JÜLICH
FORSCHUNGSZENTRUM

Parallel I/O and Portable Data Formats

MPI I/O

18. March 2013 | Florian Janetzko

Outline

- Introduction
- Derived Datatypes Revisited
- File Operations
- Advanced File Operations

JÜLICH
FORSCHUNGSZENTRUM

Parallel Programming with MPI

Introduction

18. March 2013 | Florian Janetzko

Amenities of MPI-I/O

Portability

- Standardized in 1997 and widespread support among vendors.
- Open Source implementation ROMIO is publicly available.

Ease of use

- It blends into syntax and semantic scheme of point-to-point and collective communication of MPI.
- Writing to a file is like sending data to another process.

Efficiency

- MPI implementors can transparently choose the best performing implementation for a specific platform.

Amenities of MPI-I/O

High level Interface

- It provides coordinated and structured access to a file for multiple processes
- Distributed I/O to the same file through collective operations

Handling of heterogeneous environments

- Automatic data conversion in heterogeneous systems
- File interoperability between systems via external representation

MPI I/O requirements

Understanding collective communication

- A file handle works like a communicator
- Coordination of file access can be of collective nature

Handling of immediate operations

- Non-blocking calls may overlap computation and I/O

Derived data types

- Non-contiguous file access is defined using MPI's derived data types

Handling of `MPI_Info` objects

- Some performance-critical information can be passed to the MPI layer

MPI terminology – Properties of procedures (I)

Blocking

A procedure is blocking if return from the procedure indicates that the user is allowed to reuse resources specified in the call to the procedure.

Non-blocking

If a procedure is non-blocking it will return as soon as possible from to the calling process. However, the user is not allowed to reuse resources specified in the call to the procedure before the communication has been completed by an appropriate call at the calling process.

Examples

- Blocking 
- Non-Blocking 

MPI terminology – Properties of procedures (II)

Collective

A procedure is collective if all processes in a group (e.g. in a communicator) need to invoke the procedure

Synchronous

A synchronized operation will complete successfully only if the (required) matching operation has started (send – receive).

Buffered (Asynchronous)

A buffered operation may complete successfully before a (required) matching operation has started (send – receive).

MPI_Info object (MPIS 3.0, 9+13.2.8)

- Can be used to pass hints for optimization to MPI (file system dependent)
- Consists of (key,value) pairs, where key and value are strings
- A key may have only one value
- `MPI_INFO_NULL` is always a valid MPI_Info object
- The maximum key size is `MPI_MAX_INFO_KEY`
- The maximum value size is `MPI_MAX_INFO_VALUE` (implementation dependent)

`MPI_MAX_INFO_VALUE` might be very large! It is not advisable to declare strings of that size!

Create and free MPI_Info objects

C/C++

```
int MPI_Info_create(MPI_Info info)
```

Fortran

```
MPI_INFO_CREATE(INFO, IERROR)
  INTERGER :: INFO, IERROR
```

- The created info objects contains no (key,value) pairs

C/C++

```
int MPI_Info_free(MPI_Info info)
```

Fortran

```
MPI_INFO_FREE(INFO, IERROR)
  INTERGER :: INFO, IERROR
```

- The info object is freed and set to `MPI_INFO_NULL`

Set and delete (key,value) pairs

C/C++

```
int MPI_Info_set(MPI_Info info, char *key, char *value)
```

Fortran

```
MPI_INFO_SET(INFO, KEY, VALUE, IERROR)
  CHARACTER(*) :: KEY, VALUE
  INTERGER     :: INFO, IERROR
```

C/C++

```
int MPI_Info_delete(MPI_Info info, char *key)
```

Fortran

```
MPI_INFO_DELETE(INFO, KEY, IERROR)
  CHARACTER(*) :: KEY
  INTERGER     :: INFO, IERROR
```

Retrieve active (key,value) pairs of an info object

C/C++

```
int MPI_Info_get_nkeys(MPI_Info info, int *nkeys)
```

Fortran

```
MPI_INFO_GET_NKEYS(INFO, NKEYS, IERROR)
  INTERGER      :: INFO, NKEYS, IERROR
```

C/C++

```
int MPI_Info_get_nthkey(MPI_Info info, int n, char *key);
```

Fortran

```
MPI_INFO_GET_NTHKEY(INFO, N, KEY, IERROR)
  CHARACTER(*):: KEY
  INTERGER      :: INFO, N, IERROR
```

Retrieve active (key,value) pairs of an info object

C/C++

```
int MPI_Info_get_valuelen(MPI_Info info, const char *key,
                          int *valuelen, int *flag)
```

Fortran

```
MPI_INFO_GET_VALUELEN(INFO, KEY, VALUELEN, FLAG, IERROR)
  CHARACTER (*) :: KEY
  INTERGER      :: INFO, VALUELEN, IERROR
  LOGICAL       :: FLAG
```

Retrieve active (key,value) pairs of an info object

C/C++

```
int MPI_Info_get(MPI_Info info, char *key,
                 int valuelen, char *value, int *flag)
```

Fortran

```
MPI_INFO_GET(INFO, KEY, VALUELEN, VALUE, FLAG, IERROR)
  CHARACTER(*):: KEY, VALUE
  INTERGER     :: INFO, VALUELEN, IERROR
  LOGICAL      :: FLAG
```

The function returns in `flag` either `true` if `key` is defined in `info`, otherwise it returns `false`

MPI Terminology – Basics

Task

An instance, sub-program or process of an MPI program

Communicator

All or a subset of MPI tasks

Rank

A unique number assigned to each task of an MPI program within a communicator

Handle

MPI reference to an internal MPI data structure, for example `MPI_COMM_WORLD` is a handle for the communicator which contains all MPI ranks

MPI Terminology – Datatypes

Basic datatypes

Datatypes which are defined within the MPI standard

- Basic datatypes for Fortran and C are **different**
- Examples:

Fortran

Fortran type	MPI basic type
INTEGER	MPI_INTEGER
REAL	MPI_REAL
CHARACTER	MPI_CHARACTER

C/C++

C type	MPI basic type
signed int	MPI_INT
float	MPI_FLOAT
char	MPI_CHAR

Derived datatypes

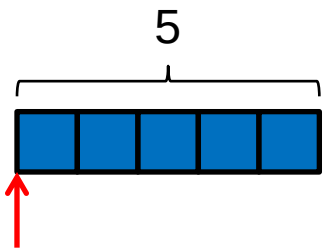
Datatypes which are constructed from basic (or derived) datatypes

MPI Terminology – Messages

Message

A packet of data which needs to be exchanged between processes

- Packet of data:
  - An array of elements of an MPI datatype (basic or derived datatype)
  - Described by
    - *Position in memory (address)*
    - *Number of elements*
    - *MPI datatyp*



- Information for sending and receiving messages
  - Source and destination process (ranks)
  - Source and destination location
  - Source and destination datatype
  - Source and destination data size

Access to JUROPA

Login

1. open a terminal
2. `ssh -A -X hpclabXX@juropa`

Compilation

1. Default compiler
   - ®Intel 11.1.072 with mkl 10.2.5.35
   - MPI compiler wrapper: `mpif77, mpif90, mpicc, mpicxx`
2. GCC compiler suite
   - `module purge`
   - `module load gcc parastation/mpi2-gcc-mt-5.0.26-1`
   - MPI compiler wrapper: `mpif77`, `mpif90`, `mpicc`, `mpicxx`

