Data Management

AQTIVATE Training Workshop I

Dirk Pleiter

CST | EECS | KTH

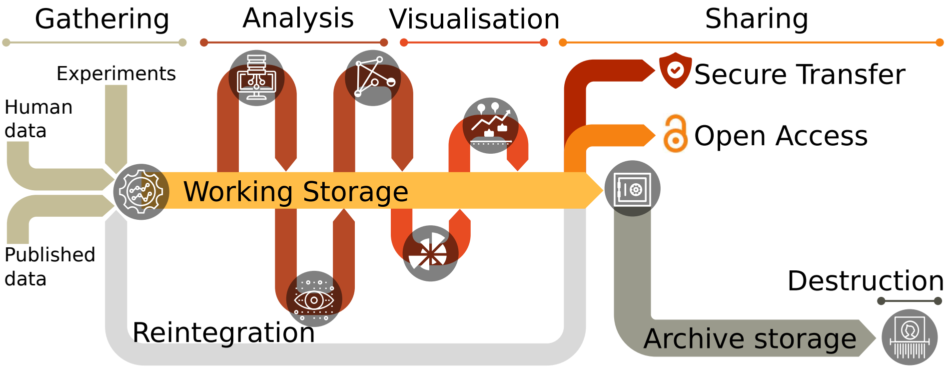
December 2023

Content

What is Data Management?

- **Data management** $=$ All efforts related to handling data as a valuable resource
- The topic of data management has many aspects (and only very few can be addressed here)
  - **Data storing**
  - Data integrity
  - Data quality management
  - **Data sharing**
  - Data provenance tracking
  - Ethical aspects, e.g. protection of sensitive data
  - Legal aspects, e.g. licencing data re-use
  - Data publishing
  - ...

Data Lifecycle View

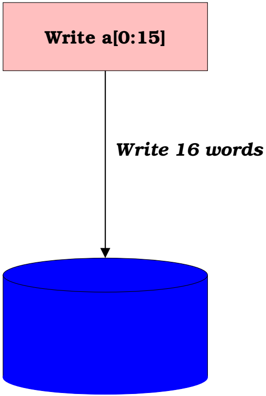


[Thomas Shafee, 2020] (CC BY)

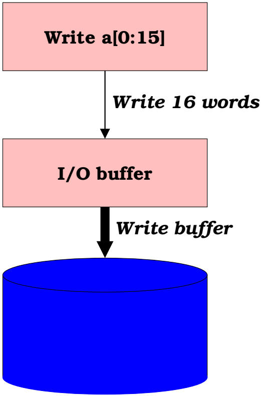
Content

Serial Case: Schematic View on I/O

Naive view:



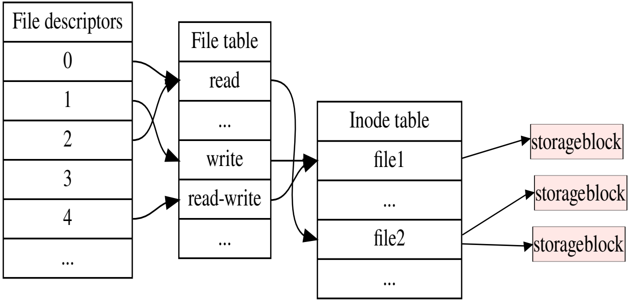
More accurate view:



Unix-like File Systems

Simplified view on Unix-like file systems:

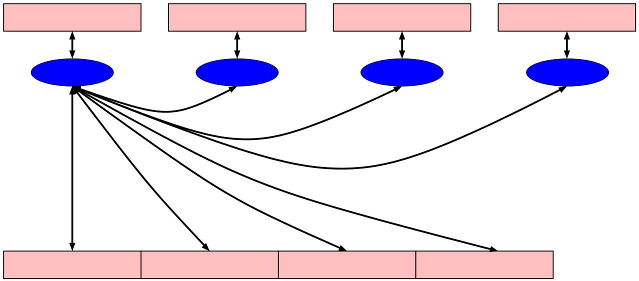
- inode as basic data structure of the file system
- inodes store file attributes and storage block locations



