Best Practice Guide - Parallel I/O

Sandra Mendez, LRZ, Germany

Sebastian Lührs, FZJ, Germany

Dominic Sloan-Murphy (Editor), EPCC, United Kingdom

Andrew Turner (Editor), EPCC, United Kingdom

Volker Weinberg (Editor), LRZ, Germany

Version 2.0 by 07-02-2019

Table of Contents

1. Introduction

1.1. About this Document

This best practice guide provides information about High Performance I/O systems, parallel I/O APIs and I/O optimisation techniques. The guide presents a description of the storage architecture and the I/O software stack.

1.2. Guide Structure

The best practice guide is split into chapters covering specific topics. These chapters are described in more detail below.

Introduction

This chapter! Describes the guide and its structure.

High Performance I/O Systems

Covers the basic concepts of parallel I/O in HPC, including: general parallel I/O strategies (e.g. shared files, file per process), parallel file systems, the parallel I/O software stack and potential performance bottlenecks.

Parallel File Systems

Describes the major parallel file systems in use on HPC systems: Lustre and Spectrum Scale/GPFS. An overview is given of BeeGFS, a file system gaining popularity, and object storage, a data storage method which is not based on files.

MPI-IO

Brief introduction to MPI-IO with links to further, detailed information along with tips on how to get best performace out of the MPI-IO library on parallel file systems.

File per process

Describes the file per process model for parallel I/O along with performance considerations.

High-level parallel I/O libraries

Covers the use of high-level libraries (HDF5, NetCDF, pNetCDF, SIONlib) and considerations for getting best peformance.

Parallel I/O performance analysis

Information on how to gather performance data on your use of I/O. Covers general advice along with the built-in options for MPI-IO and tools such as Darshan and Vampir.

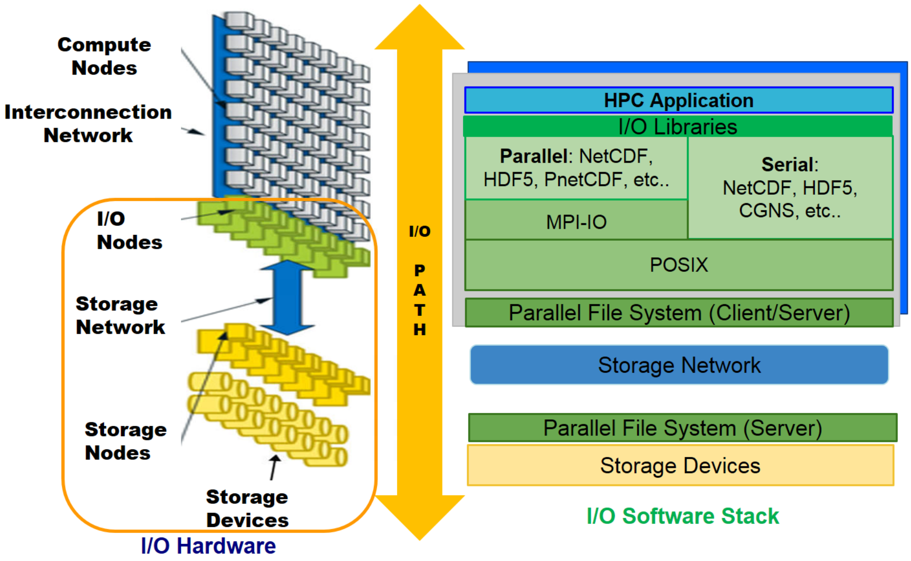
2. High Performance I/O Systems

2.1. Introduction

Current HPC facilities are composed of thousands of compute nodes, high performance networking infrastructures, and parallel I/O environments capable of scaling to terabytes/second of I/O bandwidth while providing tens of petabytes of capacity. In HPC systems, the I/O is a subsystem composed by software and hardware as can be seen in Figure 1, “Typical High Performance I/O System” . The I/O software stack includes the I/O libraries, file system and some operating system utilities. The I/O hardware, also known as I/O infrastructure, is composed of the storage network, storage nodes, I/O nodes and the I/O devices. In most cases the available I/O resources for a single application scale together with the amount of reserved compute resources. However their efficient usage depend on the individual application implementation.

The performance of many research applications is limited by the I/O system. Infrastructure under-utilisation or performance bottlenecks can be related to the I/O software stack and its interaction with the application I/O patterns. Efficient use of I/O on HPC systems requires an understanding of how parallel I/O functions work (in both software and hardware) so that the correct decisions can be made to extract best performance.

Figure 1. Typical High Performance I/O System

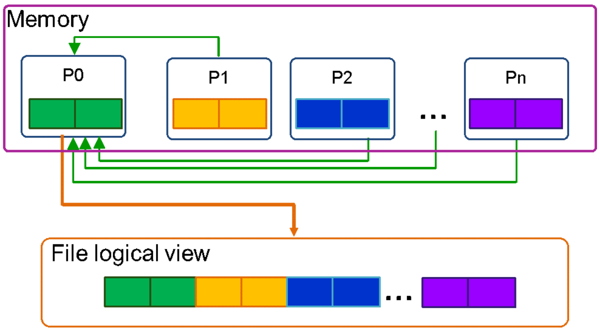


2.2. I/O Strategies in Parallel Applications

Parallel applications usually perform I/O in both serial and parallel. We briefly describe serial I/O below but the remainder of this guide will concentrate on parallel I/O.

2.2.1. Serial I/O

Figure 2. Typical serial I/O in parallel applications



In serial I/O (see Figure 2, “Typical serial I/O in parallel applications”), a single process accesses a single file. As all of the data must flow to a single process, the bandwidth is generally limited by the amount of data that can be passed from the I/O system to a single client (usually a single compute node) leading to low I/O performance. Serial I/O operations should be limited to small data volume access performed infrequently.

2.2.2. Parallel I/O

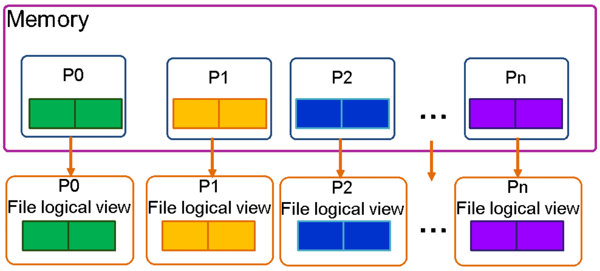
In essence, a file is simply a stream of bytes so a user may have to substantially rearrange their program data before writing it to disk. For example, if a weather model has three main arrays storing air velocity, pressure and temperature then it might make sense for all values for a particular grid point to be stored together in memory within the application (e.g. for performance reasons). However, in the file it might be preferable for the velocities for all gridpoints to be stored in sequence, then all the pressure values then the temperatures (e.g. to help in post-processing). The problem in parallel is that data rearrangement is almost always required if the parallel code should produce the same file as the serial one. For example, in a general domain decomposition parallel tasks do not own a single contiguous chunk of the global data set. Even in a simple 2D decomposition, the local data comes from many different locations in the file, with each local row coming from a different place.

This rearrangement implies communication between tasks during the IO phases, often in a new pattern that is not used within the computational phases. There are a number of ways to simplify this communication pattern which leads to four common I/O strategies. Here we concentrate on the case of writing data: HPC codes typically write much more data than they read, and also writing is a more complicated operation in parallel than reading.

File per process (Multiple files, multiple writers)

The simplest approach is to avoid the data rearrangement completely, with each task writing its data to a different file. In practice this does not avoid the issue, but simply delays it to a later time: subsequent post-processing or analysis programs will almost certainly have to access multiple files to read the data they require. For very large core counts (>10,000) this scheme starts to run up against technological limits in parallel file systems (particularly in metadata operations) and this limits the scaling of this approach at this scale.

Figure 3. File per process approach to parallel I/O



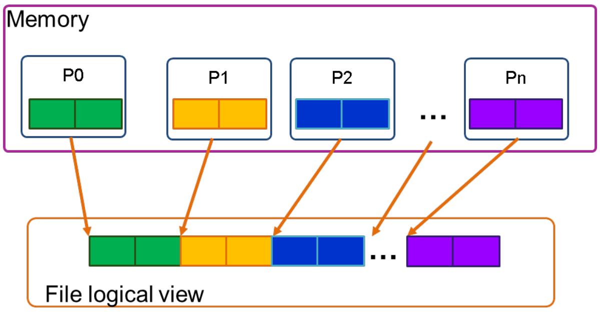
Single file, single writer

This is the other extreme, where a single master task coordinates the data rearrangement, e.g. receiving rows from many tasks and reconstructing the global data set prior to writing it out. This pattern is also called Master I/O. Normally a single process cannot benefit from the total available bandwidth and such an access scheme is also limited by the memory capabilities of the single master process. Larger chunks of data might be transferred step by step which serialises the write process even more.

Single file, multiple writers

Here the data rearrangement is achieved by each task writing its data directly to the correct place in the file, e.g each individual row is written to a different location. Although this does not involve transfer of data between tasks, the tasks will still have to communicate to avoid their writes clashing with each other if there are overlapps between the individual data chunks.

Figure 4. Single file, multiple writers approach to parallel I/O

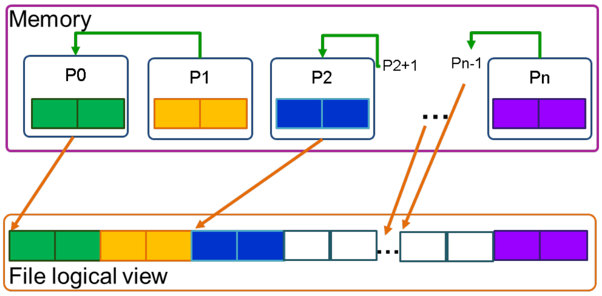


Single file, collective writers

This sits between the two approaches above, where either one or all of the parallel tasks perform I/O; here we identify a subset of tasks to perform the I/O operations. These I/O tasks must communicate with the computational tasks to receive and rearrange the data, and must coordinate with each other to avoid I/O clashes.

This technique can also be implemented using an I/O server approach where a subset of the processes in the application are specialised to handle parallel I/O operations. This allows the I/O to potentially proceed asynchronously to the rest of the application and enable more efficient use of HPC resources.

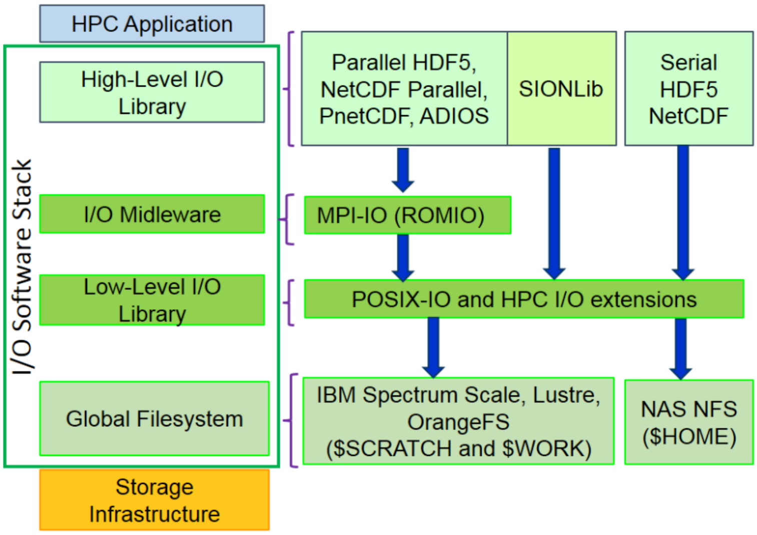
Figure 5. Single file, collective writers approach to parallel I/O



Note that, other than “multiple files, multiple writers”, all these methods should produce identical output to each other on any number of processors. However, they may have very different performance characteristics. The “multiple files, multiple writers” scheme creates a local process related data view (in comparision of the global data view for all other schemes).

2.3. The I/O Software Stack

Figure 6. The I/O Software Stack



The I/O software stack provides users and administrators with application programming interfaces (APIs) that allow them to use the hardware in a common way without worrying about the specific hardware technology. Parallel I/O libraries provide APIs that enable parallel access to a single or several files. Unlike the parallel versions, serial I/O libraries (such as those that provide the basic file operations in high-level programming languages: C/C++, Fortran, Python) do not usually offer specific APIs for parallel access.

2.3.1. Low-level: POSIX

The lowest level in the software stack we find is the POSIX interface, which refers to file operations such as open, close, read, write, stat and so on. POSIX HPC extensions were designed to improve performance of POSIX on large-scale HPC environments where the requirement for performance usually outweighs consistency considerations. For example, it often relaxes the rather strict POSIX consistency semantics [1] . In general most other APIs are build on top of POSIX and POSIX approaches are often used in context of task local file access.

At the lowest level is the file system software itself which manages access to the hardware resources and implements the functions required by the POSIX API. File systems have two key roles: i) Organising and maintaining the file name space and; ii) storing contents of files and their attributes.

On HPC facilities, we usually find networked file systems (NFS) and parallel file systems. Networked file systems must solve two problems: 1) File servers coordinate sharing of their data by many clients and 2) Scale-out storage systems coordinate actions of many servers. We will not consider NFS further in this document as they generally only use limited levels of parallelism (in the form of RAID configurations), so do not have the ability to provide high performance when using parallel I/O strategies in HPC. Generally NFS is not used for reading/writing data during large parallel calculations for this reason. Parallel file systems (usually) distribute single file data across multiple servers and provide for concurrent access to single files by multiple tasks of a parallel application. They have the potential to provide high levels of read/write bandwidth to single files by parallelising the access across multiple I/O streams.

We look at parallel file systems in the next chapter of this guide, Chapter 3.

2.3.2. Middle-level: MPI-IO

On HPC systems the middle level in the I/O software stack is dominated by MPI-IO. By using MPI-IO, it is possible to apply optimisation techniques such as collective buffering and data sieving. ROMIO [12] is the most common implementation of the MPI-IO standard and it is used in MPI distributions such as MPICH (which also covers Cray MPT and HPE MTP), MVAPICH, IBM PE and Intel MPI.

MPI-IO is discussed in more detail in Chapter 4.

2.3.3. High level I/O software

The highest level in I/O software stack (See Figure 6, “The I/O Software Stack”) presents the high-level libraries. These are APIs that help to express scientific simulation data in a more natural way such as multi-dimensional data, labels and tags, non-contiguous data and typed data. Parallel versions sit on top of the MPI-IO layer and can use MPI-IO optimisations. High-level libraries provide simplicity for visualisation and analysis; and portable formats. HDF5 [7] and NetCDF [8] are the most popular high level libraries. Over the last years PnetCDF [9] and ADIOS [10] have also been selected by HPC users to perform parallel I/O. Another library that is gaining popularity is SIONLib [11] , a scalable I/O library for parallel access to task-local files. The library not only supports writing and reading binary data to or from several thousands of processors into a single or a small number of physical files but also provides global open and close functions to access SIONlib file formats in parallel.

These high-level libraries are described in more detail in Chapter 6.

All of the libraries and interfaces described above implement parallel I/O using a shared file approach with multiple processes writing to the same logical file (some approaches also allow to use multiple physical files, which are treated together as one logical file). An alternative approach is to use the standard programming language I/O interfaces can be used to implement a file per process model of parallel I/O where every parallel process writes its own file. This approach has its own advantages and disadvantages and is described in more detail in Chapter 5.

2.4. General Pointers for Efficient I/O

A few "rules of thumb" are given below to consider when running or designing I/O-intensive applications on HPC systems.

- Avoid unnecessary I/O. For example, switch off debug output for production runs.
- Perform I/O in few and large chunks. In parallel file systems, the chunk size should be a multiple of the block size or stripe size.
- Prefer binary/unformatted I/O instead of formatted data.
- Avoid unnecessary/large-scale open/close statements. Remember that metadata operations are latency bound.
- Use an appropriate file system. Parallel file systems may not scale well for metadata operations, but provide high/scalable bandwidth. NFS-based file systems may show the reversed behaviour.
- Avoid explicit flushes of data to disk, except when needed for consistency reasons.
- Use specialised I/O libraries based on the I/O requirements of your applications. These provide more portable way of writing data and may reduce metadata load when properly used.
- Convert to target / visualisation format in memory if possible.
- For parallel programs, a file-per-process strategy can provide high throughput, but usually needs a further post-processing stage to collate the outputs.

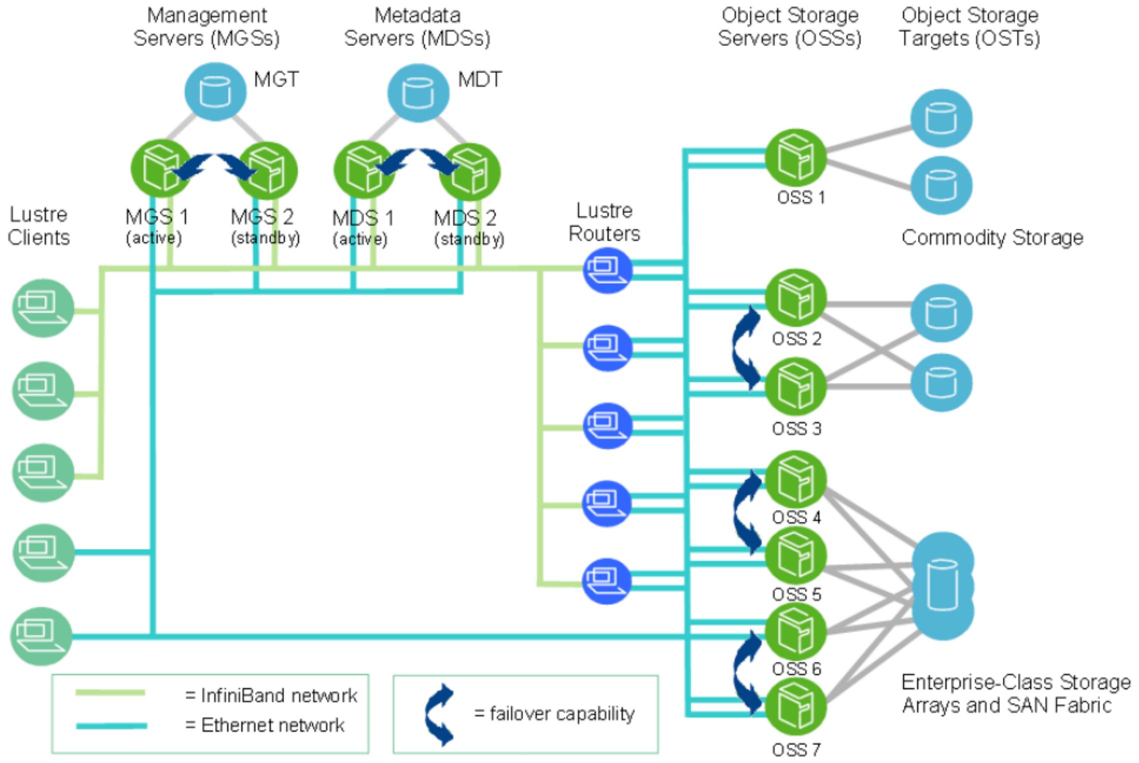
3. Parallel File Systems

3.1. Introduction

Parallel file systems provide high-performance I/O when multiple clients (a "client" usually meaning a compute node, in this instance) share the same file system. The ability to scale capacity and performance is an important characteristic of a parallel file system implementation. **Striping** is the basic mechanism used in parallel file systems for improving performance, where file data is split up and written across multiple I/O servers. Primarily, striping allows multiple servers, disks, network links to be leveraged during concurrent I/O operations, thus increasing available bandwidth. The most popular parallel file systems on HPC platforms are Lustre and IBM Spectrum Scale (formerly GPFS) [1] and form the focus of this chapter. A brief overview is given of BeeGFS, a file system gaining popularity, and object storage, a data storage method which is not based on files.

3.2. Lustre

Figure 7. Lustre cluster at scale



Lustre is a Linux file system implemented entirely in the kernel and provides a POSIX standards-compliant file system interface. Its architecture is founded upon distributed object-based storage. This delegates block storage management to its back-end servers and eliminates significant scaling and performance issues associated with the consistent management of distributed block storage metadata.

Lustre file systems are typically optimised for high bandwidth: they work best with a small number of large, contiguous I/O requests rather than a large number of small ones (i.e. small numbers of large files rather than large numbers of small files).

A Lustre file system consists of the following components:

- **Metadata Servers (MDS)** : The MDS makes metadata stored in one or more Metadata Targets (MDTs) available to Lustre clients. Each MDS manages the names and directories in the Lustre file system(s) and provides network request handling for one or more local MDTs. Operations such as opening and closing a file can require dedicated access to the MDS and it can become a serial bottleneck in some circumstances.

- **Metadata Targets (MDT)** : For Lustre software release 2.3 and earlier, each file system has one MDT; multiple MDTs are supported on later versions. The MDT stores metadata (such as filenames, directories, permissions and file layout) on storage attached to an MDS. An MDT on a shared storage target can be available to multiple MDSs, although only one can access it at a time. If an active MDS fails, a standby MDS can serve the MDT and make it available to clients. This is referred to as MDS failover.
- **Object Storage Servers (OSS)** : The OSS provides file I/O service and network request handling for one or more local Object Storage Targets (OSTs). Typically, an OSS serves between two and eight OSTs, up to 16 TB each. A typical Lustre configuration consists of an MDT on a dedicated node, two or more OSTs on each OSS node, and a client on each of a large number of compute nodes.
- **Object Storage Target (OST)** : User file data is stored in one or more objects, each object on a separate OST in a Lustre file system. Each OST can write data at around 500 MB/s. You can think of an OST as being equivalent to a disk, although in practice it may comprise multiple disks, e.g. in a RAID array. An individual file can be stored across multiple OSTs; this is called striping. The default is dependent on the particular system (1 is a common choice), although this can be changed by the user to optimize performance for a given workload.
- **Lustre clients** : Lustre clients are compute, visualisation or login nodes that are running Lustre client software, allowing them to mount the Lustre file system. Good performance is achieved when multiple clients (usually compute nodes for HPC calculations) simultaneously access the file system.

