Parallel I/O Interfaces

Parallel I/O Concepts

Serial I/O

Parallel I/O Examples

- IBM PIOFS
- Intel PFS
- PIOUS
- PASSION
- MPI-IO

MPI-IO BTIO Walkthrough

I/O Interface Concepts

OS vs. library based

Collective vs. independent

Canonical vs. partitioned file view

Synchronous vs. asynchronous

File pointer vs. explicit offset

OS vs. Library Based

OS Based

- Data accessed with UNIX "read" and "write" calls
- Special functions accessed through system calls
- E.g.: IBM PIOFS, Intel PFS

Library Based

- Special read/write calls
- Special functions are part of I/O library
- E.g.: MPI-IO, Vesta, PIOUS, PASSION

Comparison

OS Based

- Familiar interface (+)
- Compatible with existing applications (+)
- Not well suited for complex data distributions (-)
- Extended functionality complicates interface (-)
- Standard FORTRAN interface not well suited for parallel I/O (-)

Comparison

Library Based

- Interface is not familiar to new users (-)
- Applications must be re-written (-)
- Can support complicated data distributions (+)
- Extended functionality can be integrated with the interface (+)
- Same interface can be implemented for C and FORTRAN (+)

Collective vs. Independent

Independent

- Processors read or write data at any time
- No coordination among processors
- E.g.: PFS M_UNIX, MPI-IO Independent, PIOFS

Collective

- All processors (within a group) write data at once
- Processors may coordinate transfer
- E.g.: PFS M_SYNC mode, MPI-IO Collective

Comparison

Independent

- **Less synchronous (+)**
- **More flexible (+)**
- **Can not utilize collective algorithms (-)**

Collective

- **More synchronization (-)**
- **Restrictive, I/O must be in "phases" (-)**
- **Can use collective algorithms (+++)**

Canonical vs. Partitioned File View

Canonical view

- **File appears as a single byte stream**
- **All processors may access the entire file**
- **E.g.: PFS M_UNIX, PIOFS, MPI-IO**

Partitioned view

- **Each processor sees a different portion of the file**
- **Data accesses can not overlap**
- **E.g.: Vesta, PFS M_RECORD, MPI-IO**

Comparison

Canonical view

- Familiar, i.e., like a normal UNIX system (+)
- Handles unstructured file access (i.e., log files) (+)
- Requires many “seek” operations for partitioned data access (-)
- Requires additional (possibly unnecessary) synchronization to provide consistency (-)

Comparison

Partitioned view

- Unfamiliar (-)
- Can not handle unstructured access (-)
- Does not require “seek” operations for partitioned data access (+)
- No added synchronization needed to provide consistency (+)

Synchronous vs. Asynchronous

Synchronous (blocking)

- **Processes must wait for I/O operation to (partially) complete**

Asynchronous (non-blocking)

- **Operations return immediately**
- **Must later determine if operation has completed**

Comparison

Synchronous

- **Can not overlap I/O with computation (-)**
- **Do not have to check whether data is available for re-use (+)**
- **For collective I/O, processors may have to wait for other processors (-), but may perform better (+)**

Comparison

Asynchronous

- Can overlap computation with I/O (+)
- Must check whether operation has completed before re-using data (-)
- May be less effective for collective operations (-)
- To be useful, requires memory for buffering (-)

File Pointer vs. Explicit Offset

File pointer

- File read/write position is relative to a file pointer
- User must call "seek" if desired position is not current one
- How is file pointer updated (shared, independent, what about asynchronous?)

Explicit offset

- File position is included with each read or write operation

Comparison

File pointer

- **Familiar (+)**
- **Can support "log" files (+)**
- **Not well suited to some distributions (lots of seeks) (-)**
- **Semantic differences can be confusing (-)**
- **Shared offset can add synchronization overhead (-)**
- **Asynchronous access, when do I update?? (-)**

Comparison

Explicit offset

- **Unfamiliar (-)**
- **Does not support "log" files (-)**
- **Fits well with some distributions (i.e., when every access needs a seek) (+)**
- **No problem with asynchronous access (+)**
- **No additional overhead (+)**

Interface Examples

Example interfaces

- **IBM PIOFS**
- **Intel PFS**
- **PIOUS**
- **PASSION**
- **MPI-IO**
- **Others**

Example

Data is a 100x100x100 matrix

All processes read/write portions of a single file

File written out in serial order

Serial I/O

C

```
double A[100][100][100];
read(fd, A, sizeof(A));
write(fd, A, sizeof(A));
```

FORTRAN

```
double precision A(100,100,100)
read (iunit) A
write (iunit) A
```

Parallel File Distributions

1D

```
double A[100/nodes][100][100];
double precision(100,100,100/nodes)
```

2D

```
nodes=x*x
double A[100/x][100/x][100];
double precision(100,100/nodes,100/nodes)
```

