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

Chris Demetriou
School of Computer Science
Carnegie Mellon University

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

Scalable I/O Initiative

- Looking for solutions to parallel systems' I/O performance problems.
- OS group includes IBM, Intel, Princeton, Arizona, Washington, CMU.

OS group has produced low-level parallel file system API.

- Encourages efficient implementation of common operations.

Scalable I/O System Structure

High level API(s)

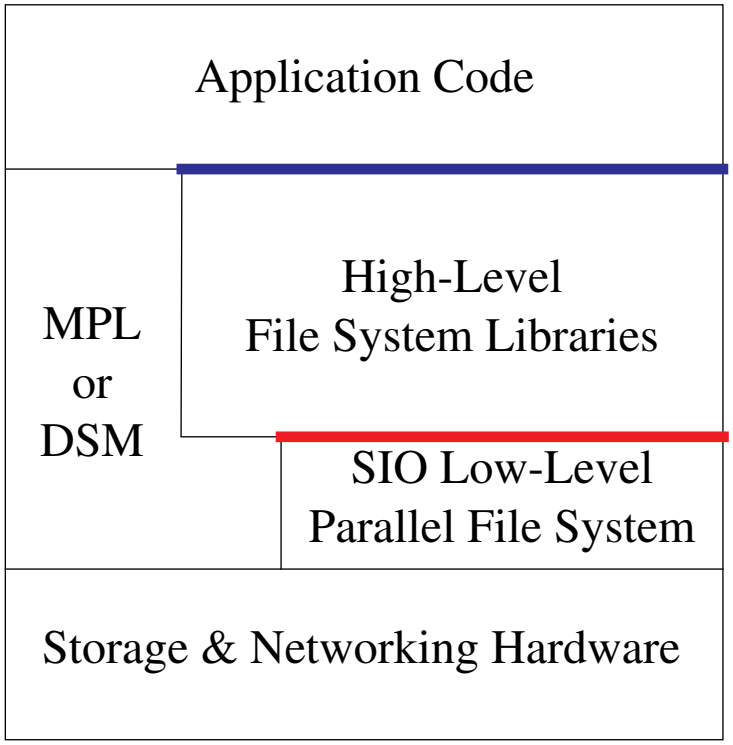
provided by libraries, used by applications

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 constraints on parallelism and efficiency)
- Small number of functions
- Extensibility

Not necessarily:

- All-encompassing
- Easy to use

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.

Can express combinations of simple strided patterns.

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

Caching helps performance, but unnecessary cache synchronization can hurt.

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 before reader's cache is guaranteed to be up to date.

Control Operations

Like UNIX ioctl() operations, but:

- Multiple controls can be performed 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.
- File label requested by library and application writers to help file 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 the 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:

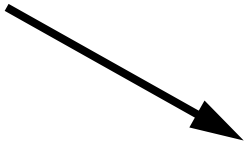
- **process groups and identifiers**
- **application's IPC system**
- **application fault tolerance**

Example of Collective I/O

Client A

H = define(FD, 2 iterations,
2 participants, F_{hint})

Send H to Client B



Client B

Receive H from Client A

join(H, “A”, 1, F_{a1}, M_{a1}, &R_{a1})
join(H, “A”, 2, F_{a2}, M_{a2}, &R_{a2})

join(H, “B”, 1, F_{b1}, M_{b1}, &R_{b1})
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 (e.g. using copy-on-write) must be done in the file system.

"FastCopy" control operation:

- **Efficiently copies contents of one open file to another open file.**
- **Destination is a regular file (i.e. not special in any way, behaves like other files).**
- **Semantics and naming of checkpoints are left up to higher-level libraries.**

Low-level API Status

Revision 1.0 released to the public on October 1, 1996.

Revision 1.0 comments due November 21, 1996.

Princeton has done an initial implementation for the Intel Paragon.

CMU is doing an implementation for networks of workstations.