3.2.1. Lustre File Layout (Striping)

Lustre file systems have the ability to stripe data across multiple OSTs in a round-robin fashion. Users can optionally configure for each file the number of stripes, stripe size, and OSTs that are used. Although these parameters can be set on a per-file basis they are usually set on directory where your output files will be written so that all output files inherit the same settings. The *stripe_size* indicates how much data to write to one OST before moving to the next OST. The *stripe_count* indicates how many OSTs to use. The default values for *stripe_count* and *stripe_size* are system dependent but are often 1 and 1 MiB respectively.

You can use the `lfs getstripe` command to see the layout of a specific file:

```
>lfs getstripe reads2.fastq
reads2.fastq
lmm_stripe_count: 1
lmm_stripe_size: 1048576
lmm_pattern: 1
lmm_layout_gen: 0
lmm_stripe_offset: 42
obdidx objid objid group
42 37138823 0x236b187
0
```

In this case, the file reads2.fastq has a single stripe of size 1 MiB (1048576 bytes). To change the layout of specific directory (or file) we can use the `lfs setstripe` command. (Note that this command will not repartition the data on existing files but will ensure that new files created within the directory use the updated settings.)

```
>lfs setstripe -c -1 results_dir/
```

In the example, we set the stripe count to -1 (maximal striping) to make the largest potential bandwidth available to new files in this directory.

3.2.2. Choosing a stripe count

The stripe count sets the number of OSTs (Object Storage Targets) that Lustre stripes the file across. In theory, the larger the number of stripes, the more parallel write performance is available. However, large stripe counts for small files can be detrimental to performance as there is an overhead in using more stripes.

The stripe count has the largest potential impact on performance on Lustre files systems. The following advice generally applies:

- When using shared files you should use maximal striping (`lfs setstripe -c -1`) to give the largest potential bandwidth for parallel access.
- Using multiple stripes with large numbers of files (for example in a file per process scheme with large core counts) can have an adverse impact on the performance of Lustre file systems. You will generally see the best performance for large file counts with a stripe count of 1.

3.2.3. Choosing a stripe_size

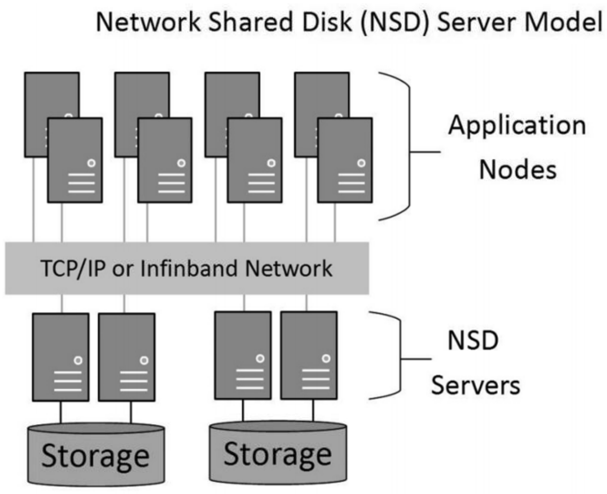
The size of each stripe generally has less of an impact on performance than the stripe count but can become important as the size of the file being written increases. We outline some considerations below but bear in mind that the impact is linked to the I/O pattern used in the application (and that stripe count is usually a more important parameter anyway). Note:

- The stripe size should be a multiple of the page size
- The smallest recommended stripe size is 512 KB
- A good stripe size for sequential I/O using high-speed networks is between 1 MB and 4 MB
- The maximum stripe size is 4 GB

Further information can be found on [32]

3.3. IBM Spectrum Scale (formerly GPFS)

Figure 8. NSD Server Model


Network Shared Disk (NSD) Server Model
Application Nodes
TCP/IP or Infinband Network
NSD Servers
Storage
Storage

Picture Source, High Performance Parallel I/O Book, Chapter 9.
Editors Prabhat, Quincey Koziol. October 2014.

IBM Spectrum Scale is a cluster file system that provides concurrent access to a single file system or set of file systems from multiple nodes. The nodes can be SAN attached, network attached, a mixture of SAN attached and network attached, or in a shared nothing cluster configuration. This enables high performance access to this common set of data to support a scale-out solution or to provide a high availability platform.

Its main characteristics are:

- Scalability: It uses the concept of wide striping that means distribute the data and metadata across all resources. Large files are divided into equal-sized blocks and the consecutive blocks are placed on different disks in a round-robin fashion.

- Caching: It is client-side, which is kept in a dedicated and pinned area of each node called the pagepool. The cache is managed with both read-ahead techniques and write-behind techniques.
- Cache coherence and protocol: It uses the distributed locking to synchronize the access to data and metadata on a shared disk.
- Metadata management: It uses inodes and indirect blocks to record file attributes and data block addresses.

At the user level, the GPFS main parameter to consider is the block size. Parallel applications should use request sizes (chunk size) that are multiples of the block size to obtain a high data transfer rate.

Unlike other parallel file systems such as Lustre, in GPFS, the user cannot change the block size or select the number of data servers. A large file is distributed across all the disks that are part of GPFS storage.

Parallel applications that use MPI-IO should write/read in multiples of the block size and align the request size to block size to avoid file lock contention that can seriously degrade I/O performance.

3.3.1. Block size

To display the amount of available disk space for each filesystem:

```
di98het@login05:~> df -Th
Filesystem Type Size Used Avail
Use% Mounted on
/dev/fs1 gpfs 12P 8.8P 3.0P 75% /gpfs
/dev/fs2 gpfs
5.2P 3.8P 1.4P 73% /gss/scratch
```

The I/O operation size should be a multiple of the block size. To display the properties of a GPFS, this command can be used:

```
di98het@login05:~> mmlsfs fs1
flag value description
-------------------------------------------------------
-f 262144
Minimum fragment size in bytes
...
-B 8388608 Block size
...
```

Further information can be found on [31]

3.4. BeeGFS (formerly FhGFS)

BeeGFS, also known as FhGFS prior to 2014, is a file system which has gained popularity in recent years in the European HPC community. Attributed in part to its relative ease of deployment, it being free and open source (in contrast to proprietary solutions like IBM Spectrum Scale), and its support of a wide range of Linux kernel versions.

Comparable to Lustre in terms of system architecture, BeeGFS also consists of Management Server (MS), Metadata Server (MDS) and Object Storage Server (OSS) components[4] . It similarly uses the concept of file striping for parallelism and provides tools to enable users to query and configure stripe parameters.

The `beegfs-ctl` utility enables clients to affect BeeGFS parameters. To query the existing stripe pattern for a file, `beegfs-ctl --getentryinfo` is used[34]:

```
    > beegfs-ctl --getentryinfo /mnt/beegfs/testdir/test.txt

    Path: /testdir/test
    Mount: /mnt/beegfs
    EntryID: 0-5429B29A-AFA6
    Metadata node: meta01 [ID: 1]

    Stripe pattern details:
    + Type: RAID0
    + Chunksize: 512K
    + Number of storage targets: desired: 4; actual: 4
    + Storage targets:
        + 102 @ storage01 [ID: 1]
        + 101 @ storage01 [ID: 1]
        + 201 @ storage02 [ID: 2]
        + 202 @ storage02 [ID: 2]
```

Indicating the `test.txt` file has four stripes, each with a size of 512KB. Similarly, setting the stripe pattern can be accomplished with `beegfs-ctl --setpattern` as follows:

```
> beegfs-ctl --setpattern --chunksize=1m --numtargets=4 /data/test
```

which sets the stripe count to four, each with a size of 1MB, for all files created under the /data/test directory. Note that default installations of BeeGFS restrict the `--setpattern` mode to superuser/root access only. Non-root users may still set striping configurations but only if the site has explicitly allowed it by enabling the `sysAllowUserSetPattern` setting on the MDS[5].

More information on BeeGFS stripe settings and performance considerations is at [6]

3.5. Object Storage

A relatively new data storage technique being driven by advances in cloud and internet-based services is object storage. This eschews the directory hierarchy of a traditional file system in favour of a collection of data "objects", each containing their own unique identifier. A comparison can be drawn between object storage and key-value pairs, such as those implemented in Java Maps or Python dictionaries.

The primary advantage of object storage over a traditional file system is improved scalability. As traditional file systems grow, they are often limited by the increasing complexity required to keep track of their hierarchical structure. In contrast, the simple flat structure of object storage allows further capacity to be added without introducing further complexity. The drawback to the approach is it is suited more for long-term storage of large volumes of unstructured data. Frequently changed data is generally not suited for object storage due to performance limitations, which can have implications for HPC applications.

A functional consideration for application authors is that specialised methods must be used to interact with an object store, rather than typical read/write file system calls. The API used by Amazon's Simple Storage Service (S3) has become a de facto standard in this area, with multiple storage vendors implementing it as the interface to their services and a wide variety of tools supporting it. However, directly interfacing with cloud stores may not be suitable for many HPC I/O patterns, due to the performance limitations on transactional data. A possible workflow for HPC use would be writing to a scratch space in a traditional file system before pushing the data to an object store for future analysis or archiving. This does not require any change to the software or compute stage of a task but does mean the user must still operate under the fundamental limitations of traditional file systems.

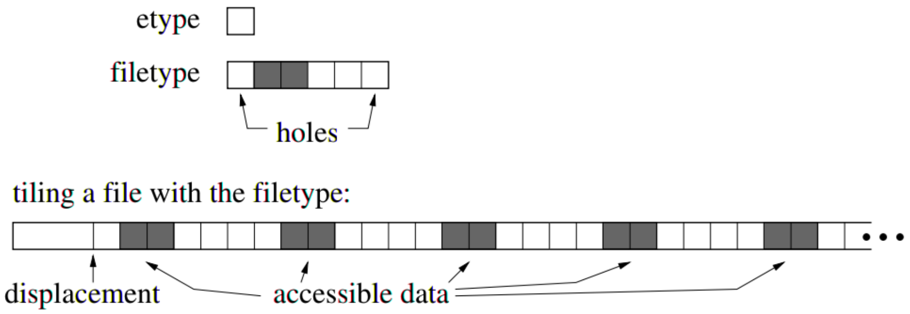
4. MPI-IO

4.1. Introduction

POSIX is a standard that maintains compatibility with the broadest base of applications while enabling high performance. Parallel file systems used across HPC provide parallel access while retaining POSIX semantics. However, the portability and optimisation needed for parallel I/O cannot easily be achieved with the POSIX interface. In order to face this issue, the HPC community defined the MPI-IO interface. MPI-IO provides a high-level interface supporting partitioning of file data among processes and a collective interface supporting complete transfers of global data structures between process memories and files. An implementation of MPI-IO is typically layered on top of a parallel file system that supports the notion of a single and common file shared by multiple processes. (Both of the common parallel file systems currently in use: Lustre and IBM Spectrum Scale, support this concept.)

MPI-IO was initially defined as part the MPI-2 (Message Passing Interface) Standard in 1997. MPI-IO provides capabilities to exploit I/O performance improvements that can be gained via support for asynchronous I/O, strided accesses, and control over physical file layout on storage devices (disks). Instead of defining I/O access modes to express the common patterns for accessing a shared file, the approach in the MPI-IO standard is to express the data partitioning using derived datatypes. Figure 9, “MPI-IO File concepts” depicts the logical view of a MPI-IO file and the accompanying table shows the main concepts defined for MPI-IO in the MPI-3 standard [33].

Figure 9. MPI-IO File concepts



Concept	Definition
file	An ordered collection of typed data items
etype	The unit of data access and positioning. It can be any MPI predefined or derived datatype
filetype	The basis for partitioning a file among processes and defines a template for accessing the file. A filetype is either a single etype or a derived MPI datatype constructed from multiple instances of the same etype.
view	Defines the current set of data visible and accessible from an open file as an ordered set of etypes. Each process has its own view of the file, defined by three quantities: a displacement, an etype, and a filetype. The pattern described by a filetype is repeated, beginning at the displacement, to define the view.
Offset	It is a position in the file relative to the current view, expressed as a count of etypes. Holes in the view’s filetype are skipped when calculating this position.
Displacement	It is an absolute byte position relative to the beginning of a file. The displacement defines the location where a view begins.
file size and end of file	The size of an MPI file is measured in bytes from the beginning of the file.
file pointer	A file pointer is an implicit offset maintained by MPI. “Individual file pointers” are file pointers that are local to each process that opened the file. A “shared file pointer” is a file pointer that is shared by the group of processes that opened the file.
file handle	A file handle is an opaque object created by MPI_FILE_OPEN and freed by MPI_FILE_CLOSE.

4.1.1. MPI-IO data access operations

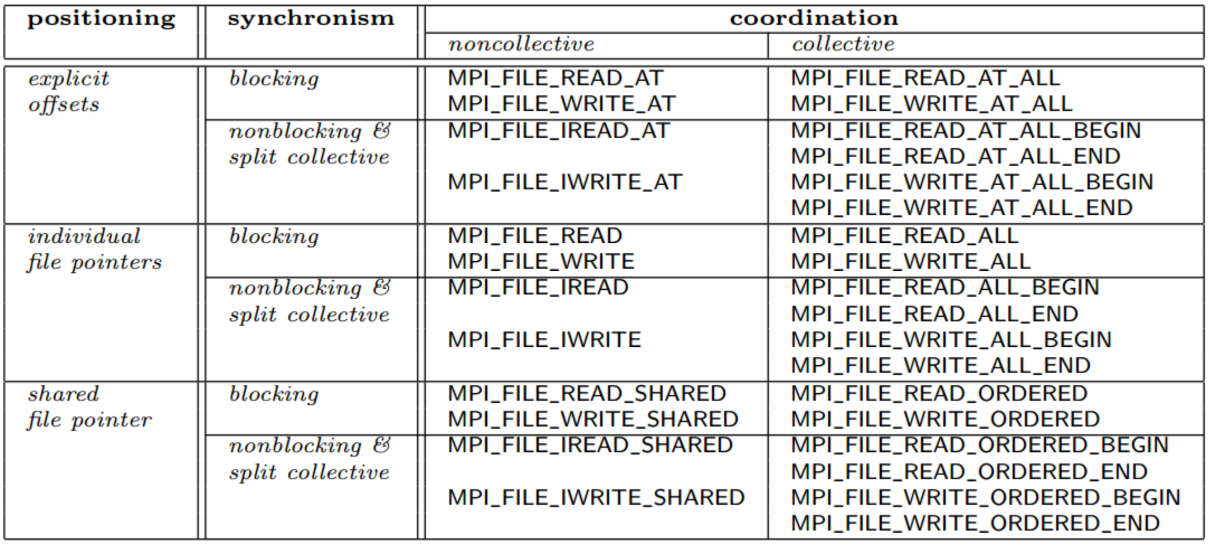
MPI-IO defines three orthogonal aspects to data access from processes to files: positioning, synchronism and coordination. Positioning can be an explicit offset or implicit through the file pointer. Synchronism provides three access modes blocking, nonblocking and split collective. Coordination allows to the MPI processes to perform noncollective or collective operations.

Collective operations are generally required to be able to achieve best performance as they allow the MPI-IO library to implement a number of important optimisations:

- Nominate a subset of MPI processes as writers, the number being selected automatically to match the file system configuration.
- Aggregate data from multiple processes together before writing, ensuring a smaller number of larger IO transactions;
- Ensure that there are no clashes between different writers so that IO transactions can take place in parallel across all the OSTs.

The strategy is automatically selected by MPI-IO but can also be manipulated by setting MPI-IO hints or environment variables. Depending on the data layout and distribution a certain collective strategy might perform better than others.

Figure 10. Data Access Operations



positioning	synchronism	coordination	
		noncollective	*collective*
explicit offsets	*blocking*	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	nonblocking & split collective	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
individual file pointers	*blocking*	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	nonblocking & split collective	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
shared file pointer	*blocking*	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	nonblocking & split collective	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

4.1.2. General Hints

- Application developers should aim to use the highest level of abstraction possible to allow the MPI-IO library to implement I/O in the optimal way.
- Collective operations should be used as the non-collective operations limit the optimisation that the MPI-IO library can perform. Collective operations often perform orders of magnitude better than non-collective operations for large amounts of I/O. This is not always the case, but should be tested for all read and write commands where possible.
- Getting good performance out of MPI-IO also depends on configuring and using the parallel file system correctly, this is particularly important on Lustre file systems.

4.2. Manipulating Files in MPI

File manipulation in MPI-IO is similar to POSIX-IO:

- Open the file: `MPI_File_open`
- Write/Read to/from the file: `MPI_File_write` or `MPI_File_read`
- Close the file: `MPI_File_close`

4.2.1. Opening a File

```
int MPI_File_open(MPI_Comm comm, const char *filename, int amode, MPI_Info
info, MPI_File *fh)
```

MPI_File_open opens the file identified by the file name **filename** on all processes in the **comm** communicator group. It is a collective routine, all processes must provide the same value for **amode**, and all processes must provide filenames that reference the same file. A process can open a file independently of other processes by using the MPI_COMM_SELF communicator. The file handle returned, **fh**, can be subsequently used to access the file until the file is closed using **MPI_File_close(fh)**.

Initially, all processes view the file as a linear byte stream, and each process views data in its own native representation.

The supported **amode** are:

- `MPI_MODE_RDONLY` read only,
- `MPI_MODE_RDWR` reading and writing,
- `MPI_MODE_WRONLY` write only
- `MPI_MODE_CREATE` create the file if it does not exist,
- `MPI_MODE_EXCL` error if creating file that already exists,
- `MPI_MODE_DELETE_ON_CLOSE` delete file on close,
- `MPI_MODE_UNIQUE_OPEN` file will not be concurrently opened elsewhere,
- `MPI_MODE_SEQUENTIAL` file will only be accessed sequentially,
- `MPI_MODE_APPEND` set initial position of all file pointers to end of file.

C users can use bit vector OR (|) and in Fortran addition "+" to combine these constants; or the bit vector **IOR** intrinsic in Fortran 90.

info is an opaque object with a handle of type **MPI_Info** in C and Fortran with the mpi_f08 module, and INTEGER in Fortran with the mpi module or the include file mpif.h. `MPI_INFO_NULL` can be used as a default.

Hints specified via **info** allow a user to provide information such as file access patterns and file system specifics to direct optimisation. Providing hints may enable an implementation to deliver increased I/O performance or minimise the use of system resources.

info is a parameter in MPI_FILE_OPEN, MPI_FILE_SET_VIEW,MPI_FILE_SET_INFO.

To create a new **info** object `MPI_INFO_CREATE(info,ierror)` is used and `MPI_INFO_SET(info, key, value, ierror)` to add an entry to **info**.

4.3. File View

In MPI-IO it is possible to define a view for MPI process using MPI_File_set_view. A view is specified by a triplet (displacement, etype, and filetype). The default view is a linear byte stream, represented by the triple (0, MPI_BYTE, MPI_BYTE).

```
MPI_FILE_SET_VIEW (fh, disp, etype, filetype, datarep, info)
```

A file view:

- changes the process’s view of the data in the file
- resets the individual file pointers and the shared file pointer to zero
- is a collective operation
- the values for `datarep` and the extents of `etype` in the file data representation must be identical on all processes in the group
- the datatypes passed in `etype` and `filetype` must be committed
- `datarep` argument is a string that specifies the format in which data is written to a file: “native”, “internal”, “external32”, or user-defined
- “external32” is a data representation that is supposed to be portable across architectures. However not all MPI implementations support the "external32" representation, so in general MPI-IO files are not portable between all combinations.

Below a simple example of a view is shown:

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;

// Create the contig datatype composed by two integers
MPI_Type_contiguous(2, MPI_INT, &contig);
lb = 0; extent = 6 * sizeof(int);

// Create the filetype with lb as lower bound
// and extent from the contig datatype
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);


// Define the displacement
disp = 5 * sizeof(int);
etype = MPI_INT;

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", \
MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);

MPI_File_set_view(fh, disp, etype, filetype, "native",
MPI_INFO_NULL);

MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

4.4. ROMIO optimisation

One of the most common MPI-IO implementation is ROMIO, which is used in the major MPI distributions such as MPICH, MVAPICH, PE IBM and Intel MPI.

Hints for Data Sieving:

- `ind_rd_buffer_size` controls the size (in bytes) of the intermediate buffer used when performing data sieving during read operations.

- ind_wr_buffer_size Controls the size (in bytes) of the intermediate buffer when performing data sieving during write operations.
- romio_ds_read Determines when ROMIO will choose to perform data sieving for read. Valid values are enable, disable, or automatic.
- romio_ds_write Determines when ROMIO will choose to perform data sieving for write. Valid values are enable, disable, or automatic.

Hints for Collective Buffering (Two-Phase I/O):

- cb_buffer_size Controls the size (in bytes) of the intermediate buffer used in two-phase collective I/O.
- cb_nodes Controls the maximum number of aggregators to be used.
- romio_cb_read Controls when collective buffering is applied to collective read operations. Valid values are enable, disable, and automatic.
- romio_cb_write Controls when collective buffering is applied to collective write operations.
- romio_no_indep_rw This hint controls when “deferred open” is used.
- cb_config_list Provides explicit control over aggregators (i.e., one process per node).