MPI starter: `mpiexec`

Running parallel jobs on JUROPA

Interactive jobs

1. open a terminal
2. `ssh -A -X hpclabXX@juropa`
3. `msub -I -X -l nodes=1:ppn=16`
4. wait for the prompt
5. start applications with `n` tasks with
   `mpiexec -np n <application>`

Batch jobs

1. open a terminal
2. `ssh -A -X hpclabXX@juropa`
3. to start applications with `n` tasks with submit the following job script with
   `msub <name_of_the_jobscript>`

```
#!/bin/bash -x
#MSUB -l nodes=1:ppn=16
#MSUB -l walltime=00:10:00
#MSUB -v tpt=1

mpiexec -np n <application>
```

JÜLICH
FORSCHUNGSZENTRUM

Parallel Programming with MPI

Derived datatypes

18. March 2013 | Florian Janetzko

Motivation

With MPI communication calls only multiple consecutive elements of the same type can be sent

Buffers may be non-contiguous in memory

- Sending only the real/imaginary part of a buffer of complex doubles
- Sending sub-blocks of matrices

Buffers may be of mixed type

- User defined data structures

```
struct buff_layout {
  int i[3];
  double d[5];
} buffer;
```

Solutions without MPI derived datatypes (I)

Non-contiguous data of a single type

- Consecutive MPI calls to send and receive each element in turn
  - ! *Additional latency costs due to multiple calls*
- Copy data to a single buffer before sending it
  - ! *Additional latency costs due to memory copy*

Solutions without MPI derived datatypes (II)

Contiguous data of mixed types

- Consecutive MPI calls to send and receive each element in turn
  - ! *Additional latency costs due to multiple calls*
- Use `MPI_BYTE` and `sizeof()` to avoid the type-matching rules
  - ! *Not portable to a heterogeneous system*

Derived datatypes

- General MPI datatypes describe a buffer layout in memory by specifying
  - *A sequence of basic datatypes*
  - *A sequence of integer (byte) displacements*
- Derived datatypes are derived from basic datatypes using constructors
- MPI datatypes are referenced by an opaque handle

MPI datatypes are opaque objects! Using the `sizeof()` operator on an MPI datatype handle will return the size of the handle, neither the size nor the extent of an MPI datatype.

Creating a derived datatype: Type map

Any derived datatype is defined by its type map

- A list of basic datatypes
- A list of displacements (positive, zero, or negative)
- Any type matching is done by comparing the sequence of basic datatypes in the type maps

General type map:

Datatype	Displacement
datatype 0	displacement of datatype 0
datatype 1	dispalcement of datatype 1
…	…

Example of a type map

```
struct buff_layout {
  int i[3];
  double d[5];
} buffer;
```

Datatype	Displacement
MPI_INT	0
MPI_INT	4
MPI_INT	8
MPI_DOUBLE	16
MPI_DOUBLE	24
MPI_DOUBLE	32
MPI_DOUBLE	40
MPI_DOUBLE	48

Padding:

- Alignment of data positions
- Holes

Contiguous data

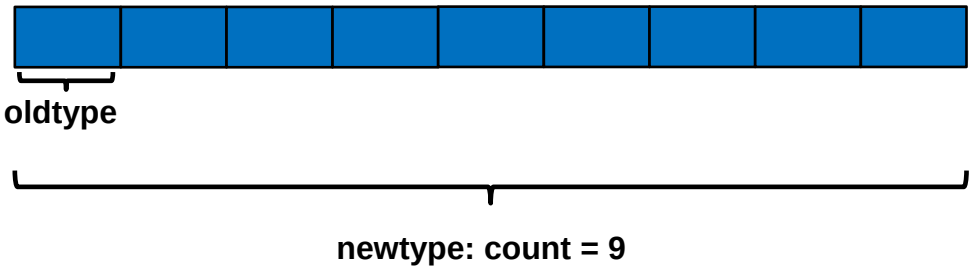
C/C++

```
int MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                        MPI_Datatype *newtype)
```

Fortran

```
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
  INTEGER :: COUNT, OLDTYPE, NEWTYPE, IERROR
```

- Simplest derived datatype
- Consists of a number of contiguous items of the same datatype



Vector data

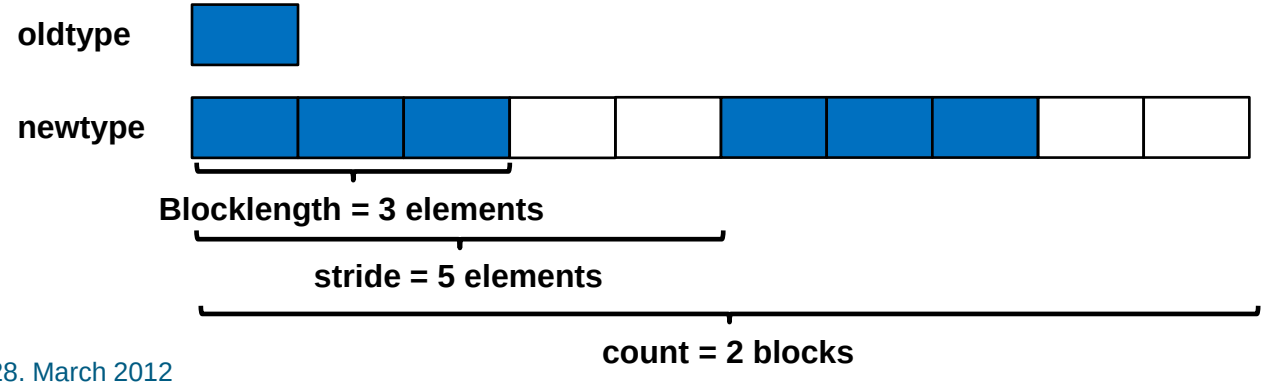
C/C++

```
int MPI_Type_vector(int count, int blocklength,
                    int stride, MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
```

Fortran

```
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,
                NEWTYPE, IERROR)
  INTEGER :: COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```

- Consists of a number of elements of the same datatype repeated with a certain stride



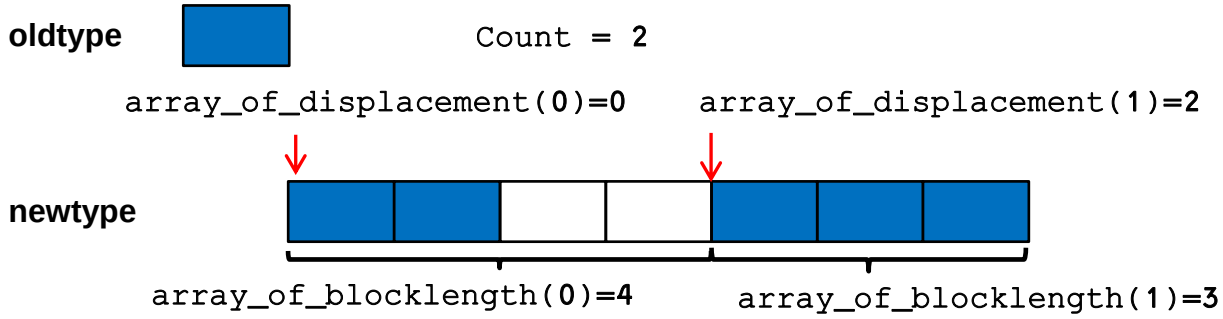
Indexed blocks

C/C++

