

# Chapter 30

---

## *Storage Models: Past, Present, and Future*

**Dries Kimpe and Robert Ross**

*Argonne National Laboratory*

30.1	The POSIX Era .....	335
30.2	The Current HPC Storage Model .....	336
30.2.1	The POSIX HPC I/O Extensions .....	337
30.2.2	MPI-IO .....	339
30.2.3	Object Storage Model .....	339
30.3	Post POSIX .....	340
30.3.1	Prior Work .....	340
30.3.2	Object Abstractions in HPC .....	341
30.3.3	Namespaces .....	343
30.4	Conclusion .....	343
Bibliography	.....	344

The storage component of HPC systems, like most components in these systems, is built of parts borrowed from other communities and markets. Early in the development of HPC systems it was recognized that a globally accessible storage system was desirable, and the HPC community converged on the use of the Portable Operating System Interface for UNIX (POSIX) I/O model as its *de facto* standard: a globally accessible directory tree holding files that each contain a stream of bytes of user data, with a strong consistency model enforcing immediate (global) visibility of updates.

As systems have grown in scale, supporting this model has become increasingly problematic, both in terms of performance and in terms of reliability. In this chapter we will discuss the POSIX model, how the HPC community has worked to adapt the POSIX model over time to meet its needs, and alternative models that are emerging from current research.

### 30.1 The POSIX Era

The POSIX I/O standard was developed by the IEEE and is currently defined as part of “IEEE Std 1003.1-1988” [13] (often referred to as “POSIX.1-1998.” There have been numerous updates and corrections since; the latest version (at the time of this publication) was IEEE Std 1003.1-2013 [1]. The first version of the POSIX standard was developed when a single computer operating system managed its own (local) file system, and issues of concurrent access were limited to the processes running on that operating system. Since all file accesses went through a single operating system on a single machine, enforcing strict consistency semantics was relatively easy. Likewise, data (and metadata such as current file size or last access time) could easily be cached, lowering the cost of accurately tracking last access, update time, and file size. This is reflected in the design of the API. For example, when retrieving the list of files in a directory, the `readdir` function only retrieves file names; To obtain extra information (such as file size), a call to `fstat` needs to be made for each file found. In a time when disks were local and uniquely accessed by a single computer and metadata could easily be cached, the cost of performing these extra calls was minimal. However, fast forwarding to modern times, where file systems are often remote (i.e., exported by file servers) and shared between multiple client computers, each call requires a round trip to a remote server. Likewise, caching is no longer straightforward as remote invalidation is required to keep cache contents consistent. In this environment, the cost of providing a single global, consistent view of the file system becomes exceedingly large.

Another problem with the POSIX model is that it forces a single, high-level data storage model for all applications, with associated costs, regardless of whether the semantics of the model are appropriate or not for the application at hand. For example, data can only be stored in a file. Each file has metadata such as file size and last access time, that are globally visible and consistent across all clients, whether an application requires this information or not. Likewise, each file needs to be in a directory. Creating a file in a directory is an atomic operation, with immediate global visibility. Because of this, file creation in a distributed file system can be highly synchronizing and consequently fundamentally unscalable.

Thus, while many distributed applications use more scalable methods internally for both I/O and data organization, POSIX offers no possibility of relaxing its strict rules, needlessly limiting application scalability. To mitigate this, numerous groups have developed additional layers that provide new organizational models and reorganize access prior to interacting with POSIX storage, many of which have been discussed previously in this book, and enhancements to POSIX to address scalability limitations have been proposed.

## 30.2 The Current HPC Storage Model

Considering the high cost (during development as well as at runtime) of implementing full POSIX I/O compliance in a distributed environment, it should not come as a surprise that some file systems instead aim to be *mostly* compliant. For example, the NFS client emulates the common unlink-after-open approach of creating temporary files by renaming the file with the goal of later removing those files (something which does not always succeed). Attribute consistency is another area where NFS is not fully compliant. In the default mode, client-side caching is used to improve performance and reduce server load. However, this means that full POSIX attribute consistency is not provided. In addition, there is no guaranteed, portable method to enforce consistency. Some of these issues are being addressed in newer versions (v4) of the NFS protocol.

