Review of Proposed High-level APIs for Parallel-I/O

Anurag Acharya, Joel Saltz, Jeff Hollingsworth
Department of Computer Science
University of Maryland, College Park 20742

June 24, 1996

(revised 6/26/96)

1 Introduction

This note serves two purposes. First, it identifies a set of features that we believe should be considered for inclusion in a high-level parallel-I/O API. Second, it describes three proposed high-level parallel-I/O APIs, PIOFS from IBM, PFS from Intel and MPI-IO, in terms of these features – which of them have been provided and in what form.

List of features::

1. Support for asynchronous I/O: whether completion has to be polled for or can an event be delivered a la select().
2. Support for collective I/O: what are the synchronization semantics associated with collective I/O; whether it is possible to specify subgroups of processes for collective I/O.
3. Support for batching I/O requests
4. Support for processor-specific file views, *i.e.*, global-to-local mapping. If allowed, when can these views be created and whether it is possible to modify them. What is semantics of trying to add to extend the end of a view.
5. Support for file pointers
6. Support for direct I/O (explicit offset), the specific offset to use is passed with the I/O request and not implicitly specified with via a file pointer
7. Support for implicit synchronization: different modes of use and implicit synchronization
8. Mapping between memory locations and file locations: how are memory and file locations involved in an I/O operation specified and how they are mapped to each other.
9. Support for hints: data placement hints, caching hints, access order hints, sharing hints, collective hints
10. Support for inquiry functions: which of the hints have been honored, what are the parameters of the underlying file systems (striping unit, block size, num of async operations allowed etc)

11. Support for memory mapping
12. Support for file checkpoints
13. Support for error handling: if an error happens, is it possible to get information to help locate the error.
14. Support for file locking: enforced/advisory
15. Support for application-level cache control: whether it is possible for an application to control the consistency operations for the client and/or the server cache.

2 PIOFS

Support for asynchronous I/O: PIOFS does not support asynchronous I/O.

Support for collective I/O: PIOFS does not support collective I/O.

Support for batching I/O requests: PIOFS does not support batching of I/O requests. It does provide the `readv()/writev()` interface which allows data from a single I/O operation to either be placed in multiple buffers (`readv()`) or to be taken from multiple buffers (`writev()`).

Support for processor-specific file views: PIOFS provides extensive support for processor-specific file views. PIOFS files are layed out as a two-dimensional matrix where the horizontal dimension is the number of *cells* and the vertical dimension is the number of striping units in each cell. A processor-specific file view can be created by specifying a partition of the file and specifying which partition is to be associated with the calling processor. Partitions can be along coordinate axes of the layout matrix or they can be constructed by specifying (stride,initial_block) pair for each dimension. PIOFS supports dynamic creation of views – a view creation command can be issued as long as the file is open. There is no implicit synchronization associated with file partitioning or view creation. It is perfectly legal for each processor to define a different partitioning and to operate on overlapping sections of a file. Extending a file view (by writing at its end) can lead to holes in the file since it is possible that only a subset of the processors extend their views.

Support for file pointers: PIOFS provides two kinds of file pointers – PIOFS file pointers and AIX file pointers. PIOFS file pointers are 8 bytes long; AIX file pointers are 4 bytes long. Both these file pointers point to positions in the *file-view* and not in the entire file.

Support for direct I/O :

Support for implicit synchronization: PIOFS provides two synchronization modes – NORMAL and CAUTIOUS. The NORMAL mode provides no synchronization or serialization. It is the responsibility of the application to ensure that overlapping operations are properly synchronized. The CAUTIOUS mode guarantees serializablity – the effects of every I/O operation are guaranteed to be atomic. For example, after concurrent two writes to the same set of locations, the contents are guaranteed to correspond to only one of the operations.

Mapping between memory locations and file locations: PIOFS supports the `read()` and `readv()` interfaces (with the corresponding write routines). The former interface associates a linear range in the memory with a linear range in the file; the latter interface associates a vector of linear ranges in the memory to a linear range in the file. That is, the latter interface supports an explicit scatter into memory and an explicit gather into the file. Note that with processor-specific file views, it is possible that this may implicitly cause both a gather into memory and a scatter into the file.

Support for hints and file attributes: PIOFS supports only data placement attributes. It is possible to specify the initial server for striping purposes, the number of servers and the striping unit. The set of servers and their order is, however, fixed. It is not possible, for example, to specify a custom order in the file is to be striped or to specify only a subset of servers. Note that these are commands and not hints – and error is returned if PIOFS is unable to honor the command. PIOFS does not support user control of caching or access order hints. It also does not support sharing hints which might be useful for copy avoidance.

