# ESD Middleware Architecture

for

# ESIWACE

Release 0.2.2
March 9, 2017
Approved: (Internal Draft)

The WP4 Team

TODOs

**TODOs**: 22 - 8 - 9 - 10 - 14 - 15 - 17 - 17 - 24 - 27 - 30 - 41 - 42 - 43 - 43 - 43 - 44 - 44 - 45 - 45 - 46 - 46

# Contents

# Revision History

| Version | Date | Who | What |
|---|---|---|---|
| 0.1.0 | 2017 Feb | Team | Initial Version. |
| 0.2.0 | 2017 March | BNL | Major restructuring. |
| 0.2.1 | March 6, 2017 | BNL | Pivot from RM-ODP to 4+1 Data Model. |

**Incomplete Document**

This version of the ESIWACE middleware architecture does not include material beyond section 4.1 because at this time (March 9, 2017), much material has either been removed for re-ordering, or it is subject to reconsideration as part of the first architectural review. The material that is included is **as-is** and should only be used by ESIWACE project particpants, as it to may be reconsidered, extended, or altered in the review.

# 1 Introduction

This document provides the architecture for new Earth System Data middleware, aimed at deployment in both simulation and analysis workflows where the volume and/or rate of data leads to performance and/or data management issues with traditional approaches.

This architecture is one of the deliverables from the Centre of Excellence in Weather and Climate in Europe (http://esiwace.eu).

## 1.1 General Objectives

In this section we outline the general challenges, and some specific challenges which this work needs to address. Detailed consequential requirements appear in Chapter 3.

### 1.1.1 Challenges and Goals

There are three broad data related challenges that weather and climate workflows need to deal with, which can be summarised as needing to handle

1. the velocity of high volume data being produced in simulations, and
2. the economic and performant persistence of high volume data,
3. high volume data analysis workflows with satisfactory time-to-solution.

Currently these three challenges are being addressed independently by all major centres, the aim here is to provide middleware architecture that can go someway to providing economic performance portability across different environments.

There are some common underlying characteristics of the problem:

1. **I/O intensity (volume and velocity)**. Multiple input data sources can be used in any one workflow, and the volume and rate of output can vary drastically depending on the problem at hand. In weather and climate use-cases,
   - during simulations, input checkpoint data needs to be distributed from data sources to all nodes and high volume output is likely to come from multiple nodes (although not necessarily all) using domain decomposition and MPI.
   - existing analysis workflows primarily use time-decomposition to achieve parallelisation which has implications for input data storage, and output data organisation — but at least is easy to understand. More complex parallelisation strategies for analysis are being investigated and may mix multiple modes of parallelisation (and hence routes to and from storage).
2. **Diversity of data formats and middleware**. In an effort to allow for easier exchange and inter-comparison of models and observations, data libraries for standardized data description and optimized I/O such as NetCDF, HDF5 and GRIB were developed but many more legacy formats exist. Many I/O optimizations used in common libraries do not adequately reflect current data intensive system architectures, as they are maintained from domain scientists and not computer scientists.

3. **Code portability**. Code is long-living, it can potentially live for decades — with some modules moving like DNA down through generations of new code. Historically such modules and parent codes have been optimised for specific supercomputers and I/O architectures but with increasingly complex systems this approach is not feasible.

4. **Sharing of data between many stakeholders** Many new stakeholders are using data on multiple different systems. As a consequence the underlying data systems need to support that multi-disciplinary research through shared, interoperable interfaces, based on open standards, allowing different disciplines to customise their own workflows over the top.

5. **Time criticality and reliability** Weather and climate applications often need to be completed in specific time windows to be useful, and all data must be reliably stored and moved — there can be no question of data being corrupted in transit or in the storage.

There are some conclusions one can draw from these general challenges: Data systems needs to scale in such a way as to support expected data volume and velocity with cost-effective and acceptable data access latencies and data durability — and do so using mechanisms which are portable across time and underlying storage architectures. So the goals of any solution should be to be:

1. Performant — coping with volume/velocity and delivering adequate bandwidth and latency.

2. Cost-Effective — affordable in both financial and environmental terms at exascale.

3. Reliable — storage is durable, data-corruption in transit is detected and corrected,

4. Transparent — hiding specifics of the storage landscape and not ***requiring*** users to change parameters specifically to a given system.

5. Portable — should work in different environments.

6. Standards based — using interfaces, formats and standards which maximise re-usability.

Of course it is clear that some of these goals are contradictory: performance, transparency, and portability are not necessarily simultaneously achievable, but we should aim to maximise these. It is also clear that a storage system may not be able to deliver these goals for all possible underlying data formats.

There are two more important objectives that do not reflect the domain per se, but reflect the desire for any solutions to be maintainable and actually used. To that end, reflecting the characteristics of software which is widely deployed, solutions should also:

7. be easily maintainable and exploiting as much as possible other libraries and components (as opposed to implementing all capabilities internally), and

8. involve open-source software with an open-development cycle.

## 1.2 Architecture Philosophy and Methodology

A middleware approach, providing new functionality which insulates applications from storage systems provides the only practical solution to the problems outlined in **??**. To that end we have designed the "Earth System Data" middleware. This new middleware needs to be inserted into existing workflows, yet it must exploit a range of existing and potential

storage architectures. It will be seen that it also needs to work within and across institutional firewalls and boundaries.