3D

```
nodes=y*y*y
double A[100/y][100/y][100/y];
double precision(100/nodes,100/nodes,100/nodes)
```

IBM PIOFS

Looks like a shared UNIX file system

- **OS based**
- **Independent file pointers**

Use normal UNIX read/write operations

- **No collective I/O support**
- **Canonical file view, can be partitioned using `piofs_fcntl` command (based on Vesta)**
- **UNIX sharing semantics not supported by default (reckless mode)**

Optimization possible using `piofs_fcntl` functions

PIOFS Usage (1D)

C

```
double A[100/nodes][100][100];
lseek(fd, 100*100*(100/nodes)*nodenum, SEEK_SET);
write(fd, A, 100*100*(100/nodes)*sizeof(double));
```

FORTRAN

- **1 Record/processor? 1 Record per plane?**
- **Use C style I/O (not possible from FORTRAN on AIX, can call C)?**
- *Lets ignore this problem for now*

PIOFS Usage (2D)

```
double A[100/x][100/x][100];
jcoord = (nodenum/x);
icoord = (nodenum%x);
for (i=0; i<(100/x); i++){
    lseek(fd, (100*sizeof(double)*(icoord*(100/x) +
               100*(jcoord*(100/x) + i),SEEK_SET);
    write(fd, A[i][0], 100*(100/x)*sizeof(double));
    }
```

PIOFS Usage (3D)

```
double A[100/y][100/y][100/y];
kcoord = (nodenum/y*y);
jcoord = ((nodenum/y)%y);
icoord = (nodenum%y);
for (k=0; k<(100/y); k++)
    for (j=0; j<(100/y); j++){
        lseek(fd, sizeof(double)*(icoord*(100/y) +
                   100*(jcoord*(100/y) + j) +
                   100*(kcoord*(100/y) + k), SEEK_SET);
        write(fd, A[k][j], (100/y)*sizeof(double));
        }
```

Summary

Works well for 1D distributions

Number of writes and seeks increases as number of dimensions increases

File pointers are not really used

Vesta interface can be used to support partitioned 1D and 2D views, 3D must still be performed by hand

Intel PFS

Similar to PIOFS

- **OS based**
- **Supports asynchronous I/O**
- **C style I/O supported from FORTRAN**

Intel PFS

Adds "file modes" to increase performance

- **Some file modes are collective**
- **Different file pointer semantics possible**
- **Different file views are possible**
- **Set using special open or setiomode**

```
void setiomode(int fildes, int iomode );
SUBROUTINE SETIOMODE(unit, iomode)
int gopen(const char *path,int oflag,int iomode,mode_t mode );
SUBROUTINE GOPEN(unit, path, iomode)
```

File Modes

M_UNIX

- **Unique file pointer**
- **Independent access**
- **Variable length, unordered records**
- **I/O atomicity guaranteed**

Closest to normal UNIX I/O and PIOFS

File Modes

M_LOG

- Shared file pointer
- Independent access
- Variable length, unordered records
- I/O atomicity guaranteed

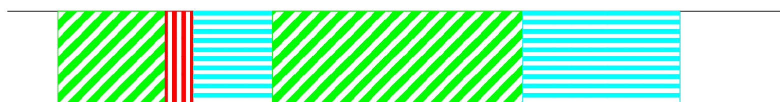
Useful for creating "log" files, accesses serialized

File Modes

M_SYNC

- Shared file pointer
- Collective access
- Variable length records, stored in process order
- I/O atomicity guaranteed

Collective access mode for some regular distributions

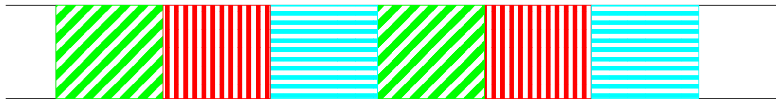


File Modes

M_RECORD

- Unique file pointer
- Independent access
- Fixed length records, process order
- I/O atomicity guaranteed
- Fully parallel

Fast non-collective mode for some regular data distributions



File Modes

M_GLOBAL

- Shared file pointer
- Collective access
- Variable length, unordered records
- I/O atomicity guaranteed
- All processes access same data

Collective broadcast mode

File Modes

M_ASYNC

- **Unique file pointer**
- **Independent access**
- **Variable length, unordered records**
- **I/O atomicity NOT guaranteed**
- **Fully parallel**

Fast independent mode, assumes programmer will not violate atomicity constraints

File Modes Summary

PIOFS code will work for M_UNIX, and possibly M_ASYNC modes

Collective access is possible using M_SYNC mode

M_SYNC and M_RECORD only support our 1D distribution

M_LOG and M_GLOBAL not useful for example

PFS 1D Usage

M_RECORD mode

```
double A[100/nodes][100][100];
lseek(fd, 100*100*(100/nodes)*nodenum, SEEK_SET);
write(fd, A, 100*100*(100/nodes)*sizeof(double));
```