```
int MPI_Type_indexed (int count, int array_of_blocklength,
                      int array_of_displacement,
                      MPI_Datatype oldtype,
                      MPI_Datatype newtype)
```

Fortran

```
MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTH,
                 ARRAY_OF_DISPLACEMENT, OLDTYPE, NEWTYPE,
                 IERROR)
  INTEGER :: COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR
```



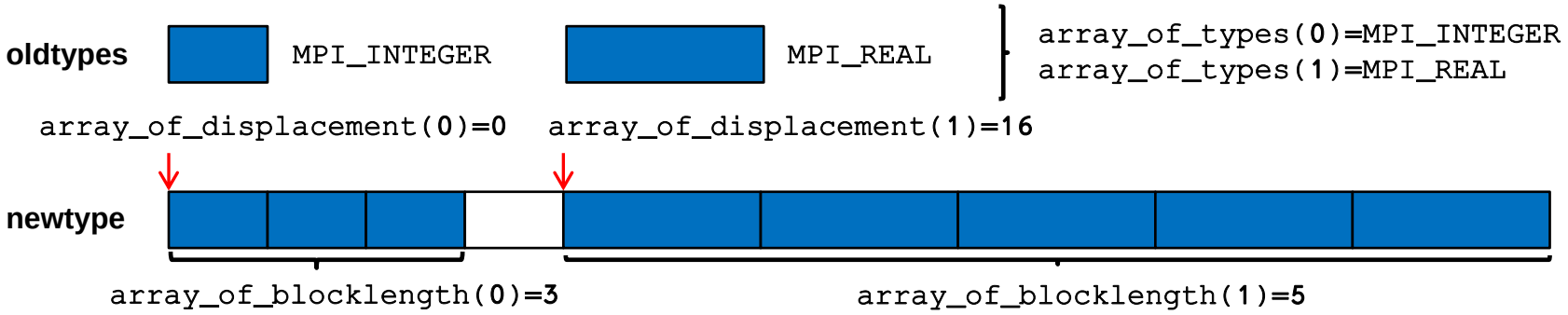
Struct data

C/C++

```
int MPI_Type_create_struct(int count, int *array_of_blocklengths,
      MPI_Aint *array_of_displacements, MPI_Datatype *array_of_types,
      MPI_Datatype *newtype)
```

Fortran

```
MPI_TYPE_CREATE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,ARRAY_OF_DISPLACEMENTS,
                       ARRAY_OF_TYPES, NEWTYPE,IERROR)
  INTEGER :: COUNT, ARRAY_OF_BLOCKLENGTHS(*), ARRAY_OF_TYPES(*), NEWTYPE,
             IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) :: ARRAY_OF_DISPLACEMENTS(*)
```



Sub-array data

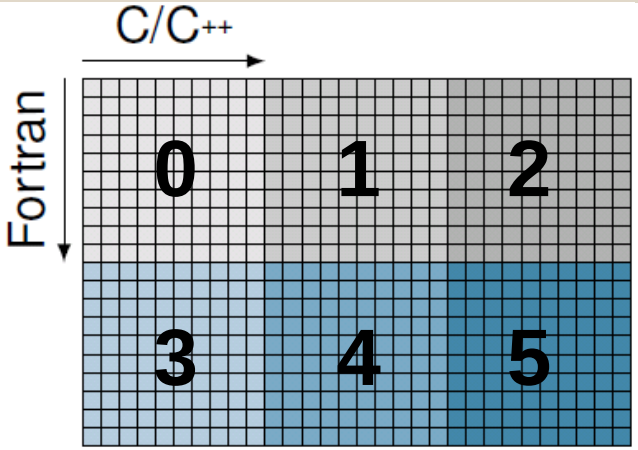
C/C++

```
int MPI_Type_create_subarray(int ndims,int array_of_sizes[],
                             int array_of_subsizes[],
                             int array_of_starts[], int order,
                             MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Fortran

```
MPI_TYPE_CREATE_SUBARRAY(NDIMS, ARRAY_OF_SIZES, ARRAY_OF_SUBSIZES,
                         ARRAY_OF_STARTS, ORDER, OLDTYPE, NEWTYPE, IERROR)
  INTEGER :: NDIMS, ARRAY_OF_SIZES(*), ARRAY_OF_SUBSIZES(*),
             ARRAY_OF_STARTS(*), ORDER, OLDTYPE, NEWTYPE, IERROR
```

Process	ARRAY_OF_STARTS
0	(0,0)
1	(0,10)
2	(0,20)
3	(10,0)
4	(10,10)
5	(10,20)



Distributed array data MPIS3.0, 4.1.4

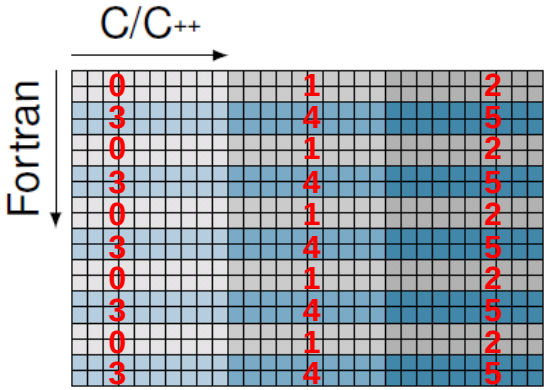
C/C++

```
int MPI_Type_create_darray(int size, int rank, int ndims,
                           int array_of_gsizes[], int array_of_distribs[],
                           int array_of_dargs[], int array_of_psizes[],
                           int order, MPI_Datatype oldtype,
                           MPI_Datatype *newtype)
```

Fortran

```
MPI_TYPE_CREATE_DARRAY(SIZE, RANK, NDIMS, ARRAY_OF_GSIZES,
                       ARRAY_OF_DISTRIBS, ARRAY_OF_DARGS, ARRAY_OF_PSIZES,
                       ORDER, OLDTYPE, NEWTYPE)
  INTEGER :: SIZE, RANK, NDIMS, ARRAY_OF_GSIZES(*), ARRAY_OF_DISTRIBS(*),
             ARRAY_OF_DARGS(*), ARRAY_OF_PSIZES(*), ORDER, OLDTYPE, NEWTYPE
```

- N-dimensional distributed/strided sub-array of an N-dimensional array
- Fortran and C order allowed
- Fortran and C calls expect indices starting from 0



Committing and freeing derived datatypes

C/C++

```
int MPI_Type_commit(MPI_Datatype *datatype)
```

Fortran

```
MPI_TYPE_COMMIT(DATATYPE, IERROR)
  INTEGER :: DATATYPE, IERROR
```

- Before it can be used in a communication, each derived datatype has to be committed

C/C++

```
int MPI_Type_free(MPI_Datatype *datatype)
```

Fortran

```
MPI_TYPE_FREE(DATATYPE, IERROR)
  INTEGER :: DATATYPE, IERROR
```

- Mark a datatype for deallocation
- Datatype will be deallocated when all pending operations are finished

Finding the address of a memory location

C/C++

```
int MPI_Get_address(void *location, MPI_Aint *address)
```

- Finding addresses and relative displacements of memory blocks

```
MPI_Aint addr_block_1, addr_block_2;
MPI_Aint displacement = 0;

MPI_Get_address(&block_1, &addr_block_1);
MPI_Get_address(&block_2, &addr_block_2);

displacement = addr_block_2 - addr_block_1;
```

Do not rely on C's address operator &, as ANSI C does not guarantee pointer values to be absolute addresses. Furthermore, address space may be segmented. Always use `MPI_GET_ADDRESS`, which also guarantees portability.