To meet these goals, the design philosphy needs to respect aspects of the weak coupling concepts of a microservices web design, of the stronger coupling notions of distributed systems design, and the tight-coupling notions associated with building appliances (such as those sold which provide transparent gateways between parallel file systems and object stores).

The design philosphy also needs to reflect the reality that while we have a good sense of the general requirements, specific requirements are likely to become clearer as we actually build and implement the ESD. It is also being built in a changing environment of other standards and tools - for example, the advent of the Climate and Forecast conventions V2.0 is likely to occur during this project, and that could have significant impact on data layouts, which might impact the ESD middleware design. Similarly, the new HDF server library being built by the HDF Group is likely to be an important component of the ESD middleware thinking, as are the changing capabilities of both the standard object APIS such as S3 and Swift, and the proprietary APIs of vendors (including, but not limited to that of our partner, Seagate).

All of these trends mean that the design philosophy, and the design itself, need to be flexible and responsive to evolving understanding and external influences. One direct consequence of this is that we might expect different components of the ESD middleware to be themselves evolving at different rates: give the complexity of the problem it is unlikely that a coherent overall architecture can be mandated and controlled *and all components deployed simultaneously at all sites and in all clients*. To that end, our underlying philosophy for all components will conform to Postel's Law:

> *Be conservative in what you send, be liberal in what you accept.*

We architect the ESD middleware system using a modified version of the "4+1 view system" consisting of the four primary views (described in the following chapters):

1. The **Development View** which *describes* the system from a software point of view, identifying the components which need to be constructed and deployed.

2. The **Logical View** which

   a) *describes* the functionality needed, and

   b) *defines* the data models underlying any information artifacts needed to implement that functionality.

   (For ESD middleware, the relevant data models will include those necessary to import and export data, to describe backend components, and to configure the layout of ESD data on those backend components.)

3. The **Physical View** which *describes* how the software components and libraries within the ESD middleware can be deployed on the hardware that the ESD middleware supports (so of necessity it defines what hardware is needed, and what it would mean for hardware to be ESD compliant).

4. The **Process View** which

   a) *defines* what services and APIS exist to deploy, manage and exploit the ESD middleware from both the administration and user perspectives, and

   b) *describes* how the system will perform and scale.

Supplemented by a number of

5. **Scenarios** (or Use Cases) which provide an integrated view of how the ESD middleware can be deployed and used. Here, our use case views will describe the primary use-cases for ESIWACE.

**TODO**: Sort out a proper reference for the 4+1 system

## 1.3 Document Structure

Before delving into the formal software architecture from a software engineering perspective, we introduce some key aspects of background information about data layout and data formats which provide context for both actual architectural decisions and some of the directions in which the architecture might evolve. This chapter (2) concludes with a description of key storage components which we consider for targeting in the architecture proper.

We extract the general properties of requirements from the Logical View and present them in Chapter 3, where we also introduce elements of related work which a priori influence the architecture itself (e.g.to introduce why we have introduced specific third party dependencies).

We then proceed to the architecture proper, beginning with an overview, before addressing the various viewpoints. The final architecture chapter addresses the scenarios and use cases, including the key first implementation scenarios that will be necessary to meet ESIWACE requirements.

This document concludes with a summary chapter which relates the specfic functional requirements to specific aspects of the architecture.

# 2 Background

## 2.1 Data Generated by Simulations

**TODO**: Need a clearer exposition of why this work is related to the architecture in terms of next generation burst buffers etc.

With the progress of computers and increase of observation data, numerical models were developed. A numerical weather/climate model is a mathematical representation of the earth's climate system, that includes the atmosphere, oceans, landmasses and the cryosphere. The model consists of a set of grids with variables such as surface pressure, winds, temperature and humidity. A numerical model can be encoded in a programming language resulting in an application that simulates the behavior based on the model. Inside an application, a grid is used to describe the covered surfaces of the model, which often is the globe. Traditionally, the globe has been divided based on the longitude and latitude into rectangular boxes. Since this produced unevenly sized boxes and singularities closer to the poles, modern climate applications use hexagonal and triangular meshes. Particularly triangular meshes have an additional advantage, that one can refine regions and, thus, can decide on the granularity that is needed locally – this leads to numeric approaches of the multi-grid methods. Grids that follow a regular pattern such as rectangular boxes or simple hexagonal grids are called structured grids. With partially refined grids or when covering complex shapes instead of the globe, the grids become unstructured, as they form an irregular pattern.

To create an hexagonal or triangular grid from the surface of the earth, the grid can be constructed starting from an icosahedron and repetitively refining the triangle faces until a desired resolution is reached. Variables contain data that can either describes a single value for each cell, the edges of the cells, or the vertices of the cells.

Figure 2.1 shows this localization – the scope of data – for the triangular and hexagonal grids.

Larger grids are shown in Figure 2.3 and in (Figure 2.2). There are figures provided that illustrate the neighborhood between data points and for different data localization.