M_SYNC mode

```
double A[100/nodes][100][100];
lseek(fd, 100*100, SEEK_SET);
write(fd, A, 100*100*(100/nodes)*sizeof(double));
```

Additional Functions

FORTRAN

```
SUBROUTINE CREAD(unit, buffer, nbytes)
SUBROUTINE CWRITE(unit, buffer, nbytes)
```

Asynchronous Read/Write

```
long iwrite(int fildes, char *buffer, unsigned int nbytes );
INTEGER FUNCTION IWRITE(unit, buffer, nbytes)
long iread(int fildes, char *buffer, unsigned int nbytes );
INTEGER FUNCTION IREAD(unit, buffer, nbytes)
long iodone(long id );
INTEGER FUNCTION IWRITE(unit, buffer, nbytes)
```

PFS Summary

OS Based

Asynchronous support

Limited collective support

FORTRAN interface for C read and write

File modes control view and access semantics

PIOUS

Library based, FORTRAN and C interfaces

- Built on top of PVM
- Uses servers that are part of the PVM job

UNIX-like interface

- Functions closely resemble UNIX system calls
- Shared, independent, or explicit file pointer
- Canonical or "segmented" file views
- No collective I/O support

PIOUS Calls

Open

```
pious_popen(group, path, view, map, faultmode, oflag, mode, seg);

pious_sopen(dsv, dsvcnt, group, path, view, map, faultmode, oflag,
           mode);

pious_open(path, oflag, mode);
```

Close

```
pious_close(fd);
```

pious_open Parameters

Group

- Which processes will use file
- Character name

View

- PIOUS_INDEPENDENT - independent file pointers
- PIOUS_GLOBAL - shared file pointers
- PIOUS_SEGMENTED - segmented view

pious_open Parameters

Map

- **Specifies which segment a process will see**

Faultmode - fault tolerance

- **PIOFS_VOLATILE, PIOFS_STABLE**

Oflag, mode - access mode, UNIX permission mode

Seg - the number of segments a file is striped across

Dsv - Specifies where to stripe

pious_open

Segments

- **Like disks, files are striped round robin**
- **Similar to PFS M_RECORD, but node order does not restrict view**

pious_open

- **equivalent to:**

```
pious_popen(group, path, PIOUS_INDEPENDANT, 1, PIOUS_VOLATILE,
          oflag, mode, seg);
```

Other PIOUS Calls

Read

```
pious_read(fd, buf, nbytes);
pious_oread(fd, buf, nbytes, offset); /* updates file pointer */
pious_pread(fd, buf, nbytes, offset); /* does not update pointer */
```

Write

```
pious_write(fd, buf, nbytes);
pious_owrite(fd, buf, nbytes, offset); /* updates file pointer */
pious_pwrite(fd, buf, nbytes, offset); /* does not update pointer */
```

PIOUS Usage (1D)

pious_write, independent mode similar to PIOFS

Explicit pointers

```
double A[100/nodes][100][100];
pious_swrite(fd, A, 100*100*(100/nodes)*sizeof(double),
           100*100*(100/nodes)*nodenum);
```

Segmented View, not useful for this program

PIOUS Usage (2D, 3D)

Similar to PIOFS

Explicit pointers eliminate need for lseeks (or can use pious_lseek if you want)

Segmented views may help performance, but are only useful if block size is the same as size of every write

PIOUS Summary

Provides library interface built on a popular message passing library

Explicit offsets or (shared/independent) file pointers

Segmented file views possible

PASSION

Library based, C interface only (so far)

- **Built on top of Intel NX**

Array oriented interface

- **Supports 2D arrays only (for now)**
- **Local or global placement model**
- **Extensive support for "out-of-core" arrays**
- **Collective and independent I/O support**
- **Uses 2-phase I/O algorithm for performance**
- **Array section pre-fetching (like asynchronous)**

PASSION Arrays

Local Placement Model (LPM)

- **One sub-array per processor**
- **Sub-arrays stored in separate files**
- **Local arrays can be in-core or out-of-core**

Global Placement Model (GPM)

- **Each processor can access any portion of the array**
- **Entire array is stored in a single file**

PASSION Usage

OCAD - out of core array descriptor

- Dimensions (2D only for now)
- Size of the array
- Processors in each dimension
- Array distribution