4.5. Setting Hints for MPI-IO

Hints for MPI-IO can be set by using:

- an "info" object, as in this example:

```
integer info, ierr
call MPI_Info_create(info, ierror)
call MPI_Info_set(info, ’romio_cb_read’, ’disable’, ierr)
call MPI_Info_set(info, ’romio_cb_write’, ’disable’, ierr)
...
call MPI_File_open(comm, filename, amode, info, fh, ierror)
```

- the ROMIO_HINTS environment variable. Here the user must create a file with a list of hints to be set at execution time:

```
>cat $HOME/romio-hints
romio_cb_read enable
romio_cb_write enable
```

  Before running the application, the ROMIO_HINTS variable is set as follows: `export ROMIO_HINTS=$HOME/romio-hints`

4.6. MPI-IO General Considerations

- MPI-IO has many features that can help users achieve high performance. The different MPI-IO routines provide flexibility as well as portability.
- The most important features are the ability to specify non-contiguous accesses, the collective I/O functions, and the ability to pass hints to the MPI implementation.
- When accesses are non-contiguous, users must create derived datatypes, define file views, and should use the collective I/O functions.

- Use of MPI I/O is often limited to parallel file systems; do not expect to obtain good performance using it with NFS.

5. File Per Process

5.1. Introduction

Often maligned in HPC circles, the file-per-process model can be a useful approach to parallel I/O for many applications if they can live within the constraints it introduces.

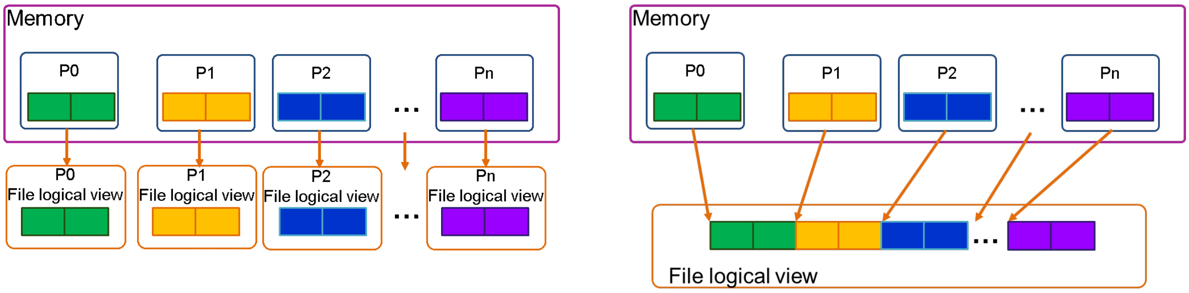
5.2. File Per Process vs. Shared File

When deciding on a parallel I/O strategy, it is important to be informed of the pros and cons of available approaches in order to select one that best meets your requirements. File-per-process is often not given appropriate consideration, with a single shared file assumed to be superior in all situations. However, this is not the case. The following section details the relative advantages and drawbacks to the two opposite extremes patterns of file-per-process and a single shared file, and seeks to justify why the lesser considered strategy should not be discarded so readily.

The simplicity of file-per-process is the most significant advantage over a shared file approach. With each process creating its own file, all file accesses are made independently of one another so it is not necessary for the implementer to introduce any synchronisation between I/O operations, sidestepping much of the complexity of parallel programming. This advantage additionally extends beyond initial implementation, as easier to understand code is also easier to maintain, leading to improved software sustainability.

However, it could be argued that file-per-process only delays introducing complexity, as, following process completion, the data is distributed over the various files and must be reconstituted or otherwise pre-processed before analysis can begin. Alternatively, the analysis tools themselves may be modified to support input from multiple files but this is a trade of application complexity for increased complexity in the tools. In addition there are normally regulations by each hosting site which only allow a limited number of individual files within one computing project.

Figure 11. Recap of file per process (left) and single shared file (right)



In terms of performance, file-per-process is entirely capable of achieving reasonable speeds approaching the maximum attainable bandwidth of the underlying system (see the following Sample Performance Data section for a demonstration of this) but is often bottlenecked by technological limits of the file system, in particular those relating to metadata operations. Metadata includes file permissions, creation date, name, owner – generally anything relating to a file other than the data itself. The more files you have, the more requests you must make for this information, and it can strain the metadata server(s) significantly. This can result in an enforced upper limit on the scalability of the approach, as your application I/O ends up serialising on the metadata resources. This serialisation is often observed by HPC service users when common commands such as `ls` run slowly or hang completely when run on directories with large numbers of files.

The shared file strategy has the advantage in terms of disk storage footprint. Each file comes with its own overhead, such as header information, which is only recorded once for a single file but must be recorded for each file in the file-per-process model. However, in practice, users are rarely limited by disk space so this is seldom a concern.

To summarise, file-per-process **pros**:

- Easier to implement
- More straightforward to maintain
- Can achieve performance on par with single shared file

and file-per-process **cons**:

- Adds complexity to data analysis
- Metadata operations limit scalability
- Has a larger disk storage footprint

5.3. Optimising File Per Process

Several of the drawbacks of file-per-process can be mitigated by the user. Use of data management approaches can alleviate much of the strain from metadata operations in the approach. Breaking up the files into a set of directories, for instance, can reduce the number of metadata queries made when copying, moving, reading or otherwise accessing the files.

Archiving utilities such as `tar` and `zip` are designed with the purpose of converting collections of files into a single file for later use and easier management. This is ideal for addressing the limitations of file-per-process and comes with additional benefits such as optional compression and error checking using, for example, the CRC values stored in zip files by default. These formats are ubiquitous in the computing world so come with the further advantage that many standard tools and libraries for handling these files already exist and can be utilised without much additional effort from the user.

File systems that support user tuning can be reconfigured to better support the file-per-process pattern. For example, a Lustre space is best configured to a single stripe per file on the corresponding directory, to minimise the overhead of communication with the OSTs:

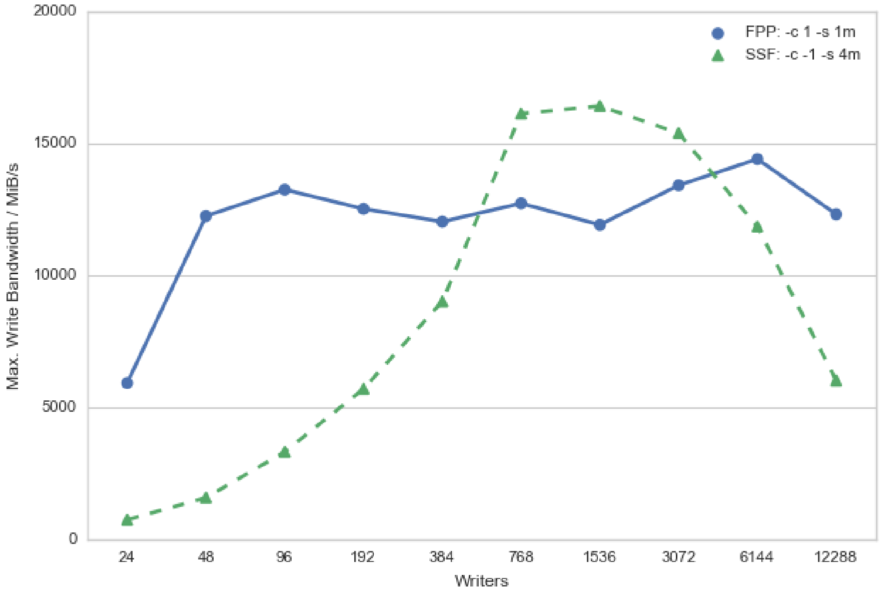
```
lfs setstripe -c 1 files_dir/
```

Refer to Chapter 2 for further details on parallel file system technology and options for tuning for large numbers of smaller files.

5.4. Sample Performance Data

Experimental data is provided by the National Supercomputing Service of the UK, ARCHER[13] [14]. The service is based around a Cray XC30 supercomputer with a Seagate Sonexion implementation of Lustre and results published primarily focus on its `fs3` file system consisting of 48 available OSTs and 12 OSSes. The file-per-process (FPP) and single-shared-file (SSF) schemes are directly compared in terms of the maximum achievable bandwidth of standard unformatted Fortran writes. Results are gathered from 1-512 fully-populated ARCHER nodes, i.e. 24-12288 CPU cores at 24 processes per node, and the plot in Figure 12, “ARCHER sample FPP vs. SSP results” produced.

Figure 12. ARCHER sample FPP vs. SSP results



While SSF reaches a higher maximum bandwidth than FPP between 768-3072 processes, FPP is still comparable at these core counts and even noticeably outperforms the shared file approach at a lower number of cores, 24-192. Furthermore, FPP outperforms SSF at the maximum of 12288 cores where SSF performance drops off, suggesting a bottleneck other than metadata performance.

FPP additionally achieves a much more consistent speed overall, with the bandwidth reached at 48 cores being approximately what is measured at all points up to 12288 cores. For these reasons, FPP is the approach recommended by ARCHER in most cases and not SSF, as may be expected.

6. High-Level I/O Libraries

6.1. Introduction

Scientific applications work with structured data and desire more self-describing file formats. A high-level I/O library is an API which helps to express scientific simulation data in a more natural way. Using high-level I/O libraries it is possible represent multi-dimensional data, labels and tags, noncontiguous data and typed data. These kind of libraries offers simplicity for visualisation and analysis, portable formats can run on one machine and take output to another; and longevity where output will last and be accessible with library tools with no need to remember a specific version number of the code. Many HPC applications make use of higher-level I/O libraries such as the Hierarchical Data Format (HDF) and the Network Common Data Format (NetCDF). Parallel versions of high-level libraries are built on top of MPI-IO and they can use MPI-IO optimisations.

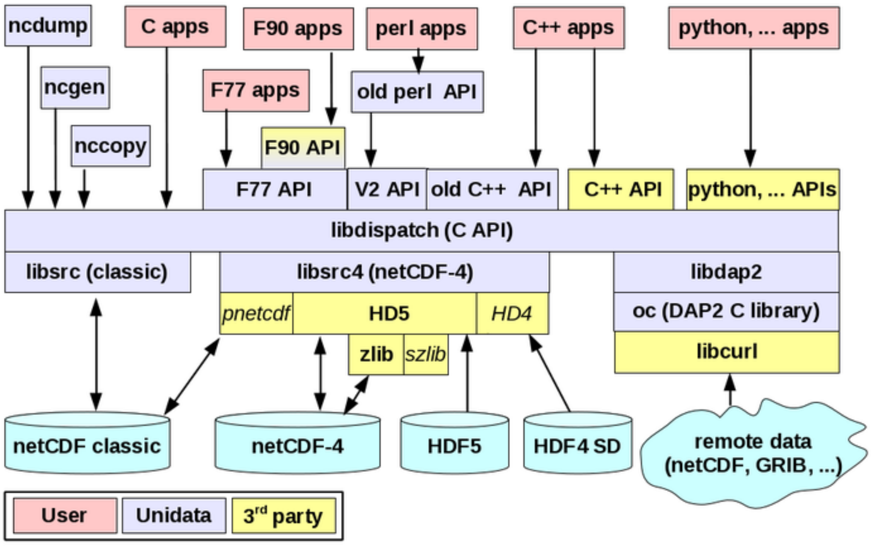
6.2. NetCDF

NetCDF (Network Common Data Form) is a set of software libraries and self-describing, machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data sets (vectors, matrices and higher dimensional arrays). The file format defined by netCDF allows scientists to exchange data and makes the I/O independent of any particular machine architecture. The portability of its files is the most important feature in netCDF. It is commonly used in climatology, meteorology and oceanography applications (e.g., weather forecasting, climate change) and GIS applications.

6.2.1. Architecture of NetCDF APIs and Libraries

Figure 14, “ Enhanced NetCDF Data Model” shows the layering architecture of netCDF C/C++/Fortran libraries and applications.

Figure 13. Architecture of NetCDF APIs and Libraries



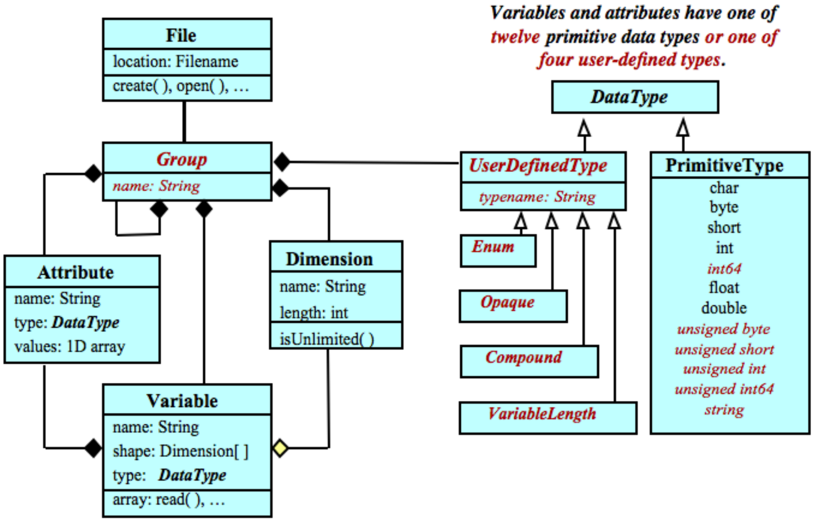
The software libraries provide read-write access to the netCDF format, encoding and decoding the necessary arrays and metadata. The core library is written in C, and provides an API for C, C++, Fortran 77 and Fortran 90. Additionally, a complete independent implementation is written in Java, which extends the core data model and adds additional functionality. Interfaces to netCDF based on the C library are also available in other languages including R, Perl, Python, Ruby, MATLAB, IDL and Octave.

Parallel I/O has been incorporated for netCDF from version 4, Unidata's netCDF supports parallel I/O either through PnetCDF or HDF5. Through PnetCDF, netCDF-4 can access files in CDF formats in parallel. Similarly, through HDF5, netCDF-4 can access files in HDF5 format (so called netCDF-4 format).

6.2.1.1. The netCDF data model

A netCDF dataset contains dimensions, variables, and attributes, which all have both a name and an ID number by which they are identified. These components can be used together to capture the meaning of data and relations among data fields in an array-oriented dataset.

Figure 14. Enhanced NetCDF Data Model



Variables and attributes have one of twelve primitive data types or one of four user-defined types.

A file has a top-level unnamed group. Each group may contain one or more named subgroups, user-defined types, variables, dimensions, and attributes. Variables also have attributes. Variables may share dimensions, indicating a common grid. One or more dimensions may be of unlimited length.

- Dimension: An entity that can either describe a physical dimension of a dataset, such as time, latitude, etc., as well as an index to sets of stations or model-runs.
- Variable: An entity that stores the bulk of the data. It represents an n-dimensional array of values of the same type.
- Attribute: An entity to store data on the datasets contained in the file or the file itself. The latter are called global attributes.

6.2.2. Parallel I/O with netCDF

The netCDF library supports parallel I/O based on MPI-IO+pnetcdf or MPI-IO+HDF5. Parallel I/O support must be enabled at configure time when building these libraries:

- To build netCDF-4 with HDF5 parallel support: `$ CC=mpicc CPPFLAGS=-I${H5DIR}/include LDFLAGS=-L${H5DIR}/lib ./configure --disable-shared --enable-parallel-tests --prefix=${NCDIR}` , where H5DIR must be a parallel HDF5 installation (`--enable-parallel` in HDF5 configure).
- To build netCDF-4 with PnetCDF parallel support: `$ CC=mpicc CPPFLAGS="-I${H5DIR}/include -I${PNDIR}/include" LDFLAGS="-L${H5DIR}/lib -L${PNDIR}/lib" ./configure --enable-pnetcdf --enable-parallel-tests --prefix=${NCDIR}`

Parallel I/O support establishes a dependency on the MPI implementation. The pnetcdf library enables parallel I/O operations on files in classic formats (CDF-1 and 2), and CDF-5 format since the release of NetCDF 4.4.0.

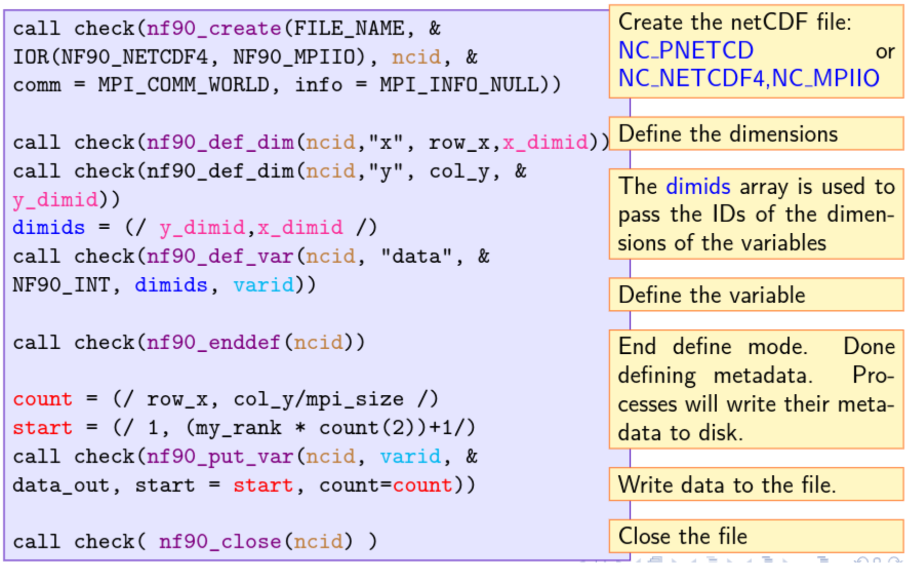
To support parallel I/O there are additional parameters and new functions:

- Fortran `nf90_create()` and `nf90_open()` have two additional optional arguments: an MPI communicator `comm`, and an `MPI_Info` object `info` (may be `MPI_INFO_NULL`)

- For switching between collective and independent access: `nf90_var_par_access(ncid, varid, access)`, where `access` may be `NF90_INDEPENDENT` or `NF90_COLLECTIVE`. The default value is independent access. This applies for writes of variables while the file is open.

Figure 15, “netCDF Parallel Example: 2D array” gives a Fortran program for a 2D array with dimensions = 5 x 8 where each process writes a subset of `5 x (dimsf(2)/mpi_size)`. The main parameters are: `count = ( 5, 2)` and `start = ( 1, mpi_rank*count(2)+1)`

Figure 15. netCDF Parallel Example: 2D array



```
call check(nf90_create(FILE_NAME, &
IOR(NF90_NETCDF4, NF90_MPIIO), ncid, &
comm = MPI_COMM_WORLD, info = MPI_INFO_NULL))

call check(nf90_def_dim(ncid,"x", row_x,x_dimid))
call check(nf90_def_dim(ncid,"y", col_y, &
y_dimid))
dimids = (/ y_dimid,x_dimid /)
call check(nf90_def_var(ncid, "data", &
NF90_INT, dimids, varid))

call check(nf90_enddef(ncid))

count = (/ row_x, col_y/mpi_size /)
start = (/ 1, (my_rank * count(2))+1/)
call check(nf90_put_var(ncid, varid, &
data_out, start = start, count=count))

call check( nf90_close(ncid) )
```

Create the netCDF file: NC_PNETCD or NC_NETCDF4,NC_MPIIO

Define the dimensions

The dimids array is used to pass the IDs of the dimensions of the variables

Define the variable

End define mode. Done defining metadata. Processes will write their metadata to disk.

Write data to the file.

Close the file

6.3. HDF5

HDF (Hierarchical Data Format) is an I/O library that serves the same purposes as NetCDF and more. As NetCDF, HDF Version 4 (HDF4) data models include annotated multidimensional arrays (called also scientific data sets), as well as raster files and lists of records. HDF4 does not support parallel I/O and files are limited to 2GB. To address these limitations, a new version was designed: HDF Version 5 (HDF5). HDF5 has no file size limitation and is able to manage files as big as the largest allowed by the operating system. Unlike classic NetCDF, HDF5 supports more than one unlimited dimension in a data type. HDF5 provides support for C, Fortran, Java and C ++ programming languages.

The HDF5 library can be compiled to provide parallel support using the MPI library. An HDF5 file can be opened in parallel from an MPI application by specifying a parallel 'file driver' with an MPI communicator and info structure. This information is communicated to HDF5 through a 'property list,' a special HDF5 structure that is used to modify the default behavior of the library.

6.3.1. HDF5 Design

HDF5 is designed at three levels:

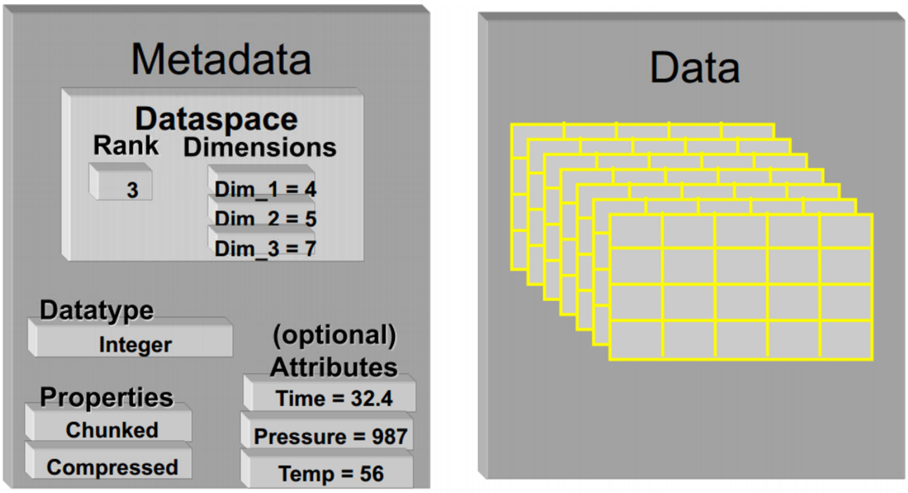
- A data model: consists of abstract classes, such as files, datasets, groups, datatypes and dataspaces, that developers use to construct a model of their higher-level concepts.