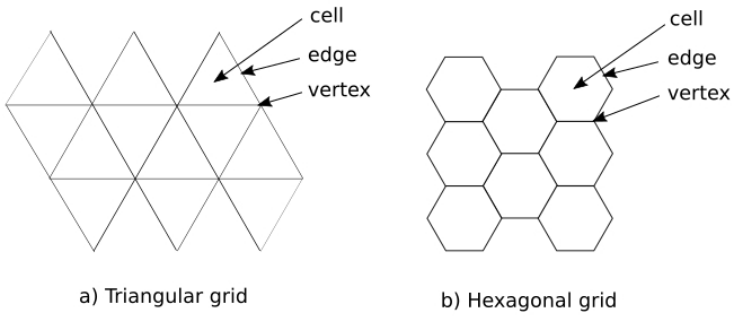



Figure 2.1: Scope of variables inside the grids

**A triangular grid** consists of cells shaped as a triangle (Figure 2.3a). Values can be located at the centers of the primal grid hexagons Figure 2.3b, and if we connect it to each other, we would see the grid of triangles Figure 2.3c. If values are located at the edges (Figure 2.3d) and they are connected with its neighbours, then the grid is given as in Figure 2.3e. If the values are located at the vertices and they are connected with its neighbours, then the grid is given as in Figure 2.3f.

**Hexagonal grid** consists of cells shaped as a flat topped hexagon (Figure 2.2a). Two ways can be used to map data to the grid: vertical or horizontal. Values can be located at the centers of the primal grid (hexagons Figure 2.2b), and if we connect it to each other, we would see the grid of triangles Figure 2.2c. If values are located at the edges (Figure 2.2d) and edges are connected with those of the neighbours, then a grid emerges (Figure 2.2e.) If the values are located at the vertices and vertices are connected with those of the neighbours, then a different grid emerges (Figure 2.2f).

### 2.1.1 Serialization of Grids

The abstractions of grids need to be serialized as data structures for the programming languages and for persisting them on storage systems. In a programming language, regular grids can usally be addressed by n-dimensional arrays. Thus, a 2D array can be used to store the data of a regular 2D longitude/latitude-based grid.

However, storing irregular grids is not so trivial. For example, a 1D array can be used to hold the data but then the index has to be determined. Staying with our 2D example, to map a 2D coordinate onto the 1D array, a mapping between the 2D coordinate and the 1D index has to be found. One strategy to provide the mapping are space-filling curves. These curves have the advantage that the indices to some extent preserve locality for points that are close together – which can be beneficial, as often operations are conducted on neighboring data (stencil operations, for example). A Hilbert curve is an example for one possible enumeration of a multi-dimensional space.

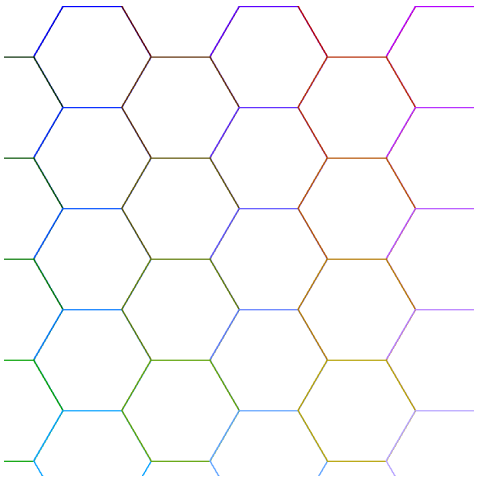
**The Hilbert curve** is a continuous space-filling curve, that helps to represent a grid as n-dimensional-array of values. To visualize its behavior, a 2D grid is shown in Figure 2.5. In 2D, the basic element of the Hilbert curve is a square with one open side. Every such square has two end-points, and each of these can be the entry-point or the exit-point. So, there are four possible variations of an open side. A first order Hilbert curve consists of one basic element. It is a 2x2 grid. The second order Hilbert curve replaces this element by four (smaller) basic elements, which are linked together by three joins (4x4 grid). Every next order repeats the process by replacing each element by four smaller elements and three joins (8x8 grid). On the Figure 2.5 the 5th level Hilbert curve is represented for the 256x256 data, that is mapped to a 32x32 grid.

The characteristics of a Hilbert curve can be extended to more than two dimensions. The first step in the figure can be wrapped up in as many dimensions as is needed and the points/neighbours will be always saved.

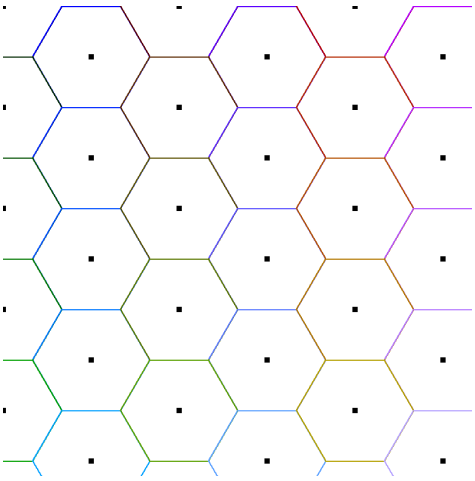
#### Considerations when serializing to storage systems

**TODO**: This is where we link the previous material to storage system futures and hence relevance to architecture, but it needs to be clearer.