  NO_DISTRIBUTION

  BLOCK_DISTRIBUTION

  CYCLIC_DISTRIBUTION
- Overlap

PASSION Usage

Creating an OCAD

```
int Size[dimensions] = {rows, cols};
int Procs[dimensions] = {procs_dim_1, procs_dim_2};
int Distribution[dimensions][2] = {{BLOCK_DISTRIBUTION,0},
                          {CYCLIC_DISTRIBUTION, block_size}};
int OCLA_size[dimensions] = {OCLA_DIM_0, OCLA_DIM_1];
int ICLA_size[dimensions] = {ICLA_DIM_0, ICLA_DIM_1];
int overlap_info[dimensions][2] = {{up, down}, {left, right}};
OCAD *OCADp;
OCADp = PASSION_mallocOCAD(dimensions, ROW_MAJOR);
PASSION_fill_OCAD(OCADp, Size, Distribution, Procs, OCLA_size,
    ICLA_size, overlap, sizeof(double));
```

PASSION Usage

Open

```
fp=PASSION_open(filename, header_size);
```

Close

```
PASSION_close(fp);
```

Headers

```
PASSION_write_header(fp, buf);
PASSION_read_header(fp, buf);
```

PASSION Usage

Local arrays

```
PASSION_write(fp, OCADp, Array);
PASSION_read(fp, OCADp, Array);
```

Array sections

```
int AccessArray[dimensions][3]; /* specifies what part of the larger
                            out-of-core array you want to access */
PASSION_read_section(fp, ACADp, Array, i, j, AccessArray);
PASSION_write_section(fp, OCADp, Array, i, j, AccessArray);
pfptr = PASSION_read_prefetch(fp, OCADp, Array, i, j, AccessArray);
PASSION_prefetch_wait(pfptr);
```

PASSION Usage

Global Arrays

```
AccessArray[dimensions][3]; /* specifies what part of the larger
                               out-of-core array you want to access */
PASSION_global_read(fp, OCADp, Array, i, j, AccessArray, nprocs);
PASSION_global_write(fp, OCADp, Array, i, j, AccessArray, nprocs);
```

PASSION Summary

Library based interface with many advance features

- Collective I/O (global access)
- Asynchronous (read prefetch)
- Extensive support for reading and writing arrays sections (OCADs, ArrayAccess, etc.)
- Only supports arrays, not general data structures

Limited utility for version 1.0 (Feb. 1995)

- 2D arrays only, C only
- Intel machines only

MPI-IO

A proposed standard high-level interface for parallel I/O.

Goals:

- **Portable (standard) interface which allows for significant system-level optimizations**
- **Targets scientific applications, common usage patterns, and real world requirements**
- **High level interface describing data distribution**
- **Favor performance over functionality**

Why “MPI”-IO?

Two goals: Portability and Performance

I/O can share some infrastructure with message passing

- I/O can use MPI communicators, datatypes, process numbers, etc.
- Non-blocking I/O is like non-blocking messages, use same interface (e.g., MPI_Write*)

MPI derived datatypes can express file access patterns

- More on this later!

MPI-IO Support

Data Partitioning (e.g., file views)

- Mapping “the file” to processes

Data Access

- Positioning (explicit offsets/file pointers)
- Synchronism (blocking/non-blocking)
- Coordination (independent/collective)

Physical File Distribution - ignore this for now

MPI-IO Data Partitioning

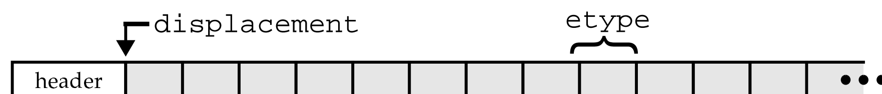
Described by three parameters:

- **`displacement` — offset to beginning of file**
- **`etype` — elementary data type**
- **`filetype` — data partitioning and access pattern**

Defined at `MPIO_Open()` **time:**

- **more flexible than file creation time (allows multiple simultaneous "views")**
- **less flexible than access time, but allows for better optimization possibilities (prefetching,...)**

MPI-IO Canonical File View



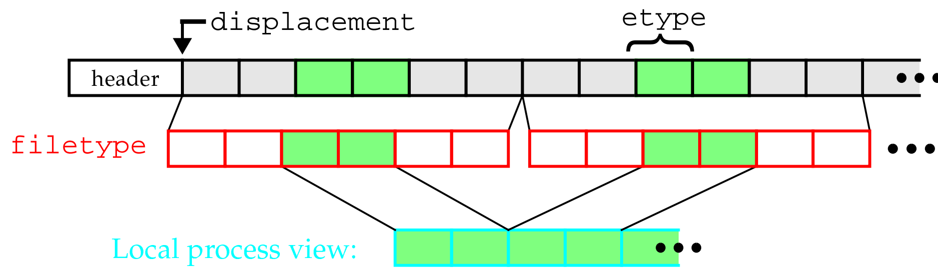
`displacement` — byte offset to beginning of file data