- A software library: to provide applications with an object-oriented programming interface that is powerful, flexible and high performance.

- A file format: provides portable, backward and forward compatible, and extensible instantiation of the HDF5 data model.

6.3.2. HDF5 File Organization and Data Model

Figure 16. HDF5 Dataset Model



Source: http://press3.mcs.anl.gov/computingschool/files/2014/01/QKHDF5-Intro-v2.pdf

HDF5 files are organized in a hierarchical structure, with two primary structures: groups and datasets.

- HDF5 group: a grouping structure containing instances of zero or more groups or datasets, together with supporting metadata.
- HDF5 dataset: a multidimensional array of data elements, together with supporting metadata.

As can be seen in Figure 16, “HDF5 Dataset Model”, a dataset relies on two parts: the data itself and the metadata. The metadata covers the datatype, optional attributes, data properties and the dataspace which represents the data dimensionality and the data layout. The file dataspace can be handled completely independently from the data layout within the memory.

6.3.3. Selecting a Portion of a Dataspace

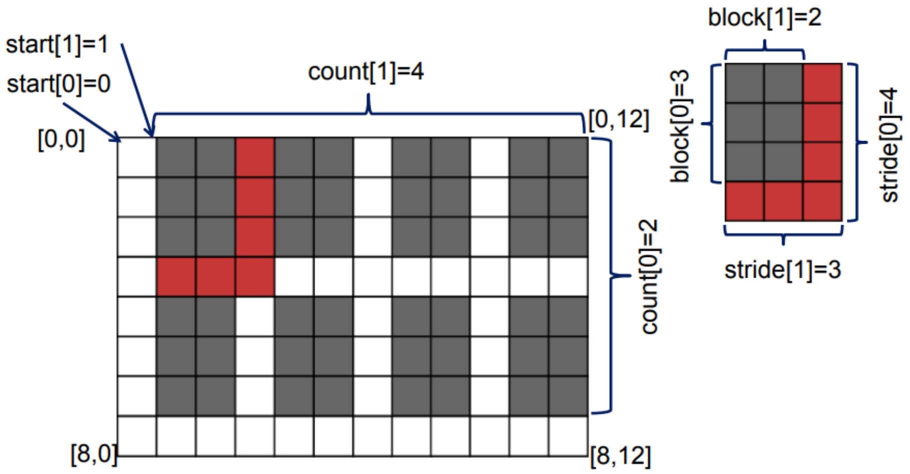
HDF5 allows reading or writing to a portion of a dataset by use of hyperslab selection. A hyperslab selection can be a logically contiguous collection of points in a dataspace, or it can be a regular pattern of points or blocks in a dataspace. A hyperslab can be selected with the function: `H5Sselect_hyperslab/h5sselect_hyperslab_f`.

Hyperslabs are described by four parameters:

- start: (or offset): starting location
- stride: separation blocks to be selected
- count: number of blocks to be selected
- block: size of block to be selected from dataspace

The dimensions of these four parameters correspond to dimensions of the underlying dataspace. Figure 17, “Hyperslab example” shows a hyperslab example and values of four parameters to select a portion of a dataset.

Figure 17. Hyperslab example



Creating a Hyperslab:

- In C: `herr_t H5Sselect_hyperslab(hid_t space_id, H5S_seloper_t op, const hsize_t *start, const hsize_t *stride, const hsize_t *count, const hsize_t *block )`
- In Fortran: `H5SSELECT_HYPERSLAB_F(SPACE_ID, OPERATOR, START, COUNT, HDFERR, STRIDE, BLOCK) INTEGER(HID_T), INTENT(IN) :: SPACE_ID INTEGER, INTENT(IN) :: OP INTEGER(HSIZE_T), DIMENSION(*), INTENT(IN) :: START, COUNT INTEGER, INTENT(OUT) :: HDFERR INTEGER(HSIZE_T), DIMENSION(*), OPTIONAL, INTENT(IN) :: STRIDE, BLOCK`

The following operators are used to combine old and new selections:

- `H5S_SELECT_SET[_F]`: Replaces the existing selection with the parameters from this call. Overlapping blocks are not supported with this operator.
- `H5S_SELECT_OR[_F]`: Adds the new selection to the existing selection.
- `H5S_SELECT_AND[_F]`: Retains only the overlapping portions of the new selection and the existing selection.
- `H5S_SELECT_XOR[_F]`: Retains only the elements that are members of the new selection or the existing selection, excluding elements that are members of both selections.
- `H5S_SELECT_NOTB[_F]`: Retains only elements of the existing selection that are not in the new selection.
- `H5S_SELECT_NOTA[_F]`: Retains only elements of the new selection that are not in the existing selection.

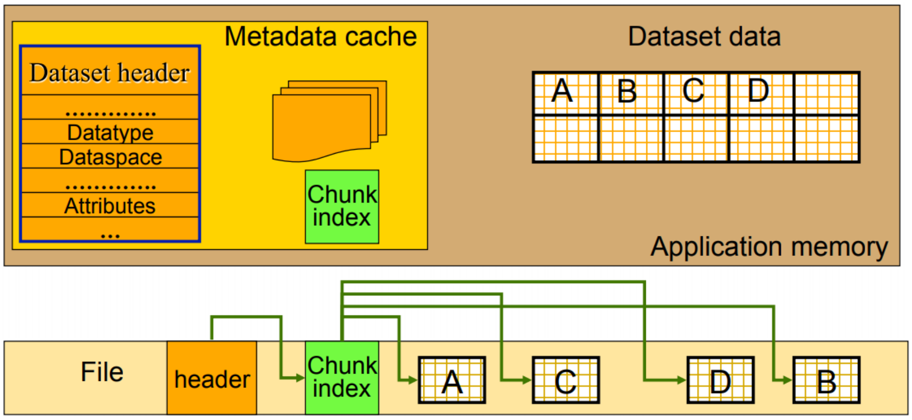
6.3.4. Chunking in HDF5

Datasets in HDF5 not only provide a convenient, structured, and self-describing way to store data, but are also designed to do so with good performance. In order to maximise performance, the HDF5 library provides ways to specify how the data is stored on disk, how it is accessed, and how it should be held in memory. The way in which the multidimensional dataset is mapped to the serial file is called the layout. The most obvious way to accomplish this is to simply flatten the dataset in a way similar to how arrays are stored in memory, serialising the entire dataset into a monolithic block on disk, which maps directly to a memory buffer the size of the dataset. This is called a contiguous layout.

An alternative to the contiguous layout is the chunked layout. Whereas contiguous datasets are stored in a single block in the file, chunked datasets are split into multiple chunks which are all stored separately in the file. The chunks can be stored in any order and any position within the HDF5 file. Chunks can then be read and written individually, improving performance when operating on a subset of the dataset[24].

Figure 18, “Chunking Scheme”[25] shows how a file is stored using chunking in HDF5.

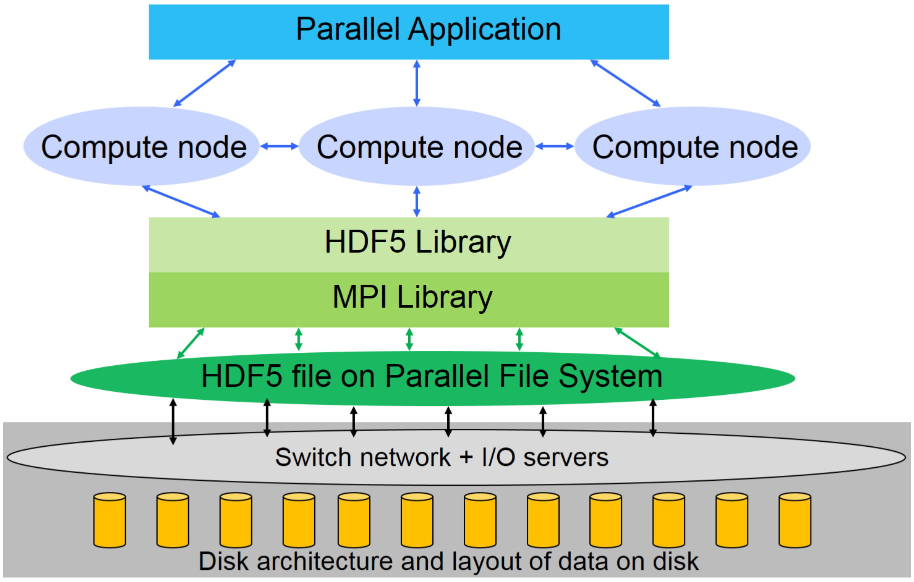
Figure 18. Chunking Scheme



6.3.5. Parallel HDF5

The parallel HDF5 (PHDF5) library is implemented upon the MPI-IO layer, meaning users can directly benefit from MPI-IO optimisation techniques such collective buffering and data sieving. Figure 19, “Parallel HDF5 Layers” shows the different layers to implement parallel HDF5.

Figure 19. Parallel HDF5 Layers


Parallel Application
Compute node
Compute node
Compute node
HDF5 Library
MPI Library
HDF5 file on Parallel File System
Switch network + I/O servers
Disk architecture and layout of data on disk

PHDF5 has the following programming restrictions:

- PHDF5 opens a parallel file with a MPI communicator,
- Returns a file ID and future access to the file is done via the file ID,
- All processes must participate in collective PHDF5 APIs and
- Different files can be opened via different communicators.

Considerations for collective calls [26]:

- All HDF5 APIs that modify structural metadata are collective
  - File operations: `H5Fcreate, H5Fopen, H5Fclose, etc`
  - Object creation: `H5Dcreate, H5Dclose, etc`
  - Object structure modification (e.g., dataset extent modification): `H5Dset_extent, etc`
- Array data transfer can be collective or independent. Dataset operations: `H5Dwrite, H5Dread`
- Collectiveness is indicated by function parameters, not by function names as in MPI.

PHDF5 presents the following characteristics:

- After a file is opened by all the processes of a communicator:
  - All parts of the file are accessible by all processes.
  - All objects in the file are accessible by all processes.
  - Multiple processes may write to the same data array (i.e. collective I/O).
  - Each process may write to individual data array (i.e. independent I/O).
- API languages: C and F90, 2003 language interfaces.
- Programming model: HDF5 uses an `access property list` to control the file access mechanism. Therefore, the general model to access HDF5 file in parallel must follow these steps: 1) set up MPI-IO file access property list, 2) open file, 3) access data and 4) close file.

6.3.6. Parallel HDF5 Example

Figure 20, “ PHDF5 Example: 2D dataset” corresponds to an 5 x 8 array dataset `dimsf = (/5,8/)`, where each process will write a subset of the data `5 x (dimsf(2) / mpi_size)`. In this example, the main parameters for the hyperslab are set as follows: `offset = (/ 0, mpi_rank * count(2) /); count = (/ 5, dimsf(2)/ mpi_size /); stride = (/ 1, 1 /)`

Figure 20. PHDF5 Example: 2D dataset

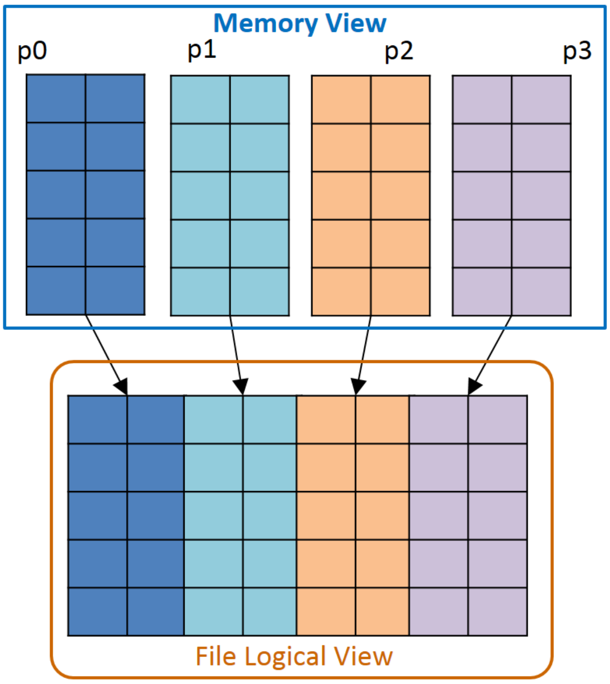
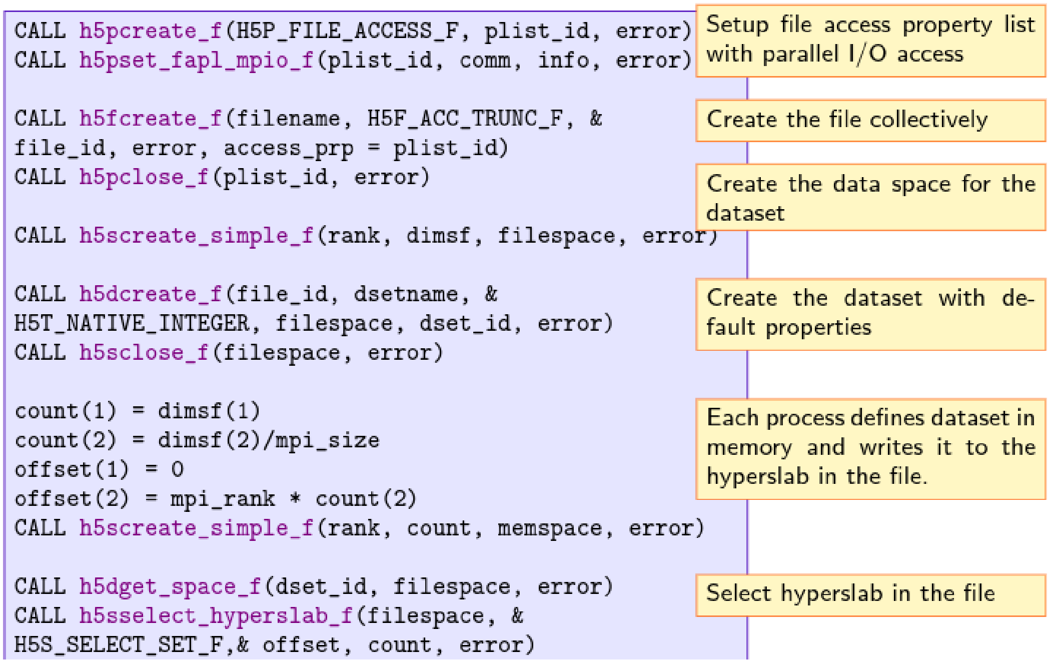


Figure 21, “ Fortran program: file, dataspace, dataset and hyperslab” is the code for the PHDF5 example that shows the file open, dataspace and dataset definition and hyperslab selection.

Figure 21. Fortran program: file, dataspace, dataset and hyperslab



```
CALL h5pcreate_f(H5P_FILE_ACCESS_F, plist_id, error)
CALL h5pset_fapl_mpio_f(plist_id, comm, info, error)

CALL h5fcreate_f(filename, H5F_ACC_TRUNC_F, &
file_id, error, access_prp = plist_id)
CALL h5pclose_f(plist_id, error)

CALL h5screate_simple_f(rank, dimsf, filespace, error)

CALL h5dcreate_f(file_id, dsetname, &
H5T_NATIVE_INTEGER, filespace, dset_id, error)
CALL h5sclose_f(filespace, error)

count(1) = dimsf(1)
count(2) = dimsf(2)/mpi_size
offset(1) = 0
offset(2) = mpi_rank * count(2)
CALL h5screate_simple_f(rank, count, memspace, error)

CALL h5dget_space_f(dset_id, filespace, error)
CALL h5sselect_hyperslab_f(filespace, &
H5S_SELECT_SET_F,& offset, count, error)
```

Setup file access property list with parallel I/O access

Create the file collectively

Create the data space for the dataset

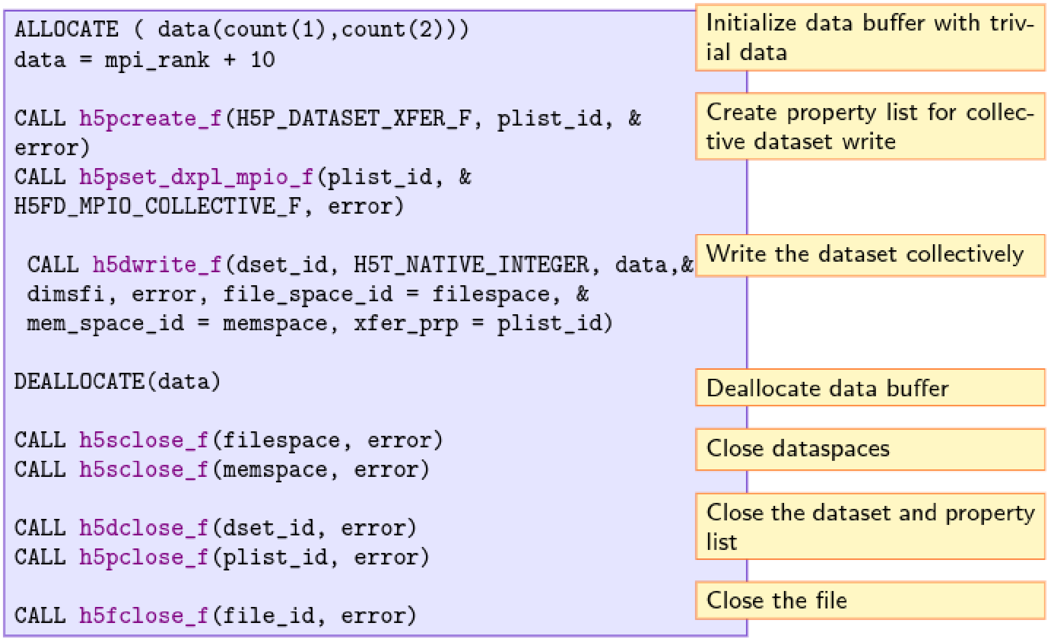
Create the dataset with default properties

Each process defines dataset in memory and writes it to the hyperslab in the file.

Select hyperslab in the file

Figure 22, “ Fortran program: property list, write and close” continues from the previous figure, showing collective operations, the writing of the data and finally the close step for all HDF5 elements.

Figure 22. Fortran program: property list, write and close



```
ALLOCATE ( data(count(1),count(2)))
data = mpi_rank + 10

CALL h5pcreate_f(H5P_DATASET_XFER_F, plist_id, &
error)
CALL h5pset_dxpl_mpio_f(plist_id, &
H5FD_MPIO_COLLECTIVE_F, error)

 CALL h5dwrite_f(dset_id, H5T_NATIVE_INTEGER, data,&
 dimsfi, error, file_space_id = filespace, &
 mem_space_id = memspace, xfer_prp = plist_id)

DEALLOCATE(data)

CALL h5sclose_f(filespace, error)
CALL h5sclose_f(memspace, error)

CALL h5dclose_f(dset_id, error)
CALL h5pclose_f(plist_id, error)

CALL h5fclose_f(file_id, error)
```

Initialize data buffer with trivial data

Create property list for collective dataset write

Write the dataset collectively

Deallocate data buffer

Close dataspaces

Close the dataset and property list

Close the file

6.4. pNetCDF

Parallel netCDF (officially abbreviated PnetCDF) is a library for parallel I/O providing higher-level data structures (e.g. multi-dimensional arrays of typed data). PnetCDF creates, writes, and reads the same file format as the serial netCDF library, meaning PnetCDF can operate on existing datasets, and existing serial analysis tools can process PnetCDF-generated files. PnetCDF is built on top of MPI-IO, which guarantees portability across various platforms and high performance. Originally PnetCDF was created as a first approach to provide parallel netCDF capabilities on top of netCDF-3. However with netCDF-4 a separate approach was implemented on top of HDF5. That's why two approaches are available today. However only PnetCDF is able to read netCDF-3 files in parallel (netCDF-4 uses PnetCDF underneath if available).

In order for easy code migration from sequential netCDF to PnetCDF, PnetCDF APIs mimic the syntax of the netCDF APIs with only a few changes to add parallel I/O concept. These changes are highlighted as follows:

- All parallel APIs are named after the originals with prefix of "ncmpi_" for C/C++, "nfmpi_" for Fortran 77, and "nf90mpi_" for Fortran 90.
- An MPI communicator and an MPI_Info object are added to the argument list of the open/create APIs
- PnetCDF allows two I/O modes, collective and independent, which correspond to MPI collective and independent I/O operations. Similar to the MPI naming convention, all collective APIs carry an extra suffix "_all". The independent I/O mode is wrapped by the calls of ncmpi_begin_indep_data() and ncmpi_end_indep_data().