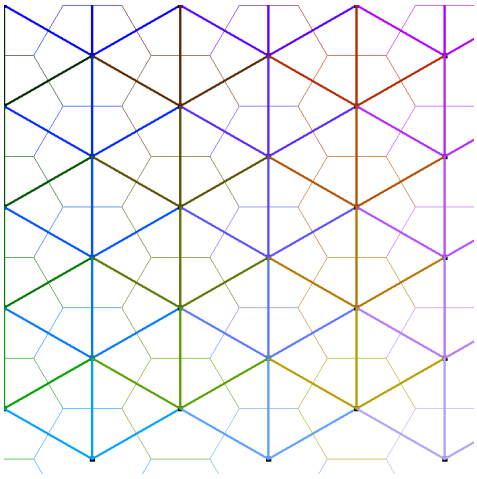
When serializing a data structure to a storage system, in essence this can be done similarly as in main memory. The address space exported by the file API of a traditional file



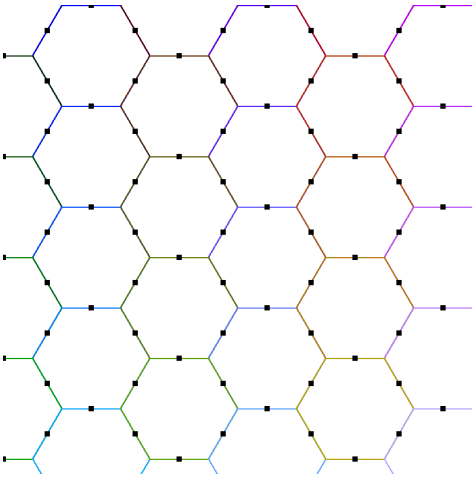
(a) Empty hexagonal grid



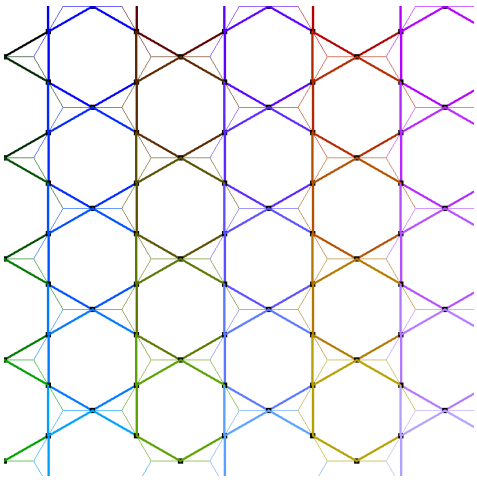
(b) Hexagonal grid with data at the cell centers



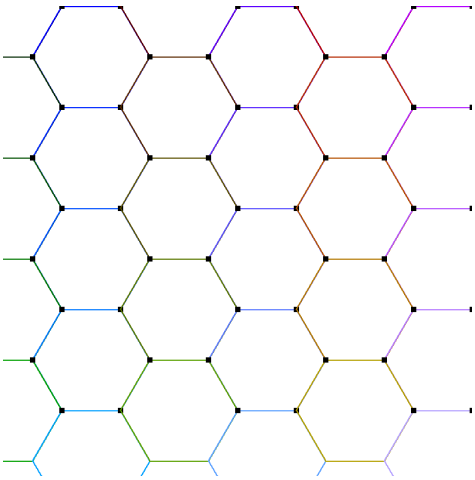
(c) Hexagonal grid with data at the cell's centers, connected neighbours