Parallel I/O Challenges

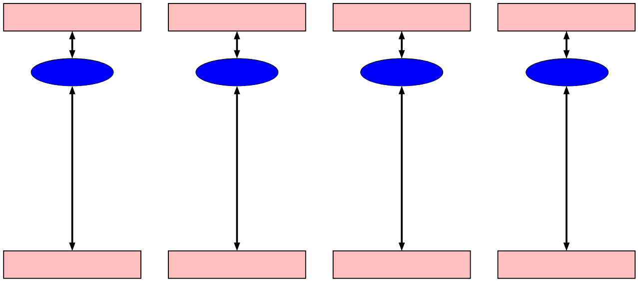
- **POSIX ordering**: The standard was designed assuming a single stream of Bytes
  - POSIX semantics assume only a single process would ever want exclusive write access
  - Parallel I/O generates many streams of Bytes
- **Block structure**: Concurrent updates of storage blocks may occur
  - Updates are implemented as read-modify-write
  - Locking is required to avoid modifications getting lost in case of concurrent updates
    ☞ Risk of lock contention
- **POSIX coherence**: Once a write completes, any read must see that write
  - Challenge in the parallel case: Write and read may be performed by different processes

Parallel Case: Non-Parallel I/O



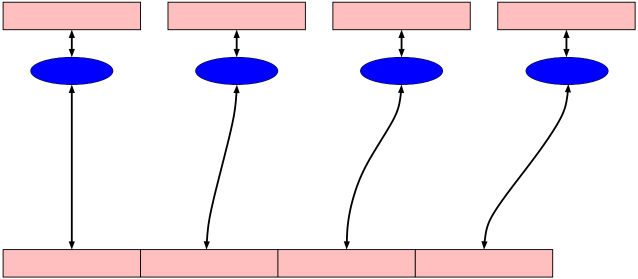
- Advantages:
  - Easy to implement
  - Single file
- Disadvantages:
  - All I/O routed through a single node
  - Non-scalable approach

Parallel Case: Independent I/O



- Advantage: Highly parallel without dependencies
- Disadvantage: Many small files
  - Metadata performance bottleneck, inode limits in file system

Parallel Case: Single File



- Advantages:
  - Highly parallel
  - Single file
- Disadvantage: high risk of concurrent storage block update

IO500 List

- Bi-annually published list of HPC systems ranked according to I/O performance
- I/O performance determined by benchmark suite including the following
  - **IOR easy**: Benchmark evaluating read/write performance using a large transfer size of 2 MByte
  - **IOR hard**: Same benchmark but now using a small transfer size of 47,008 Byte
  - **mdtest easy**: Benchmark evaluating metadata performance
- Ranking based on geometric mean of benchmark scores

IO500 List: Shaheen III

- **System**: Shaheen III (Kaust)
  - Highest ranked system with Lustre file system (no accelerating layer)
  - Top500 position #20 (peak: 39 PFlop/s)
- **Results**: November 2023 edition

IOR easy / write	2,340 GiB/s
IOR easy / read	3,950 GiB/s
IOR hard / write	24 GiB/s
IOR hard / read	1,157 GiB/s
MDEasy / write	780 kIOP/s
MDEasy / stat	1,740 kIOP/s
MDEasy / delete	703 kIOP/s

- **Observations:**
  - Read is faster than write
  - Hard reads/writes 3.4×/98× slower
  - Metadata operations are expensive

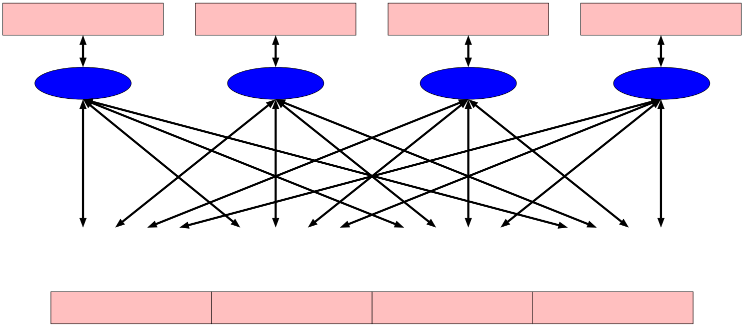
Content

Why Using MPI for I/O?

- HPC networks offer high-performance data transport capabilities
- MPI provides useful features for optimising parallel I/O
  - Non-blocking communication operations
  - Collective communication operations
  - Communicators to separate application-level message passing from I/O-related data transport
  - Support for defining application-specific data types

Cooperative Parallel I/O

Idea: Use MPI as a data transport engine to move data efficiently from or to storage.