6.5. SIONLib

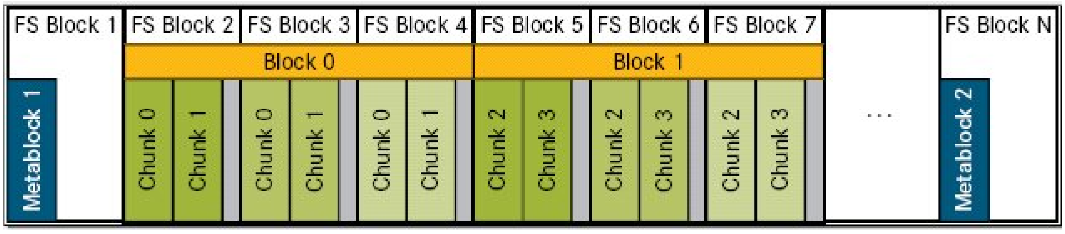
SIONlib is a scalable I/O library for parallel access to task-local files. The library allows mixing the good performance of a task local approach together with a small amount of created files. Data stored within a SIONlib fileset is a direct byte stream, no additional metadata information is stored by SIONlib. Due to the close connection to task-local files a separate post-processing step might be necessary to create a global data view of multiple local data views. The library provides different interfaces: parallel access using MPI, OpenMP, or their combination and sequential access for post-processing utilities.

6.5.1. SIONLib file format

One of the strategies for SIONlib to increase I/O performance is preventing file system block contention, i.e. different tasks trying to modify the same file system block at the same time. To avoid this, SIONlib needs additional information from the user when opening a file. The chunksize supplied during the open call is communicated globally and lets SIONlib calculate the ranges inside the file which belongs to each task. In case where there is not enough space left for a write request in the current block, SIONlib skips all the file ranges that belong to other tasks and has new chunksize bytes of space available[27].

Figure 23, “ SIONLib file layout”[27] shows file format and the different concepts used to its implementation.

Figure 23. SIONLib file layout



6.5.2. SIONLib API and utilities

The general API includes:

- Parallel interface: `sion_paropen_mpi, sion_parclose_mpi`.
- Serial interface: `sion_open, sion_open_rank, sion_close, sion_get_locations`.
- Common interface: `sion_ensure_free_space, sion_feof, sion_bytes_avail_in_block, sion_seek, sion_seek_fp, sion_fwrite, sion_fread`.

Detailed API description can be found in [28].

SIONLib also provides utilities to file manipulation[29]:

- `siondump`: Dump meta data information of a SIONlib file.
- `sionsplit`: Split one SIONlib file into separate files.
- `siondefrag`: De-fragment a SIONlib file.
- `sioncat`: Extract all data or data of one tasks.
- `sionconfig`: Print compiler and linker flags.
- `partest`: Benchmark using SIONlib.

6.6. Touching on Low-level I/O: POSIX, C and Fortran File Manipulation

The POSIX OS standard, C standard library and Fortran provide different methods for manipulating files in parallel applications. Although they are broadly discouraged for HPC applications due to the availability of the higher-level libraries detailed in this chapter, they do see use regardless. In this section, a brief overview of POSIX, C and Fortran file operations is given, as well as optimisation techniques that should be considered when I/O is performed with these interfaces. Note that threading or other concurrency models for low-level I/O are not covered in this section.

6.6.1. POSIX and the C Standard Library

In the C library, a stream is represented by a FILE* pointer. File closing is done with the `fclose` function. Functions such as `fputc`, `fputs`, and `fwrite` can be used to write data to the stream, while `fscanf`, `fgetc`, `fgets`, and `fread` read data. For lower-level I/O operations in POSIX operating systems such as Linux, a handle called a file descriptor is used instead of a FILE* pointer. A file descriptor is an integer value that refers to a particular instance of an open file in a single process. The function `open` creates a file descriptor and subsequent `read`, `write` and `close` functions take this as an argument.

Opening a file:

- `FILE *fopen(const char *path, const char *mode);`
- `FILE *fdopen(int fildes, const char *mode);`
- `int open(const char *pathname, int flags [, mode_t mode]);`

The function `fdopen()` converts an already open file descriptor (`fildes`) to a stream. The `flags` arguments to `open` refers to the access mode, for example:

- `O_DIRECTORY, O_NOFOLLOW` (enforce directory / not-a-link)
- `O_SYNC, O_ASYNC` (synchronous vs. asynchronous)
- `O_LARGEFILE, O_DIRECT` (large files / bypass cache)
- `O_APPEND, O_TRUNCO`

The `mode` argument for `open` sets file access permissions and for `fopen` refers to read/write/read-write and so on.

It is recommended to perform unformatted `fwrite()/fread()` calls rather than formatted I/O `fprintf()/fscanf()`. For repositioning within the file use `fseek()`.

6.6.1.1. O_DIRECT flag

Using the O_DIRECT flag when opening a file will bypass the buffer cache and send data directly to the storage system. This can be useful in some special cases to avoid memory copies and improve multi-client consistency/parallel access. When performing direct I/O, the request length, buffer alignment, and file offsets generally must all be integer multiples of the underlying device's sector size. There is no block size requirement for O_DIRECT on NFS, although this is not the case for local filesystems and IBM Spectrum Scale.

For example, Spectrum Scale may provide some performance benefits with direct I/O if:

- The file is accessed at random locations.
- There is no access locality.

In which case, direct transfer between the user buffer and the disk can only happen if all of the following conditions are also true:

- The number of bytes transferred is a multiple of 512 bytes.
- The file offset is a multiple of 512 bytes.
- The user memory buffer address is aligned on a 512-byte boundary.

6.6.1.2. Buffering

C-standard I/O implements three types of user buffering, and provides developers with an interface for controlling the type and size of the buffer. There are three types:

- Unbuffered: No user buffering is performed. Data is submitted directly to the kernel.
- Line-buffered: Buffering is performed on a per-line basis (default for e.g., stdout). With each newline character, the buffer is submitted to the kernel.
- Block-buffered: Buffering is performed on a per-block basis. This is ideal for files. By default, all streams associated with files are block-buffered. Standard I/O uses the term full buffering for block buffering.

Additionally, C-standard I/O provides an interface for controlling the type of buffering:

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

This must be done after opening a file but before any I/O operations. The buffering mode is controlled by macros: `_IONBF` (Unbuffered), `_IOLBF` (Line-Buffered) or `_IOFBF` (Block-buffered). The default buffer size for block buffering is `BUFSIZ`, defined in `stdio.h`. Function `fflush()` can be used to force block out early.

Example code snippet demonstrating changing the buffering type:

```
#define BUF 100000000
double data[SIZE];
char* myvbuf;
FILE* fp;
fp=fopen(FILENAME, "w");
myvbuf = (char *) malloc(BUF)
setvbuf(fp, myvbuf,_IOFBF,BUF);
fseek(fp, 0, SEEK_SET);
// start of file
fwrite(data, sizeof(double),SIZE, fp);
// close file, then deallocate buffer
```

6.6.1.3. POSIX Hints

The POSIX standard allows certain hints to be given to the OS. This specifies the future intentions for how a file will be manipulated to influence buffer cache behaviour and read-ahead. The actual effect of these hints is implementation specific - even different versions of the Linux kernel may react dissimilarly.

```
#include <fcntl.h>
int posix_fadvise (int fd, off_t offset, off_t len, int advice);
```

A call to `posix_fadvise()` provides the OS with the hints for file descriptor `fd` in the interval `[offset,offset+len)`. If len is 0, the advice will apply to the range `[offset,length of file]`. Common usage is to specify 0 for len and offset, applying the advice to the entire file.

Available options for `advice` are one of the following [2] :

- `POSIX_FADV_NORMAL`: The application has no specific advice to give on this range of the file. It should be treated as normal. The kernel behaves as usual, performing a moderate amount of readahead.
- `POSIX_FADV_RANDOM`: The application intends to access the data in the specified range in a random (non-sequential) order. The kernel disables readahead, reading only the minimal amount of data on each physical read operation.
- `POSIX_FADV_SEQUENTIAL`: The application intends to access the data in the specified range sequentially, from lower to higher addresses. The kernel performs aggressive readahead, doubling the size of the readahead window.
- `POSIX_FADV_WILLNEED`: The application intends to access the data in the specified range in the near future (asynchronous prefetch). The kernel initiates readahead to begin reading into memory the given pages.
- `POSIX_FADV_NOREUSE`: The application intends to access the data in the specified range in the near future, but only once.
- `POSIX_FADV_DONTNEED`: The application does not intend to access the pages in the specified range in the near future. The kernel evicts any cached data in the given range from the page cache.

6.6.2. Fortran Files

In fortran, a file is associated with a logical device which is in turn associated with a file by a unit specifier (UNIT=). A unit is connected or linked to a file through the OPEN statement in standard Fortran. Filenames are given with the FILE= specifier in the OPEN statement:

```
OPEN(UNIT=11, FILE="filename", options)
```

Options and considerations for making Fortran I/O more efficient:

- Specify the operation you intend with the ACTION keyword: read, write or both for `ACTION='READ' / 'WRITE' / 'READWRITE'`
- Perform direct access (`ACCESS='DIRECT'`) with a large maximum record length (`RECL=rl`). If possible, making `rl` a multiple of the disk block size. Note that data in direct-access files can be read or written to in any order and that records are numbered sequentially, starting with record number 1. The units used for specifying record length depend on the form of the data:
  - Formatted files (`FORM= 'FORMATTED'`): Specify the record length in bytes.
  - Unformatted files (`FORM= 'UNFORMATTED'`): Specify the record length in 4-byte units.
- Consider using unformatted files (`FORM='UNFORMATTED'`) for the following reasons:
  - Unformatted data avoids the translation process, so I/O tends to be faster.
  - Unformatted data avoids the loss of precision in floating-point numbers when the output data will subsequently be used as input data.
  - Unformatted data conserves file storage space (stored in binary form).

- If you need sequential formatted access, remember to access data in large chunks.

6.6.2.1. I/O Formatting

Formatted I/O can be the list-directed (i.e. using the Fortran default formats):

```
write(unit,fmt=*)
```

or `fmt` can be given a format string representing a statically compiled or dynamically generated format:

```
write(unit,fmt='(es20.13)')
```

```
write(unit,fmt=iof)
```

For unformatted I/O, it is possible to use sequential:

```
write(unit)
```

or direct access

```
write(unit, rec=i)
```

As stated, unformatted I/O is recommended over formatted I/O due to the performance benefits.

6.6.2.2. Conversion of Fortran Unformatted Files

Although formatted data files are more easily ported to other systems, Intel Fortran can convert unformatted data in several formats.

Unformatted output may not be readable by C or other Fortran processors (Fortran record markers) or may not be transferable between different platforms due to differing byte order. If the data consists of only intrinsic types, certain non-standard compiler extensions are available which may help in moving binary files between platforms.

By default, the Intel compiler uses little-endian format but it is possible to write unformatted sequential files in big-endian format, as well as read files produced in big-endian format by using the little-endian-to-big-endian conversion feature:

```
F_UFMTENDIAN=MODE | [MODE;] EXCEPTION
```

with the following options:

```
MODE = big | little
```

```
EXCEPTION = big:ULIST | little:ULIST | ULIST
```

```
ULIST = U | ULIST,U
```

```
U = decimal | decimal -decimal
```

Examples:

`F_UFMTENDIAN=big;` File format is big-endian for all units.

`F_UFMTENDIAN=big:9,12` ;big-endian for units 9 and 12, little-endian for others.

`F_UFMTENDIAN="big;little:8"` ;big-endian for all except unit 8.

6.6.2.3. I/O Pattern Issues

To eliminate unnecessary overhead, write whole arrays or strings at once rather than individual elements in different operations. Each item in an I/O list generates its own calling sequence. This processing overhead becomes most significant in an implicit loop. When accessing whole arrays, use the array name (Fortran array syntax) instead of using an implicit loop. The following snippets give the code for each of these I/O approaches. Only the last option is recommended with the others given as examples of what to avoid.

Implicit loop

```
write(...) ((a(i,j),i=1,m),j=1,n)
```

Array section

```
write(...) a(1:m,1:n)
```

Complete Array

```
write(...) a
```

Use the natural ascending storage order whenever possible. This is column-major order, with the leftmost subscript varying fastest and striding by 1. If the whole array is not being written, natural storage order is the best order possible.

6.6.2.4. Tuning Fortran Using the Intel Compiler

Buffering

In an Intel-compiled Fortran application, any records, read or written, are transferred between the user's program buffers and one or more disk block I/O buffers. These buffers are established when the file is opened by the Intel Fortran Run-Time Library. Unless very large records are being read or written, multiple logical records can reside in the disk block I/O buffer when it is written to disk or read from disk, minimising physical disk I/O.

The `OPEN` statement `BUFFERCOUNT` keyword specifies the number of I/O buffers. The default for `BUFFERCOUNT` is 1. Any experiments to improve I/O performance should increase the `BUFFERCOUNT` value and not the `BLOCKSIZE` value, to increase the amount of data read by each disk I/O.

If the `OPEN` statement has `BLOCKSIZE` and `BUFFERCOUNT` specifiers, then the internal buffer size in bytes is the product of these specifiers. If the open statement does not have these specifiers, then the default internal buffer size is 8192 bytes.

- `BLOCKSIZE=<bytes>` specifier (rounded up to multiples of 512).
- `BUFFERCOUNT=<count>` specifier (default 1, at most 127 possible).

To enable buffered writes; that is, to allow the disk device to fill the internal buffer before the buffer is written to disk, use one of the following:

- The `OPEN` statement `BUFFERED` specifier.
- The `FORT_BUFFERED` run-time environment variable.

Intercepting the Run-time's libc calls (Linux)

The Intel Fortran Run-time Library allows uses of the `USEROPEN` specifier in an `OPEN` statement to pass control to a routine that directly opens a file. The called routine can use system calls or library routines to open the file and establish special context that changes the effect of subsequent Intel Fortran I/O statements.

The `USEROPEN` specifier takes the following form:

```
USEROPEN = function-name
```

function-name is the name of an external function; it must be of type INTEGER(4) or (INTEGER*4).

The external function can be written in Fortran, C, or other languages.

For example, the following Intel Fortran code might be used to call the `USEROPEN` procedure `UOPEN` (associated with linker uopen_):

```
EXTERNAL UOPEN
INTEGER UOPEN
.
.
.
OPEN (UNIT=10, FILE='/usr/test/data', STATUS='NEW', USEROPEN=UOPEN)
```

Where `uopen_` should be:

```
int uopen_ ( char *file_name, int *open_flags, int *create_mode, int
*unit_num, int filenam_len )
```

An implementation done in C must call:

```
result = open(fname, *oflags, *cmode);
```

And can do other things such buffering and so on.

6.7. I/O Libraries Summary

- The netCDF classic data model is simple and flat composed by Dimensions, Variables and Attributes. The netCDF enhanced data model adds primitive types, multiple unlimited dimensions, hierarchical groups and user-defined data types.

- The HDF5 data model has even more features such as non-hierarchical groups, user-defined primitive data types, References (pointers to objects and data regions in a file) and Attributes attached to user-defined types.

- HDF5 has a more complex structure therefore it is more powerful and flexible than NetCDF. However, this also may have disadvantages because it more complex and possibly error-prone to develop against (difficult call sequence). Simplification is possible by using the HDF5 “lite“ high level interface. H5LT makes usage easier by providing a way to aggregate several API calls. Also image processing with H5IM provides a standard storage scheme for data which can be interpreted as images, e.g. 2-dimensional raster data. From version 1.6 to 1.8, the API has undergone evolution. HDF5-1.10.x contains several important new features for Parallel I/O. Performance issues for parallel I/O can be found in [30] that provides several techniques to improve performance for the different elements of HDF5 and taking into account the file system.

- SIONLib optimises binary one-file-per-processes approach by usage of a shared container file using a custom file format, as is more convenient where portability is not the main priority. Furthermore, SIONLib provides a transparent mechanism to avoid file system block contention.

- Specialised I/O libraries may provide more a portable way of writing data and may reduce metadata load when properly used.

- For parallel programs the output to separate files for each process can provide high throughput, but usually needs post-processing.

- Binary files may need to use library/compiler support for conversion. If binary files are transferred between different architectures (little vs.big-endian byte order) then the limitations may apply on file sizes and data types.

7. I/O Performance Analysis

7.1. Introduction

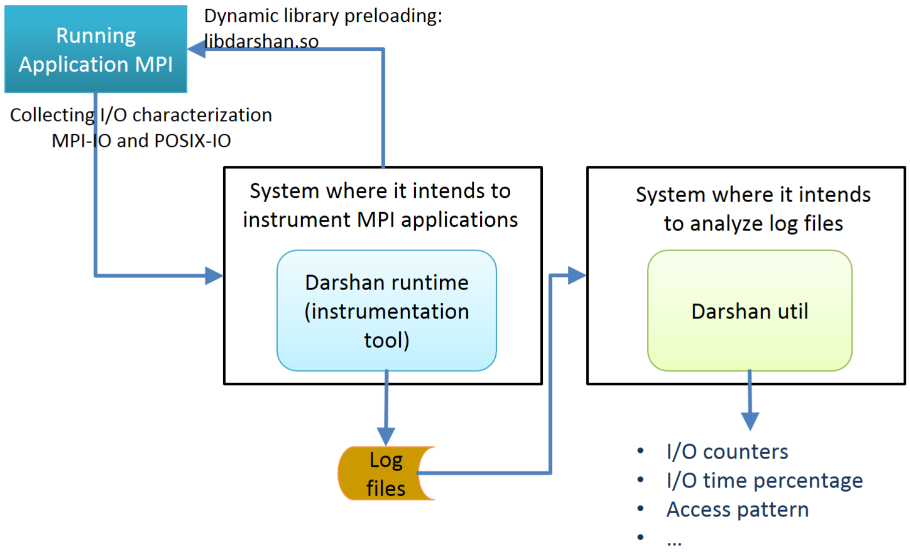
Parallel I/O performance evaluation for HPC applications is a nontrivial task because it depends on the I/O software stack, variety of application I/O patterns, hardware configuration and heterogeneity of the I/O system infrastructure.

I/O profiling tools characterise the I/O performance of HPC applications by counting I/O-related events. This is less intrusive than full tracing and is useful for identifying potential I/O bottlenecks in performance. However, more information is required for a fully-detailed understanding of I/O. Currently, the most popular tool in the HPC community is Darshan [15]. Darshan is especially designed for the HPC-IO field by providing small log files and global counters for the MPI-IO and POSIX-IO.

Conversely, we have tracing tools such as TAU [20] and Vampir [21] , that save individual event records with precise timestamps and per process, log the timing of each I/O function calls and their arguments, and construct a complete timeline.

7.2. Darshan Tool

Figure 24. Darshan Overview


Running Application MPI
Dynamic library preloading: libdarshan.so
Collecting I/O characterization MPI-IO and POSIX-IO
System where it intends to instrument MPI applications
Darshan runtime (instrumentation tool)
Log files
System where it intends to analyze log files
Darshan util
I/O counters
I/O time percentage
Access pattern
...

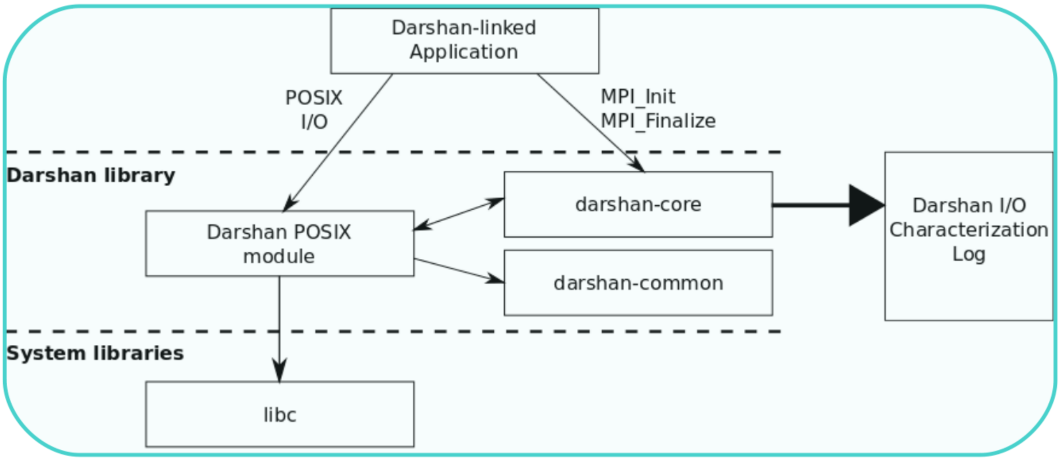
Darshan is a lightweight, scalable I/O characterisation tool that transparently captures I/O access pattern information from production applications. It was developed by the Argonne Leadership Computing Facility (ANL). Darshan provides I/O profile for C and Fortran calls including: POSIX and MPI-IO (and limited to HDF5 and PnetCDF). Darshan does not provide information about the I/O activity along the runtime. It uses a LD_PRELOAD mechanism to wrap the I/O calls.

Figure 24, “Darshan Overview” shows Darshan's two components: darshan-runtime and darshan-util. Darshan-runtime must be installed in the HPC system where the application is executed. Darshan-util can be in another machine. Using Darshan utilities and pdflatex it is possible to obtain plots with the main I/O metrics for a parallel application.

7.3. Darshan Runtime

Starting with version 3.x, the Darshan runtime environment and log file format have been redesigned such that new "instrumentation modules" can be added without breaking existing tools. Darshan can then manage these modules at runtime and create a valid Darshan log regardless of how many or what types of modules are used [19]. Figure 25, “Darshan Runtime Enviroment” depicts the components of Darshan Runtime.

Figure 25. Darshan Runtime Enviroment



7.3.1. Using the Darshan Runtime

Darshan can trace MPI applications linked statically or dynamically. The instrumentation method to use depends on whether the executables produced by the MPI compiler are statically or dynamically linked (for more detail refer to [16]).