(d) Hexagonal grid with data on the edges

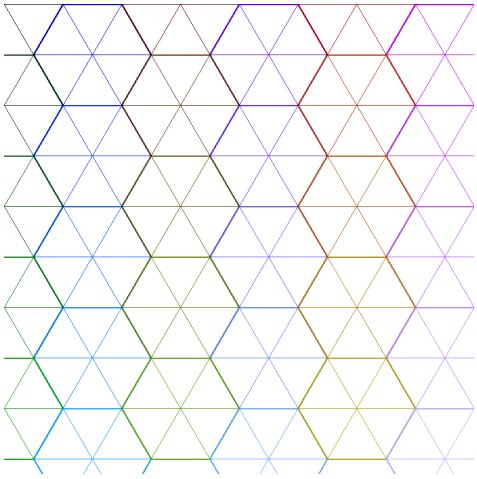


(e) Hexagonal grid with data on the edges, connected neighbours

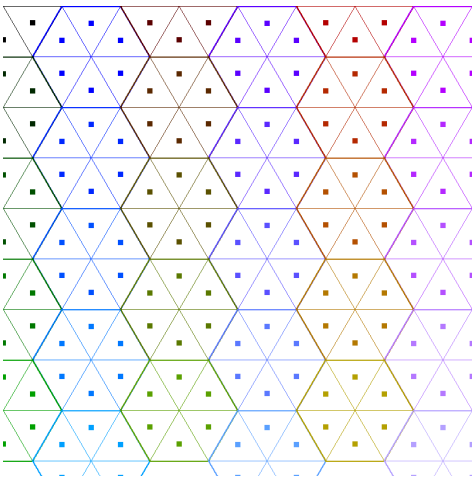


(f) Hexagonal grid with data at the vertices / connected neighbours

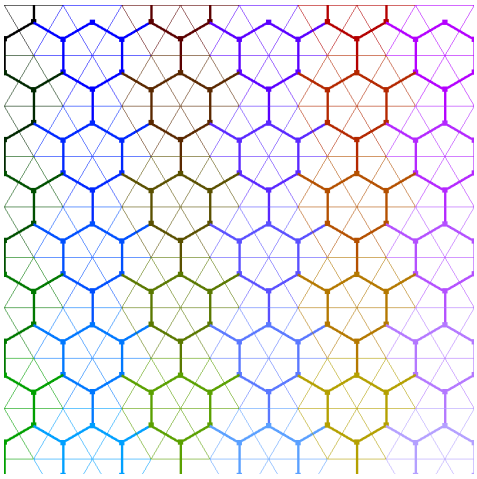
Figure 2.2: Hexagonal grid



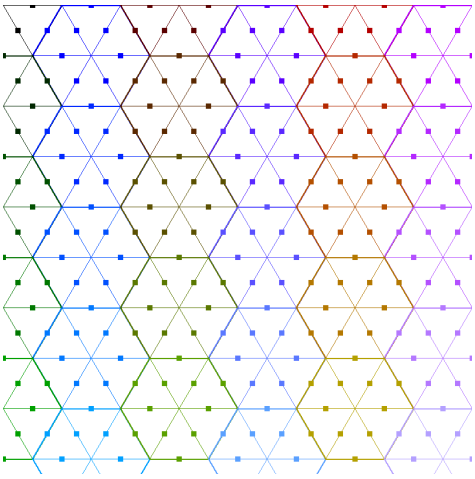
(a) Empty triangular grid



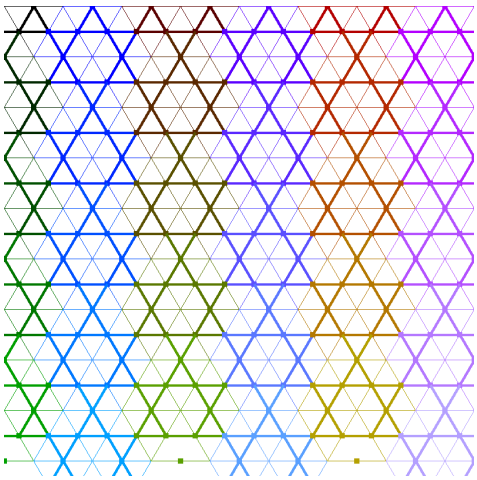
(b) Triangular grid with data at the cell centers



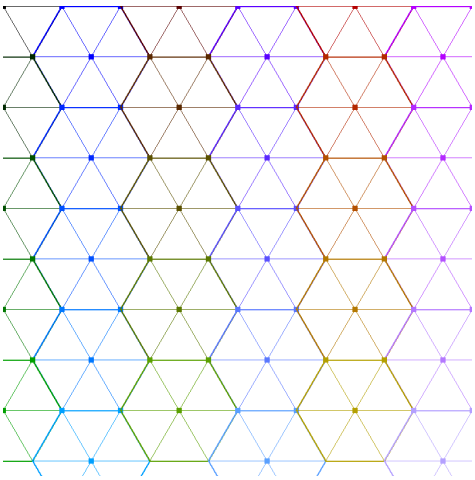
(c) Triangular grid with data at the cell centers, connected neighbours