In many cases, the POSIX semantics are unnecessarily strict, and consequently most applications continue to function correctly even on these mostly POSIX-compliant file systems. Thus the current HPC storage model aims to work around the POSIX shortcomings by adjusting or breaking POSIX where necessary and leveraging new software layers to optimize I/O before it hits the POSIX interface. Simultaneously, new storage concepts are being deployed *below* the POSIX API that have potential for larger benefits to HPC storage.

### 30.2.1 The POSIX HPC I/O Extensions

POSIX HPC I/O extensions [9] were designed to improve performance of POSIX I/O in large scale HPC environments. Software running on these systems differs from most other software in that on HPC systems, many processes, distributed over many nodes, work *collectively* on a problem. Specifically, focusing on I/O operations, this means many processes on many nodes are opening the same file(s) concurrently. Since HPC applications tend to be more synchronized as well, often all of the operations performed on these files (such as open) will be issued within a short time interval, leading to very high and bursty metadata workloads on the file system.

Since POSIX file handles are only valid on the local node, in a distributed environment—despite accessing the same file—each node is required to open the file. This causes each individual node to traverse the directory hierarchy to locate the requested file, causing high metadata overhead at the (remote) file system. The POSIX HPC extensions seek to reduce this load by allowing a single node to open the file and then *export* some representation of the resulting file handle (for example containing a direct pointer to the enclosing directory) to other nodes (`openg` function), which then convert the exported handle directly to a file handle (`sutoc` function) without having to perform a full `open` call.

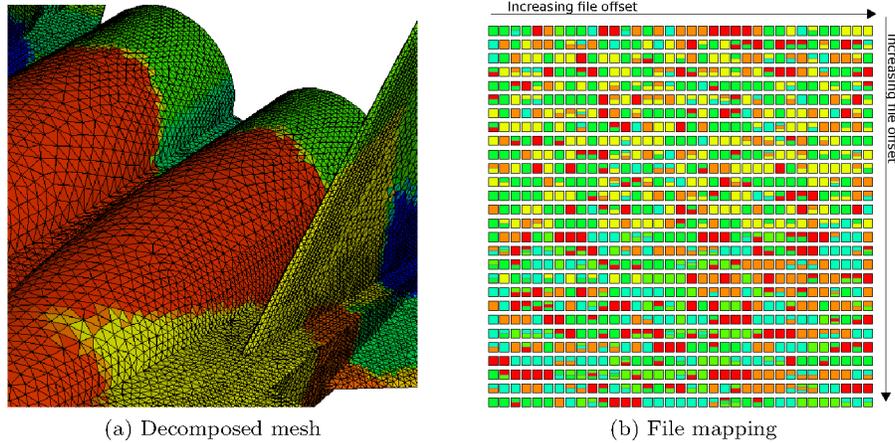


FIGURE 30.1: Domain decomposition resulting in highly non-contiguous access pattern. Color indicates node access.

HPC applications also tend to differ in how files are accessed. Often, a computational problem is partitioned between a set of nodes working on a single problem. For example, in the case of a weather simulation application, the three dimensional region for which the weather is computed is sliced into multiple pieces and distributed among the nodes. When a part of higher dimensional region gets mapped to a single dimensional file (when each node writes its portion of the simulated region to disk) a highly non-contiguous access pattern emerges. This is demonstrated in Figure 30.1, which shows a detailed 3D mesh (a), where color indicates the partitioning across nodes, and the resulting mapping to a one-dimensional file (b).