Currently, Darshan is a tool commonly provided in HPC centers through a module system. Site-specific documentation for facilities that deploy Darshan in production can be found in [17]. Although, each center provides some specific steps to enable Darshan, there are some commonalities and relevant commands are often similar. For example, at Leibniz Supercomputing Centre (LRZ), Darshan [18] is available on all its HPC systems and is enabled by the following commands in the user submission:

1. `module load darshan`: prepare environment for both dynamically and statically linked applications. Other HPC systems often provide two modules for Darshan, e.g. `module load darshan-runtime` for I/O profiling and `module load darshan-util` for analyzing Darshan logs.

2. ``export LD_PRELOAD=`darshan-user.sh $FORTRAN_PROG` ``: load the appropiate library depending on the programming lenguage of the parallel application. For dynamically-linked executables, Darshan relies on the `LD_PRELOAD` environment variable to insert instrumentation at run time. For Fortran applications compiled with MPICH, users may have to take the additional step of adding libfmpich.so to the `LD_PRELOAD` environment variable. At LRZ, this is checked by the `$FORTRAN_PROG` variable, so that the appropiate library is loaded.

3. ``export JOBID_LL=`darshan-JOBID.sh $LOADL_STEP_ID`; export DARSHAN_JOBID=JOBID_LL``: Darshan gives the log file a name based on the job identifier assigned by the job management system, to facilitate correlating logs with a specific job. At LRZ, the `DARSHAN_JOBID` environment variable is set with the Loadleveler identifier in SuperMUC. In a Linux Cluster with Slurm as the job management system `DARSHAN_JOBID` is set to `SLURM_JOB_ID`.

4. ``export LOGPATH_DARSHAN_LRZ=`darshan-logpath.sh` ``: set up the folder to save the Darshan logs. This option is available if darshan-runtine was built with `--with-log-path-by-env`. Otherwise, the `darshan-mk-log-dirs.pl` utility is applied to specify the path at configure time to include log file subdirectories organised by year, month, and day. At LRZ by default log files are placed in `$SCRATCH/.darshan-log`, but users can change the log path by setting `LOGPATH_DARSHAN_LRZ` to a more convenient folder.

Once the application executes, a log file is generated in log path, e.g. `LOGPATH_DARSHAN_LRZ`. If the execution finishes without errors, the user can analyze the *.darshan file by using darshan-util tools. The log file can also be analysed on another system.

7.4. Darshan Util

Darshan provides command line tools that enable the analysis of I/O performance metrics. Keys tools:

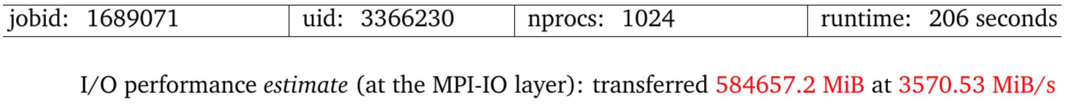
- `darshan-job-summary.pl`: creates a pdf with graphs useful for initial analysis.
- `darshan-summary-per-file.sh`: creates a separate pdf for each file opened by the application
- `darshan-parser`: dumps all information into ASCII (text) format.

```
>darshan-parser --help
Usage: darshan-parser [options] <filename>
--all   : all sub-options are enabled
--base  : darshan log field data [default]
--file  : total file counts
--file-list  : per-file summaries
--file-list-detailed  : per-file summaries with additional detail
--perf  : derived perf data
--total : aggregated darshan field data
```

7.4.1. Darshan Plots: FLASH-IO Benchmark

FLASH-IO [36] is a block-structured adaptive mesh hydrodynamics code. The computational domain is divided into blocks which are distributed across the processors. Typically a block contains 8 zones in each coordinate direction (x,y,z) and a perimeter of guardcells (presently 4 zones deep) to hold information from the neighbors. FLASH-IO will produce a checkpoint file (containing all variables in 8-byte precision) and two plotfiles (4 variables, 4-byte precision, one containing corner data, the other containing cell-centered data). The plotfiles are smaller than the checkpoint file.

Figure 26. Job Information and Performance



jobid: 1689071	uid: 3366230	nprocs: 1024	runtime: 206 seconds

I/O performance *estimate* (at the MPI-IO layer): transferred 584657.2 MiB at 3570.53 MiB/s

Figure 26, “ Job Information and Performance” shows the ID, number of MPI processes and runtime of a FLASH-IO job. The Darshan pdf file associated with this job will include executable name and date as a header, and the binary location and commands used as footer.

Figure 27. Average I/O Cost and Operation Count

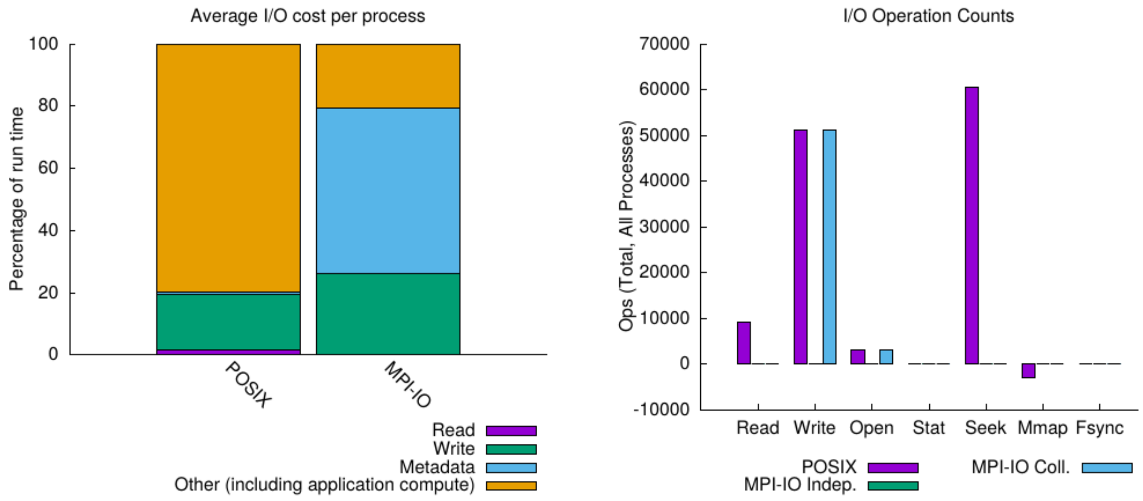


Figure 27, “ Average I/O Cost and Operation Count” presents the average I/O per process. This is the initial metric to consider when determining if an application has I/O problems. Usually, an I/O percentage greater than 10% indicates that the application requires I/O performance improvements. Also, it is possible to observe whether that percentage relates to data access (write/read) or metadata(open/close) operations. In this case, FLASH-IO is using parallel HDF5 and is dominated by metadata performance (the most significant component of the MPI-IO bar). Other listed metrics are the I/O operation counts, which indicate the MPI-IO operations are collective. These MPI-IO operations are implemented by lower-level POSIX operations such as read, write, seek, etc., as indicated in the figure. It is important to note that FLASH-IO only writes three files but the I/O library performs additional read and seek operations.

Figure 28. Access Sizes at POSIX and MPI-IO level

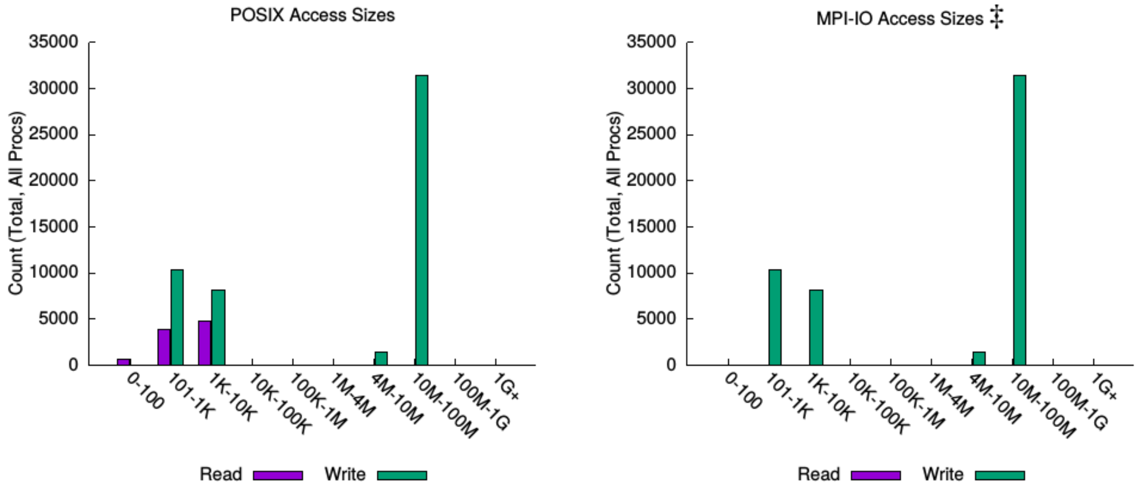
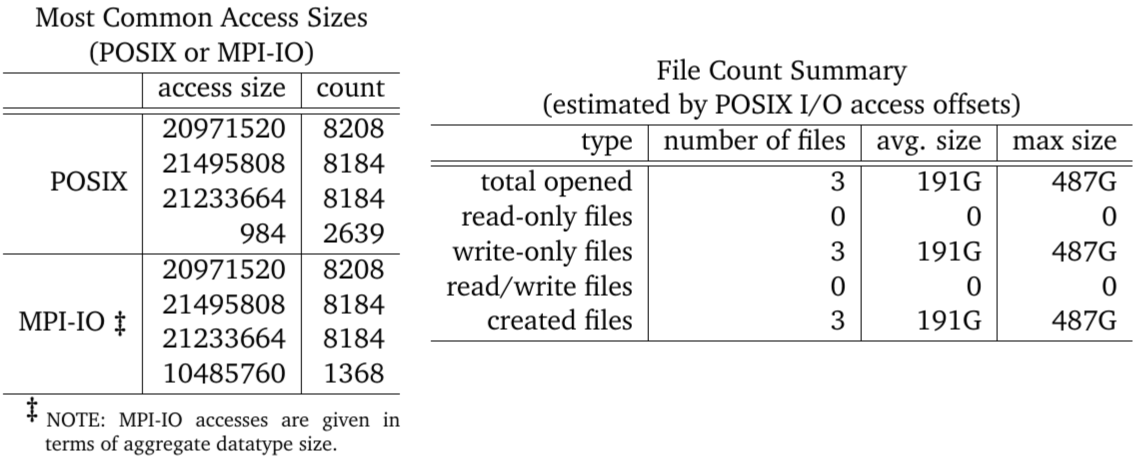


Figure 28, “ Access Sizes at POSIX and MPI-IO level” shows a histogram of MPI-IO and POSIX reads/write and their associated sizes. This information is useful for identifying small I/O operations. Reasons for small I/O include poorly implemented I/O or, occasionally, incorrect optimisation attempts by an I/O library which changes operation sizes and negatively affects the I/O pattern.

Figure 29. Common Access Sizes and File Count



Most Common Access Sizes
(POSIX or MPI-IO)

	access size	count
POSIX	20971520	8208
	21495808	8184
	21233664	8184
	984	2639
MPI-IO ‡	20971520	8208
	21495808	8184
	21233664	8184
	10485760	1368

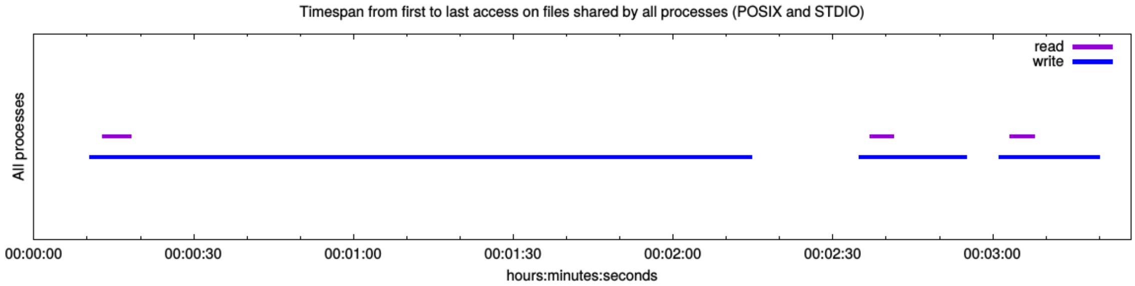
‡ NOTE: MPI-IO accesses are given in terms of aggregate datatype size.

File Count Summary
(estimated by POSIX I/O access offsets)

type	number of files	avg. size	max size
total opened	3	191G	487G
read-only files	0	0	0
write-only files	3	191G	487G
read/write files	0	0	0
created files	3	191G	487G

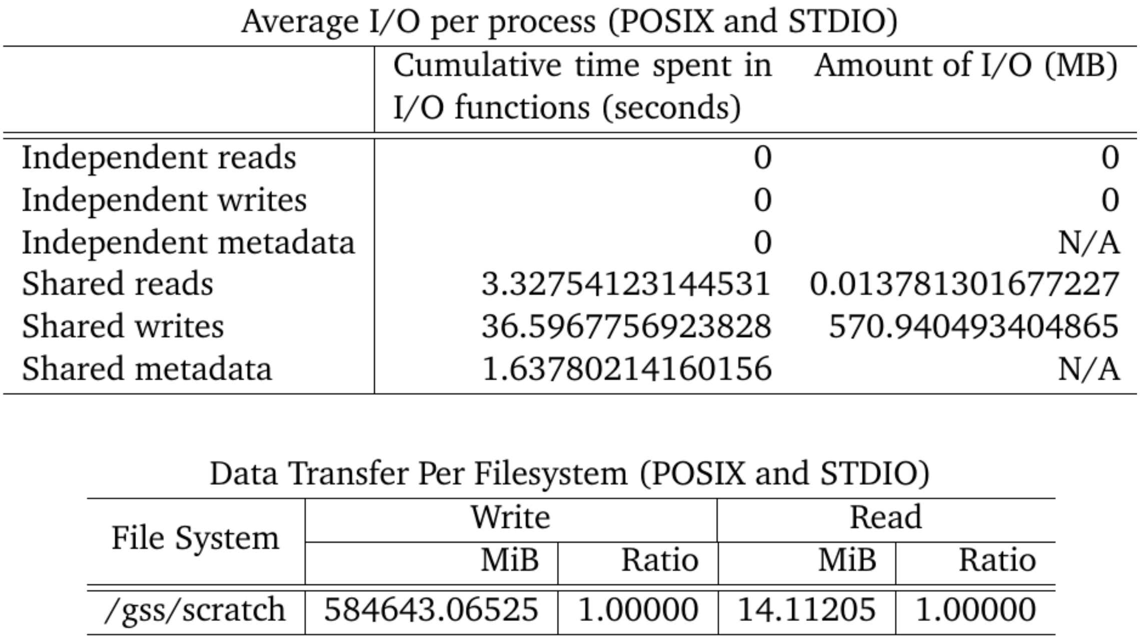
Figure 29, “ Common Access Sizes and File Count” presents the most common sizes for MPI-IO and POSIX. This information helps confirm whether collective operations are being performed or whether this has been disabled by the I/O library. Usually, the MPI-IO implementation attempts to apply an appropriate optimisation technique based on the I/O pattern. Further information in this figure mainly relates to access mode and file size.

Figure 30. Timespan from first to last access on shared files



Darshan is also capable of producing a timespan of I/O activity, for example Figure 30, “ Timespan from first to last access on shared files”. These plots are useful for observing the temporal I/O pattern. Information such as I/O sequentiality or overlapping can help identify possible inefficiency in the data access operations.

Figure 31. Average I/O and Data Transfer



Average I/O per process (POSIX and STDIO)

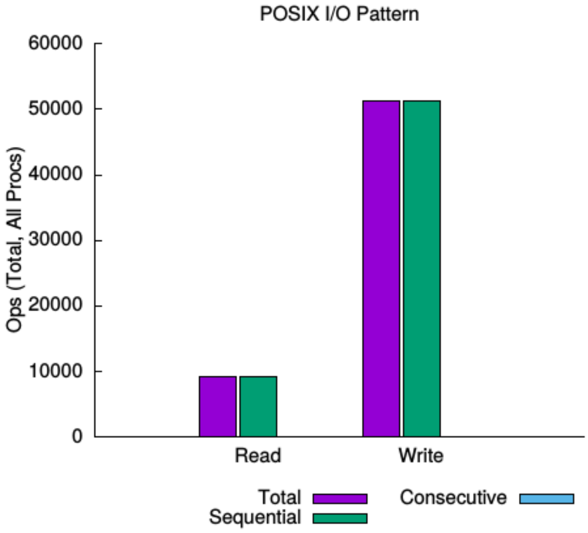
	Cumulative time spent in I/O functions (seconds)	Amount of I/O (MB)
Independent reads	0	0
Independent writes	0	0
Independent metadata	0	N/A
Shared reads	3.32754123144531	0.013781301677227
Shared writes	36.5967756923828	570.940493404865
Shared metadata	1.63780214160156	N/A

Data Transfer Per Filesystem (POSIX and STDIO)

File System	Write		Read	
	MiB	Ratio	MiB	Ratio
/gss/scratch	584643.06525	1.00000	14.11205	1.00000

In Figure 31, “ Average I/O and Data Transfer”, the average I/O per process is given, categorised by I/O strategy (shared or independent files). This information is important for perceiving possible problems at a large scale. In this figure, it is possible to observe the impact of read and metadata operations on run time and I/O size. Read operations within the I/O library could be an I/O bottleneck for a larger number of processes, optimisation at the MPI-IO level could be applied to mitigate this impact.

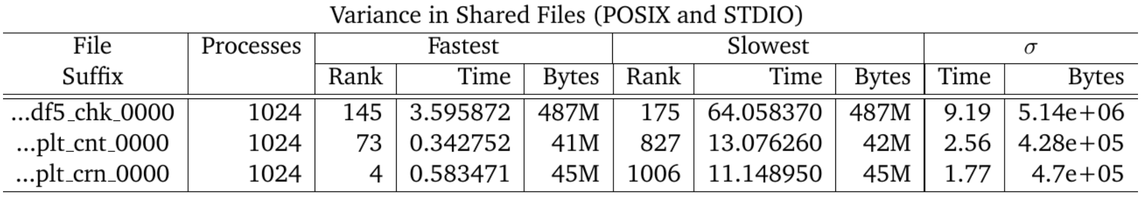
Figure 32. I/O Pattern



sequential: An I/O op issued at an offset greater than where the previous I/O op ended.
consecutive: An I/O op issued at the offset immediately following the end of the previous I/O op.

Figure 32, “ I/O Pattern” shows the I/O pattern for each file opened by the application. These plots should be analysed per file, as the noise of other files is removed and the sequential or consecutive pattern is clearer and easier to identify. Usually, a consecutive pattern is higher performance than an sequential pattern. In this case, the three files present a sequential pattern.

Figure 33. Variance in shared files



Variance in Shared Files (POSIX and STDIO)

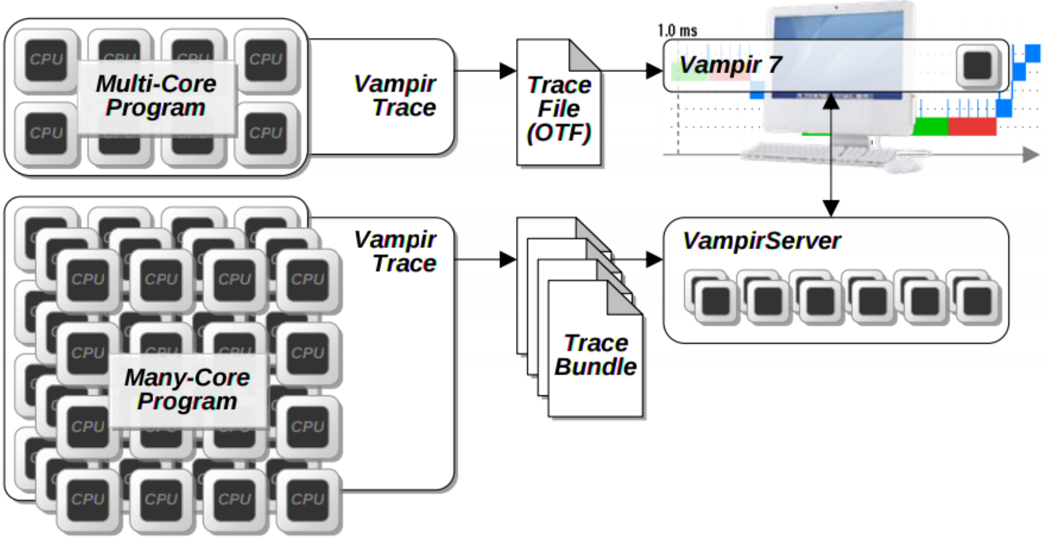
File Suffix	Processes	Fastest			Slowest			σ	
		Rank	Time	Bytes	Rank	Time	Bytes	Time	Bytes
...df5_chk_0000	1024	145	3.595872	487M	175	64.058370	487M	9.19	5.14e+06
...plt_cnt_0000	1024	73	0.342752	41M	827	13.076260	42M	2.56	4.28e+05
...plt_crn_0000	1024	4	0.583471	45M	1006	11.148950	45M	1.77	4.7e+05

Figure 33, “ Variance in shared files” presents the variance for I/O time and bytes for shared files. This is useful for identifying possible mapping problems (I/O time) or I/O imbalance (bytes).

