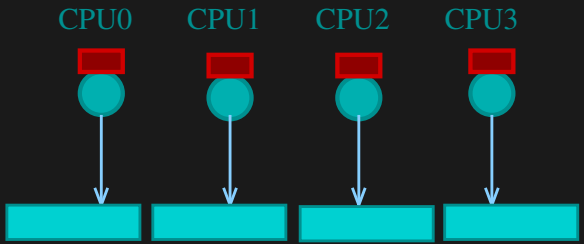
- Pro
  - Trivially simple for small I/O
  - Some I/O libraries not parallel
- Con
  - Bandwidth limited by rate one client can sustain
  - May not have enough memory on node to hold all data
  - Won't scale (built in bottleneck)



Common Ways of Doing Parallel I/O

N files for N Processes

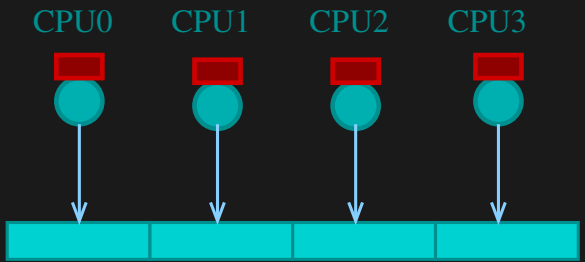
- Pro
  - No interprocess communication or coordination necessary
  - Possibly better scaling than single sequential I/O
- Con
  - As process counts increase, lots of (small) files, won't scale
  - Data often must be post-processed into one file
  - Uncoordinated I/O may swamp file system (File LOCKS!)



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.

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 Non-contiguous 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

MPI-IO

- Would like I/O to be parallel and not serial
- But writing one file per process is inconvenient and inefficient.
- MPI-IO = The parallel I/O part of the MPI-2 standard.
- Many other parallel I/O solutions are built upon it.
- Versatile and better performance than standard unix IO.
- Usually collective I/O is the most efficient.

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. non-contiguous data layout
- IO latency hiding much like communication latency hiding

  (IO may even share network with communication)
- All functionality through function calls.

MPI-IO Example: Hello World

- cd advanced-mpi/mpiio
- Compile and run it:

```
$ source $SCRATCH/advanced-mpi/setup
$ cd $SCRATCH/advanced-mpi/mpiio
$ make
$ srun -n 4 ./helloworldc
Rank 1 has message <World!>
Rank 3 has message <World!>
Rank 0 has message <Hello >
Rank 2 has message <Hello >
$ cat helloworld.txt
Hello World!Hello World!$
```

MPI-IO Example: Hello World

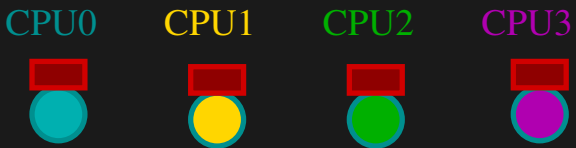
```c
#include <stdio.h>
#include <string.h>                              /* helloworldc.c */
#include <mpi.h>
int main(int argc, char **argv) {
  int rank, size;
  MPI_Offset offset;
  MPI_File   file;
  MPI_Status status;
  const int msgsize=6;
  char message[msgsize+1];
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if(rank%2)strcpy(message,"World!");else strcpy(message,"Hello ");
  printf("Rank %d has message <%s>\n", rank, message);
  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();
}
```

MPI-IO Example: Hello World

```
program MPIIO_helloworld
    use mpi
    implicit none                                          ! helloworldf.f90
    integer(mpi_offset_kind) :: offset
    integer, dimension(mpi_status_size) :: wstatus
    integer :: fileno, ierr, rank, comsize
    integer, parameter :: msgsize=6
    character(msgsize) :: message
    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,ierr)
    call MPI_File_close(fileno, ierr)
    call MPI_Finalize(ierr)
end program MPIIO_helloworld
```

