# Issues arising in the SIO-OS Low-Level PFS API

Garth Gibson and Daniel Stodolsky,
Parallel Data Lab,
School of Computer Science,
Carnegie Mellon University

## 1 Introduction

At the Scalable I/O Initiative All-Hands meeting in Pasadena (December 12-14, 1994), the SIO-OS agreed to attempt to standardize on a “low-level” parallel file system application programmers interface. This interface was intended to provide a portable, high-performance interface for parallel libraries and compilers. In particular, the interface was to be structured as to admit high-performance implementations in both MPP and multicomputer (workstation network) environments.

In many aspects, this API is analogous to the block device interface found in many operating systems. The interface is flexible, but minimal, and it is expected that most users will use either compilers or libraries that directly manipulate the interface.

This document represents a first pass at addressing many of the issues that arise in establishing such an API, and was discussed at the ARPA PI meeting in San Diego, Jan 19th, 1995. We are circulating this document to solicit comments prior to the sunday Feb 5 API workshop called by Alok Choudhary and Paul Messina (McLean Hilton, McLean VA 703-761-5111). We (Garth and Danner) will attend and would like to represent the OS working group.

## 2 Document Outline

- Goals
- Position of the API in the programming tools hierarchy
- Targeted Environments
- Basic access model
- Feature: UNIX compatibility
- Feature: Flat File Namespace
- Feature: Implementation-dependent protection model
- Feature: No file pointers
- Feature: underlying hardware visibility
- Feature: asynchronous variants
- Feature: fbuf style allocation
- Feature: Support for cache consistency
- Feature: hint interface
- Feature: collective variants of read and write.
- Feature: no direct support of memory mapping
- Summary

## 3 Goal of this document

This document is intended to provide an outline of the issues and features for the OS group proposal for a low-level SIO API. It is hoped by initially obtaining agreement on high level issues such as fea-

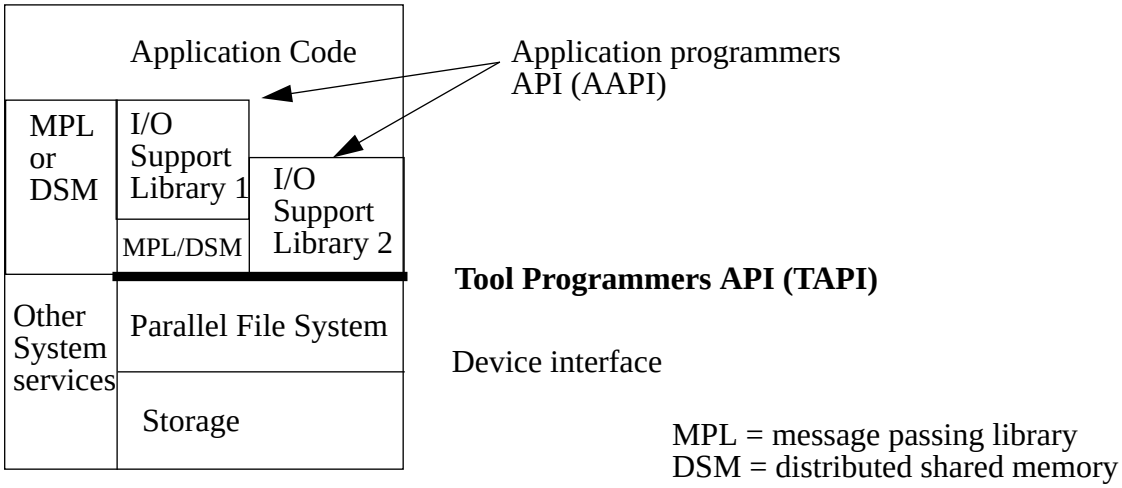


Figure 1: Position of our API in the programming hierarchy.

tures sets and the targeted environment, the generation of detailed call level API can be expedited.

From here on in, API should be taken to mean “the API proposed and supported by the OS working group of the Scalable I/O initiative”.

## 4 Position of the API

This API is intended as the boundary between the OS group and the library writers for both disk and network I/O. Its primary goal is expressing, in a portable manner, primitives that allow efficient utilization of high bandwidth storage and network I/O devices. It is intended to be sufficiently complete to support many styles of I/O, but functionality and performance are chosen over simplicity and ease of programming. Consequently, the API is primarily one for “tool writers” or highly sophisticated application programers.

Another key design goal is to allow application parallelism to be reflected in the I/O subsystem -- the interface should not imply or require synchronization.

Figure 1 shows the logical positioning of the API in the programming hierarchy. Note that the parallel file system can not rely on the application’s choice of either a message passing library or distributed shared memory system.