Unfortunately, the standard POSIX functions do not offer a way of expressing an I/O operation which is non-contiguous in the file (non-contiguous in memory is supported through the `readv` family of functions), forcing the application programmer to issue many small contiguous I/O requests instead, leading to severely reduced I/O performance. The `readx` and `writex` functions, proposed by the POSIX HPC extensions, aim to resolve this by offering functionality to describe an I/O transfer which can be non-contiguous in memory as well in the file.

The POSIX HPC I/O extensions also enable relaxing the rather strict POSIX I/O consistency semantics. For example, a new open flag is proposed, `O_LAZY`. The effect of this flag is to reduce the consistency scope of data and metadata to the scope of the process or local node, as opposed to all nodes mounting the same file system. When system wide consistency is required, an application can call `lazy_io_propagate` to propagate modifications to other processes or node, and `lazy_io_synchronize` to invalidate any locally cached data or metadata.

Other extensions include functionality to retrieve both directory contents and associated file metadata, as well as the ability to retrieve only a subset of the file metadata. The latter avoids forcing the file system to compute updated values for all file attributes, some of which (e.g., file size) can be very costly to obtain.

Adoption and support for the POSIX HPC extensions has been minimal. However, a number of calls similar to the `openg` and `sutoc` functions proposed in the POSIX I/O extensions have made it into the linux kernel: `name_to_handle_at` and `open_by_handle_at`. These functions were added in May 2011 in linux kernel version 2.6.39.

### 30.2.2 MPI-IO

The MPI standard is widely used to program HPC systems. In version 2 of this standard, support for file I/O (MPI-IO) was added. Being designed for distributed computing environments, the MPI-IO model provides a more suitable API for HPC applications. For example, non-contiguous accesses are directly supported (through the use of *MPI datatypes*), and default data consistency is more relaxed, and formally defined. MPI-IO also defines *collective* I/O operations, enabling many optimizations such as group open, which offers similar functionality to the `openg` POSIX HPC extension call.

However, since MPI-IO is frequently built on top of POSIX I/O functionality, many of the issues described in Section 30.1 resurface. Whenever possible, MPI-IO uses file system specific calls (for example through the use of `ioctl`) to work around some of the POSIX limitations, but these calls are non-standard and often vary even between version of the same file system. Widespread adoption of the POSIX HPC extensions would provide MPI-IO with a portable method to obtain the same effect. When specialized support is not available for the underlying file system, MPI-IO is limited to POSIX functionality, including the inability to communicate relaxed data consistency requirements. Consequently, MPI-IO performance is highly variable from system to system [12], causing some application and library developers to avoid MPI-IO all together.

### 30.2.3 Object Storage Model

In 1998, Gibson et al. [10] proposed a new storage interface, called Network Attached Secure Disk (NASD), proposing to replace the low-level sector access by an object based access API allowing a client to access an object (typically by numerical identifier) and offset within that object, while the device itself controls the mapping of the object data to the disk platters. The NASD work influenced the design of the ANSI T10 Object-based Storage Device (OSD) standard. This work does not aim to replace the POSIX-I/O functionality; instead it is intended to offer a basic building block on which other, high-level I/O functionality can be built. In a sense, these form a replacement for

plain disks. OSD differ in that they typically offer byte access granularity (as opposed to sector granularity). The OSD implements the storage operations (data transfer and layout), without defining any policy.

Many parallel file system implementations today, while exporting a mostly POSIX compliant model to their clients, internally access local storage using an object storage model similar to the T10 standard [6, 7, 19, 18]. Unfortunately, as seen with MPI-IO, these object storage abstractions are frequently built on top of an existing local POSIX file system. This means potential efficiency gains (for example in avoiding directory overhead) are often not fully realized. At least one reason for this insistence on building on top of POSIX is that no other data storage model comes close in terms of availability and portability.