MPI-IO Example: Hello World

```
srun -n 4 ./helloworldc
```

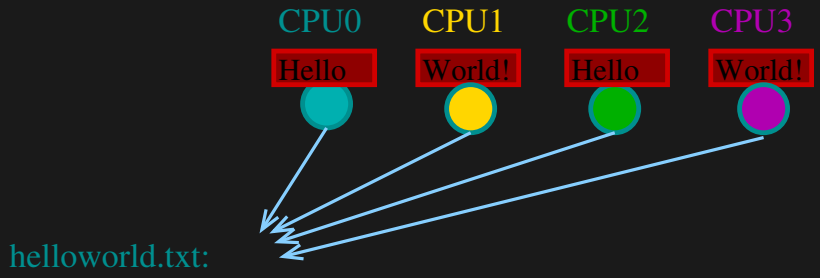


MPI-IO Example: Hello World

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

MPI-IO Example: Hello World

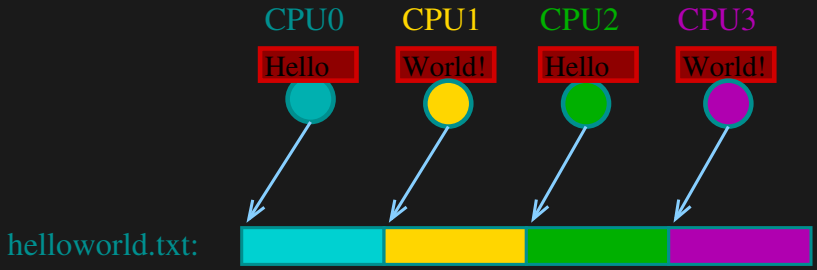
```
MPI_File_open(MPI_COMM_WORLD, "helloworld.txt",
              MPI_MODE_CREATE|MPI_MODE_WRONLY,
              MPI_INFO_NULL, &file);
```



MPI-IO Example: Hello World

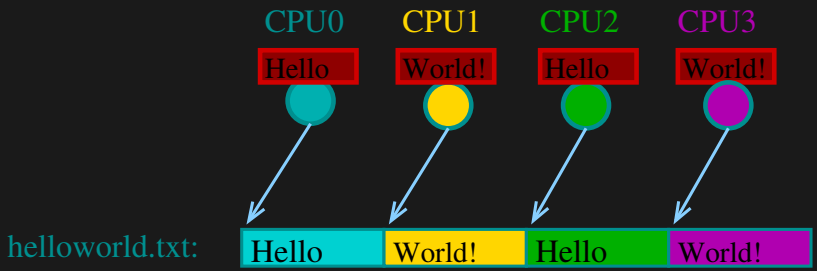
```
offset = (msgsize*rank);

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



MPI-IO Example: Hello World

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



MPI-IO Example: Hello World

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

Basic IO 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)
```

Basic IO Operations (Fortran)

```
MPI_FILE_OPEN(comm,filename,amode,info,fh,err)
character*(*) filename
integer comm,amode,info,fh,err

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

MPI_FILE_SET_VIEW(fh,disp,etype,filetype,datarep,info,err)
integer(kind=MPI_OFFSET_KIND) disp
integer fh,etype,filetype,info,err
character*(*) datarep

MPI_FILE_READ(fh,buf,count,datatype,status,err)
<type> buf(*)
integer fh,count,datatype,status(MPI_STATUS_SIZE),err

MPI_FILE_WRITE(fh,buf,count,datatype,status,err)
<type> buf(*)
integer fh,count,datatype,status(MPI_STATUS_SIZE),err

MPI_FILE_CLOSE(fh)
integer fh
```

Opening and closing a file

As in regular I/O, files are maintained through file handles. A file gets opened with MPI_File_open. E.g. 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, err
call MPI_FILE_OPEN(MPI_COMM_WORLD,"test.dat",MPI_MODE_RDONLY,MPI_INFO_NULL,fh,err)
call MPI_FILE_CLOSE(fh,err)
```

Opening a file requires...

- communicator,
- file name,
- file handle, for all future reference to file,
- info structure, or `MPI_INFO_NULL`,
- file mode, made up of combinations of the following

Opening a file requires...

- communicator,
- file name,
- file handle, for all future reference to file,
- info structure, or `MPI_INFO_NULL`,
- file mode, made up of combinations of the following

- `MPI_MODE_RDONLY`: read only
- `MPI_MODE_RDWR`: reading and writing
- `MPI_MODE_WRONLY`: write only
- `MPI_MODE_CREATE`: create the file if it does not exist
- `MPI_MODE_EXCL`: error if creating a 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 file-end

etypes, filetypes, file views

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

- displacement: Where to start in the file.
- etype: Allows to access the file in units other than bytes.
- filetype: Each process defines what part of a shared file it uses.
  - Filetypes specify a pattern which gets repeated in the file.
  - Useful for non-contiguous access.
  - For contiguous access, often etype=filetype.

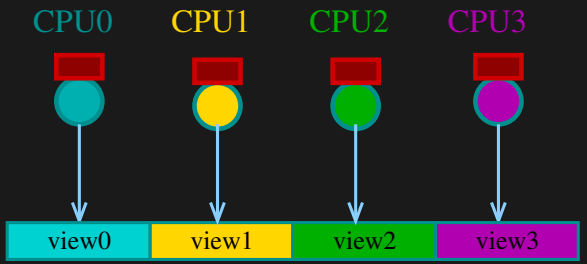
Together, these three specify the file view.

File views have to be defined collectively with `MPI_File_set_view`.

If no view is defined, a default view is active, with etype `MPI_BYTE`, and displacement 0.

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);
```



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

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

Choices

Collective?