- **skip over headers easily, supports multiple etypes**

`etype` — elementary data type, basic (typed) unit of access

- **Flexible:** `MPI_BYTE`, `MPI_DOUBLE`, **(R, G, B), etc.**
- **Supports multiple views of a file, e.g. (R, -, -)**
- **Allows explicit offsets to be portable**
- **Guarantees match between filetype and buftype**

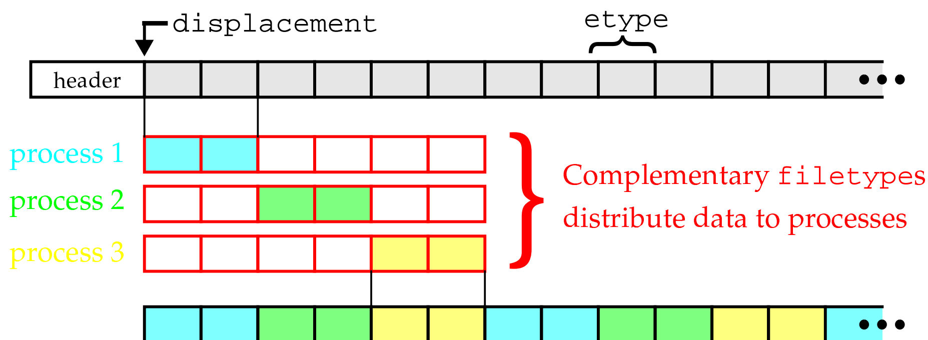
MPI-IO Local Process View



`filetype` — MPI derived datatype, "tiles" the file

- **Constructed from `etype`s, and `etype` sized holes**
- **Only file data "covered" with `etype`s is accessible**
- **Data under holes is not visible to this process**
- **Very general, e.g. permits reordering file data**

MPI-IO Global Data Partitioning

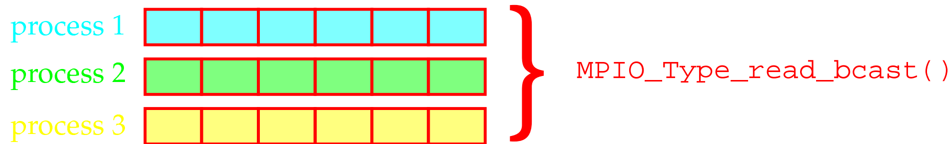


Permits very general partitioning schemes

- **non-uniform sizes**
- **overlapping access**
- **arbitrary order (not just rank ordering)**

Denotes file access pattern (useful for prefetching, etc.)

Common Patterns



Filetype constructors create filetypes for common patterns

- **Broadcast read — all nodes read identical data**
- **Write reduce — all nodes write identical data**
- **Scatter/gather — rank order access**
- **HPF distributions — BLOCK and CYCLIC**
- **General N-dimension embedded**

Rare patterns are handled via arbitrary MPI datatypes

MPI-IO Data Access

All operations are orthogonal (simple semantics)

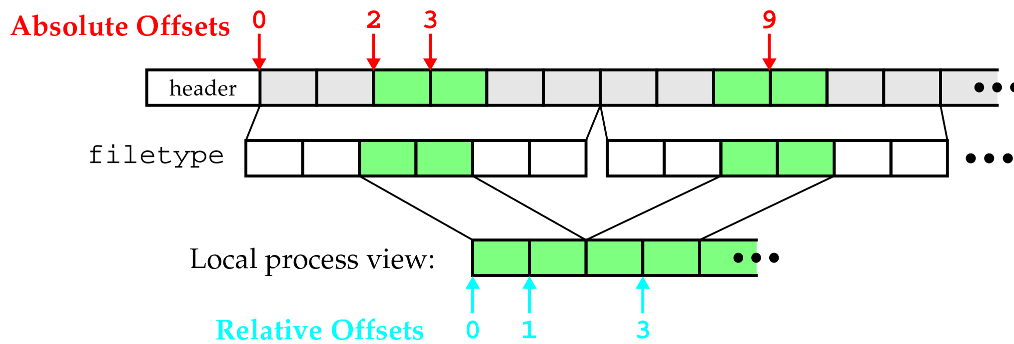
- **Transfer (read/write)**
- **Positioning (explicit offsets/file pointers)**
- **Synchronism (blocking/non-blocking)**
- **Coordination (independent/collective)**

Filetypes eliminate the need for special access functions (e.g. broadcast read, scatter/gather, etc.)

