Design & Implementation of MACSio

(pronounced “max-ee-oh”)

A Multi-purpose, Application-Centric, Scalable I/O Proxy Application

Mark C. Miller

Introduction

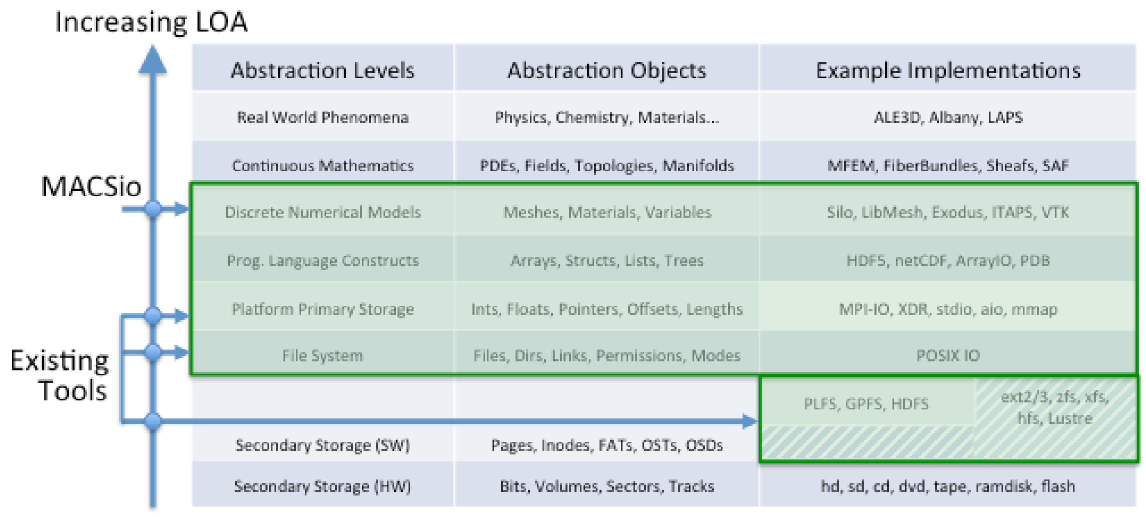
We are proposing to develop “MACSio” (pronounced “max-ee-oh”), a Multi-purpose, Application-Centric, Scalable I/O proxy application. It is being proposed to fill a long existing void in co-design proxy applications that allow for I/O performance testing and evaluation of tradeoffs in data model interfaces and parallel I/O paradigms for multi-physics, HPC applications.

Two key design features of MACSio will set it apart from existing I/O benchmarking tools. The first is the *level of abstraction (LOA)* at which MACSio is being designed to operate. The second is the degree of flexibility MACSio is being designed to provide in driving an HPC I/O workload through parameterized, user-defined data objects and a variety of parallel I/O paradigms and I/O interfaces.

Combined, these features will allow MACSio to closely mimic I/O workloads for a wide variety of real applications and, in particular, multi-physics applications where data object distribution and composition vary dramatically both within and across parallel tasks. These data objects can then be marshaled using one or more I/O interfaces and parallel I/O paradigms, allowing for direct comparisons of software interfaces, parallel I/O paradigms, and file system technologies with the same set of customizable data objects.

Level of Abstraction (LOA) and the HPC I/O Stack

The significance of LOA and its potential for impact on I/O workload cannot be understated. Application programming interfaces (APIs) supporting HPC I/O are implemented in layers of ever more sophisticated data modeling abstractions. These layers of data abstraction from what is commonly known as the HPC I/O “stack” illustrated in Figure 1.



Abstraction Levels	Abstraction Objects	Example Implementations
Real World Phenomena	Physics, Chemistry, Materials...	ALE3D, Albany, LAPS
Continuous Mathematics	PDEs, Fields, Topologies, Manifolds	MFEM, FiberBundles, Sheafs, SAF
Discrete Numerical Models	Meshes, Materials, Variables	Silo, LibMesh, Exodus, ITAPS, VTK
Prog. Language Constructs	Arrays, Structs, Lists, Trees	HDF5, netCDF, ArrayIO, PDB
Platform Primary Storage	Ints, Floats, Pointers, Offsets, Lengths	MPI-IO, XDR, stdio, aio, mmap
File System	Files, Dirs, Links, Permissions, Modes	POSIX IO
Secondary Storage (SW)	Pages, Inodes, FATs, OSTs, OSDs	PLFS, GPFS, HDFS; ext2/3, zfs, xfs, hfs, Lustre
Secondary Storage (HW)	Bits, Volumes, Sectors, Tracks	hd, sd, cd, dvd, tape, ramdisk, flash

Figure 1 Various abstraction levels of the HPC I/O stack supporting modern, multi-physics HPC applications. Included are levels not typically mentioned but are nonetheless here for completeness. Levels highlighted in green are the main focus of MACSio.

Each layer implements its abstractions in terms of the abstractions available in the layer below. At the top, applications simulate real-world phenomena by solving a collection of coupled, partial differential equations (PDEs). In a multi-physics simulation the phenomena of interest may include, for example, a combination of structural mechanics, electromagnetics, fluid dynamics, heat conduction, chemical kinetics and multi-phase materials.

The phenomena of the "real-world" are characterized by continuous mathematical equations (e.g. partial differential equations, fields, topologies, manifold spaces, etc.). These continuous equations are then expressed in terms of discrete numerical models (meshes, elements and variables, etc.) suitable for implementation as computational processes. In turn, these models are implemented in programming languages in terms of algorithms and data structures (arrays, structs, linked lists, trees, etc.). Ultimately, these data structures are composed of large volumes of numbers (ints, floats, pointers, offsets, lengths, etc.) occupying locations in primary memory. When an application needs to dump this data to secondary storage, it typically does so through a POSIX I/O interface (files, directories, links, permissions, etc.).