See also:

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

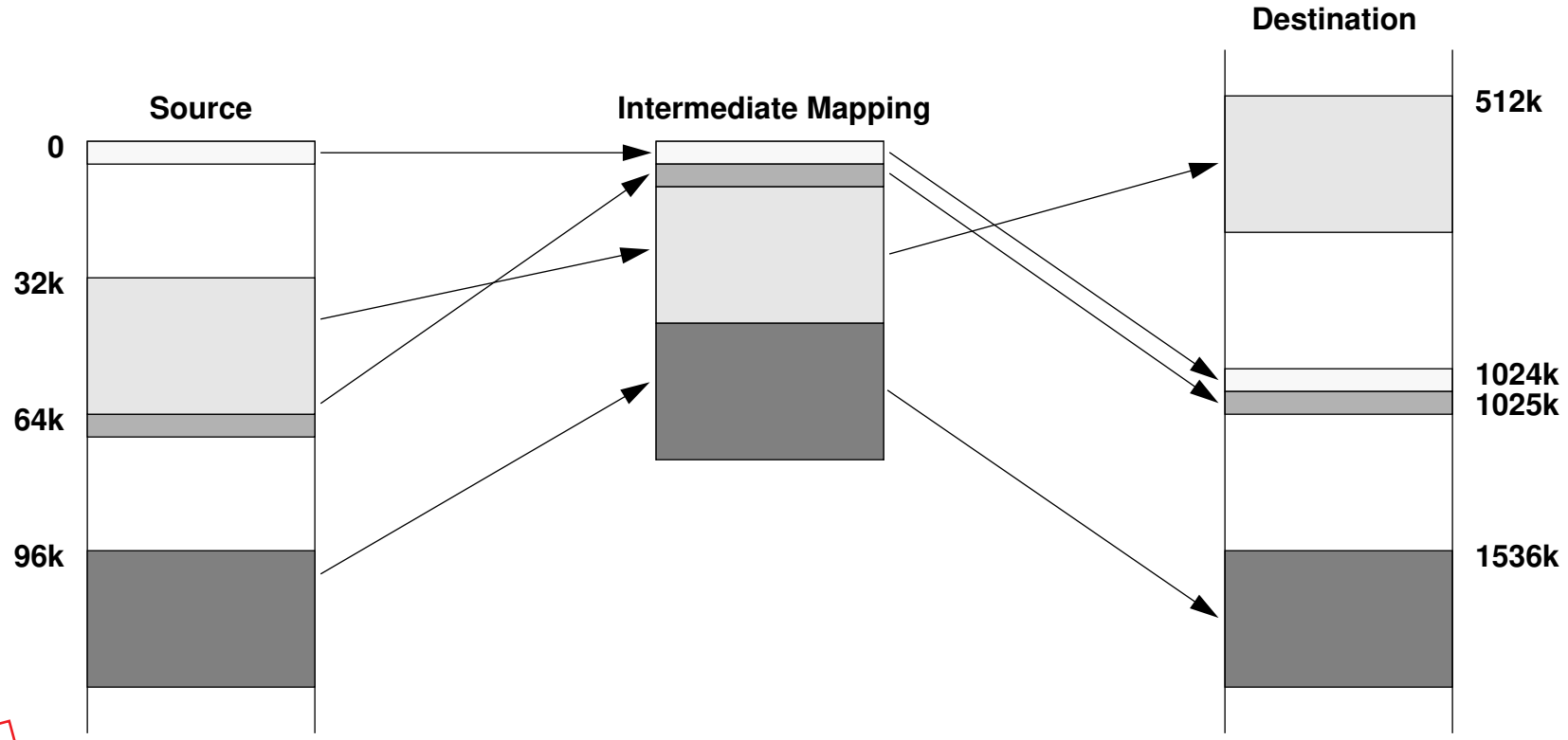
Scatter-Gather Example

Source list:

(offset = 0, size = 1k, stride = 64k, count = 2)
(offset = 32k, size = 32k, stride = 64k, count = 2)

Destination list:

(offset = 1024k, size = 2k, stride = 0, count = 1)
(offset = 512k, size = 32k, stride = 0, count = 1)
(offset = 1536k, size = 32k, stride = 0, count = 1)



Ordered Hints

- **Exactly describe expected in-order sequence of reads and writes.**
- **Are part of a total order (over all files) per task.**
- **Example:**

  Will read blocks 0, 100, 200, and 300 of file A, then

  Will write block 10 of file B, then

  Will read blocks 100-299 of file B, then

  Will write all blocks of file C.

Unordered Hints

- Describe expected use of file regions, like UNIX madvise() does for memory regions.
- Completely independent of each other, may not overlap.
- Examples:

  Blocks 100-299 of file A will be written in reverse sequential order.

  Blocks 3, 5, 7, and 9 of file B will be read.

  Blocks 2, 4, 6, and 8 of file B will not be read.

  All blocks of file C will be read and/or written in random order.

Outstanding Research Topics

- Copy avoidance interfaces
- More complex scatter-gather I/O interfaces
- Caching of I/O region descriptions
- File redundancy specification
- Quality of service specification
- “Dynamic Set” functionality (get first available record, etc.)