Chapter 2
Parallel I/O Basics

Robert Latham and Robert Ross

As Earth systems simulations grow in sophistication and complexity, developers need to be concerned not only about computational constraints but also about storage resources. When dealing with the large data sets produced by high-resolution simulations, the storage subsystem must have both the capacity to store the data and the capability to access that data efficiently. The computing facilities of today and tomorrow will provide increasing computational power, but storage capabilities are not increasing at a corresponding rate. One challenge for all applications will be how to best manage and mitigate the growing input/output (I/O) bottleneck.

By understanding the structure of high-performance storage systems, scientists can better direct their efforts as they strive to get optimum performance out of storage. High-performance storage systems rely on an entire stack of software tools and libraries (Fig. 2.1). This chapter provides an overview of the software stack, discusses tuning strategies for applications, and briefly covers some best practices for high-performance parallel I/O.

2.1 The I/O Software Stack

Each year, high-end computing systems offer ever more powerful hardware. Computational scientists in turn refine software models of physical processes to take advantage of faster machines, doing so with the help of libraries optimized for specific platforms. Similarly, computational scientists can rely on I/O abstraction libraries to both hide platform-specific aspects and deliver higher performance.This organization of I/O software is commonly referred to as the *I/O software stack*. Using

R. Latham · R. Ross
Argonne National Laboratory, Lemont, USA
e-mail: robl@mcs.anl.gov

R. Ross
e-mail: rross@mcs.anl.gov

V. Balaji et al., *Earth System Modelling – Volume 4*, SpringerBriefs in Earth System Sciences, DOI: 10.1007/978-3-642-36464-8_2, © The Author(s) 2013

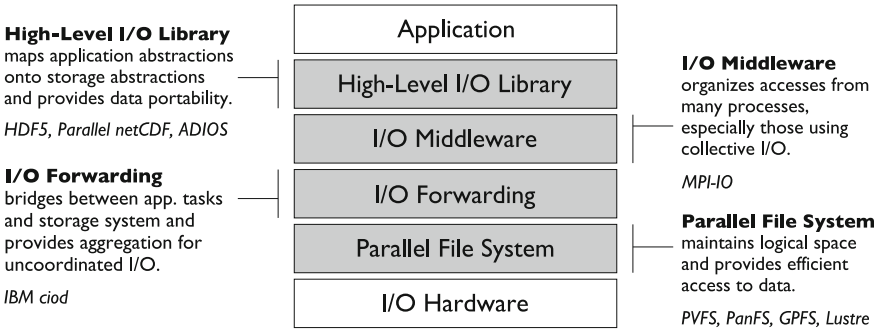

Fig. 2.1 I/O software stack. The higher levels are tailored for applications while the lower levels combine and maintain multiple channels to storage for fast parallel access

this I/O software stack, an application scientist can access I/O resources through interfaces better suited to a particular domain, while lower levels of the I/O software stack carry out optimizations and deal with storage-specific matters behind the scenes. Although application scientists do not directly interact with these lower, more device-oriented layers, a deeper understanding of these libraries can be helpful when the time comes to tune an application's I/O, and in fact should drive code design for I/O-heavy applications.

At the highest level, a computational model operates on structured data, usually a multidimensional grid of latitude, longitude, altitude, and quantities such as temperature or pressure, or possibly an even more elaborate structure. This application model fits the science and has very little, if any, relationship to the way storage is organized.

Every application will at some point need to carry out I/O, either to read in a data set or to write out a checkpoint or intermediate result. Ideally it would do so while maintaining the data structures used for the science model. Further, scientists need to collaborate with other groups or run on different computer systems. Data format libraries (often referred to as high-level I/O libraries) solve both these issues. We will cover these libraries in more detail in Sect. 2.4.

Isolating data format libraries from the file system, the I/O middleware (e.g., MPI-IO, the I/O layer of MPI-2, The MPI Forum 1997) component introduces several approaches for achieving performance portability. Collective I/O and MPI datatypes can help extract maximum performance from the underlying file system. Application scientists, however, often find these constructs a poor fit for their scientific models. Sect. 2.3 provides greater detail about the MPI-IO optimizations and how the libraries built on top of MPI-IO present these features to scientists.