## 5 Targeted Environments

This API is intended to be portable to both supercomputer and network-of-workstation environments. The following characteristics are assumed to exist in both of these environments

- A reliable high-bandwidth network/interconnect
- Programs are written in FORTAN, C or C++
- Rare concurrent write-sharing of a file by two or more parallel applications.
- Frequent concurrent write-sharing of a file by multiple processes/tasks that comprise a parallel application.
- Routine Recompilation

However, the following supercomputer characteristics are **not** assumed to obtain portability to the multicomputer (workstation network) environment

- Low-latency interconnect (< 1 ms)
- Homogenous CPUs

The multicomputer characteristic of virtual memory hardware is **not** assumed to obtain portability to supercomputer environments.

A parallel application (task) consists of a collection of processes. In order to detect accidental write sharing (and collective I/O), some mechanism needs to exist for processes to indicate to the parallel file system of which task they are a part (see Section 16). A process can belong to only one task.

## 6 Basic Access Models

### 6.1 Disk Storage

A parallel file is viewed as an infinite linear sequence of bytes, and can be modified on byte granularity. The basic interface consists of primitives to

- Create and delete files
- Open and close a file
- Determine the position of the last byte written in a file that is not open for write access[1].
- Read or write a specified byte range within a file.

### 6.2 Network I/O

Need help here - connection or connection-less? packet stream or byte stream?
The rest of this document concentrates on features primarily related to disk I/O.

## 7 Feature: Unix Compatibility

Compatibility with UNIX is not mandatory. It would be nice if it would be easy to construct a mostly unix-compatible library on top of this API, especially at the read(), write(), and lseek() level. Compatibility with fork() semantics and open() and close() is much less important.

## 8 Feature: File namespace

A hierarchical namespace is unnecessary baggage for high-performance storage systems. Cray users have long since proven the ability to manage without them.

A flat name space will be provided. It could consist either of short binary identifiers (e.g, 128 bit ids) or ASCII strings of limited length (e.g, less than 128 characters). It is expected that an implementation-dependent name service will be availible to support a more general name space

1. We are attempting to avoid implicit synchronization of allocation operations.

## 9 Feature: Protection model

File system access control and resource quotas are important aspect of any file system. However, the protection model of the file system is intimately tied to a particular computing environment's concept of identity, authorization and audit. While implementations are encouraged to validate access rights at least on open, the protection model is left up to the implementor.

## 10 Feature: No file pointers

Shared file pointers require implicit synchronizaton and are therefore not desirable. Independent file pointers are inapplicable to network I/O (and any other non-seekable I/O device), inadequate for scatter/gather, multithreaded, or collective I/O, and are therefore not particularly useful either. Therefore, all I/O is performed with explicit offsets relative to the start of the file.

Consequently, the basic access primitives are read_at_offset and write_at_offset. Scatter/gather variants (where both the location of the data in the address space and the offsets of the data in the file are scattered) will be provided.

We expect a file pointer abstraction to be provided by I/O support libraries when appropriate for the application programmer's API.

## 11 Underlying hardware visibility

To achieve high efficiency, it is desirable to have the application perform I/O operations that exploit features of the underlying I/O hardware. For disk arrays, these features would include the stripe size and number of disks. For a network, this might be the hardware packet size and fifo depths. Consequently, an interface should be provided to allow the system to return important characteristics of the underlying hardware.

A set of standard characteristics to describe the important characteristics of all hardware seems impractical. Instead, we propose a standard inquiry mechanism an application can use to request system characteristics, in a manner similar to the SCSI "mode sense" command. Implementations can return the appropriate information for options that make sense for that system and reject other requests. Implementation dependent inquires are permitted.

## 12 Feature: Asynchronous forms

Asynchronous (two-phase) forms of read and write should be present. The initial request specifies the desired data range and target address(es). This call initiates the request and returns an identifier. This identifier can then be used to poll to determine if the request has completed, or make a potentially blocking call that returns when the request has completed with the completion status of the request.

## 13 Feature: Fbuf-style calls

For both networks and caching and prefetching filesystems, data copies can often be avoided by allowing the I/O subsystem to determine the address to which data is read and to surrender a buffer containing data on a write. The value of this interface has been demonstrated in both the MACH device

interface and in network work by Peterson & co.

This interface is extremely natural for packet-oriented networks and for disk requests that are the size of a block and block aligned. For multi-block and sub-block disk requests, the system might want to return a vector of memory locations where data is stored.

To support integration with FORTRAN (which lacks pointers), the application should be able to specify a pool of buffers into which data must be placed. An allocating read then returns a list of buffers and the amount of data in each buffer. The API should include a function to copy the data in a list of buffers into a contiguous area of memory.

## 14 Cache consistency

The Charisma studies of supercomputer I/O (Kotz and others) show that many supercomputer I/O requests are under 200 bytes but show substantial spatial locality. Consequently, small caches/prefetch buffers located at the compute nodes can lead to substantial improvements in performance and decrease in interconnect utilization. Write-behind buffers at the nodes lead to similar improvements. In the multicomputer environment this advantage is expected to be especially significant. Consequently, the design of the system should support client process (node) level caching, which immediately introduces a cache consistency problem.