Basic Example (1/3)

- Simple sequence of POSIX I/O calls:
  - Open a file: `fopen`
  - Write data to the file: `fwrite`
  - Close the file: `fclose`
- Same sequence of I/O calls using MPI:
  - Open a file: `MPI_File_open`
  - Write data to the file: `MPI_File_write`
  - Close the file: `MPI_File_close`

Basic Example (2/3)

```c
#include <stdio.h>
#include "mpi.h"

#define N 1000

int main(int argc, char *argv[])
{
  MPI_File fh;
  int buf[N], rank;

  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  MPI_File_open(MPI_COMM_WORLD, "myfile.out", MPI_MODE_CREATE|MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);

  if (rank == 0) MPI_File_write(fh, buf, N, MPI_INT, MPI_STATUS_IGNORE);

  MPI_File_close(&fh);

  MPI_Finalize();
  return 0;
}
```

Basic Example (3/3)

- MPI_File_open is collective over the communicator
  - The communicator is used to support collective I/O
  - The modes are similar to POSIX open
  - MPI_Info provides additional hints for performance
- The MPI_File_write calls are independent
  - The call may be executed only on a subset of ranks
- MPI_File_close is collective

Shared File Access

Ways to access a shared file in MPI:

- Local file handle:
  - Unix-style approach:
    - Seek to file position using MPI_File_seek
    - Read from or write to file using MPI_File_write or MPI_File_read
  - Combination of both steps:
    - MPI_File_read_at
    - MPI_File_write_at
- Shared file handle:
  - MPI maintains exactly one file handle shared by all processes
  - Available functions: MPI_File_read_shared, MPI_File_write_shared

Write with Off-set: `MPI_File_write_at`

- Syntax:

```
int MPI_File_write_at(
  MPI_File fh,            /* File handle */
  MPI_Offset offset,      /* File offset */
  const void *buf,        /* Initial address of buffer */
  int count,              /* Number of elements */
  MPI_Datatype datatype,  /* Data type */
  MPI_Status *status      /* Status object */
);
```

- Writes a file at an explicitly specified off-set
- Off-set will typically depend on the rank

Write with Off-set: Example

```c
#include "mpi.h"

int main() {
  const int n = 32;
  int x[n];
  int irank;
  MPI_File fh;
  MPI_Status status;

  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &irank);

  for (int i = 0; i < n; i++) x[i] = n * irank + i;

  MPI_File_open(MPI_COMM_WORLD, "myfile.out", MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &fh);
  MPI_File_write_at(fh, irank * n * sizeof(int), x, n, MPI_INT, &status);
  MPI_File_close(&fh);

  MPI_Finalize();

  return 0;
}
```

Collective I/O Operations

- I/O functions where programmer promises call by all processes in a communicator
  - `MPI_File_read_all`
  - `MPI_File_write_all`
  - `MPI_File_read_at_all`
  - `MPI_File_write_at_all`
- Allows MPI to merge requests from different processes
  - Large collective access to file system may improve performance

Optimisation Recommendations

- Try to reduce the number of I/O operations by operating on larger chunks of data
  - It may be beneficial to extract the necessary data after reading (data sieving)
- Avoid concurrent storage block updates (lock contention)
  - Consider block-aligned writes
- Use collective instead of independent I/O functions
- Avoid performing a large number of file system metadata operations
  - In particular, avoid creating a large number of files

Content

Motivation

[Veerle Van den Eynden et al., 2011]

- It encourages scientific enquiry and debate
- It maximises transparency and accountability
- It enables scrutiny of research findings
- It encourages the improvement and validation of research methods
- It reduces the cost of duplicating data collection
- It leads to new collaborations between data users and data creators
- It can increase the impact and visibility of research
- It can provide a direct credit to the researcher as a research output in its own right
- It provides important resources for education and training

FAIR Principles

[Mark D. Wilkinson et al., 2016]

Findable

Accessible

Interoperable

Reusable

- It is becoming a mandatory requirement by funding agencies
  - Example: “The [European] Commission will work with global policy and research partners to foster cooperation and to create a level playing field in scientific data sharing and data-driven science.” [EU Commission, COM(2016)178]
- Data sharing is an important aspect of Open Science [Vicente-Saez+Martinez-Fuentes, 2018]
- These are guiding principles, not an implementation

What Does "Findable" Mean?

F1 Globally unique and persistent identifiers (PID) are assigned to the (meta)data

F2 The data is described with rich metadata

F3 The metadata includes the PID

F4 The (meta)data is registered or indexed in a searchable resource

- Popular system for PIDs: Digital Object Identifier
  - Can also be used for data
  - Benefit: data becomes citable

doi Foundation

What Does “Accessible” Mean?

A1 The (meta)data is retrievable by the PID using standardised protocols
- A1.1 The protocol is open, free, and universally implementable
- A1.2 The protocol supports authentication/authorization procedures where necessary

A2 The metadata is accessible even if the data is no longer available

- A1 can be achieved, e.g., by means of a file catalogue (PID $\rightarrow$ storage location(s))
- Accessible does not mandate public access without authentication
- The metadata is valuable even without the associated data

What Does “Interoperable” Mean?