Parallel file systems (Sect. 2.2) are currently one of the base technologies of scientific data storage. Parallel file systems aggregate multiple storage devices into a single, unified, high-performance storage space. The file systems have a linear data layout, in contrast to the structured data of scientific applications. Parallel file systems do not have the concept of collective I/O, though some might have data access

routines capable of expressing fairly sophisticated data access methods. While several abstraction layers insulate the application author from any specific parallel file system, knowledge about the underlying file system can still give a clue as to which access patterns are more likely to perform well.

Having these multiple layers of abstraction between scientific applications and the storage subsystem can complicate matters, but each layer provides key features needed to achieve the highest I/O performance. Parallel file systems provide a general-purpose interface in addition to managing storage devices, allowing for a broad array of uses. Middleware libraries can target multiple file systems but still need a Application Programming Interface (API) broad enough to apply across a variety of science domains. Data format libraries can dictate programing structures and file formats tailored for their end users; and because multiple data format libraries can target a single middleware library, each application domain can have a data format library that best meets its needs.

2.2 Parallel File Systems

Many different file systems provide shared access from a collection of clients, spanning the range from centralized to completely multiplexed. One way to categorize these is by the degree to which they enable efficient concurrent access. At one end of the spectrum are parallel file systems (PFS), designed for concurrent access from hundreds or thousands of processors to a single file. At the other end of the spectrum, a distributed file system like NFS[1] is more of a connectivity solution, providing access from many clients, but not performing well when many accesses take place concurrently. Between these two are what are sometimes termed cluster file systems, such as CXFS (Shepard and Eppe 2006). These file systems provide data distribution that enables some level of concurrent access, but they are not architected to support the high degree of concurrency seen in high-performance computing (HPC) workloads.

In many ways a parallel file system acts just like any other file system. The same tools to create files, make directories, and read and write data work on both serial and parallel file systems. This intentional similarity allows legacy applications and familiar tools to seamlessly operate alongside high-performance applications.

The biggest contribution a parallel file system makes to the software stack is its aggregation of multiple network links, servers, and disks into a global namespace. This aggregation provides a convenient logical "unit" that applications can interact with. Figure 2.2 depicts how a parallel file system might distribute a large file across the servers to enable concurrent access. Clients can communicate concurrently with the servers when they read or write different portions of a file. Thus they are capable of achieving better performance than a single server could provide, because the load

[1] Actually NFS isn't a file system at all, but rather a file system *protocol*; but for our purposes in this section it is easiest to just think of it as a file system.

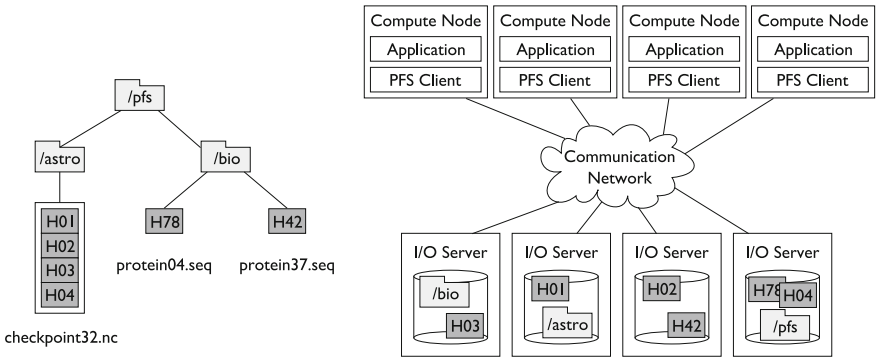

Fig. 2.2 Major components of a parallel file system: hierarchical namespace for stored files, simultaneous access by many clients, shared interconnect, and multiple server nodes managing persistent storage. Small files may be placed entirely on one server, while large files are broken up and scattered across multiple servers

is distributed across many servers, disks, and network links. The major parallel file systems today are Lustre (Braam 2003), GPFS (Schmuck and Haskin 2002), PVFS (PVFS development team 2008), and PanFS (Nagle et al. 2004).

Parallel file systems manage the data on disks. Two main designs exist: block-based and object-based. Block-based parallel file systems have their roots in older storage system designs, with GPFS being a current example of this design. Data on disk is globally allocated and managed in blocks, often many kilobytes or megabytes in size. Should an I/O request only partially fill a data block, the file system still operates on the entire block. For reads, this poses little problem: the software can merely discard any unnecessary data. Writes become trickier: the file system must carry out a *read-modify-write* sequence. First the block is read into memory, then the modifications are made, and finally the buffer is written out to disk.