Pros and cons of software abstractions

As is so often the case in computer science, abstraction is a key to addressing complex software engineering challenges. The evolution of the HPC I/O stack is no exception. With abstraction in I/O interfaces has come many benefits; APIs better match application data and are easier to use, data is more shareable and, by extension, applications more interoperable, DIT services are more abundant and data management tools more sophisticated.

On the other hand, as a general rule of thumb, abstraction in computer science has historically also proven to be in constant conflict with performance. For example, the diagram in Figure 2 illustrates an important consequence of data abstraction layering in the IP Protocol Stack.

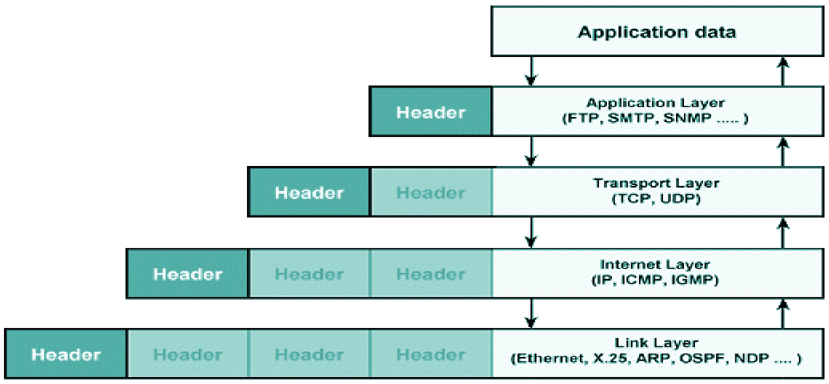


Figure 2 The IP Protocol Stack; remarkably similar in concept and purpose to the HPC I/O stack.

Each layer incurs some overhead in the form of additional data needed to implement a given layer's abstractions. At the bottom of the stack, the abstraction overhead, in terms of data packet size in the example of the IP Protocol stack, is perhaps as much as 100%. In I/O activities within software components of the HPC I/O stack, abstraction overhead can impact performance in sometimes unanticipated and often difficult to diagnose ways.

We aim to develop MACSio to operate at higher levels of abstraction than current tools. The "AC" in MACSio stands for Application-Centric. MACSio will be designed to generate I/O workloads that are representative of how real applications actually interact with secondary storage through the HPC I/O stack each layer of which has the potential to impact I/O behavior as data ultimately flows to/from disk.

I/O Performance Characteristics and Comparisons

Typical performance curves for several common layers of software in the HPC I/O stack are illustrated in the notional diagram in Figure 3. This diagram illustrates a number of performance characteristics typical of HPC I/O as a function of request size…

1. I/O performance can be no better than actual hardware disk rates.
2. I/O performance improves with increasing request sizes.

3. Performance curves for one layer can be no better than the layer below.
4. The difference in performance at any given request size can be interpreted as the overhead cost to implement/use the data abstractions that level offers.
5. As request size increases, abstraction overheads become less significant.
6. When significant overheads impact performance, this is often due to impedance mismatches between layers; either within the I/O stack of between the application and the top layer of the stack.

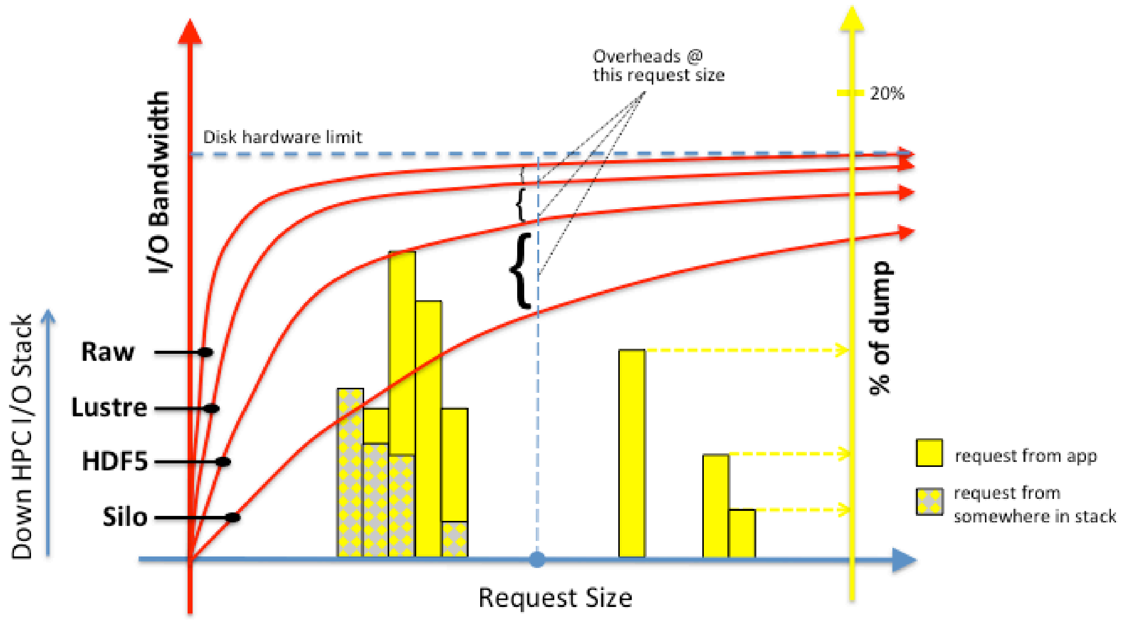


Figure 3 Notional I/O performance as a function of request size. This diagram illustrates a number of features of HPC I/O performance as a function of request sizes.