- I1 For the (meta)data a formal, accessible, shared, and broadly applicable language is used
- I2 For the (meta)data vocabularies are used that follow FAIR principles
- I3 The (meta)data includes qualified references to other (meta)data

- FAIR requires machine actionable (meta)data
  - Languages/file formats like XML can facilitate this

What Does “Reusable” Mean?

R1 (Meta)data richly described with plurality of accurate and relevant attributes

- R1.1 (Meta)data released with clear and accessible data usage license
- R1.2 (Meta)data associated with detailed provenance
- R1.3 (Meta)data meet domain-relevant community standards

- Popular choice for licence: Creative Commons
- Note relation of reusability and good scientific practices:

Reproducibility:	same data	+	same method	=	same result
Replicability:	**new** data	+	same method	=	same result
Robustness:	same data	+	**different** method	=	same result

Metadata

- **Metadata** $=$ Data that provides information about other data
- Metadata can include many kinds of information, e.g.
  - Content (using a general and domain-specific vocabulary)
  - Provenance (who, when, where, how?)
  - Access (format, path, license)
- There exist various generic metadata standards, e.g. Dublin Core DublinCore
  1. Contributor: An entity responsible for making contributions to the resource
  2. Coverage: The spatial or temporal topic of the resource, the spatial applicability of the resource, or the jurisdiction under which the resource is relevant
  3. Creator: An entity primarily responsible for making the resource
  4. Date: A point or period of time associated with an event in the lifecycle of the resource
  5. ...

Technology for Metadata Documents: XML

- Both human-readable and machine-readable
- Standardized by the World Wide Web Consortium's (W3C) XML 1.0 Specification
- Availability of the XML schema (XSD) technology that allows to define the necessary metadata for interpreting and validating XML documents
- Availability of standardised technologies to query XML documents
  - W3C's XPath (current version: XPath 3.1)
  - W3C's XQuery (current version: XQuery 3.1)
- Many tools for processing XML documents
  - For parsing: `libxml2` with interfaces for C, C++, Python, Perl
  - For formatting and validation: `xmllint`

Background on XML

- Simple example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dataEntry>
  <key>7</key>
  <value>Hello world!</value>
</dataEntry>
```

- Key elements:
  - Tag: Markup construct that begins with $<$ and ends with $>$
    - Start tag: <elem>
    - End tag: < /elem>
    - Empty-element tag: <elem/ >
  - Element: Logical document component starting/ending with a tag (or empty-element tag)
  - Attribute

XML Schema

- Technology to formally describe the elements in an XML document
- Simple example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="dataEntry">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="key" type="xs:decimal"/>
        <xs:element name="value" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Validating XML Files (1/2)

- Document validation using the `xmllint` tool:

```
% xmllint --schema ./tst-schema.xml ./tst-doc-ok.xml
<?xml version="1.0" encoding="UTF-8"?>
<dataEntry>
  <key>7</key>
  <value>Hello world!</value>
</dataEntry>
./tst-doc-ok.xml validates
```

Validating XML Files (2/2)

- Try document with incorrect tag:

```
% xmllint --schema ./tst-schema.xml ./tst-doc-bad-1.xml
<?xml version="1.0" encoding="UTF-8"?>
<dataEntry>
  <idx>7</idx>
  <value>Hello world!</value>
</dataEntry>
./tst-doc-bad-1.xml:3: element idx: Schemas validity error : Element 'idx': This element is not expected. Expected is ( key ).
./tst-doc-bad-1.xml fails to validate
```

- Try document with incorrect element content:

```
% xmllint --schema ./tst-schema.xml ./tst-doc-bad-2.xml
<?xml version="1.0" encoding="UTF-8"?>
<dataEntry>
  <key>seven</key>
  <value>Hello world!</value>
</dataEntry>
./tst-doc-bad-2.xml:3: element key: Schemas validity error : Element 'key': 'seven' is not a valid value of the atomic type 'xs:decimal'.
./tst-doc-bad-2.xml fails to validate
```

Status of Data Sharing Standards

- Lattice QCD simulations: International Lattice Data Grid (ILDG) [Karsch et al., 2022]
  - Standards for XML-based metadata documents for ensembles and configurations
  - File format standard
- Molecular dynamic simulations
  - Efforts towards standardisation of file formats [M. Abraham et al., 2019]
- Computational fluid-dynamics simulations
  - ???

Content

Main Take-Aways

- I/O can for applications become a major performance bottleneck
  - Improving the capabilities of the I/O subsystem is a major focus of the HPC community, but technologies and architectures mainly focus on improving storage capacity
  - Various options for improving I/O performance exist as performance can strongly depend on the I/O pattern
- Sharing research data based on the FAIR principles are becoming part of the good scientific practices
  - The required efforts can be very rewarding