Luckily, some alternatives are now starting to appear. Seagate, a well known American data storage company, recently commercialized a disk drive which exposes storage only as object storage API, as opposed to exposing low-level sectors, as is still common for disk drives. A widely adopted standardized object based access method for low-level storage could finally provide libraries and applications an equivalent solution providing the portability of POSIX with the flexibility to define their own access semantics (consistency) and grouping structures (directories or alternatives).

---

### 30.3 Post POSIX

Currently, the POSIX model for storage still dominates in HPC. However, there is increasing use of libraries such as PnetCDF and HDF5 that provide alternative data models to users. These libraries create an opportunity for storage system designers: new underlying storage models can be deployed by mapping these libraries directly onto the new storage model, avoiding the need to support the POSIX model at all.

Thus, the HPC community may be at the cusp of a “post POSIX” era where new HPC storage models appear in production systems. When considering what the storage model(s) in this era might look like, two needs are evident. First, highly parallel applications (and the libraries that support them) need to store multiple, concurrent streams of data and organize these conveniently. Second, with the explosion of data that is occurring, new methods for identifying data of interest are increasingly important.

#### 30.3.1 Prior Work

Of course, while production HPC systems have primarily provided POSIX storage access, software products outside of HPC and research in the HPC

community have pointed to a number of viable alternative models. This section will discuss a few, more relevant examples.

The IBM Virtual Storage Access Method (VSAM) model [15], defined in the 1970s, provides a number of features that would be compelling in an HPC system. Data is stored as records of potentially variable length with multiple fields. Data items can be referenced with a key, with a record number, or the file can be directly accessed with byte offsets. Multiple dataset organizations are provided to cater to specific use cases.

While most users do not realize it, the Microsoft New Technology File System (NTFS) also provides an interesting alternative model in the form of *alternative data streams* [5]. This functionality allows for multiple streams of data to be associated with the same file name. A default data stream holds standard “POSIX-style” data, while a colon notation is used to define and access additional named streams under the same file name.

This model of multiple streams associated with a single file name is not unique to NTFS, and in fact the approach has appeared in HPC parallel file systems research as well. The Galley parallel file system [16], developed in the 1990s, supported a concept of *subfiles*. In their model, a set of subfiles were created at the time a file was created that mapped to underlying storage devices. These subfiles then contained a set of *forks* that each could hold an array of bytes (like a normal POSIX file). The authors showed how upper software layers could map astronomical data into this organization.

The Vesta parallel file system [8] was developed at IBM in the 1990s specifically for HPC. Vesta exposes a 2D structure for files, with *physical partitions* holding sequences of records. Physical partitions are similar to subfiles in the Galley model and are meant to map to storage nodes. This provides a notion of parallelism of access that has been adopted by current research in the area.

### 30.3.2 Object Abstractions in HPC

Work in object based file systems set the stage for one possible alternative: providing direct access to storage objects. Researchers are investigating how to expose an object abstraction while maintaining the existing name space abstraction. In this model, a directory entry refers to a collection of objects, each individually accessible.

The “End of Files” (EOF) [11] project is one such example. Goodell et al. developed a prototype atop PVFS [7] that allows for a static set of objects to be associated with a directory entry. Conceptually this is best thought of as the file system no longer owning the distribution of data into objects, but rather delegating this to higher level software layers. This approach exposes the natural unit of concurrency (i.e., the object) and provides multiple data streams that may be used by upper layers for organizational purposes.

Figure 30.2 shows how the PnetCDF (Chapter 15) library maps netCDF datasets to a POSIX file (left) or to the EOF object model (right). In the POSIX file mapping, PnetCDF lays out variables across the single file byte