Finding the address of a memory location

Fortran

```
MPI_GET_ADDRESS(LOCATION,ADDRESS,IERROR)
  <type>  :: LOCATION(*)
  INTEGER(KIND=MPI_ADDRESS_KIND) :: ADDRESS
  INTEGER :: IERROR
```

- Finding addresses and relative displacements of memory blocks

```
INTEGER(KIND=MPI_ADDRESS_KIND) :: addr_block_1, addr_block2
INTEGER(KIND=MPI_ADDRESS_KIND) :: displacement=0
INTEGER                        :: ierror

call MPI_Get_address(block_1, addr_block_1, ierror)
call MPI_Get_address(block_2, addr_block_2, ierror)

displacement = addr_block_2 - addr_block_1
```

Datatypes – size and extent

Size

The size of a datatype is the net number of bytes to be transferred (without “holes”).

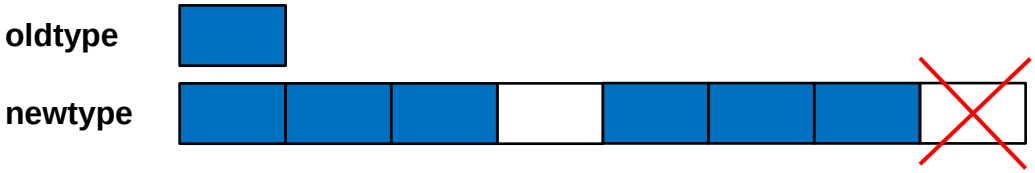
Extent

The extent of a datatype is the span from the lower to the upper bound (including inner “holes”). When creating new types, holes at the end of the new type are not counted to the extent.

Basic datatypes

- size = extent = number of bytes used by the compiler

Derived datatypes



- size = 6 x size of `oldtype`
- extent = 7 x extent of `oldtype`

Query size and extent of datatypes

C/C++

```
int MPI_Type_size(MPI_Datatype datatype,int *size)
```

Fortran

```
MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
  INTEGER :: DATATYPE, SIZE, IERROR
```

- Returns the total number of bytes of the entries in `DATATYPE`

C/C++

```
int MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint* lb,
                        MPI_Aint* extent)
```

Fortran

```
MPI_TYPE_GET_EXTENT(DATATYPE, LB, EXTENT, IERROR)
  INTEGER :: DATATYPE,IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) :: LB,EXTENT
```

- The extent is the number of bytes between the lower and the upper bound markers

Resizing datatypes

C/C++

```
int MPI_Type_create_resized(MPI_Datatype oldtype,
                            MPI_Aint lb, MPI_Aint extent,
                            MPI_Datatype* newtype)
```

Fortran

```
MPI_TYPE_CREATE_RESIZED(OLDTYPE, LB, EXTENT, NEWTYPE,
                        IERROR)
  INTEGER :: OLDTYPE, NEWTYPE, IERROR
  INTEGER(KIND=MPI_ADDRESS_KIND) :: LB, EXTENT
```

- Sets new lower and upper bound markers
- Allows for correct stride in creation of new derived datatypes

JÜLICH
FORSCHUNGSZENTRUM

Parallel Programming with MPI

MPI file operations

18. March 2013 | Florian Janetzko

MPI I/O – terminology

File

An MPI file is an ordered collection of typed data items.

Displacement

Displacement is an absolute byte position relative to the beginning of a file.

Offset

Offset is a position in the file relative to the current view. It is expressed as a count of elementary types.

File pointer

A file pointer is an explicit offset maintained by MPI.

Opening a file

C/C++

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

Fortran

```
MPI_FILE_OPEN(COMM, FILENAME ,AMODE, INFO, FH, IERROR)
  CHARACTER*(*) :: FILENAME
  INTEGER       ::COMM,AMODE,INFO,FH,IERROR
```

- Filename's namespace is implementation dependent
- Call is collective on `COMM`
- Process-local files can be opened with `MPI_COMM_SELF`
- Filename must reference the same file on all processes
- Additional information can be passed to MPI environment via the `MPI_INFO` handle.

Access modes

Access mode is a bit-vector, which is modified with

- | (Bitwise OR) in C
- `IOR` (IOR Operator) in FORTRAN 90
- + (Addition Operator) in FORTRAN 77

One and only one of the following modes is mandatory:

- `MPI_MODE_RDONLY` – read only
- `MPI_MODE_RDWR` – read and write access
- `MPI_MODE_WRONLY` – write only

The following modes are optional:

- `MPI_MODE_CREATE` – create file if it doesn't exist
- `MPI_MODE_EXCL` – error if creating file that already exists
- `MPI_MODE_DELETE_ON_CLOSE` – delete file on close
- `MPI_MODE_UNIQUE_OPEN` – file can not be opened elsewhere
- `MPI_MODE_SEQUENTIAL` – sequential file access (e.g. tapes)
- `MPI_MODE_APPEND` – all file pointers are set to end of file

Associate info objects with an open file

C/C++

```
int MPI_File_set_info(MPI_File fh, MPI_Info info)
```

Fortran

```
MPI_FILE_SET_INFO(FH, INFO, IERROR)
  INTEGER      :: FH, INFO, IERROR
```

- Info items that cannot be changed for an open file need to be set when opening the file
- MPI implementation may choose to ignore the hints in this call

Retrieve an info object associated with an open file

C/C++

```
int MPI_File_get_info(MPI_File fh, MPI_Info info_used)
```

Fortran

```
MPI_FILE_GET_INFO(FH, INFO_USED, IERROR)
  INTEGER      :: FH, INFO_USED, IERROR
```

- This function returns all info items associated with file **`fh`**
- The number of items might be more or less than the number specified when opening the file

Possible keys for MPI_Info objects (selection)

Collective buffering

- `collective_buffering` (bolean)
- `cb_block_size` (integer, bytes, data access in chunks of this size)
- `cb_buffer_site` (integer, bytes)
- `cb_nodes` (integer, number of nodes used for collective buffering)

Disk striping

- `striping_factor` (integer, number of devices to stripe over)
- `striping_unit` (integer, bytes, size of block on each device)

➢ See Lustre parameters `stripe_size`, `stripe_count`

Closing a file

C/C++

```
int MPI_File_close(MPI_File *fh)
```

Fortran

```
MPI_FILE_CLOSE(FH, IERROR)
  INTEGER :: FH, IERROR
```

- Collective operation
- If `MPI_MODE_DELETE_ON_CLOSE` was specified on opening, the file is deleted after closing
- The user must ensure that all outstanding requests of a process connected with `FH` have completed before that process calls `MPI_FILE_CLOSE`

Deleting a file

C/C++

```
int MPI_File_delete(char * filename, MPI_Info info)
```

Fortran

```
MPI_FILE_DELETE(FILENAME, INFO, IERROR)
  CHARACTER(*) :: FILENAME
  INTEGER      :: INFO, IERROR
```

- May be used to delete a file that is not currently opened
- Call is not collective, if called by multiple processes on the same file, all but one will return an error code ≠ `MPI_SUCCESS`

JÜLICH
FORSCHUNGSZENTRUM

Parallel Programming with MPI

File views

18. March 2013 | Florian Janetzko

MPI I/O terminology

Elementary type

The elementary type is the basic entity of a file. It must be the same on all processes with the same file handle.