Operations specify a `buftype` **(an MPI derived datatype) as the process memory source or destination for file data**

- **Supports complex memory patterns**

Positioning — Explicit Offsets



Explicit Offsets

- supports "atomic" seek and transfer operations
- expressed in `etype` units (portable)
- absolute offsets — global view of entire file
- relative offsets — local view of accessible data

Positioning — File Pointers

Two types of file pointers supported simultaneously

- Individual file pointers — local to each process
- Shared file pointer — single global shared value

Equivalent semantics to explicit offset operations using the current file pointer position

File pointer is *always* updated prior to operation

```
offset = get_file_pointer_value();
update_file_pointer(size_of_request);
MPIO_xxx(fh, offset,...)
```

- Identical semantics regardless of synchronism
- Unexpected behavior with non-blocking and EOF

Coordination — Collective I/O

Collective I/O — all processes must participate

Allows implementation to optimize access

- **Better prefetching and caching strategies**
- **Collective buffering**
- **Access scheduling (e.g., disk directed I/O)**

Does not imply a barrier synchronization (same as in MPI)

Semantically identical to non-collective accesses

MPI-IO Functions

MPIO Open/close

```
MPIO_Open(comm, filename, amode, disp, etype, filetype, moffset,
          hints, fh)
MPIO_Close(fh)
MPIO_Fle_control(fh, size, command, arg)
MPIO_Seek(fh, offset, whence)
```

MPI-IO Data Access Functions

Positioning	Synchronism	Coordination	
		independent	*collective*
explicit offset	*blocking (synchronous)*	MPIO_Read	MPIO_Read_all
		MPIO_Write	MPIO_Write_all
	nonblocking (asynchronous)	MPIO_Iread	MPIO_Iread_all
		MPIO_Iwrite	MPIO_Iwrite_all
individual file pointers	*blocking (synchronous)*	MPIO_Read_next	MPIO_Read_next_all
		MPIO_Write_next	MPIO_Write_next_all
	nonblocking (asynchronous)	MPIO_Iread_next	MPIO_Iread_next_all
		MPIO_Iwrite_next	MPIO_Iwrite_next_all
shared file pointer	*blocking (synchronous)*	MPIO_Read_shared	MPIO_Read_shared_all
		MPIO_Write_shared	MPIO_Write_shared_all
	nonblocking (asynchronous)	MPIO_Iread_shared	MPIO_Iread_shared_all
		MPIO_Iwrite_shared	MPIO_Iwrite_shared_all

NAS

MPI-IO Usage - UNIX Style I/O

UNIX Command Equivalents

```
MPIO_Open(MPI_COMM_WORLD,"file", MPIO_RDWR|MPIO_CREATE, 0,
        MPI_BYTE, MPI_BYTE, NULL, &fh);
MPIO_Write_next(fh, buff, MPI_BYTE, count, status);
MPIO_Read_next(fh, buff, MPI_BYTE, count, status);
MPIO_Seek(fh, offset, whence);
```

1D distribution (using MPIO_Open shown above)

```
double A[100/nodes][100][100]; MPIO_Status status;
MPIO_Seek(fh, 100*100*(100/nodes)*nodenum, MPIO_SEEK_SET);
MPIO_Write_next(fh, A, MPI_BYTE,
        100*100*(100/nodes)*sizeof(double), &status);
```

NAS

MPI-IO Usage - UNIX Style I/O

2D distribution (using UNIX equivalent MPIO_Open)

```
double A[100/x][100/x][100];MPIO_Status status;
jcoord = (nodenum/x);
icoord = (nodenum%x);
for (i=0; i<(100/x); i++){
    MPIO_Seek(fd, (100*sizeof(double)*(icoord*(100/x) +
                100*(jcoord*(100/x) + i), MPIO_SEEK_SET);
    MPIO_Write_next(fh, A[i][0], MPI_BYTE,
            100*(100/x)*sizeof(double), &status);
    }
```

MPI-IO Usage - UNIX Style I/O

3D distribution (using UNIX equivalent MPIO_Open)

```
double A[100/y][100/y][100/y];MPIO_Status status;
icoord = (nodenum/y*y);
jcoord = ((nodenum/y)%y);
kcoord = (nodenum%y);
for (i=0; i<(100/y); i++)
    for (j=0; j<(100/y); j++){
        MPIO_Seek(fd, sizeof(double)*(kcoord*(100/y) +
                    100*(jcoord*(100/y) + j) +
                    100*(icoord*(100/y) + i), MPIO_SEEK_SET);
        MPIO_Write_next(fh, A[i][j], MPI_BYTE,
                (100/y)*sizeof(double), &status);
        }
```

MPI-IO Usage - Typed I/O

Basic types

```
double A[100/nodes][100][100]; MPIO_Status status;
MPIO_Open(MPI_COMM_WORLD,"file", MPIO_RDWR|MPIO_CREATE, 0,
        MPI_DOUBLE, MPI_DOUBLE, NULL, &fh);
MPIO_Seek(fh, 100*100*(100/nodes)*nodenum, MPIO_SEEK_SET);
MPIO_Write_next(fh, A, MPI_DOUBLE, 100*100*(100/nodes), &status);
```