In parallel applications, care must be taken to prevent *false sharing* during writes, when two or more processes want to modify different regions of the same data block at the same time. Locking mechanisms are deployed to control such problems, but at a cost of serializing what could have otherwise been a parallel data access. Small I/O requests are never good for a parallel file system, but users of block-based file systems would be well advised to align accesses to block boundaries (the other layers in the software stack often do so automatically) and to read and write in multiples of the block size.

More recently, parallel file systems have adopted an object-based storage model (e.g., PVFS, Lustre, PanFS). In this scheme, clients operate not on blocks but on higher-level abstractions called *objects*. This object abstraction imposes some structure on the data in the storage devices and gives the storage servers more intelligence about how clients are accessing data. Additionally, this approach allows storage servers to locally manage allocation of space on disks, distributing the overhead of this process. The object-based storage model, with a more sophisticated client-server

API and more flexible servers, is an important step toward addressing the problems of false sharing and read-modify-write overhead. While the underlying storage model might still be block-based, the object layers provide scope for optimizing low-level data access.

2.3 MPI-IO

In the early days of supercomputers, each vendor had its own communication library, and moving a code to a different supercomputer meant rewriting a significant portion of that code. The MPI Forum came together as a response to this byzantine landscape and proposed a single, standard message passing interface. Each supercomputer vendor implemented MPI, and as a result programmers could go back to writing scientific codes instead of worrying about machine-specific communication interfaces.

The MPI Forum reconvened a few years later to round out the MPI standard with additional features (The MPI Forum 1997), including a section on I/O, often called MPI-IO. Just as MPI provides an abstraction for communication networks, MPI-IO provides a common interface to both parallel and legacy file systems. Today, nearly every MPI library also contains an implementation of MPI-IO.

The MPI-IO API feels a lot like message passing, but for files. Sending a message becomes a write to a file (likewise for receiving/reading). MPI-IO brings two important concepts from MPI to the I/O software stack: noncontiguous and collective access. These features allow for a wide range of optimizations in the MPI-IO library, such as two-phase I/O (Thakur et al. 1999) or data shipping (Prost et al. 2001), that can significantly improve access rates in scientific codes. Depending on the level of file system support, a sophisticated MPI-IO implementation can turn what would normally be a poorly performing workload into something close to the ideal (but uncommon in scientific applications) large-block sequential I/O (Ching et al. 2002, 2003a, b).

File system APIs typically offer only a linear view of data: all data fits into one dimension, from byte 0 at the beginning of the file up to the last byte of the file. Scientific applications, however, have more sophisticated data structures. Even something as simple as reading a column out of a row-major array turns a logically contiguous access into one that is scattered across the file. This noncontiguous I/O access is fairly common in scientific applications.

In the same way that programmers use MPI datatypes to describe communication patterns, programmers can also use MPI datatypes in MPI-IO to describe noncontiguous data in memory and define a file view with these datatypes to describe the layout of data in the file.

Even if the file system does not support such a rich data access API, the MPI library can still optimize these noncontiguous accesses. With data sieving (Thakur et al. 1999), a single, large I/O operation replaces several smaller ones. Like a sieve, unneeded data is discarded. This optimization works well if the "density" of the

request is high—if there is not much wasted data. In the write case, just like a block-oriented file system, the programmer must be vigilant to prevent false sharing during the read-modify-write phases, ensuring no concurrent operations within a block boundary, if high performance is the goal.

The MPI-IO layer introduces another important optimization: collective I/O. A parallel program often has distinct phases where all processes are involved in an I/O call, such as when writing out a checkpoint file or reading in a data set. If all processes access the file system independently, this storm of uncoordinated requests can be hard to service efficiently. If all processes instead carry out I/O in concert, the MPI-IO collective I/O routines can aggressively optimize the access pattern.

Collective I/O yields four key benefits. First, optimizations such as data shipping and two-phase I/O rearrange the access pattern to be more friendly to the underlying file system. Second, if processes have overlapping I/O requests, the library can eliminate duplicate I/O work. Third, by coalescing multiple regions, the density of the I/O request increases, making the two-phase I/O optimization more efficient. Fourth, and perhaps most important for scaling to the largest supercomputers, the I/O request can also be "aggregated" down to a number of nodes more suited to the underlying file system. These optimizations make some form of collective I/O almost compulsory for any application requiring high I/O performance at large scale (Yu et al. 2006). Examples of some of these techniques are to be found below in Chap. 3.