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

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

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

Choices

Strategic considerations

Pros for single task I/O:

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

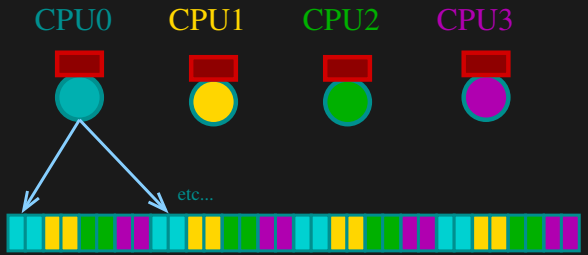
Cons:

- If there are interdependences between how processes write, there may be faster collective I/O operations.
- 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.

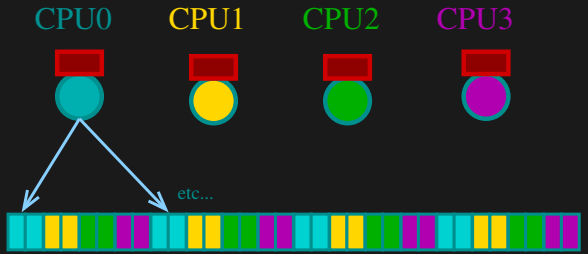
Non-contiguous data

What if the data in the file is supposed to be as follows?



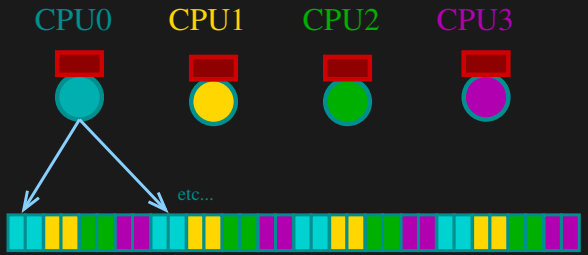
Non-contiguous data

What if the data in the file is supposed to be as follows?



- Filetypes can help!
- Or custom MPI data types (also useful in high dimensional ghost cells).

Non-contiguous data



- 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 File View


```
int MPI_File_set_view(
    MPI_File fh,
    MPI_Offset disp,        /* displacement in bytes from start */
    MPI_Datatype etype,     /* elementary type */
    MPI_Datatype filetype, /* file type; prob different for each proc */
    char *datarep,          /* 'native' or 'internal' */
    MPI_Info info)          /* MPI_INFO_NULL for today */
```

MPI-IO File View

Filetypes (made up of etypes; repeat as necessary)

```
int MPI_File_set_view(
    MPI_File fh,
    MPI_Offset disp,        /* displacement in bytes from start */
    MPI_Datatype etype,     /* elementary type */
    MPI_Datatype filetype, /* file type; prob different for each proc */
    char *datarep,          /* 'native' or 'internal' */
    MPI_Info info)          /* MPI_INFO_NULL for today */
```

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);
```

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)
```

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`.

File data representation

There are three possible representations:

- 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:

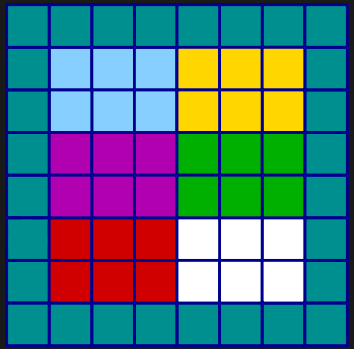
  Specific data representation, basically 32-bit big-endian IEEE format.

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.

More non-contiguous data: subarrays

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



Common special cases of non-contiguous access $\longrightarrow$ specialized functions: `MPI_Type_create_subarray` and `MPI_Type_create_darray`.

More non-contiguous data: subarrays

C code:

```
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 coordinatess for a process.

Conclusion

Recap

- MPI: Basics
- Example: 2D Diffusion
- Derived Data Types
- Application Topology
- MPI-IO

Good References

- W. Gropp, E. Lusk, and A. Skjellun, Using MPI: Portable Parallel Programming with the Message-Passing Interface. Third Edition. (MIT Press, 2014).
- W. Gropp, T. Hoefler, R. Thakur, E. Lusk, Using Advanced MPI: Modern Features of the Message-Passing Interface. (MIT Press, 2014).
- A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, Second Edition. (Addison-Wesley, 2003) (A bit old but still reasonable)
- The man pages for various MPI commands.
- http://www.mpi-forum.org/docs/ for MPI standard specification.