Scalable I/O Low-level API: A Portable Programming Interface for Parallel File Systems

Garth Gibson

Carnegie Mellon University

http://www.cs.cmu.edu/~garth

P. Corbett, J.P. Prost, IBM
C. Demetriou, G. Gibson, E. Riedel, J. Zelenka, CMU
Y. Chen, E. Felten, K. Li, Princeton
J. Hartman, L. Peterson, U. Arizona
B. Bershad, A. Wolman, U. Wasington
R. Aydt, U. Illinois

http://www.pdl.cs.cmu.edu/SIO/Docs/API_1.0.html

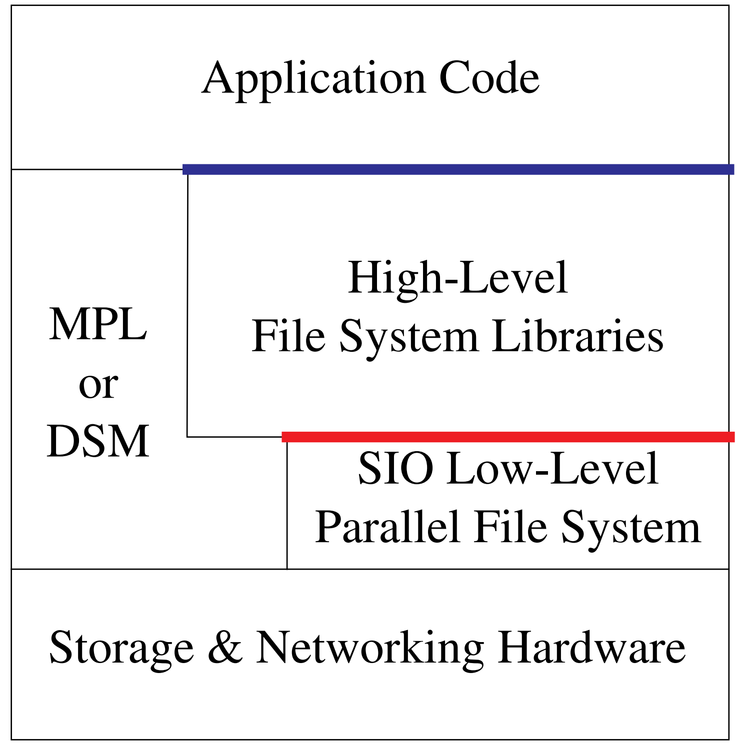
Scalable I/O System Structure

High level API(s)

- provided by libraries, used by applications
- specialized for applications' needs
- for example: MPI-IO

Low level API

- provided by parallel file system, used by high level libraries



MPL = Message Passing Library
DSM = Distributed Shared Memory

Multiple portable high-level libraries (e.g. current vendor interfaces) built on low-level API.

Low-level API Goals

- Portability
- Flexibility
- High performance
  minimal constraint on parallelism, efficiency
- Small number of functions
- Extensibility

Not necessarily:

- All-encompassing
- Trivial to program

Main Features of the API

What's There:

- Synch and async I/O operations
- Scatter-gather to/from file and memory regions
- File access pattern hints
- Application control over client caching
- "File control" operations
- Collective I/O Support
- Primitives for file checkpointing

What's Not:

- Shared file pointers
- Implicit synchronization points
- IPC facilities

Scatter-Gather To and From File and Memory Regions

Efficient support for common access patterns.

Scatter-gather specification contains:

- starting file or memory offset
- size
- stride
- count

I/O operations take a list of them.

Combinations of simple strided patterns expressible.

Nested striding is not supported.

File Access Pattern Hints

Target known optimizations.

Disclosure and advice, not directions.

- Hints can be ignored.

Vendor-extensible “hint classes” support different hint models:

- Ordered: explicit description of future accesses
- Unordered: information about what will be used

Meaning of mixed-class hints is unspecified.

Application Control Over Client Caching

Client caching helps performance, but unnecessary cache synchronization hurts.

Provide a sane default, but allow optimization.

- Clients that know their consistency needs can manage consistency themselves.

Standard options:

- Client caching completely disabled
- Sequential consistency (the default)
- Weakly consistent client caching

Plus, vendor-defined caching modes.

Example of Weakly Consistent Client Caching

Client A	Client B
write(FD, F_{shared}, M_a)	
	unsafe read(FD, F_{shared}, M_b)
propagate(FD, F_{shared})	
	unsafe read(FD, F_{shared}, M_b)

- - - - - - - - Synchronization Event -

	refresh(FD, F_{shared})
	safe read(FD, F_{shared}, M_b)

Writer must propagate and reader must refresh to guarantee reader's cache is up to date.