In order to tune behavior of these optimizations, MPI-IO provides a way for applications to pass parameters down to the library. These key-value string pairs, called hints, provide a way for developers to inform the MPI-IO library about application-specific workloads. For example, one could forcibly disable the data sieving optimization if the access pattern was going to be very sparse, or tune the intermediate buffer to be exactly the same size as a record of interest. Gropp et al. (1999) provides further information about MPI-IO hints.

With data types, file views, and hint parameters, one might consider the MPI-IO interface overwhelming for a scientific programmer. MPI-IO makes key optimizations *possible*, but it does require understanding a fairly complex API. Fortunately, scientists have access to several libraries built on top of MPI-IO to bridge this "usability gap" by hiding the details of using MPI-IO and presenting an interface more suited for applications.

2.4 Data Format Libraries

While applications have a computational model that operates on physical elements, PFS and MPI-IO interfaces work in terms of bytes in files. Converting between these two models can be a complex and tedious process. Furthermore, collaboration with other scientists means data exchange and can be facilitated by the use of a common data representation. Data format libraries (often called high-level I/O libraries because they are built on top of MPI-IO) address both of these issues.

Data format libraries bridge the gap between scientific applications and MPI-IO or lower levels of the I/O software. These libraries present an interface based on multidimensional arrays of typed variables. For example, when modeling atmospheric phenomena, a multidimensional array holds values for latitude, longitude, altitude, and some attribute like temperature or barometric pressure. Multidimensional arrays are intended for structured data, where contiguous array indices indicate nearest-neighbour relationships; these assumptions don't map perfectly to some approaches such as unstructured grids or adaptive meshes, but they do match well to common models and provide a good building block for more sophisticated abstractions.

Metadata—information about data such as timestamps, provenance, and workflow processes—can be as important as the actual data. High level libraries provide methods to annotate variables and entire data sets with attributes. With these annotations in place, collaborators (both now and in the future) have more context in which to understand the data.

Parallel NetCDF (Li et al. 2003) and HDF5 (The HDF Group 2008) are the commonly used higher-level parallel I/O libraries today. Both share many of the same features. Consider the NetCDF example in Fig. 2.3. The dataset contains multidimensional, typed variables. Both the variables and the dataset itself can have additional attributes. The resulting file, thanks to a well-defined file format, contains all the information needed to read the data on any platform. Recently, members of the serial NetCDF project introduced a third parallel I/O library. They implemented the serial NetCDF API using the HDF5 library. The new NetCDF-4 library (Rew et al. 2006) can use several HDF5 features, including parallel I/O. NetCDF, HDF and other data formats are discussed in greater detail in Chap. 5.

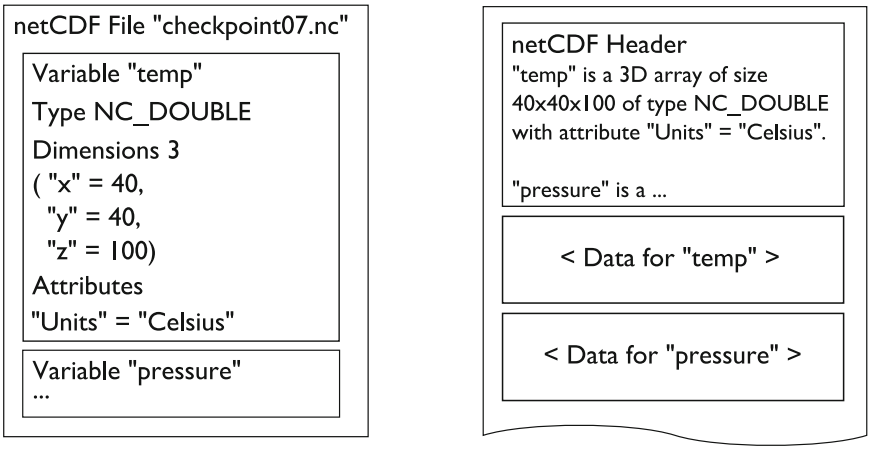

Fig. 2.3 NetCDF: one example of a data format library. The user treats data stored in NetCDF as a series of multidimensional arrays of typed data. These arrays have names and other metadata. The library manages the actual layout on disk, allocating space for a header as well as each variable. Note that both Parallel NetCDF and NetCDF use this same file format

Even though a high-level library requires another set of functions between an application and the underlying MPI-IO library, this extra layer introduces negligible overhead when used appropriately. In some cases, in fact, the abstraction allows for more aggressive optimizations, because the library API provides a richer description of the user's desires than do the lower-level APIs.