(d) Triangular grid with data on the edges



(e) Triangular grid with data on the edges, connected neighbours



(f) Triangular grid with data on the vertices / connected neighbours

Figure 2.3: Triangular grid

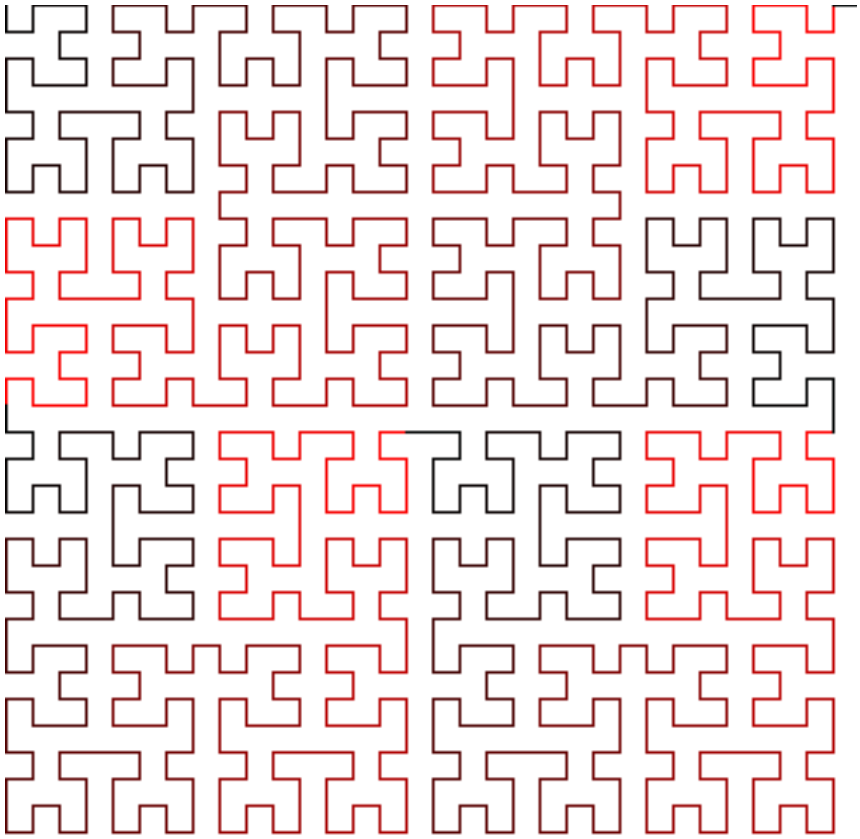


Figure 2.4: Hilbert space-filling curve

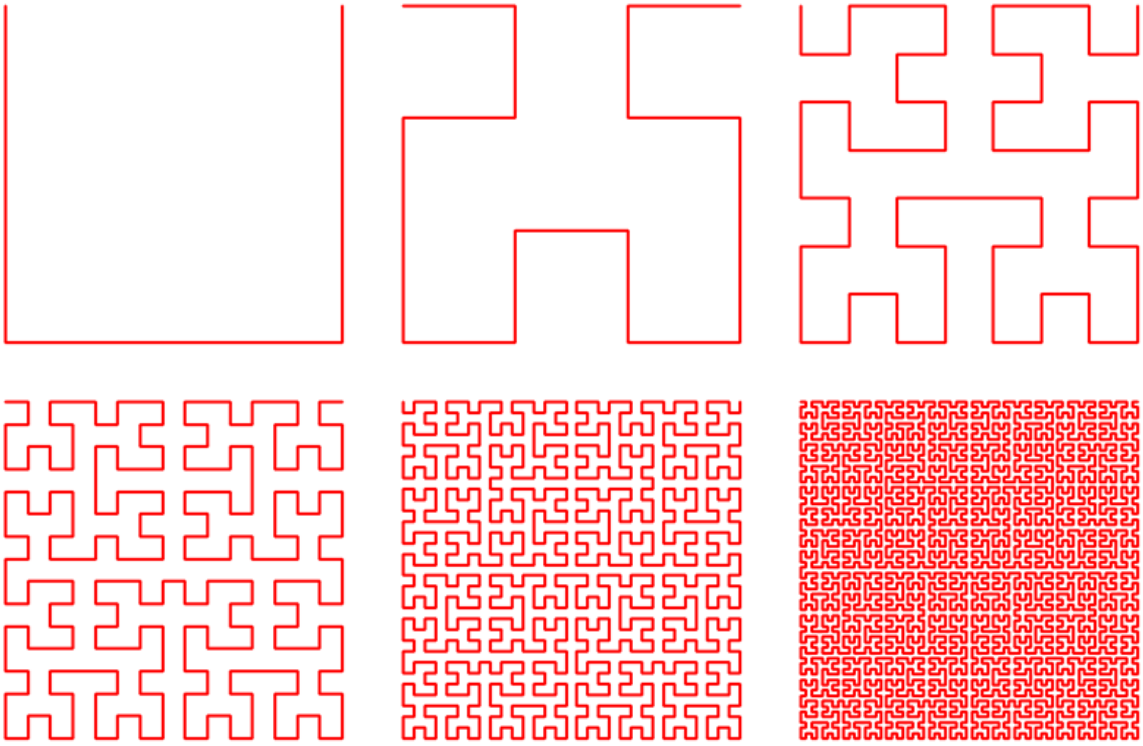


Figure 2.5: Hilbert space-filling curve

system considers the file to be an array of bytes starting from 0. This is quite similar to the 1D structure from main memory. However, a general purpose language (GPL) uses variable names to point to the data in this 1D address space. A GPL offers means to access even multi-dimensional data easily. The user/programmer does not need to know the specific addresses in memory; addresses are calculated within the execution environment or code of the application.

**TODO**: This section needs some diagrams and/or references.

The main concern here is consecutive or stride access through the array; if the programmer wishes the application to loop through a given dimension of the array, memory locations would be addressed which may not be close to each other in memory, thus leading to cache misses and hence poorer performance. The generalisation is the stride, which specifies steps through the different dimensions of the array (e.g. incrementing both dimensions of a 2D array, thus walking along the "diagonal"). Another special case is where the programmer needs to process the whole array, which would be done most efficiently by stepping through all the memory locations incrementally[1], whereas looping over the dimensions and incrementing them one at a time requires more calculations and may lead to inefficient memory access with cache misses if not done correctly[2].

When storing data from memory directly on persistent media, then the original source code is necessary to understand this data. Similarly, the interpretation of the bytes in the data must be same when reading it back, thus, the byte order and size of the datatypes of the machine reading the data must be identical to those of the machine that wrote it. Floating point numbers must be encoded in the same byte formats. Since this is not always given, it threatens the longevity of our precious data, by hindering the portability and reusability of the data.

Therefore, portable data formats have been developed that allow to serialize and de-serialize data regardless of the machine's architecture. To allow correct interpretation of a byte array, the library implementing the file format must know the data type that the bytes represent. This information must be stored besides the actual bytes representing the data to allow later reading and interpretation. From the user perspective, it is useful to also store further metadata describing the data. For instance, a name and description of the contained information. This eases not only debugging but also allows other applications to read and process data in a portable way. File formats that contain this kind of semantical and structural metadata are called **self-describing file formats**.