File type

The file type describes the access pattern of the processes on the file. It defines what parts of the file are accessible by a specific process. The processes may have different file types to access different parts of a file.

File view

A view defines the file data visible to a process. Each process has an individual view of a file defined by a displacement, an elementary type and a file type. The pattern is the same that `MPI_TYPE_CONTIGUOUS` would produced if it were passed the file type.

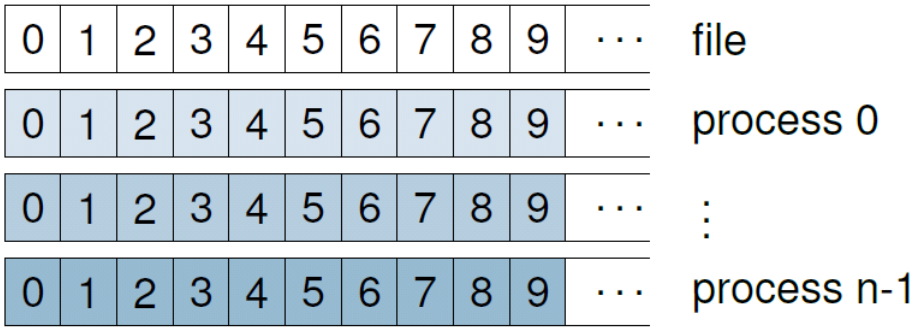
Basic file view properties

- Defines the elementary type as the atomic entity of a file
- Defines the logical view to a file through the file type
- Defines the byte position to the first data elements visible to a process with displacement
- Defines a data representation that handles encoding of data

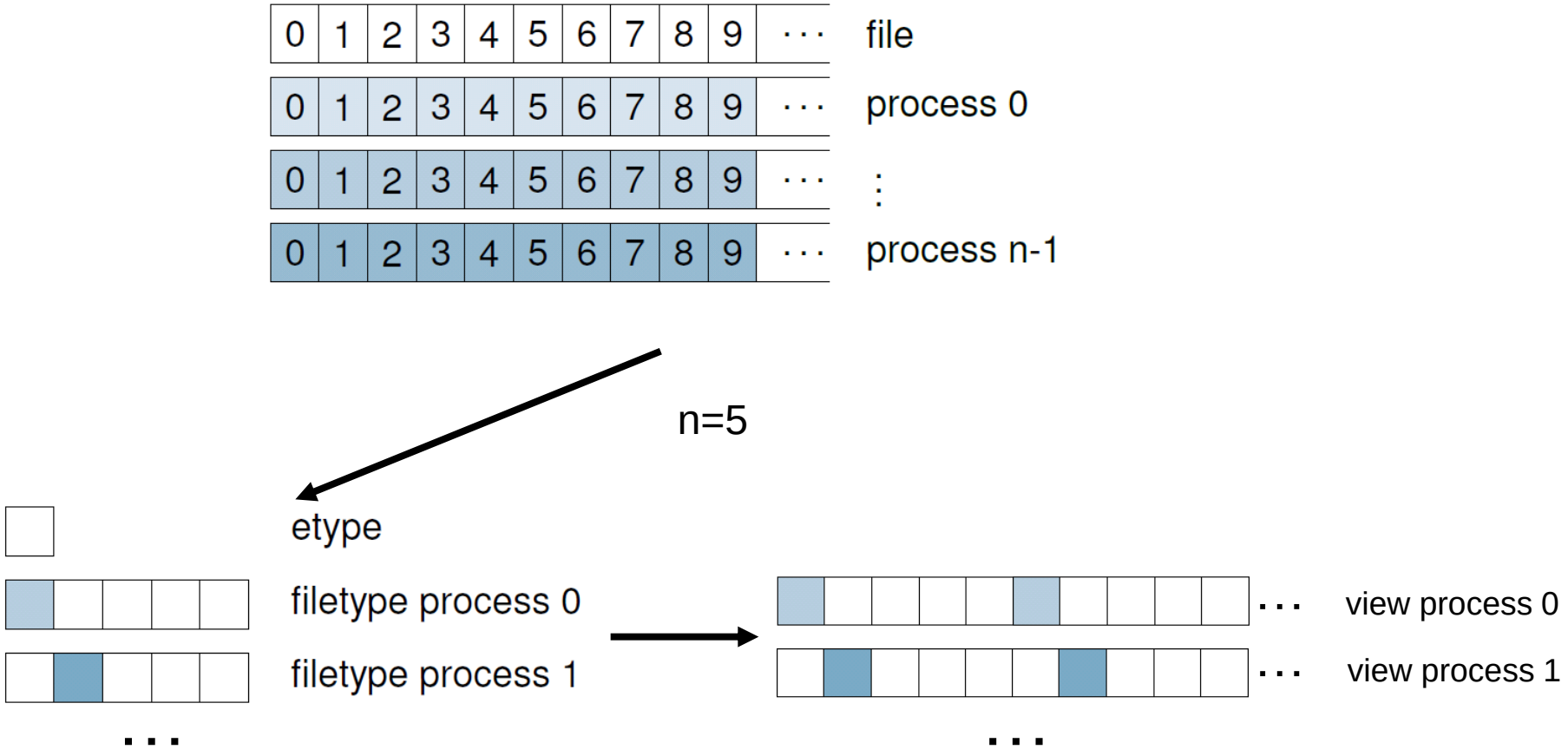
Default file view

A default view for each participating process is defined implicitly while opening the file

- No displacement
- The file has no specific structure
- All processes have access to the complete file



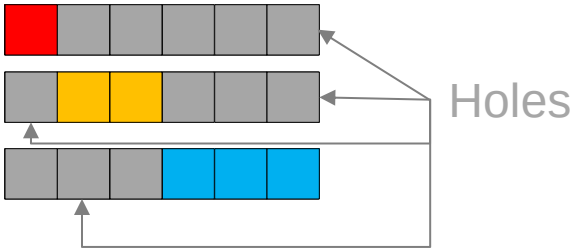
Example of a user-defined file view

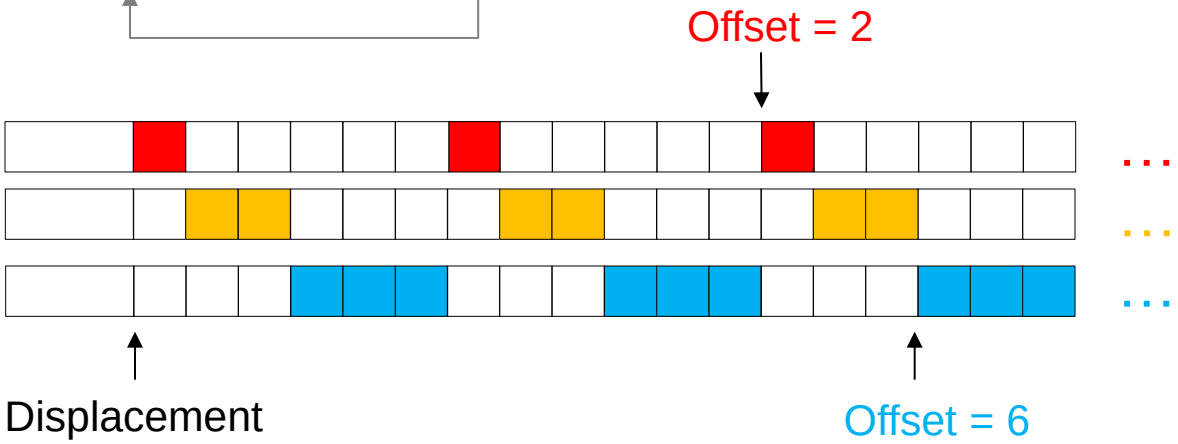


MPI I/O terminology – overview

elementary type (MPI predefined or derived datatype)

filetype process 0
filetype process 1
filetype process 2





global view

Set the file view

C/C++

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

Fortran

```
MPI_FILE_SET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP,
                  INFO, IERROR)
  INTEGER        :: FH, ETYPE, FILETYPE, INFO, IERROR
  CHARACTER*(*)  :: DATAREP
  INTEGER(KIND=MPI_OFFSET_KIND) :: DISP
```