Figure 3 also illustrates a typical I/O request histogram for a representative application in yellow. The yellow bars in the figure represent various request size ranges. For a given range, the height of a bar indicates the percent of total bytes in a given (restart or plot) dump that were read or written at that request size range. In this example, a majority of bytes were read or written at relatively small request sizes where abstraction overheads in the performance curves are severe. The application will suffer serious I/O performance losses as a consequence. However, if the application can operate at large enough I/O request sizes that most of the performance overheads are amortized away, then performance will be good.

Often, the solution to improving performance is to find a way to *aggregate* large collections of smaller requests into a handful of larger requests. When such requests originate *within* the I/O stack, it is often possible to re-engineer one or more of the layers and better performance is achieved without any need to change the application(s). On the other hand, when these requests originate from the application itself, it is sometimes necessary to adjust the application to improve the way it utilizes one or more of the underlying layers.

This aspect of I/O performance is also illustrated in the figure with either solid yellow or pattern filled to indicate the two different kinds of I/O requests. Those that originate directly from the application itself are solid filled. Those that originate from another layer are pattern filled. The requests that originate from another layer are almost invariably a result of the additional metadata necessary for the layer to implement the data abstractions it presents.

In this example, a majority of the smaller I/O requests originate as metadata from one of the layers. However, there are also quite a few small requests that originate from the application itself.

This kind of detailed analysis and in particular getting a handle on where in the stack I/O requests originate is some of the key kind of performance data MACSio will produce enabling us to identify places where a given layer could be used in better ways as well as where certain application behaviors that are bad for performance could be identified and re-engineered.

Multi-Purpose I/O Workload Generation

Another key part of MACSio's design is that it is intended to provide a lot of flexibility and customizability in the way it can drive I/O activities. This flexibility is made possible by a few key design choices.

The first, already described above, is the LOA at which MACSio is designed to operate. The level of abstraction of MACSio's data generation will be such that it can represent the highest levels of abstraction at which real HPC applications express their I/O operations. This, alone, provides a significant amount of flexibility in the resulting I/O workloads that MACSio will be able to demonstrate.

Another key design feature of MACSio is the use of a dynamic, run-time plugin architecture. This design will define how high-level data generated within MACSio is delivered to I/O plugins and will enable plugin developers the greatest flexibility in deciding how best to handle data marshaled by MACSio.

Parallel I/O Paradigms

The data generated within MACSio will be suitable to drive a variety of parallel I/O paradigms within MPI and MPI+ parallel execution contexts. Those in current use and most common among HPC applications are illustrated in Figure 4.