For example, the Parallel NetCDF library knows how to cache the file header information, but the MPI-IO layer is unable to distinguish the file header from any other region of the file. The HDF5 library uses a sophisticated allocation routine to optimize sparse matrix operations.

Section 2.3 covered some of the ways MPI-IO makes use of MPI datatypes to describe I/O access patterns. These datatypes map well to the multidimensional accesses used in data format libraries. The challenge of using MPI-IO directly lies in the relative complexity of describing these memory and file regions. Data format libraries manage the use of MPI datatypes on behalf of the application, allowing for a host of optimizations at the MPI-IO level without effort from the user.

As discussed earlier, scientific applications possess distinct periods of computation and I/O. Data format libraries make it easy to carry out the I/O in these phases collectively. When applications use the collective I/O routines in these higher level libraries, the MPI-IO implementation can do much to manipulate the I/O requests into the most appropriate format for the underlying storage system. As with and collective I/O, single-file I/O means more opportunities for optimization and a better chance at achieving higher I/O rates. For example, noncontiguous requests from different clients to adjacent regions can be coalesced.

Space limitations enforce only a brief treatment of I/O libraries. Chapter 5 will discuss more about the file formats in use in environmental models. The bibliographic references for these libraries will also lead to more in-depth materials.

2.5 Applying I/O Lessons to Applications

The biggest payoffs with respect to parallel I/O performance come from a simple rule: Perform I/O with the fewest calls, and pass as much information as possible about the intended results. Usually this means describing I/O as a single collective I/O call to or from one file for all processes. This rule gives the I/O software stack the most opportunity to optimize the I/O access pattern. Naturally, this single piece of advice won't work for all situations and might not yield all the performance gains one could expect, but it will go a long way toward that end.

Nothing harms parallel I/O performance more than small I/O requests. No matter what level of the I/O software stack, an application should describe all the I/O in as few calls as possible. MPI datatypes or the datatype features of the data format libraries (e.g., HDF hyperslabs) can select all data with a single function call instead of making many individual calls for one element at a time. Applications may benefit by setting aside some memory to combine accesses, though the lower layers of the software stack might already offer a tuned and debugged version of this optimization.

Awareness of the underlying file system might guide developers toward an ideal I/O strategy. Some file systems make extensive use of locks to enforce Portable Operating System Interface (POSIX) consistency semantics. Unfortunately, these lock-based file systems impose serialization of access and undermine most of the content in this section. The locking mechanisms are conservatively designed for safety; application scientists should investigate ways to disable locks for their site, at least for applications where it is semantically possible.

All the levels of the software stack have ways to give "hints" about what an application wants to do. These hints are sometimes simple, such as opening a file as either read-only or write-only but not both. Other hints are more complex, such as using MPI-IO Info parameters to adjust internal algorithms.

2.6 Parallel I/O Today and Tomorrow

From parallel file systems managing physical disks, to high-level abstractions, the I/O software stack includes a great deal of software, all of which contributes to high performance and convenient parallel I/O. The more contextual information each layer has, the more optimizations that layer can perform on behalf of the application. In fact, there is more to the I/O stack than could be covered in this space, and still more under development.

One topic that we did not discuss is asynchronous I/O. Asynchronous I/O interfaces provide a way for an application programmer to initiate I/O operations and then test for their completion later. Ideally the underlying I/O software would perform data transfer behind the scenes, allowing the application to go back to computation while I/O occurs. Unfortunately, in practice this is not the case. Most I/O software implements asynchronous I/O with blocking operations – either all I/O is performed on the first call, or all I/O is performed when the application tests for completion. Further, if implementations were to perform I/O behind the scenes, the I/O communication could interfere with communication performed as part of the computation, negatively impacting performance.

This said, asynchronous interfaces do provide another opportunity for libraries to combine I/O operations (e.g., as in PNetCDF), and the trend toward increasing core counts and more feature-rich networks in HPC systems may provide additional opportunities for more effective asynchronous I/O systems in the future.

The careful reader will notice that one part of Fig. 2.1 – I/O forwarding – was not discussed. I/O forwarding helps large I/O subsystems scale by hiding some number of clients behind a forwarding layer and presenting many fewer clients to the file system. This software exists only on today's very large supercomputer systems, where it is used to reduce the apparent number of clients by a factor of 25 to 250.