Developers using a self-describing file formats have to use an API to define the metadata. Such a format may support arbitrary complex data types, which implies that some kind of data description framework must be part of the API for the file format. See Section 2.3 for more information about data description frameworks.

## 2.2 File formats

[1] Assuming the whole array is stored in contiguous memory, as it is in these simple examples.

[2] Fortran historically stores 2D arrays in column-major order, whereas C and most other languages used in science store data in row-major order.

| Name | Fullname | Version | Developer |
|---|---|---|---|
| GRIB1 | GRIdded Binary | 1 | World Meteorological Organization |
| GRIB2 | GRIdded Binary | 2 | World Meteorological Organization |
| NetCDF3 | Network Common Data Form | 3.x | Unidata (UCAR/NCAR) |
| NetCDF4 | Network Common Data Format | 4.x | Unidata (UCAR/NCAR |
| HDF4 | Hierarchical Data Format | 4.x | NCSA/NASA |
| HDF4-EOS2 | HDF4-Earth Obseving System | 2 | |
| HDF5 | Hierarchical Data Format | 5.x | NCSA/NASA |
| HDF5-EOS5 | HDF5-Earth Obseving System | 5 | |

Table 2.1: Parallel data formats

**TODO**: This section is written from a "simulation" only perspective. Need to consider read/write separately, and different aspecs of apects of parallelisation. What about MPI targeted write to OSDS? What about SAFE?

Generally, parallel scientific applications are designed in such a way, they can solve complicated problems faster when running on a large number of compute nodes. This is achieved by splitting a global problem into small pieces and distributing them over the compute nodes; this is called domain decomposition. After each node has computed a local solution, they can be aggregated to one global solution. This approach can decrease time-to-solution considerably.

I/O makes this picture more complicated, especially when data is stored in one single file and is accessed by several processes simultaneously. In this case, problems can occur, when several processes access the same file region, e.g., two processes can overwrite the data of each other, or inconsistencies can occur when one process reads, while another writes. Portability is another issue: When transferring data from one platform to another, the contained information should still be accessible and identical. The purpose of I/O libraries is to hide the complexity from scientists, allowing them to concentrate on their research.

Some common file formats are listed in the Table 2.1. All of these formats are portable (machine independent) and self-describing. Self-describing means, that files can be examined and read by the appropriate software without the knowledge about the structural details of the file. The files may include additional information about the data, called "metadata". Often, it is textual information about each variable's contents and units (e.g.,"humidity" and "g/kg") or numerical information describing the coordinates (e.g., time, level, latitude, longitude) that apply to the variables in the file.

GRIB is a record format, NetCDF/HDF/HDF-EOS formats are file formats. In contrast to record format, file formats are bound to format specific rules. For example, all variable names in NetCDF must be unique. In HDF, although, variables with the same name are allowed, they must have different paths. No such rules exist for GRIB. It is just a collection of records (datasets), which can be appended to the file in any order.

GRIB-1 record (aka, 'message') contains information about two horizontal dimensions (e.g., latitude and longitude) for one time and one level. GRIB-2 allows each record to contain multiple grids and levels for each time. However, there are no rules dictating the order of the collection of GRIB records (e.g, records can be in random chronological order).

Finally, a file format without parallel I/O support, but still worth to mention, is CSV (comma-separated values). It is special due to its simplicity, broad acceptance and support by a wide range of applications. The data is stored as plain text in a table. Each line of the file is a data record. Each record consists of one or more fields, that are separated by commas (hence the name). The CSV file format is not standardized. There are many implementations that support additional features, e.g., other separators and column names.

### 2.2.1 NetCDF4

NetCDF4 with Climate Forecast (CF) metadata and GRIB evolved to the de-facto standard formats for convenient data access for the scientists in the domain of NWP and climate. For convenient data access, it provides a set of features, for example, metadata can be used to assign names to variables, set units of measure, label dimensions, and provide other useful information. The portability allows data movement between different possibly incompatible platforms, which simplifies the exchange of data and facilitates communication between scientists. The ability to grow and shrink datasets, add new datasets and access small data ranges within datasets simplifies the handling of data a lot. The shared file allows to keep the data in the same file. Unfortunately, the last feature conflicts with performance and efficient usage of the state-of-art HPC. The files, which are accessed simultaneously by several processes, cause a lot of synchronisation overhead which slows down the I/O performance. Synchronization is necessary to keep the data consistent.

The rapid development of computational power and storage capacity, and slow development of network bandwidth and I/O performance in the last years resulted in imbalanced HPC systems. The application use the increased computational power to process more data. More data, in turn, requires more costly storage space, higher network bandwidth and sufficient I/O performance on storage nodes. But due to imbalance, the network and I/O performance are the main bottlenecks. The idea is, to use a part of the computational power for compression, adding a little extra latency for the transformation while significantly reducing the amount of data that needs to be transmitted or stored.

Before considering a compression method for HPC, it is a good idea to take a look at the realization of parallel I/O in modern scientific applications. Many of them use the NetCDF4 file format, which, in turn, uses HDF5 under the hood.