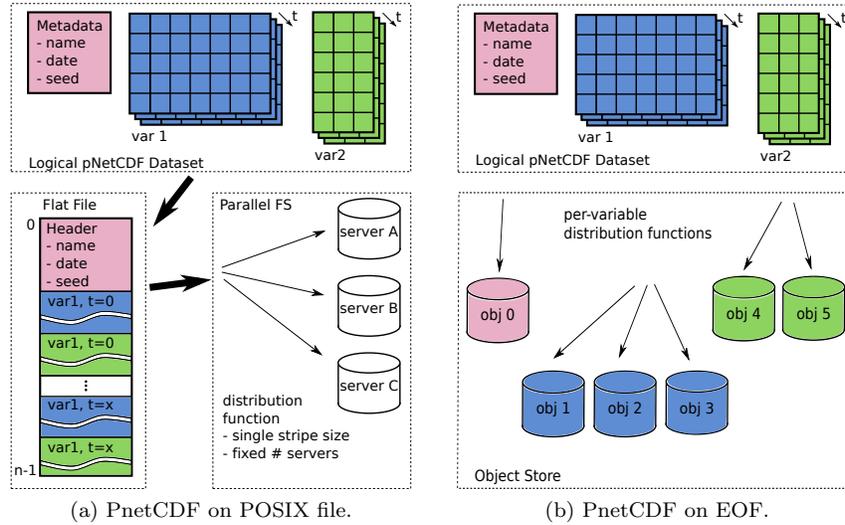


FIGURE 30.2: Mapping of PnetCDF dataset to (a) POSIX file, or (b) EOF objects.

stream. One complication of this process is *record variables*, which have one dimension of undefined length. In the netCDF standard,  $n - 1$  dimension subsets of these variables are interleaved at the end of the file in order to reduce the overall file size. This organizational choice is problematic for high-performance access, as it leads to noncontiguous I/O.

In the EOF object model, distinct objects are used to hold metadata and the data corresponding to variables. The PnetCDF library becomes responsible for mapping multidimensional variables into one or more objects. This simplifies data layout, especially for record variables that can now be mapped into distinct object sets without need for interleaving.

The Intel Storage and I/O Fast Forward project [2, 4] is similarly looking at how high-level data models can be mapped onto exascale storage architectures. The Intel activity includes three sub-projects: looking at how HDF5 (Chapter 16) can target object storage backends; examining how PLFS (Chapter 14) and I/O Forwarding Software Library (IOFSL) [3] can be adapted to manage burst buffers to optimize data layout on object storage backends; and developing a prototype Distributed Application Object Storage (DAOS) system, building on Lustre (Chapter 8) including a distributed transaction capability.

The transaction capability is one unique feature of the Intel prototype I/O stack and is built on top of a versioning capability at the DAOS level. This capability allows for transactions with Atomicity, Consistency, Isolation, Durability (ACID) properties when data moves from burst buffer to persistent

storage. Transactions are similarly supported at the burst buffer layer, and the augmented HDF5 library takes advantage of these to ensure consistent transitions of a given dataset.

Another interesting feature of the Intel Fast Forward work is the manner in which burst buffers are integrated into the overall system. I/O nodes that manage local NVRAM export three distinct data models for use by application libraries: a blob object (i.e., a byte stream), a keyword-value object, and an array object. All are used within HDF5, and the I/O node is responsible for mapping transactions on these constructs into operations on DAOS objects. This issue of translations between abstractions at various levels of the I/O stack is critical, and often overlooked.

### 30.3.3 Namespaces

Despite hierarchical file systems being declared as dead [17], this model of data organization still dominates in HPC. While both the EOF and Intel Fast Forward projects provide alternative organizations for data, neither makes radical changes to the way we organize and reference named entities.

What seems most likely is that the role of providing namespaces for HPC applications will be taken over by upper layers of the HPC storage stack, layers such as HDF5, PnetCDF, and others, with storage systems instead providing a more rich building block abstraction on which these namespaces are constructed.

What is even less clear is how search-like capabilities will be integrated into the HPC storage stack. Activities such as hFAD [17] and Spyglass [14] have provided some insight, but ultimately HPC search is about application data structures – not something that the low level storage system is necessarily even aware of. Perhaps active storage advances will be what ultimately enable application-oriented searches to be executed local to storage where they can be performed efficiently.

