The MPI Message-passing Standard Practical use and implementation (VII)

SPD Course
01/04/2015
Massimo Coppola

MPI-IO

Rationale

- MPI-IO is a subset of the MPI API designed to manipulate files by applying/extending previously discussed MPI concepts (Datatypes, Collective operations)
- MPI-IO goes beyond POSIX file semantics in order to allow
  - Non-interfering access to files from parallel processes
  - Optimization opportunities in file access to the implementation layer
    - on both ordinary and parallel file systems
  - Straightforward mapping to files of in-memory distributed data structures MPI datatypes

Basic concepts in MPI-IO

- MPI **files**
  - Ordered collection of **typed** data items
  - Opened by **groups** of MPI processes
  - Collective op.s on files are collectives over that group
- the opaque object **file handle** is used to reference a file in MPI calls
  - Created by MPI_FILE_OPEN, destroyed by MPI_FILE_CLOSE
- Collective file operations are ordinary collectives
  - We will skip MPI-IO *split collectives*, which are different
- Type matching rules are those of MPI datatypes
  - We will not deal with the "data representation" extensions of MPI-IO, that manage file representation conversions to enable files that are portable across architectures

Basic concepts in MPI-IO

- **file displacement** is an offset in bytes from file the beginning
  - all we have here in POSIX semantics
- **etype** (elementary type) unit of data access and positioning
- **filetype** a template for partitioning and accessing the file
- **view** is the way each process sees the file data:
  - what parts of the file the process can access
  - built on top of the etype and filetype

Data access and *etype*

- **etype** (elementary type) unit of data access and positioning
  - Any basic or derived MPI datatype subject to
  - Constraint that alll the typemap displacements are non-negative and monotonically increasing
    - Both size and extent are obviously significant
  - Data access is performed in whole etype units
- **filetype**
  - Either a single etype or
  - a derived MPI datatype built from multiple instances of the base etype
    - Constraint on the filetype "holes": their extent must be a multiple of etype extent

File View

- A file view dictates which portion of the file a process can access
- Size, extent and holes in the fileview are all significant
- Data in the holes of the fileview are guaranteed not to be altered by any MPI-IO operation from the current process
- `MPI_File_set_view` set a process' file view