Control Operations

Like UNIX ioctl() operations, but:

- Multiple controls in a single operation.
- Used to get basic file status
  (replacing UNIX-like stat()).

Can be performed via:

- sio_control()
- sio_open() (e.g. for layout specification)
- sio_test() (but cannot change file state)

Vendor-extensible.

Control Operations: A Few Examples

GetPreallocation and SetPreallocation

- Get and set the amount of space reserved for the file's data.

GetLabel and SetLabel

- Get and set the file's label.
- Label requested by library/application writers for file representation portability.

GetLayout and SetLayout

- Get and set the file's layout (e.g. striping) parameters.
- Layout types are vendor-extensible.

Collective I/O

Allows the file system to optimize requests (batching and coalescing).

Effectively:
coordinated asynchronous read and write.

But: participants inform file system ahead of time that their requests are made as a group.

Interface specified so that file system doesn't need to know about or handle:

- process groups and identifiers
- application's IPC system
- application fault tolerance

Example of Collective I/O

Client A	Client B
H = define(FD, 2 iterations, 2 participants, F_{hint})	
Send H to Client B	
	Receive H from Client A
join(H, “A”, 1, F_{a1}, M_{a1}, &R_{a1})	join(H, “B”, 1, F_{b1}, M_{b1}, &R_{b1})
join(H, “A”, 2, F_{a2}, M_{a2}, &R_{a2})	join(H, “B”, 2, F_{b2}, M_{b2}, &R_{b2})
async_status_any(..., {R_{a1},R_{a2}})	async_status_any(..., {R_{b1},R_{b2}})

Primitives for File Checkpointing

Long-running parallel applications want to checkpoint file state.

Efficient file checkpointing done in file system (e.g. using copy-on-write).

"FastCopy" control operation:

- **Efficiently copies contents of one open file to another open file.**
- **Destination is a regular file.**
- **Semantics and naming of checkpoints are left up to higher-level libraries.**

Low-level API Status

Public release of revision 1.0 October 1, 1996

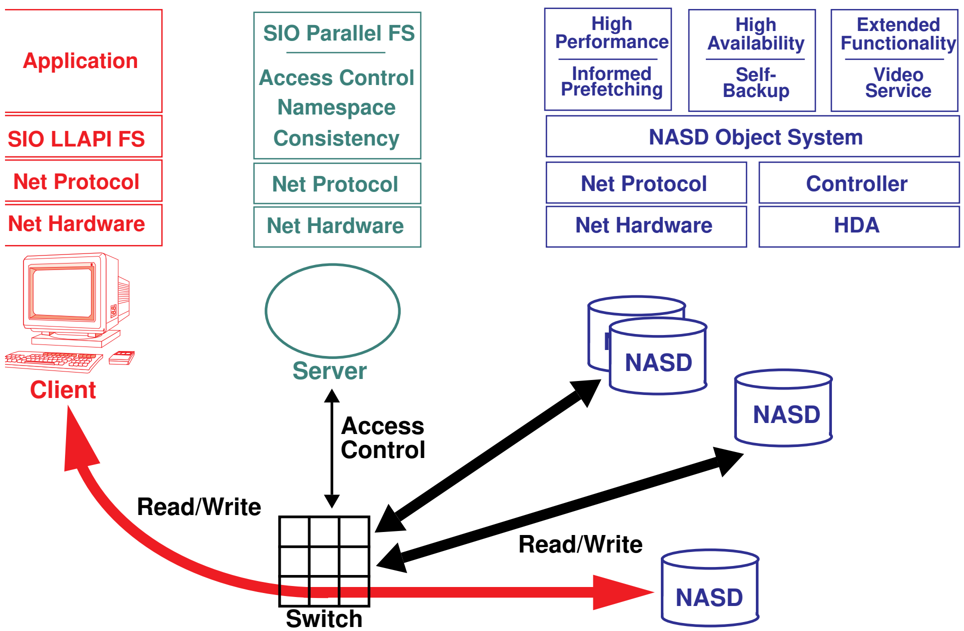
Princeton has implementation on Intel Paragon

CMU doing implementation for
networks of workstations and
network-attached storage

See API document:

http://www.pdl.cs.cmu.edu/SIO/Docs/API_1.0.html

LLAPI on Network Attached Secure Disks



Attach drives to network

- avoid server copy
- trend to commodity cluster networks (VIA)

Build NASD LL-API FS

- “traffic cop” DMA management

Raise drive functional interface

- object store, attribute conditional ops, drive execution

Integrate security into drive

- message digest based capabilities