Scalable I/O Low-Level Parallel File System API

Frequently Asked Questions

Jim Zelenka
School of Computer Science
Carnegie Mellon University

jimz+@cs.cmu.edu
http://www.pdl.cs.cmu.edu/

Highlights

Relationship with POSIX/χOpen
Shared file pointers
Third-party transfer
Checkpoint support
Data type support
File type support
Permissions and access control
Asynchronous I/O and threads
Scatter/gather complexity
Subfile and remapping support
File remove: immediate effect
Transfer size
File namespace
File-file and memory-memory operations
Collective I/O
Cache Coherence issues

POSIX Compliance

This API is intended to lead to POSIX extensions.

First, some experimentation is necessary to demonstrate utility and correctness.

- This demonstration defines functionality entirely separately from POSIX.
- This way, an implementation could be a user-level library, and need not be bound to existing implementations of the POSIX API.

When utility is clear, we will suggest that this API become part of POSIX/χOpen. At that time, redundant interfaces should be merged.

Shared File Pointers

One task's file accesses implicitly change file pointers seen by other tasks.

Shared file pointers represent distributed state, and require synchronization to update

- Local synchronization between threads
- Global synchronization between nodes

Often unnecessary

- Programmer might not need or want this
- Application synchronization/communication could be exploited

=> Best to implement in separate library

- Recoverable, distributed lock manager

Third-Party Transfer

One node transfers to/from another node's address space

=> No explicit requirement in this API

- insecure
- difficult to debug
- hard to ensure high performance across all interesting parallel architectures (dangerously close to DSM)

On systems with shared/distributed shared memory, multiple nodes may be able to map the same memory addresses

- implicit third party transfer

Checkpoint Support

Parallel applications are often long-running, and therefore a major source of I/O is checkpoints, suggesting that high-performance checkopinting is needed.

No <u>explicit</u> checkpoint support

- too many convenience issues (namespace management, semantics, recovery, etc)

=> FastCopy extension

- Section 16 (page 113) of v1.0 API

FastCopy makes a snapshot of current file state

- implementation details open, but something like copy-on-write recommended
- an application or higher-level library could use this to support efficient checkpointing

Data Type Support

File system knowledge of and specialization for structured files

=> None: files are streams of bytes

High-level libraries can provide data types (MPI-IO)

Much harder to ensure high performance in a low-level implementation

File Type Support

File system awareness of file format/data layout

No support for file type conversion, file-type-specific access or semantics

=> Support is provided for <u>identifying</u> file types verbosely: file labels

naming conventions too limited

Labels documented in v1.0 API

- Section 1.12 describes labels briefly
- Section 4.17 defines label type
- Section 13.9 defines GetLabel, SetLabel operations via sio_control()

Permissions and Access Control

Many filesystems need/desire access control

=> No explicit recognition in API

piggyback on POSIX or local implementation

Different filesystems provide different semantics (ie UNIX, AFS, etc)

Implementations may add additional error codes, indicating permissions issues

Asynchronous I/O and Threads

Why provide async I/O at all?

on some platforms, more efficient than threads

Why provide synchronous I/O?

on some platforms, may be more efficient (can hijack calling thread to do work)

What is minimal async support?

must provide at least one “outstanding” I/O

legal to implement by only performing I/O when issuing or completing function is called

Scatter/Gather Complexity

I/O descriptor is list of strided accesses

Encourages programmers to issue a lot of I/O work at once (for performance)

Strided type is fairly simple

offset, len, stride, count

=> Not nested strided accesses

Nested strided accesses are under discussion

may increase batching opportunities

harder for programmers to use than basic strided

Subfile and Remapping Support

Vesta has concept of subfiles, ability to change file data <-> address mapping

=> No requirement for subfiles or remapping in this API

High-level library can provide naming and access functionality

Utility seems specialized, and not performance-critical at lower level

File Remove: Immediate Effect

=> File may be gone (unaddressable) as soon as file is removed

System is not required to track who has a file open

Easier for some implementors, especially with respect to fault-tolerance

Transfer Size

API designed to encourage large transfers (batching)

Small transfers are equally possible under API

Up to individual implementations to handle various batch and transfer sizes efficiently

File Namespace

No explicit definition of what namespace looks like

No explicit directory support (and no basic directory operations)

Directory operations could be added as an extension.

Will be more of an issue when merged with POSIX/χOpen.

File-File and Memory-Memory Operations

FastCopy is efficient file duplication

No other direct File-File operations specified

No memory-memory operations specified

outside scope of this document

Collective I/O: Task Groups

Task groups identify participants in group I/O

No explicit definition of task groups

Too dependent upon communication
system/higher-level libraries

=> Tasks are defined with respect to one another
for the purposes of each collective I/O

- "grouping" is implicit
- no notion of relation of these tasks to one another other than the scope of the I/O itself
- no notion of relation of these tasks to other tasks not participating in this I/O whatsoever

Cache Coherence: Domain

Cache coherence is defined with respect to instances of the API implementation

- for some implementations, may be across applications on the same processor (ie, in-kernel implementation)
- for some implementations, may be per running-application (ie, a user-level library implemena-tion with no kernel support)

Cahce Coherence: Server Caching

Implementation may not *have* server(s)

Use/reuse best disclosed to the server via hint mechanism

Hints can describe future I/Os exactly, or patterns of future I/Os

Cache Coherence: Granularity

Define a consistency unit: this is caching granularity

On some platforms, may be a word or a cache line (ie, MPPs)

One some implementations, may be larger (could even be a sector or a file block, for instance)

Byte-level granularity would be extremely inefficient for either of these cases

Cache Coherence: Weak Consistency Failure Modes

Weak consistency requires application to explicitly propogate and refresh file regions

Correct sequence: writer writes, then propogates; reader refreshes, then reads

Potential failures: refresh before propogate, read without refresh, write without propogate

Implicit barrier operation between propogate and refresh; application's responsibility