Contiguous types

```
MPI_Datatype type;
MPI_type_contiguous(100*100*(100/nodes), MPI_DOUBLE, &type);
MPIO_Open(MPI_COMM_WORLD,"file", MPIO_RDWR|MPIO_CREATE, 0,
        type, type, NULL, &fh);
MPIO_Seek(fh, nodenum, MPIO_SEEK_SET);
MPIO_Write_next(fh, A, type, 1, &status);
```

MPI-IO Usage - Explicit Offsets

2D Example

```
double A[100/x][100/x][100];MPIO_Status status; MPI_Datatype type2;
jcoord = (nodenum/x);
icoord = (nodenum%x);
MPI_type_contiguous(100*(100/x), MPI_DOUBLE, &type2);
MPIO_Open(MPI_COMM_WORLD,"file", MPIO_RDWR|MPIO_CREATE, 0,
        type2, type2, NULL, &fh);
for (i=0; i<(100/x); i++){
    MPIO_Write(fh, (icoord + 100*jcoord + i*x), A[i][0],
        type2, 1, &status);
    }
```

Complex Datatypes

nD Constructor

```
MPIO_Type_nd_array(ndims, sizes, subsizes, starts, order,
        element, outtype)
```

ndims - number of dimensions

sizes - size of full data structure

subsizes - size of the submatrix being described

starts - starting point of each submatrix dimension

order - C or FORTRAN order

element - datatype of each matrix element

outtype - new datatype being created

MPI-IO Usage - 3D Distribution

Buftype - how is the file distributed in memory

```
sizes[0]=100/y; sizes[1]=100/y; sizes[2]=100/y;
subsizes[0]=100/y; subsizes[1]=100/y; subsizes[2]=100/y;
starts[0]=0; starts[1]=0; starts[2]=0;
order=0; /* C Order - column major */
MPIO_Type_nd_array(3, sizes, subsizes, starts, order,
        MPI_DOUBLE, &buftype);
```

MPI-IO Usage - 3D Distribution

Filetype - how is the processor's memory distributed on disk

```
icoord = (nodenum/y*y);
jcoord = ((nodenum/y)%y);
kcoord = (nodenum%y);
sizes[0]=100; sizes[1]=100; sizes[2]=100;
subsizes[0]=100/y; subsizes[1]=100/y; subsizes[2]=100/y;
starts[0]=icoord*100/y; starts[1]=jcoord*100/y;
starts[2]=kcoord*100/y;
order=0; /* C Order - column major */
MPIO_Type_nd_array(3, sizes, subsizes, starts, order,
       MPI_DOUBLE, &filetype);
```

MPI-IO Usage - 3D Distribution

Open

```
MPIO_Open(MPI_COMM_WORLD,"file", MPIO_RDWR|MPIO_CREATE, 0,
       MPI_DOUBLE, filetype, NULL, &fh)
```

Independent

```
MPIO_Write(fh, 0, A, buftype, 1, &status)
```

Collective

```
MPIO_Write_all(fh, 0, A, buftype, 1, &status)
```

MPI-IO Summary

Supports

- **General data distributions,**
- **Canonical or partitioned view**
- **Collective/independent I/O**
- **Explicit/implicit offsets**
- **UNIX style and high level operation support**
- **Blocking and non-blocking I/O**

MPIO Summary

Limitations

- **Requires MPI**
- **MPI datatypes can be hard to use, and are needed for best performance**

Other I/O Interfaces

PPFS

- **Library based**
- **Highly configurable, designed for experimentation**

Panda

- **Library based**
- **Good support for simple 3D distributions**

CMMD

- **OS based**
- **I/O modes, more restrictive than PFS**

Interface Summary

Interface	Base	Collective Support	Views	Asynchronous Support	Offset Support
PIOFS	OS	no	Linear, Partitioned	no	File Pointer
PFS	OS	yes	Linear, Partitioned	yes	File Pointer
PIOUS	Library	no	Linear, Partitioned	no	Explicit Offset, File Pointer
PASSION	Library	yes	Linear, Partitioned	yes	Explicit Offset
MPI-IO	Library	yes	Linear, Partitioned	yes	Explicit Offset, File Pointer

MPI-IO BTIO Code Walkthrough

BTIO Benchmark

- Based on the NAS BT benchmark
- Solution matrix written out every 5 iterations

MPI-IO Version

- Started with NPB version 1.5 code, Multi-cellular decomposition
- Simple version - UNIX style I/O
- Full version - MPI Datatype fully describes matrix