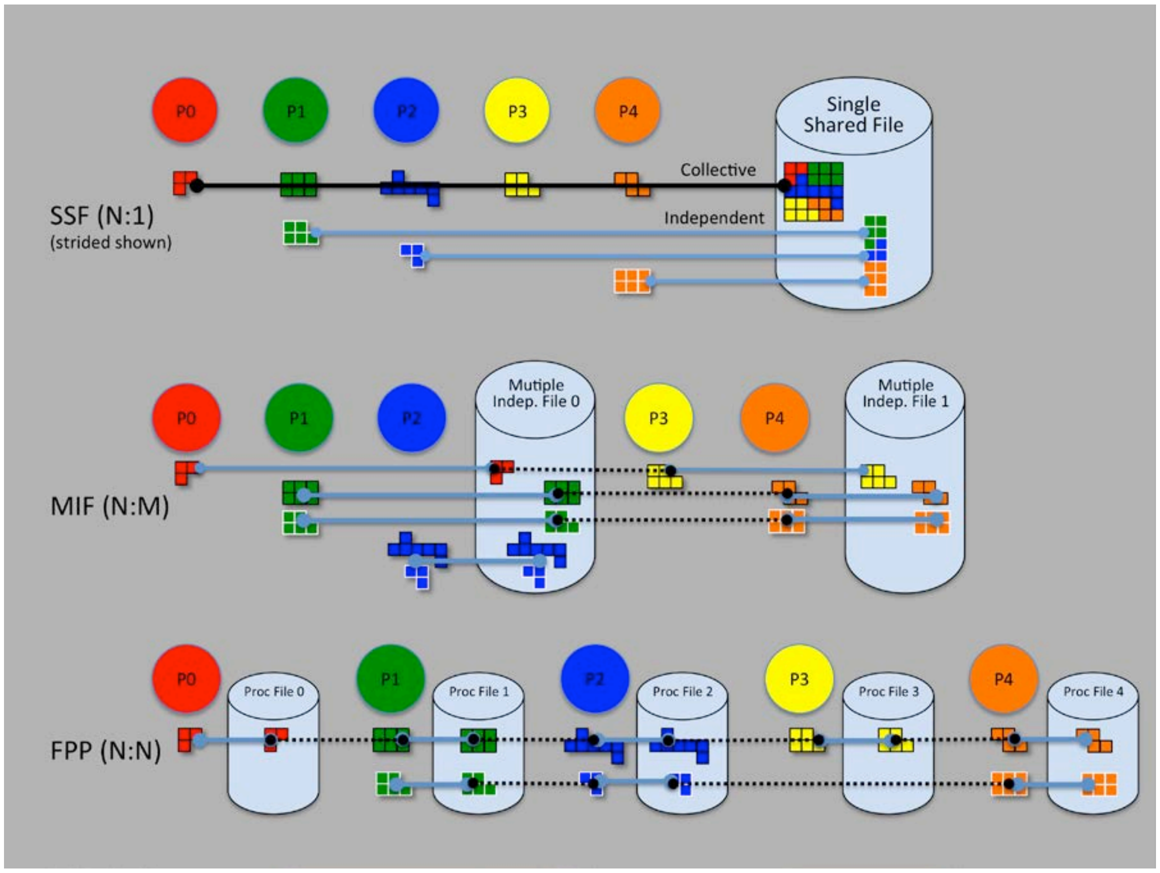


Figure 4 Common HPC Parallel I/O Paradigms. Black lines tie I/O activities that may proceed in parallel. Top: Single Shared File (SSF) both collective and independent, middle: Multiple, Independent File (MIF) and bottom: File-per-Processor (FPP). Note that these diagrams illustrate what the application sees looking down at the file system.

Single Shared File (SSF)

In the single, shared file (SSF) paradigm, parallelism is achieved through concurrent access to a single, shared file (from the perspective of the application). This paradigm is sometimes also called N->1 because it is N tasks writing to one, single file. In this paradigm I/O requests can be either independent or collective. This is illustrated by the two paths labeled collective and independent in the upper diagram of Figure 4. However, collective requests are seen as being somewhat unique to the SSF paradigm as well as potentially offering the greatest opportunity for high performance. On the other hand, there are some subtleties regarding what *collective* I/O operations truly mean in the SSF paradigm (or any other paradigm for that matter) that require further clarification in a later section of this document. SSF is sometimes referred to as *Rich Man's* parallel I/O because it utilizes (requires in fact) a parallel interface at the file system LOA (e.g. Lustre, GPFS, or PLFS).

Segmented and Strided SSF

In some descriptions, the SSF paradigm is further divided into *segmented* and *strided* access patterns. These access patterns have to do with the *granularity* at which data from different tasks intermingles in the file address space. In segmented SSF, large swaths of the file address space tend to be read/written by a single task. In strided SSF, data from many/all tasks tends to co-mingle even in very fine grained swaths.

Multiple Independent File (MIF)

In the multiple, independent file (MIF) paradigm, parallelism is achieved through simultaneous access to multiple files. The application divides itself into file groups. For each file group, the application manages exclusive access among all the tasks of the group. I/O is *serial* within groups but parallel *across* groups. The number of files (groups) is wholly independent from the number of processors and is often chosen to match the number of independent I/O pathways available in the hardware between the compute nodes and the file system. This paradigm is sometimes also called N->M because it is N tasks writing to M files (M<N).

In this paradigm I/O requests are almost exclusively independent. However, there are scenarios where collective I/O requests can be made to work and might even make sense in the MIF paradigm. MIF is often referred to as *Poor Man's* parallel I/O because the onus is on the application to manage the distribution of data across potentially many files. In truth, this illuminates the only salient distinction between SSF and MIF. In either paradigm, if you dig deep enough into the I/O stack, you soon discover that data is always being distributed across multiple files. The only difference is whether that physical arrangement of data is hidden from the application by some sort of higher level abstraction (e.g. a parallel file system) or explicitly managed by the application (and thereby also exposed to the file system).

File Per Processor (FPP)

The file per processor paradigm is just a special case of MIF where the number of files is equal to the number of processors. This paradigm is sometimes called N->N because it is N tasks writing to N files. However, FPP paradigms typically also include a throttle to govern the number of files that are being accessed at any one time to avoid overloading the underlying storage systems components.

Apart from SSF and MIF, there is indeed a multiple shared file paradigm as well. Although it is not often used in practice, ensuring MACSio has the ability to drive such a paradigm makes some sense. On the other hand, there isn't anything particularly special about a multiple shared file paradigm that involves additional work beyond that necessary to support SSF and MIF.

What Will Collective I/O Operations in MACSio mean?

Typically, the notion of collective applies to the interface through which data is marshaled. In a collective interface, all processors must call the same function with largely the same arguments though buffer contents and sizes may vary from processor to processor. Think of a parallel, distributed, 2 dimensional array as illustrated in Figure 5.

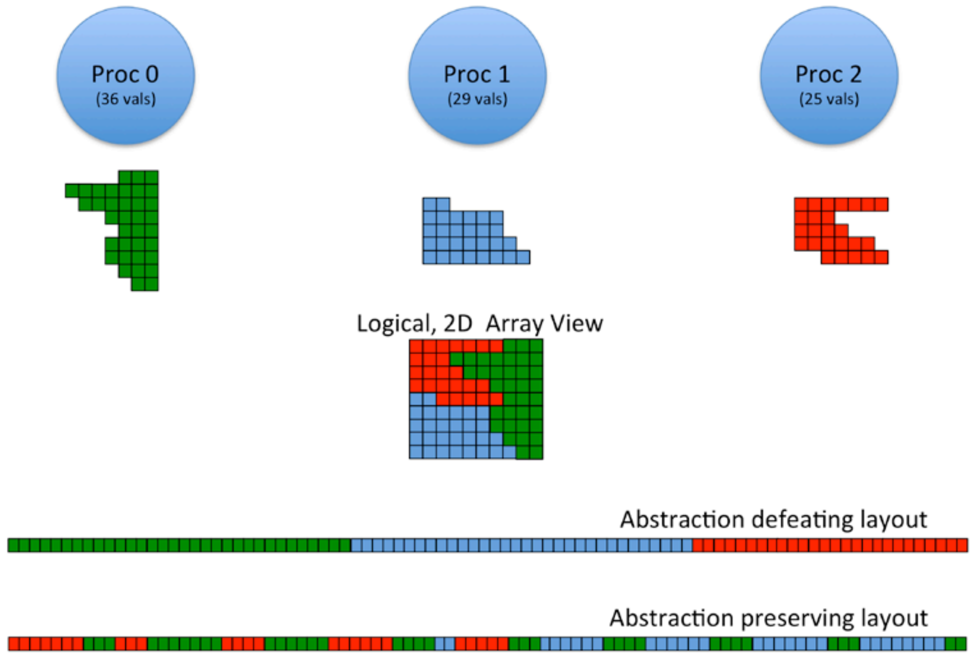


Figure 5 Examples of distributed, multi-dimensional array storage in memory and in a file.