7.5. Vampir

The Vampir performance visualisation tool consists of a performance monitor (e.g., Score-P or VampirTrace) that records performance data and a performance GUI, which is responsible for the graphical representation of the data. Figure 34, “Vampir Architecture” depicts the components of the Vampir Tool.

Figure 34. Vampir Architecture



To trace I/O events, VampirTrace is recommended as, if this is built with I/O tracing support, it intercepts calls to I/O functions of the standard C library. Therefore, it is possible to capture the serial and parallel I/O done by a parallel application.

The following functions are intercepted by VampirTrace:

```
close creat creat64 dup
dup2 fclose fcntl fdopen
fgetc fgets flockfile fopen
fopen64 fprintf fputc fputs
fread fscanf fseek fseeko
fseeko64 fsetpos fsetpos64 ftrylockfile
funlockfile fwrite getc gets
lockf lseek lseek64 open
open64 pread pread64 putc
puts pwrite pwrite64 read
readv rewind unlink write writev
```

Tracing I/O events has to be activated for each tracing run by setting the environment variable VT_IOTRACE to "yes". Setting the environment variable VT_IOTRACE_EXTENDED to "yes" enables the collection of additional function arguments for some of the I/O function mentioned above. For example, this option additionally stores offsets for pwrite and pread to the I/O event record. Enabling VT_IOTRACE_EXTENDED automatically enables VT_IOTRACE[35].

7.5.1. Using Vampir

Users can instrument a parallel application by using VampirTrace:

- compile-time instrumentation:
  - Fortran 77: `vtf77 [-g] -c <further options> myprog.f`
  - Fortran 90 and higher: `vtf90 [-g] -vt:f90 mpif90 -c <further options> myprog.f90`
  - C: `vtcc [-g] -vt:cc mpicc -c <further options> myprog.c`
  - C++: `vtcxx [-g] -c -vt:cxx mpiCC <further options> myprog.cpp`

- run-time instrumentation (vtrun): `mpirun -np 16 vtrun ./a.out`

The following environment variables must set before running the application with VampirTrace:

```
export VT_PFORM_LDIR=/gpfs/scratch/project/user/vampir-tmp
export VT_FILE_UNIQUE='yes'
export VT_IOTRACE='yes'
```

After execution a .otf file as well as a number of *.events.z files are generated. The Vampir tool is then used for visualisation and trace analysis:

- For small traces: `vampir <filename>.otf`
- For large traces: `vampirserver start -n <tasks>`

7.5.1.1. Vampir Examples: FLASH-IO and BT-IO

To demonstrate the I/O profiling information produced by Vampir, two benchmarks are detailed here. One (FLASH-IO) was previously described in the Darshan section above.

Figure 35. FLASH-IO Processes Timeline

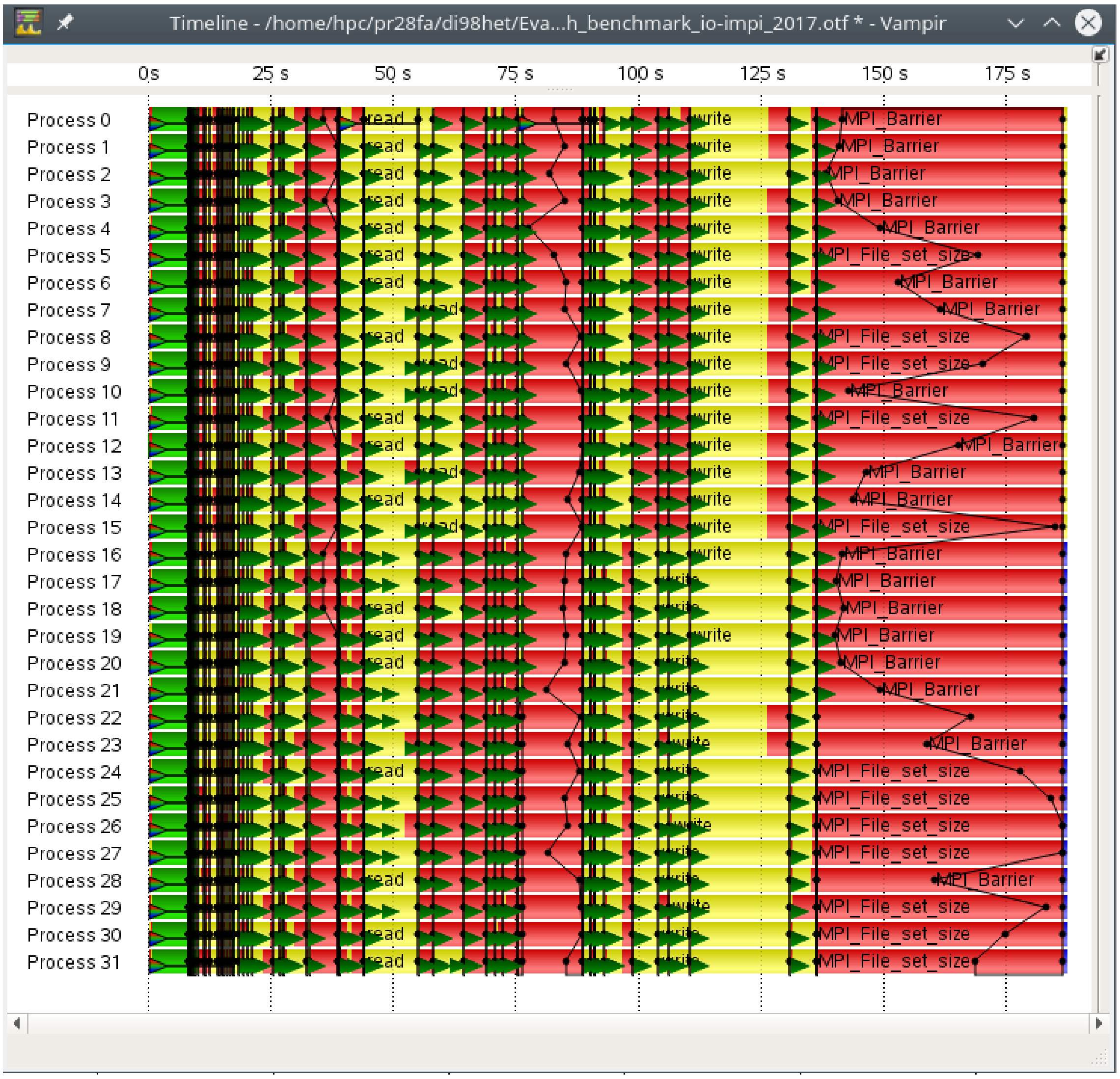
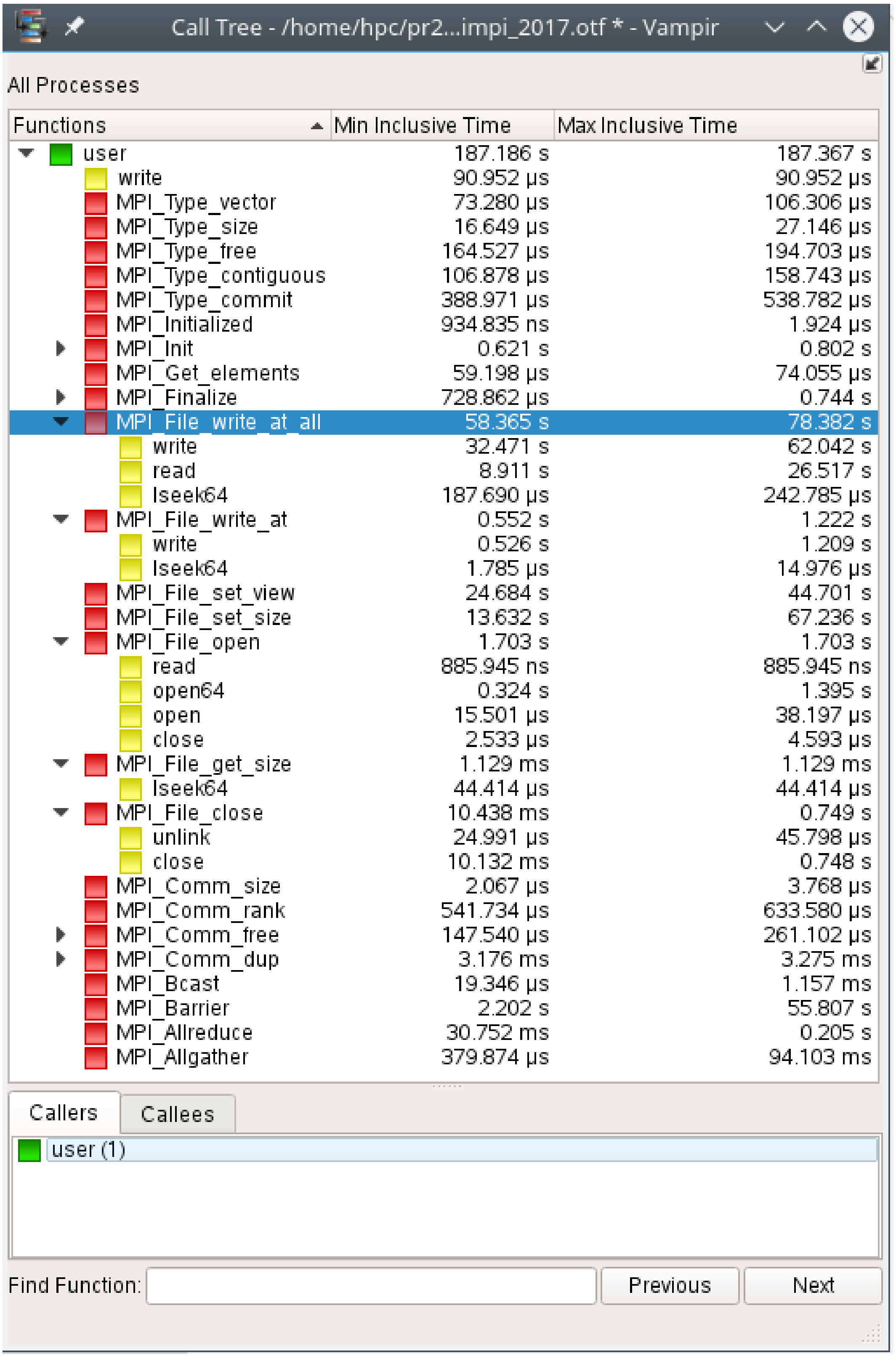


Figure 35, “ FLASH-IO Processes Timeline” depicts the execution of FLASH-IO using Intel MPI over 32 processes and HDF5 for I/O. The dark green triangles represent I/O events, yellow bars give the time of I/O operations, green bars give time of user operations, and the red bars show the time for MPI events. Here, all process are doing I/O, meaning the collective buffering optimisation technique is not being performed.

Figure 36. FLASH-IO Call Tree



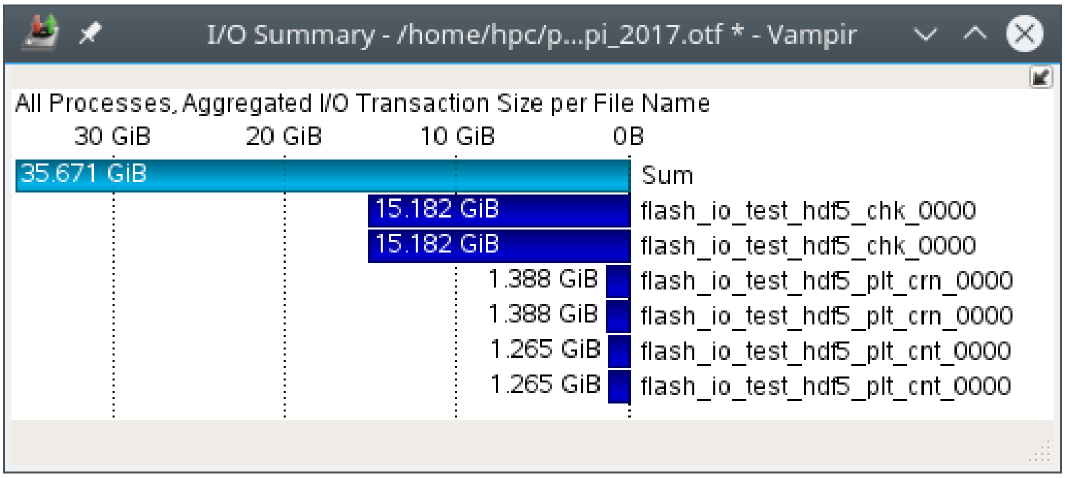
All Processes

Functions	Min Inclusive Time	Max Inclusive Time
▼ 🟩 user	187.186 s	187.367 s
🟨 write	90.952 µs	90.952 µs
🟥 MPI_Type_vector	73.280 µs	106.306 µs
🟥 MPI_Type_size	16.649 µs	27.146 µs
🟥 MPI_Type_free	164.527 µs	194.703 µs
🟥 MPI_Type_contiguous	106.878 µs	158.743 µs
🟥 MPI_Type_commit	388.971 µs	538.782 µs
🟥 MPI_Initialized	934.835 ns	1.924 µs
▶ 🟥 MPI_Init	0.621 s	0.802 s
🟥 MPI_Get_elements	59.198 µs	74.055 µs
▶ 🟥 MPI_Finalize	728.862 µs	0.744 s
▼ 🟥 MPI_File_write_at_all	58.365 s	78.382 s
🟨 write	32.471 s	62.042 s
🟨 read	8.911 s	26.517 s
🟨 lseek64	187.690 µs	242.785 µs
▼ 🟥 MPI_File_write_at	0.552 s	1.222 s
🟨 write	0.526 s	1.209 s
🟨 lseek64	1.785 µs	14.976 µs
🟥 MPI_File_set_view	24.684 s	44.701 s
🟥 MPI_File_set_size	13.632 s	67.236 s
▼ 🟥 MPI_File_open	1.703 s	1.703 s
🟨 read	885.945 ns	885.945 ns
🟨 open64	0.324 s	1.395 s
🟨 open	15.501 µs	38.197 µs
🟨 close	2.533 µs	4.593 µs
▼ 🟥 MPI_File_get_size	1.129 ms	1.129 ms
🟨 lseek64	44.414 µs	44.414 µs
▼ 🟥 MPI_File_close	10.438 ms	0.749 s
🟨 unlink	24.991 µs	45.798 µs
🟨 close	10.132 ms	0.748 s
🟥 MPI_Comm_size	2.067 µs	3.768 µs
🟥 MPI_Comm_rank	541.734 µs	633.580 µs
▶ 🟥 MPI_Comm_free	147.540 µs	261.102 µs
▶ 🟥 MPI_Comm_dup	3.176 ms	3.275 ms
🟥 MPI_Bcast	19.346 µs	1.157 ms
🟥 MPI_Barrier	2.202 s	55.807 s
🟥 MPI_Allreduce	30.752 ms	0.205 s
🟥 MPI_Allgather	379.874 µs	94.103 ms

🟩 user (1)

Vampir provides several views for performance analysis. A call tree can be produced showing the MPI-IO operations and the corresponding POSIX operations, as demonstrated in Figure 36, “ FLASH-IO Call Tree”. Here, the collective operation `MPI_File_write_at_all` is shown to be composed of POSIX write, read and lseek64 operations. This provides an explanation for the read and seek operations observed with Darshan. A similar analysis can be done for the other MPI-IO operations.

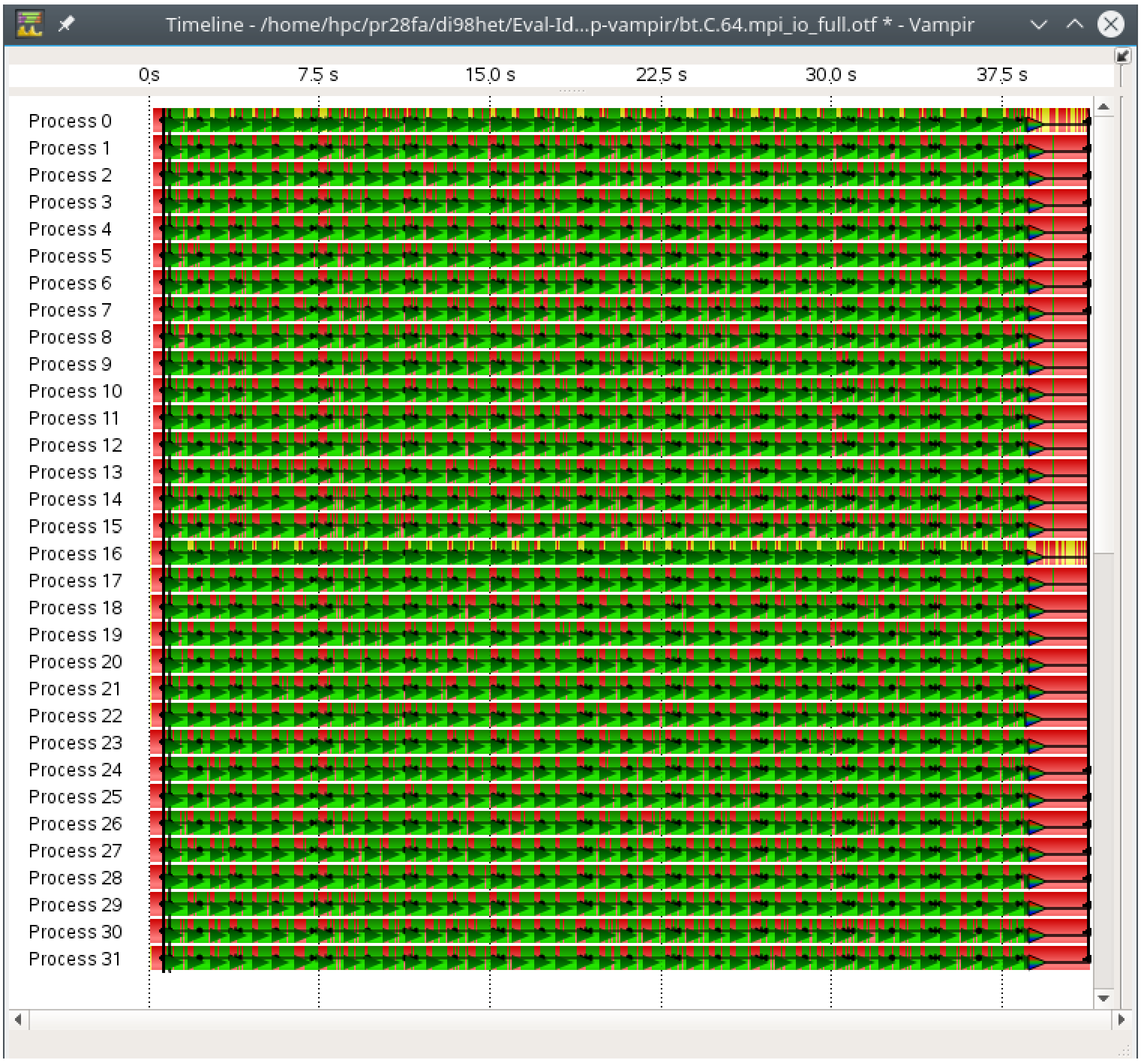
Figure 37. FLASH-IO I/O Summary


I/O Summary - /home/hpc/p...pi_2017.otf * - Vampir
All Processes, Aggregated I/O Transaction Size per File Name
30 GiB
20 GiB
10 GiB
0B
35.671 GiB
Sum
15.182 GiB
flash_io_test_hdf5_chk_0000
15.182 GiB
flash_io_test_hdf5_chk_0000
1.388 GiB
flash_io_test_hdf5_plt_crn_0000
1.388 GiB
flash_io_test_hdf5_plt_crn_0000
1.265 GiB
flash_io_test_hdf5_plt_cnt_0000
1.265 GiB
flash_io_test_hdf5_plt_cnt_0000

A further example of Vampir profiling capability is shown in Figure 37, “ FLASH-IO I/O Summary ”. Here, the aggregated I/O transaction size per file is given with Vampir providing a further set of metrics: Number of I/O Operations, Aggregated I/O Transaction Size, Aggregated I/O Transaction Time, and values of I/O Transaction Size, I/O Transaction Time, or I/O Bandwidth with respect to their selected value type. More detail about Vampir I/O summaries can be find in [22]

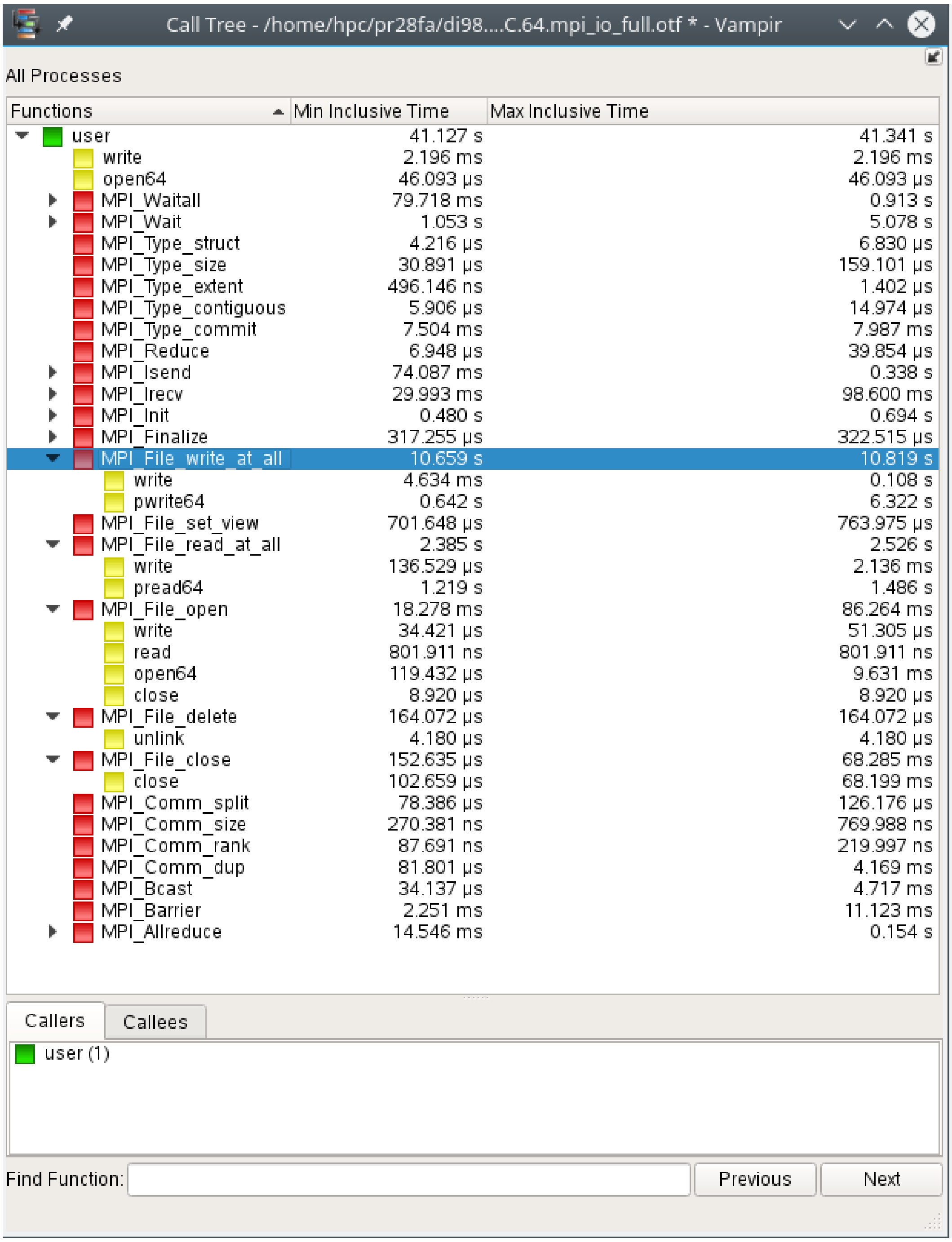
The BT-IO benchmark [37] is part of the parallel benchmark suite NPB-MPI developed by the NASA Advanced Supercomputing Division and is the second case we use to demonstrate the capabilities of Vampir. BT-IO presents a block-tridiagonal partitioning pattern on a three-dimensional array across a square number of processes. Each process is responsible for multiple Cartesian subsets of the entire data set, whose number increases with the square root of the number of processors participating in the computation. In BT-IO, forty arrays are consecutively written to a shared file by appending one after another. Each array must be written in a canonical, row-major format in the file. The forty arrays are then read back for verification using the same data partitioning [38].