Support for inquiry functions: PIOFS allows the user to query information about individual files as well as the entire file systems. For a file, it provides information about the size of the file, size of its checkpoint version, striping unit, number of servers, the initial striping server. For a file system, it provides information about the number of servers, file system block size, and number of blocks currently available.

Support for memory mapping: PIOFS does not provide support for memory mapping.

Support for file checkpoints: PIOFS provides support for fast file checkpointing. One checkpoint can be maintained for every file. The file-view scheme mentioned above can be used to read either from the main file or the checkpoint file. The checkpoint version cannot be modified. It can, however, be deleted.

Support for error handling: PIOFS provides limited support for error handling. In case of errors, the kind of error is reported but no information is provided on the location of the error. For example, if an error occurs in a large I/O request that retrieves data from disks on multiple servers, it is not possible to figure out the erring server.

Support for file locking: PIOFS does not provide file locking.

Support for cache control: PIOFS does not support application-level cache control. In particular,i t contains no analogues of `sync()` and `fsync()`.

3 MPI-IO

Support for asynchronous I/O: MPI-IO supports asynchronous I/O by the `MPIO_Iread` and `MPIO_Iwrite` groups of functions. It provides asynchronous versions of both independent and collective operations. Completion of asynchronous operations has to be polled for using the `MPIO_Test` or the `MPIO_Wait` groups of functions.

Support for collective I/O: MPI-IO provides extensive support for collective I/O. Collective versions of all I/O operations are provided. Collective I/O operations can be synchronous or asynchronous. Collective communication is done in terms of an MPI communicator. All members of a communicator must participate. The communicator is specified when the file is opened.

Support for batching I/O requests: MPI-IO does not directly support batching of I/O requests like the POSIX `lio_listio` interface. However, it is possible to cause a large number of

I/O operations to occur with a single function call. This is achieved by associating datatypes (and strides) with both the memory buffer used for I/O and files. This allows scatter-gather operations with regular patterns.

Support for processor-specific file views: MPI-IO allows processor-specific file views to be created by associating suitable datatypes with the file when it is opened. A file view can be dynamically changed using `MPIO_File_Control()`. If the file is being independently opened, different processors can define different partitioning schemes and can operate on overlapping sections of a file. Extending a file view (by writing at its end) can lead to holes in the file since it is possible that only a subset of the processors extend their views. If the file is opened collectively, all processors must specify the same partitioning scheme.

Support for file pointers: MPI-IO provides two kinds of file pointers – independent pointers which are used routines which belong to the `MPIO_xxx_next_yyy` group or shared pointers which are used by `MPIO_xxx_shared__` routines. In addition to file-pointer-based routines, MPI-IO also provides routines, the `MPIO_xxx_yyy` group which use an absolute file offset.

Support for direct I/O : Direct I/O is supported for blocking and non-blocking I/O in both collective and independent mode. The `MPIO_Read` and `MPI_Write` calls provide blocking independent direct I/O.

Support for implicit synchronization: MPI-IO provides two synchronization modes – MPIO_RECKLESS and MPIO_CAUTIOUS. The The MPIO_RECKLESS mode provides no synchronization or serialization. It is the responsibility of the application to ensure that overlapping operations are properly synchronized. The MPIO_CAUTIOUS mode guarantees serializablity – the effects of every I/O operation are guaranteed to be atomic. For example, after concurrent two writes to the same set of locations, the contents are guaranteed to correspond to only one of the operations.

Mapping between memory locations and file locations: The file locations to be accessed are specified as a base location and a count of filetype items. The base location can be a file pointer or an explicit offset. The memory locations to be accessed are similarly described by a base location and a count of buftype items. Using appropriate buftypes and filetypes, it is possible to arrange a scatter-gather operation for both read and write calls.

Support for hints and file attributes: MPI-IO supports a large set of hints. Hints can be specified on file open or via `MPIO_File_Control()`.

- access order hints: MPIO_SEQUENTIAL, MPIO_REVERSE_SEQUENTIAL, MPIO_RANDOM
- partitioning hints: BROADCAST_REDUCE, SCATTER_GATHER, HPF, HPF_BLOCK, HPF_CYCLIC
- access type hints: MPIO_READ_ONCE, MPIO_WRITE_ONCE, MPIO_READ_MOSTLY, MPIO_WRITE_MOSTLY
- data placement hints: striping unit, number of nodes to stripe over, list of nodes to stripe over.
- sharing hints: MPIO_NB_PROC - number of processes that "are likely to" access this file

Support for inquiry functions: MPI-IO allows the user to query information about individual files. Using `MPIO_File_Control()`, it is possible to query which of the hints were accepted by the

MPI-IO implementation. It is also possible to access other information associated with an open file like the access mode, datatypes associated with memory and file locations, and file-size.