---

## 30.4 Conclusion

The POSIX storage model has been a cornerstone of HPC systems for decades. As a result of its success, HPC system providers have been slow to adopt alternatives to the well-understood POSIX file model, but the increasing use of I/O libraries such as HDF5 and PnetCDF by scientific codes have eliminated many long-standing dependencies on the antiquated POSIX model and provided new ways for science teams to interact with storage.

Research has identified a number of promising alternative models, and the community seems ready to embrace a new model. New additions in the storage hierarchy, such as the inclusion of nonvolatile memory in the storage

system, have further disrupted the status quo. With current storage systems struggling to keep pace, the timing could not be better.

---

## Bibliography

- [1] IEEE Std 1003.1, 2013 Edition. [http://www.unix.org/version4/ieee\\_std.html](http://www.unix.org/version4/ieee_std.html).
- [2] Intel Fast Forward Storage and I/O Program Documents. Technical report, Intel, 2013.
- [3] Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, and Robert Ross. Scalable I/O Forwarding Framework for High-Performance Computing Systems. In *IEEE International Conference on Cluster Computing (Cluster 2009)*, New Orleans, LA, September 2009.
- [4] Eric Barton. Fast Forward I/O and Storage. Keynote at the 7th Parallel Data Storage Workshop (PDSW), November 2012.
- [5] Hal Berghel and Natasa Brajkovska. Wading into alternate data streams. *Commun. ACM*, 47(4):21–27, April 2004.
- [6] Peter J. Braam. The Lustre Storage Architecture. Technical report, Cluster File Systems, Inc., 2003.
- [7] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.
- [8] Peter F. Corbett and Dror G. Feitelson. The Vesta Parallel File System. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.
- [9] Marti Bancroft DOD, NRO John Bent DOE, NNSA LANL, Gary Grider DOE, James Nunez DOE, Ellen Salmon NASA, Lee Ward DOE, and NNSA SNL. High End Computing Interagency Working Group (HECIWG) Sponsored File Systems and I/O Workshop HEC FSIO 2011.
- [10] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *SIGPLAN Not.*, 33(11):92–103, October 1998.
- [11] David Goodell, Seong Jo Kim, Robert Latham, Mahmut Kandemir, and Robert Ross. An Evolutionary Path to Object Storage Access. In *Proceedings of the 7th Parallel Data Storage Workshop*, Salt Lake City, UT, November 2012.

- [12] William D Gropp, Dries Kimpe, Robert Ross, Rajeev Thakur, and Jesper Larsson Träff. Self-Consistent MPI-IO Performance Requirements and Expectations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 167–176. Springer, 2008.
- [13] IEEE. *2004 (ISO/IEC). IEEE/ANSI Std 1003.1, 2004 Edition. Information Technology—Portable Operating System Interface (POSIX®)—Part 1: System Application: Program Interface (API) C Language*. IEEE, New York, NY USA, 2004.
- [14] Andrew W Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L Miller. Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems. *FAST*, 9:153–166, 2009.
- [15] Dave Lovelace, Rama Ayyar, Alvaro Sala, and Valeria Sokal. VSAM Demystified (Second Edition). Technical Report SG24-6105-01, IBM International Technical Support Organization, September 2003.
- [16] Nils Nieuwejaar and David Kotz. The Galley Parallel File System. *Parallel Computing*, 23(4):447–476, June 1997.
- [17] Margo I Seltzer and Nicholas Murphy. Hierarchical file systems are dead. In *HotOS*, 2009.
- [18] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 307–320, 2006.
- [19] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 17–33, 2008.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. Please make sure you have the DOE acknowledgment at the end of your paper (before the References).

For acknowledgment information, please use:

This work was supported by the U.S. Department of Energy, Office of Science, under Contract DE-AC02-06CH11357.