- Changes the process's view of the data
- Local and shared file pointers are reset to zero
- Collective operation
- `ETYPE` and `FILETYPE` must be committed
- `DATAREP` is a string specifying the data format

Data representations (I)

`native`

- Data is stored in the file exactly as it is in memory
- On homogeneous systems no loss in precision or I/O
- performance due to type conversions
- On heterogeneous systems loss of transparent interoperability
- No guarantee that MPI files are accessible from C/Fortran

`internal`

- Data is stored in implementation-specific format
- Can be used in a homogeneous or heterogeneous environment
- Implementation will perform file conversions if necessary
- No guarantee that MPI files are accessible from C/Fortran

Data representations (II)

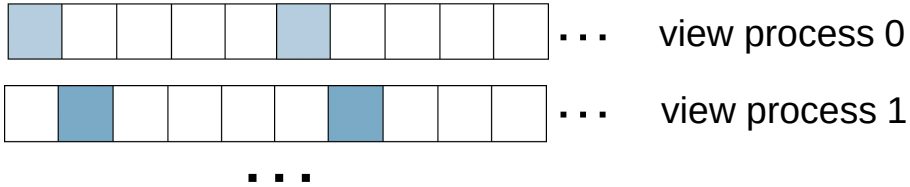
`external32`

- Standardized data representation (big-endian IEEE)
- Read/write operations convert all data from/to this representation
- Files can be exported/imported to/from different MPI environments
- Precision and I/O performance may be lost due to type conversions between `native` and `external32` representations
- Internal may be implemented as `external32`
- Can be read/written also by non-MPI programs

`User defined`

- Allow the user to insert a third party converter into the I/O stream to do the data representation conversion

Example

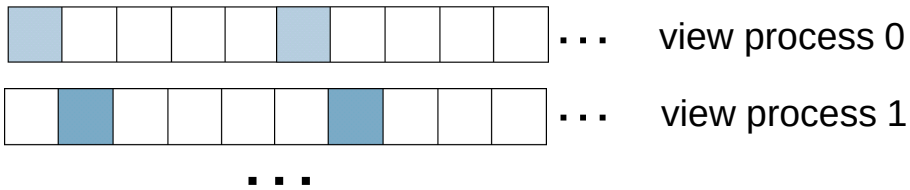


Fortran

```
INTEGER :: ndims=1, size, subsize=1, start
INTEGER :: etype=MPI_INTEGER, filetype, fh, ierror
INTEGER(KIND=MPI_OFFSET_KIND) :: disp=0

...
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, start, ierror)
call MPI_Type_create_subarray(ndims, size, subsize, start,&
&                             MPI_ORDER_FORTRAN, etype,&
&                             filetype, ierror)
call MPI_Type_commit(filetype, ierror)
call MPI_File_open(MPI_COMM_WORLD, "output.dat",&
&                  MPI_MODE_CREATE+MPI_MODE_RDWR,&
&                  MPI_INFO_NULL, fh, ierror)
call MPI_File_set_view(fh, disp, etype, filetype, "native",&
&                      MPI_INFO_NULL, ierror)
...
```

Example



C/C++

```
int          ndims=1, size, subsize=1, start;
MPI_Datatype etype=MPI_INT, filetype;
MPI_File     fh;
MPI_Offset   disp=0;

...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &start);
MPI_Type_create_subarray(&ndims, &size, &subsize, &start,
                         MPI_ORDER_C, etype, &filetype);
MPI_Type_commit(&filetype);
MPI_File_open(MPI_COMM_WORLD,"output.dat",
                  MPI_MODE_CREATE|MPI_MODE_RDWR,
                  MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, etype, filetype, "native",
                  MPI_INFO_NULL);
...
```

Querying the file view

C/C++

```
int MPI_File_get_view(MPI_File fh, MPI_Offset *disp,
                      MPI_Datatype *etype, MPI_Datatype *filetype,
                      char *datarep)
```

Fortran

```
MPI_FILE_GET_VIEW(FH, DISP, ETYPE, FILETYPE, DATAREP,
                  IERROR)
  INTEGER        :: FH, ETYPE, FILETYPE, IERROR
  CHARACTER*(*)  :: DATAREP
  INTEGER(KIND=MPI_OFFSET_KIND) :: DISP
```

Returns the process's view of the file

File pointers

Individual file pointers

- Each process has its own file pointer that is only altered on accesses of that specific process

Shared file pointers

- This file pointer is shared among all processes in the communicator used to open the file
- It is modified by any shared file pointer access of any process
- Shared file pointers can only be used if the file type gives each process access to the whole file!

Explicit offset

- No file pointer is used or modified
- An explicit offset is given to determine access position
- This can not be used with `MPI_MODE_SEQUENTIAL`!

Writing to a file using individual file pointers

C/C++

```
int MPI_File_write(MPI_File fh,void *buf, int count,
                   MPI_Datatype datatype,
                   MPI_Status *status)
```

Fortran

```
MPI_FILE_WRITE(FH, BUF, COUNT, DATATYPE, STATUS,
               IERROR)
  <type>  :: BUF(*)
  INTEGER :: FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

- Writes `COUNT` elements of `DATATYPE` from memory starting at `BUF` to the file
- `DATATYPE` is used as the access pattern to `BUF` and the sequence of basic datatypes of `DATATYPE` (type signature) must match contiguous copies of the etype of the current view
- Starts writing at the current position of the file pointer
- `STATUS` will indicate how many bytes have been written

Reading from a file using individual file pointers

C/C++

```
int MPI_File_read(MPI_File fh, void *buf, int count,
                  MPI_Datatype datatype,
                  MPI_Status *status)
```

Fortran

```
MPI_FILE_READ(FH, BUF, COUNT, DATATYPE, STATUS, IERROR)
  <type>  :: BUF(*)
  INTEGER :: FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE), IERROR
```

- Reads `COUNT` elements of `DATATYPE` from the file to the memory starting at `BUF`
- Starts reading at the current position of the file pointer
- `STATUS` will indicate how many bytes have been read

Seeking a file position using individual file pointers

C/C++

```
int MPI_File_seek(MPI_File fh, MPI_Offset offset,
                  int whence)
```

Fortran