Figure 38. BT-IO Processes Timeline



Here, BT-IO was executed under IBM MPI (IBM's Parallel Environment (PE)) on 64 processes, over 4 compute nodes and using MPI for I/O. Figure 38, “ BT-IO Processes Timeline” shows the timeline for the first 32 processes. It can be seen that only process 0 and 16 perform data access operations (yellow bars) as collective operations are performed and collective buffering is enabled for BT-IO explicitly. The system where this trace was performed has collective buffering configured for one I/O aggregator per compute node.

Figure 39. BT-IO Call Tree



Call Tree - /home/hpc/pr28fa/di98....C.64.mpi_io_full.otf * - Vampir

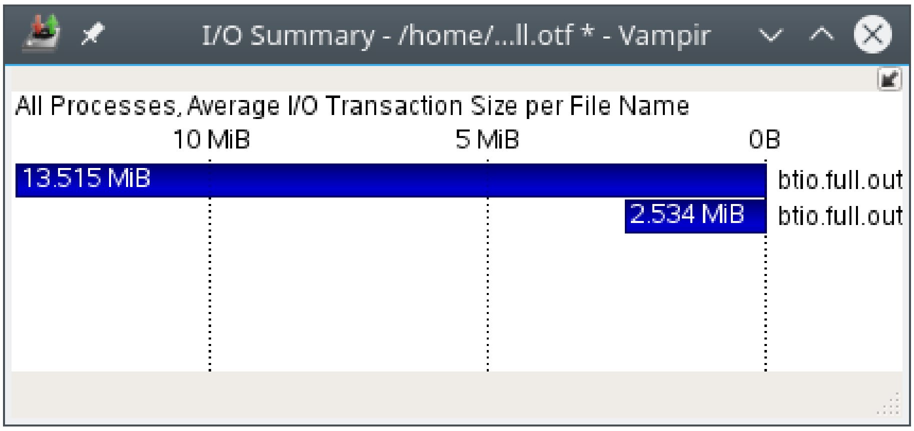
All Processes

Functions	Min Inclusive Time	Max Inclusive Time
▼ 🟩 user	41.127 s	41.341 s
🟨 write	2.196 ms	2.196 ms
🟨 open64	46.093 µs	46.093 µs
▶ 🟥 MPI_Waitall	79.718 ms	0.913 s
▶ 🟥 MPI_Wait	1.053 s	5.078 s
🟥 MPI_Type_struct	4.216 µs	6.830 µs
🟥 MPI_Type_size	30.891 µs	159.101 µs
🟥 MPI_Type_extent	496.146 ns	1.402 µs
🟥 MPI_Type_contiguous	5.906 µs	14.974 µs
🟥 MPI_Type_commit	7.504 ms	7.987 ms
🟥 MPI_Reduce	6.948 µs	39.854 µs
▶ 🟥 MPI_Isend	74.087 ms	0.338 s
▶ 🟥 MPI_Irecv	29.993 ms	98.600 ms
▶ 🟥 MPI_Init	0.480 s	0.694 s
▶ 🟥 MPI_Finalize	317.255 µs	322.515 µs
▼ 🟥 MPI_File_write_at_all	10.659 s	10.819 s
🟨 write	4.634 ms	0.108 s
🟨 pwrite64	0.642 s	6.322 s
🟥 MPI_File_set_view	701.648 µs	763.975 µs
▼ 🟥 MPI_File_read_at_all	2.385 s	2.526 s
🟨 write	136.529 µs	2.136 ms
🟨 pread64	1.219 s	1.486 s
▼ 🟥 MPI_File_open	18.278 ms	86.264 ms
🟨 write	34.421 µs	51.305 µs
🟨 read	801.911 ns	801.911 ns
🟨 open64	119.432 µs	9.631 ms
🟨 close	8.920 µs	8.920 µs
▼ 🟥 MPI_File_delete	164.072 µs	164.072 µs
🟨 unlink	4.180 µs	4.180 µs
▼ 🟥 MPI_File_close	152.635 µs	68.285 ms
🟨 close	102.659 µs	68.199 ms
🟥 MPI_Comm_split	78.386 µs	126.176 µs
🟥 MPI_Comm_size	270.381 ns	769.988 ns
🟥 MPI_Comm_rank	87.691 ns	219.997 ns
🟥 MPI_Comm_dup	81.801 µs	4.169 ms
🟥 MPI_Bcast	34.137 µs	4.717 ms
🟥 MPI_Barrier	2.251 ms	11.123 ms
▶ 🟥 MPI_Allreduce	14.546 ms	0.154 s

🟩 user (1)

The BT-IO call tree is shown in Figure 39, “ BT-IO Call Tree”. Here, the MPI-IO operations use different underlying POSIX operations than Intel MPI. As can be seen in the figure, the collective operation `MPI_File_write_at_all` is composed of write and pwrite64 operations at the POSIX level.

Figure 40. BT-IO I/O Summary


I/O Summary - /home/...ll.otf * - Vampir

All Processes, Average I/O Transaction Size per File Name

Finally, in Figure 40, “ BT-IO I/O Summary” the I/O summary is shown. BT-IO only writes and reads a single 6 GiB file but the summary presents two files. This is due to Vampir depicting the file at both the POSIX and MPI-IO levels for the metric selected. In the case, the Average I/O Transaction Size is selected, showing the effect of collective buffering techniques on operation size at the POSIX level (13.515 MiB).

7.5.1.2. Comparing traces using Vampir

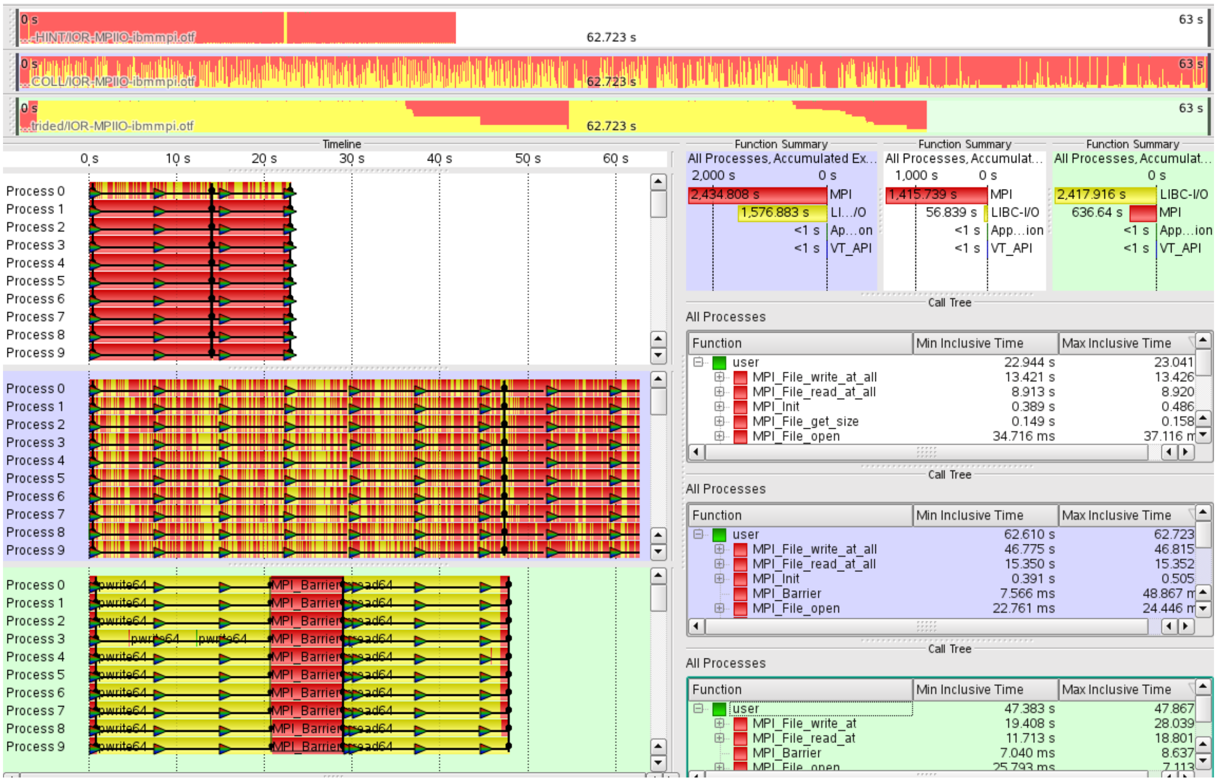
Another useful view of the Vampir Tool is the comparison of traces. To demonstrate this, the IOR benchmark is executed using MPI-IO for independent and collective operations for a strided pattern. IOR [23] is a synthetic benchmark for testing the performance of parallel filesystems. The benchmark supports a variety of different APIs to simulate I/O load. IOR can be used for testing performance of parallel file systems using various I/O libraries: MPI-IO, POSIX-IO, HDF5 and PnetCDF.

The IOR parameters for this test were: MPI processes = 64, request size = 1 MiB, 16 MPI processes per compute node, a MPI process per core, 1GiB of data per process.

and the strided pattern was configured as follows:

- Independent I/O: `IOR-MPIIO-ibmmpi -a MPIIO -s 1024 -b 1m -t 1m`
- Collective I/O by default: `IOR-MPIIO-ibmmpi -a MPIIO -c -s 1024 -b 1m -t 1m`
- Collective I/O with collective buffering enabled: `IOR-MPIIO-ibmmpi -a MPIIO -s 1024 -b 1m -t 1m`, but setting `export ROMIO_HINTS=romio-hints` where romio_hints contains `romio_cb_read enable; romio_cb_write enable`.

Figure 41. IOR traces comparison



Call Tree

All Processes

Function	Min Inclusive Time	Max Inclusive Time
user	22.944 s	23.041
MPI_File_write_at_all	13.421 s	13.426
MPI_File_read_at_all	8.913 s	8.920
MPI_Init	0.389 s	0.486
MPI_File_get_size	0.149 s	0.158
MPI_File_open	34.716 ms	37.116 m

Call Tree

All Processes

Function	Min Inclusive Time	Max Inclusive Time
user	62.610 s	62.723
MPI_File_write_at_all	46.775 s	46.815
MPI_File_read_at_all	15.350 s	15.352
MPI_Init	0.391 s	0.505
MPI_Barrier	7.566 ms	48.867 m
MPI_File_open	22.761 ms	24.446 m

Call Tree

All Processes

Function	Min Inclusive Time	Max Inclusive Time
user	47.383 s	47.867
MPI_File_write_at	19.408 s	28.039
MPI_File_read_at	11.713 s	18.801
MPI_Barrier	7.040 ms	8.637
MPI_File_open	25.793 ms	7.113

Figure 41, “ IOR traces comparison” shows the traces for the three cases. The first timeline corresponds to collective operations with collective buffering enabled, the second to collective operations by default and the final timeline to the independent I/O. Collective buffering is generally recommended for giving the best performance non-contiguous patterns. However, performance depends on the configuration of the underlying system as well. Here, the transfer size (1MiB) is less than the blocksize of the filesystem (8MiB), and the buffer size for collective operations is 16MiB. Therefore, aggregated I/O performs well for this system. When selecting an optimisation technique for your I/O pattern, the I/O configuration of the system itself must also be considered.

7.6. MPI-IO Reporting with Cray MPICH

The custom version of MPI provided by Cray (`cray-mpich`) available on XC-series clusters, as well as others based around Cray hardware, has an additional feature that reports a variety of I/O statistics to aid with profiling efforts. Setting the environment variable `MPICH_MPIIO_STATS=1` before running a parallel application enables the report:

```
export MPICH_MPIIO_STATS=1
```

which gives output such as:

```
+--------------------------------------------------------+
| MPIIO write access patterns for benchmark_files/mpiio.dat
|   independent writes      = 0
|   collective writes       = 24
|   independent writers     = 0
|   aggregators             = 24
|   stripe count            = 48
|   stripe size             = 1048576
|   system writes           = 3072
|   aggregators active      = 0,0,0,24 (1, <= 12, > 12, 24)
|   total bytes for writes  = 3221225472 = 3072 MiB = 3 GiB
|   ave system write size   = 1048576
|   read-modify-write count = 0
|   read-modify-write bytes = 0
|   number of write gaps    = 0
|   ave write gap size      = NA
| See "Optimizing MPI I/O on Cray XE Systems" S-0013-20 for explanations.
+----------------------------------------------------------+
```

for each MPI-IO operation performed in the job. This allows for confirmation that the application and MPI library have been tuned effectively by, for example, exclusively using collective operations to optimise I/O. Refer to Chapter 4 for further guidance on MPI-IO.

Additionally, the environment variable `MPICH_MPIIO_HINTS_DISPLAY=1` is available on Cray systems to print the MPI-IO hints used by open operations on each file accessed by an application. Setting:

```
export MPICH_MPIIO_HINTS_DISPLAY=1
```

before running the same job as above produces:

```
PE 0: MPIIO hints for benchmark_files/mpiio.dat:
cb_buffer_size           = 16777216
romio_cb_read            = automatic
romio_cb_write           = automatic
cb_nodes                 = 24
cb_align                 = 2
romio_no_indep_rw        = false
romio_cb_pfr             = disable
romio_cb_fr_types        = aar
romio_cb_fr_alignment    = 1
romio_cb_ds_threshold    = 0
romio_cb_alltoall        = automatic
ind_rd_buffer_size       = 4194304
ind_wr_buffer_size       = 524288
romio_ds_read            = disable
romio_ds_write           = disable
striping_factor          = 48
striping_unit            = 1048576
romio_lustre_start_iodevice = 0
direct_io                = false
aggregator_placement_stride = -1
abort_on_rw_error        = disable
cb_config_list           = *:*
romio_filesystem_type    = CRAY ADIO:
```

Refer to Chapter 4 for descriptions of these hints.

Further documentation

Books

[1] *Prabhat and Q. Koziol,High Performance Parallel I/O, 1st ed. Chapman and Hall/CRC, 2014.*

[2] *Linux System Programming: Talking Directly to the Kernel and C Library, 1st ed. O'Reilly Media, Inc., 2007.*

Websites, forums, webinars

[3] *PRACE Webpage, http://www.prace-ri.eu/ .*

[4] *BeeGFS Wiki: System Architecture, https://www.beegfs.io/wiki/SystemArchitecture .*

[5] *BeeGFS Wiki: Storage Pools, https://www.beegfs.io/wiki/StoragePools .*

[6] *BeeGFS Wiki: Striping Settings, https://www.beegfs.io/wiki/Striping .*

[7] *HDF5 Webpage, https://support.hdfgroup.org/HDF5/ .*

[8] *NetCDF Webpage, http://www.unidata.ucar.edu/software/netcdf/ .*

[9] *PnetCDF Webpage, http://cucis.ece.northwestern.edu/projects/PnetCDF/ .*

[10] *ADIOS Webpage, https://www.olcf.ornl.gov/center-projects/adios/ .*

[11] *SIONLib Webpage, http://www.fz-juelich.de/jsc/sionlib .*

[12] *ROMIO Webpage, http://www.mcs.anl.gov/projects/romio/ .*

[13] *ARCHER » Hardware, http://www.archer.ac.uk/about-archer/hardware/ .*

[14] *ARCHER » Best Practice Guide - Sample FPP/SSF Results, http://www.archer.ac.uk/documentation/best-practice-guide/io.php#summary-of-performance-advice .*

[15] *Darshan Webpage, http://www.mcs.anl.gov/research/projects/darshan/ .*

[16] *Darshan-runtime installation and usage, https://www.mcs.anl.gov/research/projects/darshan/docs/darshan3-runtime.html .*

[17] *Darshan Documentation, https://www.mcs.anl.gov/research/projects/darshan/documentation .*

[18] *Darshan at LRZ, https://doku.lrz.de/display/PUBLIC/Darshan .*

[19] *Modularized I/O characterization using Darshan 3.x, https://www.mcs.anl.gov/research/projects/darshan/docs/darshan-modularization.html .*

[20] *TAU Webpage, https://www.cs.uoregon.edu/research/tau/home.php .*

[21] *Vampir Webpage, https://www.vampir.eu/ .*

[22] *Performance Data Visualization: I/O Summary, https://vampir.eu/tutorial/manual/performance_data_visualization#sec-iosummary .*

[23] *HPC IO Benchmark Repository, https://github.com/hpc/ior .*

[24] *Chunking in HDF5, https://portal.hdfgroup.org/display/HDF5/Chunking+in+HDF5 .*

[25] *HDF5 Advanced Topics, https://support.hdfgroup.org/HDF5/doc/Advanced/Chunking/Chunking_Tutorial_EOS13_2009.pdf .*

[26] *A Brief Introduction to Parallel HDF5, https://www.alcf.anl.gov/files/Parallel_HDF5_1.pdf* .

[27] *SIONlib file format, https://apps.fz-juelich.de/jsc/sionlib/docu/fileformat_page.html* .

[28] *API overview, https://apps.fz-juelich.de/jsc/sionlib/docu/api_page.html* .

[29] *Utilities for managing of SIONlib files, https://apps.fz-juelich.de/jsc/sionlib/docu/util_page.html* .

[30] *Performance Issues, https://confluence.hdfgroup.org/display/knowledge/Performance+Issues* .

Manuals, papers

[31] *IBM Spectrum Scale Manual, https://www.ibm.com/support/knowledgecenter/STXKQY_4.2.3/com.ibm.spectrum.scale.v4r23.doc/pdf/scale_ins.pdf* .

[32] *Lustre Manual, http://doc.lustre.org/lustre_manual.pdf* .

[33] *MPI: A Message-Passing Interface Standard Version 3.0, https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf* .

[34] *ThinkParQ BeeGFS Training - Typical Administrative Tasks, https://indico.mathrice.fr/event/5/session/10/contribution/29/material/slides/0.pdf* .

[35] *VampirTrace 5.14.4 with extended accelerator support, https://tu-dresden.de/zih/forschung/ressourcen/dateien/projekte/vampirtrace/accelerator/dateien/VampirTraceManual/VampirTrace-5.14.4-gpu2-user-manual.pdf?lang=en* .

[36] *A case study for scientific I/O: improving the FLASH Astrophysics Code, https://iopscience.iop.org/article/10.1088/1749-4699/5/1/015001/pdf* .

[37] *NAS Parallel Benchmarks I/O Version 2.4, https://www.nas.nasa.gov/assets/pdf/techreports/2003/nas-03-002.pdf* .

[38] *Dynamically Adapting File Domain Partitioning Methods for Collective I/O Based on Underlying Parallel File System Locking Protocols, http://users.eecs.northwestern.edu/~wkliao/PAPERS/fd_sc08_revised.pdf* .