Traditional distributed file systems use a block-oriented distributed shared memory implementation to solve the cache consistency problem. Unfortunately, this solution has both synchronization and scalability problems. To avoid such a solution, we propose to push the cache consistency problem up to the library and application level, by providing primitives to explicitly propagate changes back to an I/O node and invalidate stale cached data. Applications should be able to trivially disable compute-node caching, prefetching and write-behind for debugging, and open and close should imply both invalidate and propagate.

We propose a cache control granularity of 1 byte.

## 15 Hint Interface

Informing the file system of an applications I/O pattern has been used to achieve substantial speedup by both improving cache management (Cao et al) and increase concurrency (Patterson et. al). This interface should be equally as valuable in a supercomputer environment.

Several systems have hint-like interfaces that allow delivery of a single hint when a file is open (e.g, MPI-IO). Other systems allow hints to be delivered over the life of the application (e.g, TIP). Because of the greater flexibility of the latter, it is preferred.

The interface should be able to describe all frequently occurring I/O calls (reads, writes, cache control). Implementors are free to extend the hint interface to include system and implementation specific hints. An unrecognized hint should be silently ignored.

## 16 Collective variants

Both the Intel PFS and Thinking Machines SFS have collective I/O calls, in which many processes perform a single I/O as a group. In his paper on disk directed I/O, Kotz shows that the flexibility this affords the parallel file system can lead to substantial performance improvements, and the Charisma studies show that collective variants are extensively used on the CM-5. Variants in which every byte

read is sent to exactly one process and in which data is broadcast to all nodes are used.

Fundamentally, a collective read or write call must specify two things

- What processes are participating in the I/O
- What data should be put where in each processes address space.

From the system's perspective, an ideal collective I/O call delivers all this information in a single call, providing a central point for analysis. However, for an parallel's application perspective, it may be more desirable to have each application describe only the data region it wants and leave it up to the system to put the pieces together. Forcing the former interface on applications can reduce parallelism, but applications often have this information (e.g, it is known at compile-time) so providing no way to hand this information to the system would limit performance.

Our proposed interface therefore allows, but does not require, applications to provide the collective description of the data moved in a collective I/O in addition to the per-destination scatter/gather lists.

One additional wrinkle is the issue of naming the processes (nodes) involved in a collective I/O. Processes in parallel task may only know each other by identifiers assigned by a message passing library or DSM system. However, we would like this API to be DSM/MPL-neutral and therefore not use either process or MPL assigned names (the use of process names is also inadequate as does not support multithreaded applications).

Our proposed solution is to have the application associate an identifier with every collective I/O. The set of processes performing a collective I/O agree upon a unique identifier for this collective I/O. Each process involved in the collective I/O request provides

- the local memory addresses this process wants to read or write
- the byte offsets within the file at which to read or write the data
- the identifier associated with the I/O

And at least one of the calls provides the following

- the total number of processes involved in this collective I/O call

And optionally, each of the calls may provide

- A description of the union of the byte offsets which will be read or written by the processes participating in the collective I/O
- A set of (pfs_port_id, address, file offset, length) tuples describing where all the data in the collective I/O will be placed. The "pfs_port_id" is an implementation-dependent identifier that must be obtained by each process by an earlier call to this API.

**17 Feature: Memory Mapping**

Memory mapped I/O has proven popular in several application domains, particularly those that are read-intensive. By avoiding system calls on every read or write, higher performance can be obtained.

The need for a memory-mapped interface in the MPP/multicomputer environment is less clear. Virtual memory hardware is not present in all MPPs, and page-faults are known to substantially degrade the performance of gang-scheduled parallel jobs. Additionally, files in the MPP/multicomputer environment often exceed 4GB, making them difficult to memory map on 32-bit processors. Page coherency and write granularity issues are also nontrivial.

For this reasons, we propose not to standardize a memory-mapped interface in the API. If desired, implementations can provide memory mapped interface.

## 18 Summary

This document has discussed many features that could be present in the SIO OS working group low-level API. In conclusion, the API should

- Support performance and parallelism over ease of us
- Do all I/O with explicit offsets
- Support a flat namespace
- Offer no specific protection model
- Make visible hardware characteristics important for performance
- Offer asynchronous, memory allocating, and collective variants of read and write
- Support incoherent compute-node (compute-process) level caching
- Allow access pattern hints to be given throughout the life on an application
- Allow implementation-specific memory-mapped interfaces

If agreement is reached on these points, the next step is to design a language binding for C and/or FORTRAN. CMU is willing to coordinate this effort, and would suggest publishing the results (along with a rationale) in the upcoming IOPADS workshop.