Each processor may have a different, potentially non-contiguous, piece of the array. In a single collective call, all processors can work together to marshal it. However, collective-ness is a constraint only on the interface between the application and the rest of the I/O stack. What happens with the data below this interface either as it moves down the I/O stack and out to the file or up the I/O stack and into the application's memory? In particular, what is the assumed structure of the array in the file on read or that winds up getting created in the file on write?

In Figure 5, two common situations are illustrated. In each case, all processors are collectively working to either read or write the whole array object. However, in one case, the array object in the file is utilized as nothing more than a glorified buffer container. We call this an *abstraction defeating* approach and is a common pitfall in parallel I/O benchmarking tools. Any consumer would be unable to correctly interpret the array object without having additional information on how to properly knit it back together. Furthermore, even if the producer provides such additional information, all consumers would have to know such information exists, where to find it in the file and how to utilize it. This effectively burdens all consumers with the work of implementing the very data model abstractions the layers in the I/O stack are designed to provide.

On the other hand, an abstraction preserving approach removes all assumptions about parallel decomposition of data and facilitates re-decomposition into different parallel distributions upon re-read. In MACSio, valid collective I/O operations, which apply primarily to SSF paradigms, will require that the resulting data objects be *abstraction preserving*.

Metadata, Burst Buffers and DIT Operations

File system metadata operations (e.g. file/directory creation and deletion) often used in conjunction with these parallel I/O paradigms to manage directory trees (breadth and depth) and forests (for many timesteps) will be included in the workloads MACSio produces.

Likewise, data staging to/from burst buffers and/or migration to/from tertiary storage (via utilities such as SCR or HIO) will be included in MACSio's design.

In addition, MACSio will enable testing of various *I/O-relevant* Data-in-Transit (DIT) services. Over the past few decades in scientific computing, I/O libraries have evolved substantially from simply moving data between memory and persistent storage, to offering a variety of other useful capabilities to operate on data in transit. There are many DIT services available within various I/O libraries in the HPC I/O stack. These include such operations as

- Numeric format conversions
- Checksumming for data integrity
- Precision reduction (e.g. double to float)
- Lossless and lossy compression
- Units conversions
- Data subsetting
- Coordinate transformations
- Data re-ordering and layout optimization for down-stream post-processing
- M-N data shuffles for aggregating I/O requests across ranks
- In rare cases, even derived variable computation and sophisticated feature detection algorithms.

MACSio will ensure the data it generates to be marshaled by I/O libraries is of a sufficient character to drive a number of I/O relevant DIT services as well as provide a means for comparing performance/utility among them.

On the other hand, among all the things an I/O library can do, one thing it ought to do well is I/O. And, this is first and foremost what MACSio is being designed to test and measure; an I/O library's ability to do I/O apart from anything else the library may be capable of doing and, in particular, the library's ability to do I/O at scales and in configurations relevant to current and future generation systems and applications.

Existing I/O Benchmarking Tools

In this section we introduce and describe several existing I/O benchmarking tools available. We first describe some well known HPC specific benchmarks. Then, for completeness, we also list several non-HPC specific benchmark tools but without much discussion of their capabilities.

In Table 1 we summarize results from a review of a number of currently available I/O benchmarking tools.

Table 1 Feature Comparison of I/O Performance Tools

	Lang	LOA	Pmode	Verify	Coll	Real Data	Abs Keep	HPC relevant DIT Services				EZ Extend			Dir Ops	Perf DB	Last Active
								M/N	Zip	Sub	Trans	Pmode	Lib	MPI+			
IOR	C	File system	SSF, MIF	Yes	Yes	No	No	Yes	No	No	No	No	Yes		Yes	No	2013
FLASH-IO	Mix	AMR Mesh	SSF	No	No	No	Yes	No	No	No	No	No	No		No	No	2011
S3D-IO	Fort	3D Array	SSF	No	Yes	No	Yes	No	No	No	No	No	No		No	No	?
BTIO	Fort	3D Array	SSF	Yes	Yes	Yes	Yes	No	No	No	No	No	No		No	No	2003
GCRM-IO	C	AMR Mesh	SSF	No	Only	No	Yes	No	No	No	No	No	No		No	No	2013?
HACC-IO	C++	1D Array	SSF	No	No	No	No	No	No	No	No	Yes	Yes		No	No	2012
b_eff_io	C	File system	SSF, FPP	No	Yes	No	No	No	No	No	No	Yes	No		No	No	2001
MPI Tile IO	C	File system	SSF	No	Yes	No	No	Yes	No	No	No	No	No		No	No	2001
Non contig IO	C	File system	SSF	Yes	Yes	No	No	No	No	No	No	No	No		No	No	2005
fs_test	C	File system	SSF	Yes	Yes	No	No	No	No	No	NO	No	No		Yes	Yes	2008
Parkbench	Fort	File system	SSF	Some	Yes	No	Yes	No	No	No	Yes	No	No		No	No	1997
Mdtest	C	Fileystem	None	No	No	No	No	No	No	No	No	No	No		Yes	No	2003
Rompio	C	File system	SSF	Yes	Yes	No	No	No	No	No	No	No	No		No	No	2007
Tiobench	C	File system	None														2000