Support for memory mapping: MPI-IO does not provide support for memory mapping.

Support for file checkpoints: MPI-IO does not support file checkpointing.

Support for error handling: MPI-IO defines several classes of errors (unrecoverable, recoverable, eof) and allows the user to register handler functions for each class.

Support for file locking: MPI-IO does not provide file locking.

Support for cache-control: MPI-IO supports limited application-level cache control. The `MPIO_File_Sync()` function can be used to flush contents of a file (presumably from both client and server caches, if they exist) to stable storage. This function does not guarantee flushing the results of asynchronous I/O requests that had not completed.

4 PFS

Support for asynchronous I/O: PSF supports asynchronous I/O by the `iread`, `ireadv`, `iwrite` and `iwritev` functions. Completion of asynchronous operations is tested using `iodone` or `iowait`. These functions use an ID returned returned from `iread, ireadv, iwrite` or `iwritev`. `iodone` determines whether the asynhrouous read or write operation is complete but does not block. `iowait` blocks until the asynchronous I/O operation has completed.

Support for collective I/O: PFS supports a set of predefined collective I/O modes via the synchronization modes (M_RECORD, M_GLOBAL, and M_ASYNC). Synchronization modes are specified at file open, and normal I/O requests are then used for the collective operations.

Support for batching I/O requests: PFS does not support batching of I/O requests. It does provide the `readv()/writev()` interface which allows data from a single I/O operation to either be placed in multiple buffers (`readv()`) or to be taken from multiple buffers (`writev()`).

Support for processor-specific file views: PFS does not directly support processor-specific file views. However, a type of file view can be created for read-only files using the F_SETSATTR request of the fcntl system call to change the stripe factor and stripe size. This is useful to read a matrix using a different decomposition than it was written.

Support for file pointers: PFS supports 4-byte OSF/1 file pointers and 8-byte extended file pointers.

Support for direct I/O : PFS supports synchronous direct I/O with the readoff/writeoff and asynchronous direct I/O with ireadoff/iwriteoff.

Support for implicit synchronization: PFS provides five synchronization modes - M_UNIX, M_LOG, M_SYNC, M_RECORD and M_GLOBAL.

- M_UNIX - each processor has its own file pointer, file operations are performed on a first-come, first-served basis, and access is unrestricted. This mode supports standard UNIX I/O atomicity, unlike the M_ASYNC mode (see below).
- M_LOG - All processors share the same file pointer and file operations are performed on a first-come first-served basis. Records may be of variable length. Because requests can be

carried out in any order, the order of data in a file may vary from run to run. There is no requirement that all processors participate. This mode is typically used for error logs.

- M_SYNC - All processors share the same file pointer and file operations are performed in order of processor number. All processors must complete their access before any processor can access the file again. In this mode, all processors must perform the same file operations in the same order.

- M_ASYNC - All processors have their own file pointer and file operations may be performed in any order. In this mode, the application must ensure synchronization between nodes (if required).

- M_RECORD - All processors are supposed to open the file and carry out the same operations on the file in the same order. The I/O request size must be the same from each node. No synchronization is enforced and each processor maintains its own file pointer. Files created in this mode store data in processor order.

- M_GLOBAL - All processors share a file pointer - each I/O request is a global operation in which all processors must carry out the same file accesses in the same order. All processors read the same data and all processors are supposed to write the same data (although this is not checked).

Mapping between memory locations and file locations: - PFS supports `cread`, `iread`, `cwrite` and `iwrite` calls which associate a linear range in memory with a linear range in a file. The `creadv`, `ireadv`, `cwritev` and `iwritev` calls associate a vector of linear ranges in the memory to a linear range in the file. That is, the latter interface supports an explicit scatter into memory and an explicit gather into the file.

Support for hints and file attributes: - Default data placement and striping characteristics are specified when a PFS file system is mounted. Data placement and striping characteristics can be changed on a per-file basis via the fcntl() system call.

Support for inquiry functions: PFS provides the `statpfs` call to allow users to find out the stripe size for the parallel file system along with a list of pathnames specifying the set of directories that define the stripe group for a parallel file system. The same data is also available to applications via the fcntl system call.

Support for memory mapping: PFS does not provide support for memory mapping.

Support for file checkpoints: PFS does not directly support file checkpointing

Support for error handling: PFS provides error codes for each call.

Support for file locking: PFS provides whole file locks using normal UNIX semantics.

Support for cache control: PFS supports buffering on a per-file basis via the fcntl system call.