```
MPI_FILE_SEEK(FH, OFFSET, WHENCE, IERROR)
  INTEGER :: FH, WHENCE, IERROR
  INTEGER(KIND=MPI_OFFSET_KIND) :: OFFSET
```

- Updates the individual file pointer according to `WHENCE`, which can have the following values:
  - `MPI_SEEK_SET`: pointer is set to `OFFSET`
  - `MPI_SEEK_CUR`: pointer is set to the current position plus `OFFSET`
  - `MPI_SEEK_END`: pointer is set to the end of file plus `OFFSET`
- `OFFSET` can be negative, which allows seeking backwards
- It is erroneous to seek to a negative position in the view

Querying the position of an individual file pointer

C/C++

```
int MPI_File_get_position(MPI_File fh, MPI_Offset* offset)
```

Fortran

```
MPI_FILE_GET_POSITION(FH, OFFSET, IERROR)
  INTEGER :: FH, IERROR
  INTEGER(KIND=MPI_OFFSET_KIND) :: OFFSET
```

- Returns the current position of the individual file pointer in `OFFSET`
- The value can be used to return to this position or calculate a displacement

Using shared file pointers

- Same semantics just add `_shared` to the calls
  - `MPI_File_write_shared`
  - `MPI_File_read_shared`
  - `MPI_File_seek_shared`
  - `MPI_File_get_position_shared`
- Blocking, individual read using the shared file pointer
- Only the shared file pointer will be advanced accordingly
- `DATATYPE` is used as the access pattern to `BUF`
- Middleware will serialize accesses to the shared file pointer to ensure collision-free file access

Using an explicit offset

- Semantics similar using individual file pointers, just add `_at` to the calls and specify `OFFSET` (MPIS 3.0, 13.4.2)
  - `MPI_File_write_at`
  - `MPI_File_read_at`
  - `MPI_File_seek_at`
  - `MPI_File_get_position_at`
- File access starts at `OFFSET` units of etype from beginning of view
- `DATATYPE` is used as the access pattern to `BUF` and the sequence of basic datatypes of `DATATYPE` (type signature) must match contiguous copies of the etype of the current view

JÜLICH
FORSCHUNGSZENTRUM

Parallel Programming with MPI

Advanced file operations

18. March 2013 | Florian Janetzko

Collective file access – benefits

Explicit offsets / individual file pointers:

- MPI implementation may internally communicate data to avoid serialization of file access
- MPI implementation may internally communicate data to avoid redundant file access
- Chance of best performance

Shared file pointer

- Data accesses do not have to be serialized by the MPI-implementation
- First, locations for all accesses can be computed, then accesses can proceed independently (possibly in parallel)
- Also here: Chance of good performance

Collective file access – function calls

Semantics identical to non-collective calls just add

- `_all` with when using individual file pointers or explicit offset
- `_ordered` when using shared file pointers

With shared file pointers data is written in the order of process ranks

- Deterministic outcome as opposed to individual writes with the shared file pointer

All processes sharing the file handle have to participate

MPIS 3.0, 13.4.4

Writing to a file collectively

C/C++

```
int MPI_File_write_all(MPI_File fh, void *buf, int count,
                       MPI_Datatype datatype,
                       MPI_Status *status)
```

Fortran

```
MPI_FILE_WRITE_ALL(FH,BUF,COUNT,DATATYPE,STATUS,IERROR)
  <type>                  :: BUF(*)
  INTEGER                 :: FH,COUNT,DATATYPE,IERROR
  STATUS(MPI_STATUS_SIZE) :: STATUS
```

- Call is collective on communicator associated with fh
- MPI can use communication between ranks to optimize I/O
- False share of file system blocks can be minimized as access pattern can be communicated an rearranged

Non-blocking I/O – characteristics

If supported by hardware, I/O can complete without intervention of the CPU (asynchronous I/O)

- overlap of computation and I/O

I/O calls have two parts (→ non-blocking communication)

- Initialization
- Completion

Implementations may perform all I/O in either part

Asynchronous I/O is not supported on the Blue Gene/Q architecture!

Non-blocking I/O – function calls

Individual function calls

- Initialized by call to `MPI_File_i[...]`
- Completed by call to `MPI_Wait` or `MPI_Test`

Collective function calls

- Also called split-collective
- Initialized by call to `[...]_begin`
- Completed by call to `[...]_end`

`STATUS` parameter is replaced by `REQUEST` parameter

File pointers are updated to the new position by the end of the initialization call

Non-blocking write with individual file pointer

C/C++

```
int MPI_File_iwrite(MPI_File fh, void *buf, int count,
                    MPI_Datatype datatype,
                    MPI_Request *request)
```

Fortran

```
MPI_FILE_WRITE_ALL(FH,BUF,COUNT,DATATYPE,STATUS,IERROR)
  <type>                  :: BUF(*)
  INTEGER                 :: FH,COUNT,DATATYPE,REQUEST,IERROR
```

- Same semantics to buffer access as non-blocking point-to-point communication
- Completed be a call to `MPI_Wait` or `MPI_Test`
- Other individual calls analogous

Split collective file access

C/C++

```
int MPI_File_read_at_all_begin(MPI_File fh,
                               MPI_Offset offset,
                               void *buf, int count,
                               MPI_Datatype datatype)
int MPI_File_read_at_all_end(MPI_File fh,
                             void *buf, MPI_Status *status)
```

- Collective operations may be split into two parts
- Rules an restrictions:
  - *Only one active split or regular collective operations per file handle*
  - *Split collective operations do not match the corresponding regular collective operation*
  - *Same buf argument in* `_begin` *and* `_end` *calls*

Split collective file access

Fortran

```
MPI_FILE_READ_AT_ALL_BEGIN(FH, OFFSET, BUF, COUNT,
                           DATATYPE, IERROR)
MPI_FILE_READ_AT_ALL_END(FH, BUF, STATUS, IERROR)
  <type>     :: BUF(*)
  INTEGER    :: FH, COUNT, DATATYPE, STATUS(MPI_STATUS_SIZE)
  INTEGER    :: IERRIR
  INEGER(KIND=MPI_OFFSET_KIND) :: OFFSET
```

- Collective operations may be split into two parts
- Rules an restrictions:
  - *Only one active split or regular collective operations per file handle*
  - *Split collective operations do not match the corresponding regular collective operation*
  - *Same buf argument in* `_begin` *and* `_end` *calls*

Categorizing MPI I/O function calls

Data access functions `MPI_File_write`…/`MPI_File_read`…

Positioning

- Individual file pointers: no special qualifier
- Shared file pointers: …`_[shared|ordered]`… (non-collective or collective, respectively)
- Explicit offset: …`_at`…

Synchronism

- Blocking: no special qualifier
- Non-blocking: either `MPI_File_i`… or …`_[begin/end]` (non-collective or collective, respectively)

Process coordination

- Individual: no special qualifier
- Collective: …`_all`…

Further MPI I/O functions

Pre-allocating space for a file [may be expensive]

- `MPI_FILE_PREALLOCATE(fh, size)`

Resizing a file [may speed up first write access to a file]

- `MPI_FILE_SET_SIZE(fh, size)`

Querying file size

- `MPI_FILE_GET_SIZE(filename, size)`

Querying file parameters

- `MPI_FILE_GET_GROUP(fh, group)`
- `MPI_FILE_GET_AMODE(fh, amode)`

JÜLICH
FORSCHUNGSZENTRUM

Parallel Programming with MPI

Blue Gene/Q Extensions

18. March 2013 | Florian Janetzko

MPI Extensions – MPIX Function Calls

IBM offers extensions to the MPI standard for Blue Gene/Q

- Currently only C interface available

C/C++