Legend:
Lang = programming language
LOA = Level of abstraction within I/O stack at which benchmark's I/O requests operate
Pmode = Parallel I/O paradigms/modes supported
Verify = Are I/O requests verified that expected data is read/written
Coll = Does the benchmark test/support collective I/O requests
Real Data = Is the data marshaled by the benchmark representative of a real application
Abs Keep = Are the I/O operations abstraction preserving
HPC Relevant DIT Services (Are the following Data In Transit services tested/supported…)
M/N = M<->N Data shuffles
Zip = Compression, including precision reductions
Sub = Data subsetting
Trans = Data re-orderings/transposes
EZ Extend (Does the implementation appear to support easily adding…)
Pmode = a new I/O paradigm/mode
Lib = a new I/O library
MPI+ = multi-core/multi-threading
Dir Ops = Are file system directory operations in concert with file I/O tested/supported
Perf DB = Are performance results captured in a curated/hosted database
Last Active = Guesstimate of last time any development work was performed

Detailed Discussion of some HPC Specific I/O Benchmarks

Here we briefly describe some of the existing HPC I/O benchmarking tools currently available. The information presented here is gleaned from reading and in some cases directly quoting documentation and where necessary actual source code.

fs_test (formerly MPI-IO Test)

fs_test was originally developed at LANL under the name MPI-IO Test. It is available on GitHub (https://github.com/fs-test) and was last updated May, 2014. Fs_test operates at the "File system" LOA (lowest level of the green levels highlighted in figure 1) It tests reads and writes, collective and independent, via PLFS, Posix and MPI-IO. Using terminology defined in the fs_test documentation, it also aims to test N-N, N-M and N-1 (strided and non-strided) DIT services. The shuffle patterns produced represent highly regularized, best case scenarios. So, the benchmark is apt to produce optimistic, upper bounds of I/O performance realizable by HPC multi-physics applications.

IOR

IOR stands for "Interleaved or Random". It is available on SourceForge (http://sourceforge.net/projects/ior-sio/) and was last updated April 2013. IOR operates at the "File system" LOA (lowest level of the green levels highlighted in figure 1). IOR is designed to test parallel file system performance by driving I/O requests, reads and writes, collective and independent through one of several available interfaces; MPI-IO, Posix (file-per-processor), PLFS, IOD*, DAOS*, HDF5 and netCDF (parallel). It supports SSF I/O (Single shared file) and there appears to be limited support for MIF I/O.

IOR employs an abstract interface for characterizing I/O operations and then includes a number of implementations of this interface for each of the aforementioned products. The IOD and DAOS products are internal interfaces to the Fast Forward I/O Stack. IOD is the "IO Dispatcher" and DAOS is the "Distributed Application Object Storage" layer interface. MPI-IO, Posix, PLFS, IOD, DAOS are file system interfaces while HDF5 and netCDF are high-level array-based I/O libraries.

Because IOR's abstract interface for characterizing I/O operations is designed more or less at the file system level of abstraction, it winds up using and driving HDF5 and netCDF in substantially restricted ways from how applications ordinarily use them. In short, IOR doesn't use HDF5 or netCDF interfaces at the same LOA multi-physics HPC applications typically would.

An important feature of IOR's design is that its very simple abstract interface makes it easy to plug in, statically, new I/O interfaces to it. Though, at present those interfaces are defined at compile time. There appears to be some work underway to add Amazon S3, HDF5 REST interfaces and mdtest (see below) functionality into IOR.

However, IOR cannot drive any of these interfaces at an LOA that is significantly different from the file system LOA. That alone makes IOR unsuitable for the kind of end-to-end performance analysis and comparisons we wish to achieve. IOR cannot test compresssion of multi-dimensional arrays. It cannot test M-N data shuffles of multi-dim arrays, traffic and performance of which is much more complicated than swaths of 1D buffers. For that matter, it cannot test the effects of different chunking and blocking factors of multi-dim arrays. In no way could it drive bi-modal interfaces like iMeshP, LibMesh or Vista. The workloads IOR generates are very regular, the same data in the same sizes from the same processors at the same intervals. It sort of represents the minimum you'd want from a benchmarking tool.

mdtest

mdtest is available on SourceForge (http://sourceforge.net/projects/mdtest/) and was last updated December, 2013. Like fs_test and IOR, mdtest operates at the "File system" LOA (lowest level of the green levels highlighted in Figure 1). However, unlike fs_test and IOR, mdtest is designed primarily to test a file system's performance in handling its own, internal metadata associated with directory tree creation and traversal, file and link creation and stat inquires. So, mdtest isn't designed so much to drive I/O requests (although it appears to have some ability to do that), as it is to drive primitive file system operations often used on conjunction with various parallel I/O paradigms where file and dir-tree creation and traversal are involved. It appears to drive both Posix and PLFS interfaces.

Darshan I/O Instrumentation Library

Darshan is not an I/O benchmark. It is an I/O instrumentation library available from ANL (http://git.mcs.anl.gov/radix/darshan.git/). It was last updated October, 2014. It enables developers to take any existing application and, with minimal effort, instrument it with Darshan to collect detailed timing information on Posix and MPI_File I/O operations during application runs. It does its work at the file system LOA (lowest level of the green levels highlighted in Figure 1), just as the other benchmarking tools do. But it makes it possible to collect detailed file system performance data from real applications in real use. It enables any application to be used more or less like a benchmarking tool.