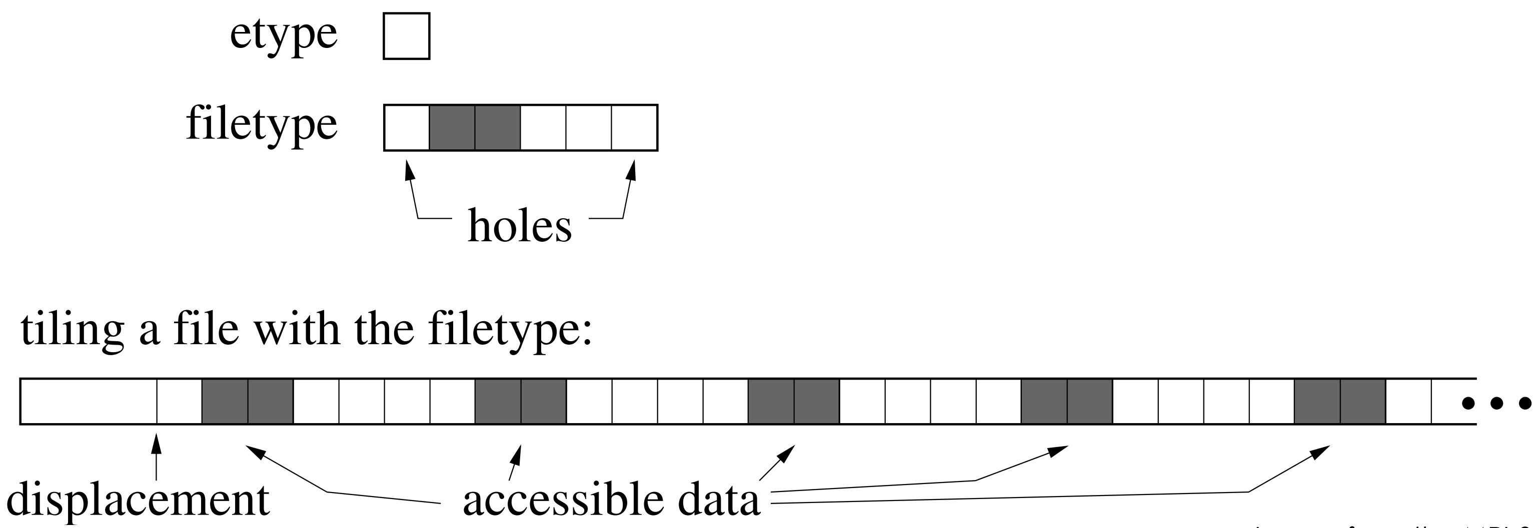

Image from the MPI-3 standard

Setting a fileview

- ```
  int MPI_File_set_view(MPI_File fh,
        MPI_Offset disp, MPI_Datatype etype,
        MPI_Datatype filetype,
        const char *datarep, MPI_Info info)
  ```
- Add or change the fileview for a file handler
  - This is done per-process
- Specify the etype and filetype
- The displacement allows to skip the initial part of the file
  - Skip headers or data with a different organization
  - displacement is an offset in bytes from the start of the file
  ```

Fileviews and collective I/O

- Multiple interlacing fileviews allow several processes to collectively read a whole file
  - Not an easy property to ensure, within Posix
    - File-block granularity issues
  - each process reads/writes only its own data according to the appropriate datatype
  - data is gathered/scattered on the actual file by MPI

Image from the MPI-3 standard

Basic concepts in MPI-IO

- **file size** : in bytes, measured from the beginning of file
  - a file size offset from the beginning of file gives the byte immediately following end of file
- **offset** : position in the file relative to the current *view*, expressed in count of etypes
- **file pointer** : an MPI-maintained implicit offset within a file
  - **Individual** file pointer are local to each process
  - **Shared** file pointers are shared by the group of processes sharing the file handle

3 types of file positioning

- Individual file pointers
  - Each process maintains its own file pointer, i.e. the offset where read and writes happen
  - Read and writes from different processes are independent
  - Method used by routines which do not have any positional qualifier in their name
- Shared file pointer
  - All processes share a common file pointer
  - Read and writes are collective operations
  - Method used by routines of type _SHARED and _ORDERED
- Explicit offsets
  - Use an explicit offset parameter
  - Do not need or modify any kind of file pointer
  - Primitives of the _AT type use explicit offsets

File open

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

- Collective within a communicator
  - Comm must be and intracommunicator
  - Use MPI_COMM_SELF by a single process is allowed
  - All processes must provide the same accessmode
  - All filenames must refer to the same file
- Initially the files is always seen as a byte stream (default fileview)
  - A specific fileview must be set later on via MPI_SET_FILE_VIEW
- A file handler is returned that is used by other MPI-IO primitives
- All file resources shall be freed via MPI_CLOSE before calling MPI_Finalize
  ```

File open

- Several obvious modes, MPI_MODE_* :
  - RDONLY, RDWR, WRONLY
  - CREATE — create file if does not exist
  - EXCL — error if file does exist
  - DELETE_ON_CLOSE
  - UNIQUE_OPEN — never concurrently open this file
  - SEQUENTIAL — file is only accessed sequentially
  - APPEND — set f.pointer to the file end

- Modes may be combined as bitmasks, where not conflicting
- Many have same semantics as POSIX
- UNIQUE_OPEN applies to MPI and non-MPI calls

MPI_Info

- Mechanism for providing additional information to the MPI implementation
  - Simple MPI API to set up and query opaque objects implementing (key,value) maps
  - MPI_Info tags can be used by MPI-IO to optimize the file system interface and its implementation
  - The semantics of any MPI-IO primitive does **not** change
    - Info tags are implementation-specific
    - Implementations are free to ignore any MPI_Info (obviously including any unsupported hints)
  - Access performance and/or resource usage can be improved as a consequence
  - MPI_INFO_NULL means no info is provided
  - Info hints are specified per file
  - Some hints constrained to match within a collective

Some Info tags reserved for MPI-IO

- access_style (list of tags)
  - Declares the kind of file access of the program
  - {read_once, write_once, read_mostly, write_mostly, sequential, reverse_sequential, random }
- collective_buffering (bool)
  - SAME on all processes, enable collective buffering
- cb_block_size (int) cb_buffer_size (int) cb_nodes (int)
  - SAME, size of each **file** buffer for collective I/O, overall size of buffers on each target node, number of target nodes
- io_node_list
  - SAME, list of I/O devices used to store the file
- striping_factor (int) striping_unit (int)
  - SAME, only relevant at file creation
  - Number of I/O devices for file striping, and suggested size in bytes of the striping units

Basic file management

- `int MPI_File_close(MPI_File *fh)`
  - Only needs the file handler

- `int MPI_File_delete(const char *filename, MPI_Info info)`
  - If the file is open, results are implementation dependent: file may not be deleted and/or further data access may fail
  - If file is not deleted, errors MPI_ERR_FILE_IN_USE or MPI_ERR_ACCESS will be triggered

- `int MPI_File_set_size(MPI_File fh, MPI_Offset size)`
- `int MPI_File_get_size(MPI_File fh, MPI_Offset *size)`
  - Both offsets here are in bytes

Overall schema of I/O primitives

positioning	synchronism	coordination	
		noncollective	*collective*
explicit offsets	*blocking*	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	nonblocking & split collective	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
individual file pointers	*blocking*	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	nonblocking & split collective	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
shared file pointer	*blocking*	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	nonblocking & split collective	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

Examples

- `int MPI_File_read_at(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
  - Explicit offset
  - Local buffer is an array of etypes
  - AT routines can't be called for MODE_SEQUENTIAL files

- `int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
  - A collective (the comm is cached by the fh)

- Analogues: write_at, write_at_all ; non blocking versions which are managed by TEST* and WAIT*

Examples

- `int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)`
- Reads using the implicit file pointer offset for this process
- Analogues for writing, collective form and for non blocking

References

- MPI-3 Chapter 13 : Sections 13.1 – 13.2.4, 13.2.6 – 13.4.4 (skip space preallocation, split collectives, data representations); read sections 13.6.2 – 13.6.9
- Optimizing Noncontiguous Accesses in MPI-IO (Thakur, Gropp, Lusk)
  – http://www.mcs.anl.gov/~thakur/papers/mpi-io-noncontig.pdf
- Skim through MPI-3 chapter 9 for details about MPI_Info structures.