Restrictions of 12/95 NAS MPI-IO Subset

No file pointers

Relative offsets only

No file pointers

Must call MPIO_Init/MPIO_Finalize

Hints not fully implemented

Synchronous I/O only

Data Distribution

u

```
double precision u(5, -2:IMAX+1,  -2:JMAX+1,  -2:KMAX+1, ncells)
```

- **5 elements per data point**
- **extra room for boundary conditions**
- **multiple cells per node - sqrt(numnodes)**

Helper variables

```
cell_size(dim, cell)
```

- **size of u along dimension “dim” for cell “cell”**

```
cell_low(dim, cell)
```

- **lowest index of dimension “dim” for cell “cell”**

Simple MPI-IO Usage

Open

```
       iseek=0
      call MPIO_Open(comm_solve,
     $           'ufs:/scratch1/out.mpiio.simple',
     $           MPIO_WRONLY+MPIO_CREATE,
     $           iseek, MPI_DOUBLE_PRECISION, MPI_DOUBLE_PRECISION,
     $           MPIO_OFFSET_RELATIVE, 0, fp, ierr)
      idump=0
```

Simple MPI-IO Usage

Write step

```
      do cio=1,no_cells
         do kio=0, cell_size(3,cio)-1
            do jio=0, cell_size(2,cio)-1
               iseek=5*(cell_low(1,cio) +
$                PROBLEM_SIZE*((cell_low(2,cio)+jio) +
$                PROBLEM_SIZE*((cell_low(3,cio)+kio) +
$                PROBLEM_SIZE*idump)))

               count=5*cell_size(1,cio)

               call MPIO_Write(fp, iseek,
$               u(1,0,jio,kio,cio),
$               MPI_DOUBLE_PRECISION, count,
$               mstatus, ierr)
            enddo
         enddo
      enddo
      idump = idump + 1
```

Full MPI-IO

Elementary type

```
   call MPI_Type_contiguous(5, MPI_DOUBLE_PRECISION,
$                            element, ierr)
   call MPI_Type_commit(element, ierr)
   call MPI_Type_extent(element, eltext, ierr)
```

Full MPI-IO

Buftype - single cell

```
      do  c = 1, ncells
          sizes(1) = IMAX+4
          sizes(2) = JMAX+4
          sizes(3) = KMAX+4
          subsizes(1) = cell_size(1, c)
          subsizes(2) = cell_size(2, c)
          subsizes(3) = cell_size(3, c)
          starts(1) = 2
          starts(2) = 2
          starts(3) = 2

          call MPIO_Type_nd_array(3, sizes, subsizes, starts,
     $          1, element, cell_btype(c), ierr)
          cell_blength(c) = 1
          cell_disp(c) = eltext*(IMAX+4)*(JMAX+4)*(KMAX+4)*(c-1)
      enddo
```

Full MPI-IO

Combining the cells to a single buftype

```
      cell_blength(ncells+1) = 1
      cell_btype(ncells+1) = MPI_UB
      cell_disp(ncells+1) =
     $            eltext*(IMAX+4)*(JMAX+4)*(KMAX+4)*ncells

      call MPI_Type_struct(ncells+1, cell_blength, cell_disp,
     $            cell_btype, combined_btype, ierr)
      call MPI_Type_commit(combined_btype, ierr)
```

Full MPI-IO

File type - single cell

```
        do  c = 1, ncells
            sizes(1) = PROBLEM_SIZE
            sizes(2) = PROBLEM_SIZE
            sizes(3) = PROBLEM_SIZE
            subsizes(1) = cell_size(1, c)
            subsizes(2) = cell_size(2, c)
            subsizes(3) = cell_size(3, c)

            starts(1) = cell_low(1,c)
            starts(2) = cell_low(2,c)
            starts(3) = cell_low(3,c)
            call MPIO_Type_nd_array(3, sizes, subsizes, starts,
     $           1, element, cell_ftype(c), ierr)
            cell_blength(c) = 1
            cell_disp(c) = 0
        enddo
```

Full MPI-IO

Combining the cells to a single filetype

```
        call MPI_Type_struct(ncells, cell_blength, cell_disp,
     $             cell_ftype, combined_ftype, ierr)
        call MPI_Type_commit(combined_ftype, ierr)
```

Full MPI-IO

Open

```
      iseek=0
      call MPIO_Open(comm_solve,'ufs:/scratch1/out.full.mpiio',
     $          MPIO_WRONLY+MPIO_CREATE,
     $          iseek, element, combined_ftype,
     $          MPIO_OFFSET_RELATIVE, 0, fp, ierr)
```

Write step

```
             call MPIO_Write_all(fp, iseek, u,
     $                    combined_btype, 1, mstatus, ierr)
            call MPI_Type_size(combined_btype, iosize, ierr)
            iseek = iseek + iosize/eltext
```