Non HPC-specific I/O Benchmarks

In this section, we briefly describe some other non-HPC specific benchmarking tools available either commercially or as Open Source.

iozone

Iozone is useful for determining a broad file system analysis of a vendor's computer platform. The benchmark tests file I/O performance for the following operations. Read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read/write, pread/pwrite variants, aio_read, aio_write, mmap,

iometer
iometer is a Windows only too. It was formerly developed by Intel tool and then made Open Source. It tests and measures performance of hardware, buses, disks, latencies and bandwidths.

Fio
Fio is an I/O Aworkload generator and statistics gatherering and reporting tool.
http://www.storagereview.com/fio_flexible_i_o_tester_synthetic_benchmark
http://pkgs.repoforge.org/fio/

Filebench
Filebench is a file system and storage benchmark that can generate both micro and macro workloads. It employs a versatile language-based approach for workload specification. Filebench includes several popular macro-workloads in its distribution including those for Web-server, Mail-server, Database-server, and others.

Bonnie++
(http://www.googlux.com/bonnie.html) Bonnie++ is a benchmark tool for testing hard disks and file system performance.

TioTest
TioTest is a threaded I/O test of sec2/mmap interfaces.

sgpdd, odbfilter, ost survey
These are Lustre specific file system performance testing tools.

testdfs
testdfs is a tool for testing performance of the distributed file system of Hadoop (HDFS)

Some Observations
Benchmarks are challenging to develop. I/O benchmarks are no exception. A 2009 study entitled "*A Nine Year Study of File System and Storage Benchmarking*" (http://www.fsl.cs.sunysb.edu/project-fsbench.html) had this to say...

> *Benchmarking is critical when evaluating performance, but is especially difficult for file and storage systems. Complex interactions between I/O devices, caches, kernel daemons, and other OS components result in behavior that is rather difficult to analyze. Moreover, systems have different features and optimizations, so no single benchmark is always suitable. The large variety of workloads that these systems experience in the real world also add to this difficulty.*
>
> *We have found that some of the most commonly used benchmarks are flawed, and many research papers do not provide a clear enough*

> *picture of file system performance. We believe that a good performance evaluation should use micro-benchmarks to highlight both the good and bad qualities of a file system, as well as general-purpose benchmarks or traces to give an idea about how it would perform under expected and realistic workloads. Nevertheless, care should be taken to ensure that general-purpose benchmarks indeed accurately reflect the real-world workloads. In addition, benchmarks should scale well, and results should be reproducible and comparable across papers.*

Add to this the additional complexities resulting from parallelism, scalability and the HPC I/O stack and its pretty easy to reach the conclusion that scalable I/O benchmarks are intractable.

A common theme in these I/O benchmarking tools is proliferation of run-time parameters for a test either via command-line arguments or from configuration files. For example, in IOR many of the features of high-level products like HDF5 (e.g. compression, metadata caching) are not controllable without changing the IOR source code perhaps in ways that are incompatible with its overall design or not relevant to other I/O interfaces IOR tests. In MACSio, this issue is addressed by supporting run-time options that are passed directly to specific plugins.

Almost without exception, all of these I/O benchmarks operate at the lowest LOA, the file system interface. Even in the tools that support higher level libraries like HDF5, the overall design of the benchmark does not admit the use of the library at its natural LOA and instead tends to use the library in far too limited a way to be representative of real HPC applications. MACSio will address this issue by freeing plugin developers from having to worry about significant parts of the overall benchmark implementation and instead allow them to focus on optimal use of the library for a handful of specific I/O operations.

None of the tests currently available offer the ability to activate/drive other system components, in particular the parallel interconnect, while simultaneously attempting to perform I/O. However, additional loads on other system components has historically proven to be a significant factor in I/O performance realized by HPC applications in actual use. What this means is that existing benchmark tools are likely to provide a best case, upper bound on I/O performance and not necessarily a realistic estimate of performance HPC applications would realize. While MACSio's design will include accommodations for driving additional system loads (including casting internal I/O operations in terms of an asynchronous paradigm), the initial implementation of MACSio will also not include these features.

Libraries supporting the HPC I/O stack are complex and require a certain degree of expertise to use effectively. Developing a benchmark for any one I/O interface in the HPC I/O stack is a significant level of effort. Doing so for more than a handful of interfaces is almost intractable. As a result, the currently available I/O benchmarks either do a decent job for a single interface or a mediocre job for several. In the case of MACSio, a key intention of the design is that the actual I/O specific coding is handled in a product specific plugin which, we expect, product experts to help to develop. As part of MACSio's initial development, we expect to develop a few key plugins such as for Silo, HDF5 and maybe LibMesh or ITAPS. These plugins will serve as examples to other plugin developers. However, we will also recognize that development of plugins for different HPC I/O libraries will involve combined efforts of other HPC library experts.

MACSio Design

In this section, we describe MACSio's design at a high level illustrated in Figure 6.