I/O forwarding and other techniques are expected play a larger role as HPC systems move toward even greater levels of parallelism. Buffered aggregation methods (Ma et al. 2003) have also shown promising results, and are being incorporated into libraries like ADIOS. Current research also focuses on abstractions that are even more domain specific. The unstructured grid and adaptive mesh models do not map

especially well to the multidimensional array models used in current high-level I/O libraries, as indicated earlier, and solutions are being investigated. Some emerging trends are discussed below in other sections, including especially Chaps. 3 and 7.

References

Braam PJ (2003) The lustre storage architecture. Technical Report, Cluster File Systems, Inc., http://lustre.org/docs/lustre.pdf

Ching A, Choudhary A, Coloma K, Liao W, Ross R, Gropp W (2003a) Noncontiguous i/o accesses through MPI-IO. In: Proceedings of the third IEEE/ACM international symposium on cluster computing and the grid (CCGrid2003)

Ching A, Choudhary A, Liao W, Ross R, Gropp W (2002) Noncontiguous i/o through pvfs. In: Proceedings of the 2002 IEEE international conference on cluster computing

Ching A, Choudhary A, Liao W, Ross R, Gropp W (2003a) Efficient structured data access in parallel file systems. In: Proceedings of cluster 2003, Hong Kong

Gropp W, Lusk E, Thakur R (1999) Using MPI-2: Advanced features of the message-passing interface. MIT Press, Cambridge http://mitpress.mit.edu/book-home.tcl?isbn=0262571331

Li J, Keng Liao W, Choudhary A, Ross R, Thakur R, Gropp W, Latham R, Siegel A, Gallagher B, Zingale M (2003) Parallel netCDF: A high-performance scientific I/O interface. In: Proceedings of SC2003: high performance networking and computing, IEEE Computer Society Press, Phoenix, AZ http://www.sc-conference.org/sc2003/paperpdfs/pap258.pdf

Ma X, Winslett M, Lee J, Yu S (2003) Improving MPI-IO output performance with active buffering plus threads. In: Proceedings of the 2003 international parallel and distributed processing symposium, IEEE, pp 10

Nagle D, Serenyi D, Matthews A (2004) The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In: SC '04: Proceedings of the 2004 ACM/IEEE conference on supercomputing, IEEE Computer Society, Washington, DC, USA, p 53, http://dx.doi.org/10.1109/SC.2004.57

Prost JP, Treumann R, Hedges R, Jia B, Koniges A (2001) MPI-IO/GPFS, an optimized implementation of MPI-IO on top of GPFS. In: Proceedings of SC2001

PVFS development team (2008) The PVFS parallel file system. http://www.pvfs.org/

Rew RK, Hartnett EJ, Caron J (2006) NetCDF-4: Software implementing an enhanced data model for the geosciences. In: 22nd international conference on interactive information processing systems for meteorology, oceanography and hydrology, AMS

Schmuck F, Haskin R (2002) GPFS: A shared-disk file system for large computing clusters. In: First USENIX conference on File and Storage Technologies (FAST'02), Monterey, CA

Shepard L, Eppe E (2006) SGI infinite storage shared filesystem CXFS: A high-performance, Multi-OS filesystem from SGI

Thakur R, Gropp W, Lusk E (1999) Data sieving and collective I/O in ROMIO. In: Proceedings of the seventh symposium on the frontiers of massively parallel computation, IEEE Computer Society Press, pp 182–189, http://www.mcs.anl.gov/ thakur/papers/romio-coll.ps

The HDF Group (2008) HDF5. http://www.hdfgroup.org

The MPI Forum (1997) MPI-2: extensions to the message-passing interface. The MPI Forum, http://www.mpi-forum.org/docs/docs.html

Yu H, Sahoo RK, Howson C, Almasi G, Castanos JG, Gupta M, Moreira JE, Parker JJ, Engelsiepen TE, Ross R, Thakur R, Latham R, Gropp WD (2006) High performance file I/O for the blue-gene/l supercomputer. In: Proceedings of the 12th international symposium on high-performance computer architecture (HPCA-12). http://www.mcs.anl.gov/ thakur/papers/bgl-io.pdf

http://www.springer.com/978-3-642-36463-1

Earth System Modelling - Volume 4
IO and Postprocessing
Balaji, V.; Redler, R.; Budich, R.
2013, XIII, 58 p. 6 illus., 2 illus. in color., Softcover
ISBN: 978-3-642-36463-1