```
#include <mpix.h>
```

- Functions start with `MPIX_` instead of `MPI_`
- Functions related to I/O discussed below
- For other MPIX functions please see

http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN/UserInfo/MPIextensions.html

MPI Extensions – MPIX Function Calls

C/C++

```
int MPIX_Pset_diff_comm_create(MPI_Comm *pset_comm_diff)
```

- Collective call on `MPI_COMM_WORLD`
- Returns a communicator which contains only MPI ranks which run on nodes belonging to different I/O Bridge Nodes
- The name of this function is chosen for backwards compatibility, since there are no psets on the Blue Gene/Q anymore.

C/C++

```
int MPIX_Pset_diff_comm_create_from_parent(
                MPI_Comm parent_comm,
                MPI_Comm *pset_comm_diff)
```

- Like `MPIX_Pset_diff_comm_create` however, collective call on `COMM`

MPI Extensions – MPIX Function Calls

C/C++

```
int MPIX_Pset_same_comm_create(MPI_Comm *pset_comm_same)
```

- Collective call on `MPI_COMM_WORLD`
- Returns a communicator which contains only MPI ranks which run on nodes belonging to the same I/O Bridge Nodes
- The name of this function is chosen for backwards compatibility, since there are no psets on the Blue Gene/Q anymore.

C/C++

```
int MPIX_Pset_same_comm_create_from_parent(
                MPI_Comm parent_comm,
                MPI_Comm *pset_comm_same)
```

- Like `MPIX_Pset_same_comm_create` however, collective call on `COMM`

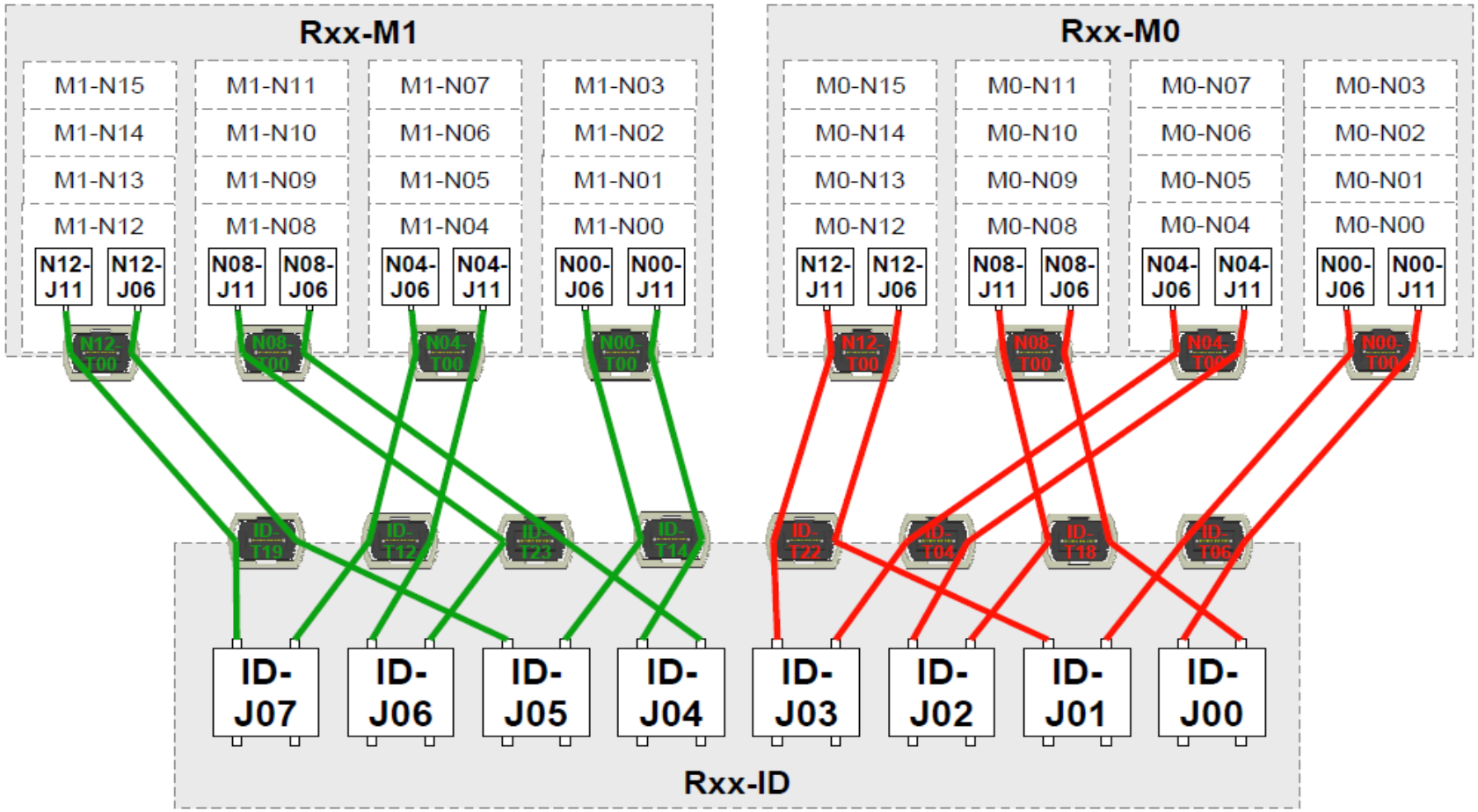
MPI Extensions – MPIX Function Calls

C/C++

```
int MPIX_Pset_io_node(int *io_node_route_id,
                      int *distance)
```

- Returns information about the I/O node associated with the local compute
- The I/O node route identifier `io_node_route_id` is a unique number, yet it is not a monotonically increasing integer
- The distance to the I/O node `distance` is the number of hops on the torus from the local compute node to the associated I/O node. On BG/Q the I/O Bridge Nodes are those nodes that are closest to the I/O node and will have a distance of '1'
- The name of this function is chosen for backwards compatibility, since there are no psets on the Blue Gene/Q anymore.

Blue Gene/Q: I/O-Node Cabling (8 ION/Rack)



JÜLICH
FORSCHUNGSZENTRUM

Parallel Programming with MPI

Blue Gene/Q Exercise

18. March 2013 | Florian Janetzko

Access to JUQUEEN

Login

1. open a terminal
2. `ssh -A -X hpclabXX@juqueen`

Compilation

1. IBM XL compiler suite
   - MPI compiler wrapper: `mpixlf77, mpixlf90, mpixlf95, mpixlf2003, mpixlc, mpixlcxx`
2. GCC compiler suite
   - MPI compiler wrapper: `mpif77`, `mpif90`, `mpicc`, `mpicxx`

MPI starter: `runjob` (batch only)

Running parallel jobs on JUQUEEN

Batch jobs

1. open a terminal
2. `ssh -A -X hpclabXX@juqueen`
3. to start applications with `n` tasks with submit the following job script with `llsubmit <name_of_jobscript>`

```
#@job_name         = LoadL_Sample_1
#@comment          = "myjob"
#@error            = job.err
#@output           = job.out
#@environment      = COPY_ALL
#@wall_clock_limit = 00:10:00
#@notification     = error
#@notify_user      = <email>
#@job_type         = bluegene
#@bg_size          = 32
#@queue

runjob -n 8 -p 1 --exe <application>
```