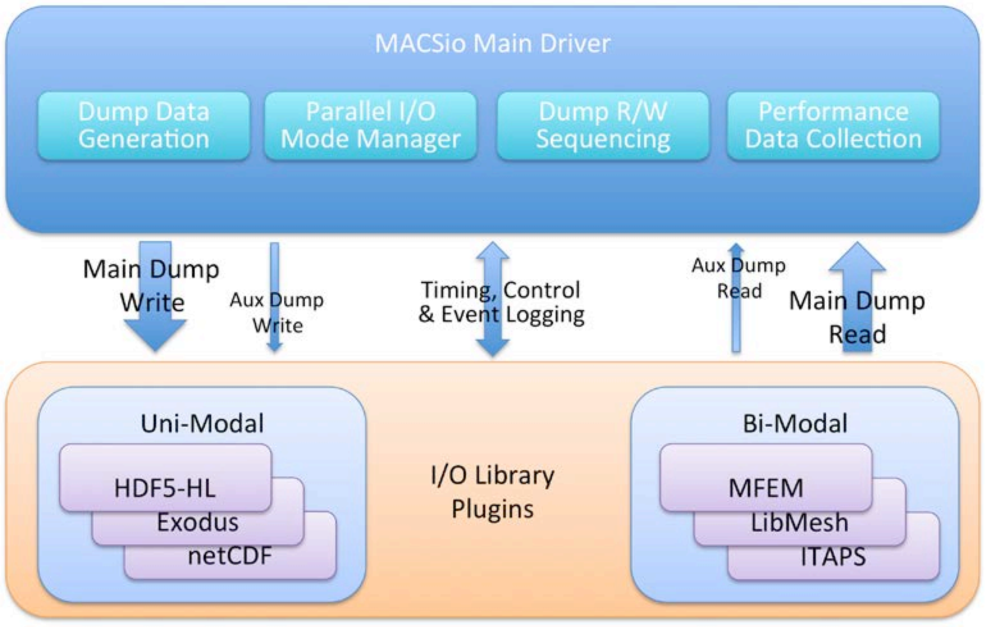


Figure 6 High Level Design of MACSio

The Two Main Pieces of MACSio

MACSio is divided into two main pieces. The core of MACSio is the upper-level driver. The MACSio driver generates data to be marshaled by I/O tests, orchestrates tests and monitors performance. The other half of MACSio is the I/O plugins. An I/O plugin implements a handful of interface methods to write/read dumps, interact with the file system and provide fine-grained performance and event logging information.

An I/O plugin in MACSio will manage dumps of data following one of the Parallel I/O paradigms; either receiving data from MACSio to write or returning data to MACSio to read. Two flavors of plugins will be supported. Uni-modal plugins support storage only as files in the file system. Bi-modal plugins support storage both as files in the file system and as an in-memory database for applications to use internally.

Data Generation

The main driver of MACSio is responsible for generating the data that will be marshaled for I/O performance testing. The data MACSio generates will be parameterized, distributed 3D meshes (structured or unstructured) and arrays. Parameterization will include

- Nominal and optionally randomization of mesh/array part counts per rank/core
- A few choices in how mesh/array parts are distributed in parallel including nicely structured distributions as well as fully unstructured distributions and distributions that utilize only a subset of all cores/ranks.
- Nominal and optionally randomization of mesh/array part sizes (in terms of numbers of zones of mesh or array entries)
- The number, type (char, int, float) and kind (nodal/zonal) of variables (mesh variables or arrays)
- Choice in algorithm used to fill variable buffers with data (e.g. constant, random/chaotic, sinusoidal, Poison distributed, etc.)
- Depth and breadth with optional randomization of auxiliary metadata hierarchies (such as might be seen in material models or various other rich metadata produced by multi-physics codes)
- Frequency of main mesh/array dump writes, reads or both as well as being optionally interleaved with auxiliary data dumps.

The data MACSio generates will be housed in an uber JSON-like object tree that is handed off to plugins for dump writes and received from plugins on dump reads. This tree will include information essential for plugins to determine parallel distribution of the main data objects.

Parallel I/O Mode Manager

The role of the Parallel I/O Mode manager in MACSio is to provide functionality to the I/O plugins necessary to support key parallel I/O modes; SSF, MIF, etc. Interaction with the underlying file system, including such operations as file and directory creation are important events that will be managed by MACSio on behalf of the plugins. When plugins need to perform these operations, they will do so by consulting functions in MACSio's Parallel I/O Mode Manager most likely through use of a call-back programming paradigm.

Though it is not pictured in Figure 6, MACSio will also include the ability to control various DIT services available in various layers of the I/O stack (and plugins) such as MPI-IO's ability to do M<->N data shuffles, or HDF5's or netCDF's ability to do compression, etc.

Dump Read/Write Sequencing

Dumps will occur at pseudo-regular intervals, with parameterized randomization. In addition, there will be support for at least two categories of dumps, the main dump plus a much smaller auxiliary dump (such as might be used for capturing time histories or marshaling a subset of the main dump to a linker code). The frequencies of each category of dump will be parameterized. In addition, while the main dump category will most likely involve new files (but not required) with each new dump, the auxiliary dump category need not and may just as well be handled by continuous updates to the same file(s) across many dumps.

Performance Data Collection

MACSio will capture, minimally, the time required for a plugin to complete the write or read of a dump and then average this over many sequences of dumps. In addition, plugins will optionally be able to use timing utilities within MACSio to record finer grained information regarding performance of individual parts of the dump process. This is primarily so that MACSio can be used to gather information on how to improve a given I/O library for a given use case MACSio drives.

MACSio will also include the ability to enable and disable Darshan I/O performance capture for any given test.

The data MACSio captures will be combined with all relevant test parameters, including information the test platform architecture to a curated/hosted performance database.

Diagnostic, Debug, Error and Logging

It is not pictured in Figure 6. However, MACSio will also include utilities for plugins (as well as MACSio proper) to log and handle error conditions and provide additional diagnostic information regarding test progress and behavior.