### 2.2.2 Typical NetCDF Data Mapping

Listing 2.1 gives an example for scientific metadata stored in a NetCDF file. Firstly, between Line 1 and 4, a few dimensions of the multidimensional data are defined. Here there are longitude, latitude with a fixed size and time with a variable size that allows to be extended (appending from a model). Then different variables are defined on one or multiple of the dimensions. The longitude variable provides a measure in “degreees east” and is indexed with the longitude dimension; in that case the variable longitude is a 1D array that contains values for an index between 0-479. It is allowed to define attributes on variables, this scientific metadata can define the semantics of the data and provide information about the data provenance. In our example, the unit for longitude is defined in Line 7. Multidimensional variables such as `sund` (Line 45) are defined on a 2D array of values for the longitude and latitude over various timesteps. The numeric values contain a scale factor and offset that has to be applied when accessing the data; since, here, the data is stored as short values, it should be converted to floating point data in the application. The `_FillValue` indicates a default value for missing data points.

Finally, global attributes such as indicated in Line 54 ff. describe that this file is written with the NetCDF-CF schema and its history describes how the data has been derived / extracted from original data.

Listing 2.1: Example NetCDF metadata

```
dimensions:
  longitude = 480 ;
  latitude = 241 ;
  time = UNLIMITED ; // (1096 currently)
variables:
  float longitude(longitude) ;
    longitude:units = "degrees_east" ;
    longitude:long_name = "longitude" ;
  float latitude(latitude) ;
    latitude:units = "degrees_north" ;
    latitude:long_name = "latitude" ;
  int time(time) ;
    time:units = "hours since 1900-01-01 00:00:0.0" ;
    time:long_name = "time" ;
    time:calendar = "gregorian" ;

  short t2m(time, latitude, longitude) ;
    t2m:scale_factor = 0.00203513170666401 ;
    t2m:add_offset = 257.975148205631 ;
    t2m:_FillValue = -32767s ;
    t2m:missing_value = -32767s ;
    t2m:units = "K" ;
    t2m:long_name = "2 metre temperature" ;
  short sund(time, latitude, longitude) ;
    sund:scale_factor = 0.659209863732776 ;
    sund:add_offset = 21599.6703950681 ;
    sund:_FillValue = -32767s ;
    sund:missing_value = -32767s ;
    sund:units = "s" ;
    sund:long_name = "Sunshine duration" ;

// global attributes:
    :Conventions = "CF-1.0" ;
    :history = "2015-06-03 08:02:17 GMT by grib_to_netcdf-1.13.1:
    ↪ grib_to_netcdf /data/data04/scratch/netcdf-atls14-
    ↪ a562cefde8a29a7288fa0b8b7f9413f7-lFD4z9.target -o /data/data04/
    ↪ scratch/netcdf-atls14-a562cefde8a29a7288fa0b8b7f9413f7-CyGl1B.nc -
    ↪ utime" ;
}
```

**TODO**: (BNL) A proper semantic discussion to allow us to introduce "cross file " aggregation would introduce the CF standard here and then discuss the CF aggregation rules and how they need to be supported in the architecture.

### 2.2.3 SAFE

**TODO**: BNL4BNL: Need to introduce SAFE and describe how much of it is supportable within any ESD architecture - leading to a specific requirement in the logical view.

## 2.3 Data Description Frameworks

Many application developers rely on data description frameworks or libraries to manage datatypes[3]. Different libraries and middlewares provide mechanisms to describe data using

[3] A datatype is a collection of properties, all of which can be stored on storage and which, when taken as a whole, provide complete information for data conversion to or from the datatype.

basic types and to construct new ones using dedicated APIs. Datatypes are provided as a transparent conversion mechanism between internal representation (as data is represented in memory) and external representation (how data is transmitted over the network or saved to permanent storage). This section gives an overview of datatypes provided by different software packages. Starting from existing middlewares' datatype definitions, we will propose a list of basic datatypes to be supported by the ESD middleware.

### 2.3.1 MPI

The Message Passing Interface supports derived datatypes for efficient data transfer as well as compact description of file layouts (through file views). MPI defines a set of basic datatypes (or type class) from which more complex ones can be derived using appropriate data constructor APIs. Basic datatypes in MPI resemble C atomic types as shown in Table 2.2.

| Datatype | Description |
|---|---|
| MPI_CHAR | this is the traditional ASCII character that is numbered by integers between 0 and 127 |
| MPI_WCHAR | this is a wide character, e.g., a 16-bit character such as a Chinese ideogram |
| MPI_SHORT | this is a 16-bit integer between -32,768 and 32,767 |
| MPI_INT | this is a 32-bit integer between -2,147,483,648 and 2,147,483,647 |
| MPI_LONG | this is the same as MPI_INT on IA32 |
| MPI_LONG_LONG_INT | this is a 64-bit long signed integer, i.e., an integer number between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 |
| MPI_LONG_LONG | same as MPI_LONG_LONG_INT |
| MPI_SIGNED_CHAR | same as MPI_CHAR |
| MPI_UNSIGNED_CHAR | this is the extended character numbered by integers between 0 and 255 |
| MPI_UNSIGNED_SHORT | this is a 16-bit positive integer between 0 and 65,535 |
| MPI_UNSIGNED_LONG | this is the same as MPI_UNSIGNED on IA32 |
| MPI_UNSIGNED | this is a 32-bit unsigned integer, i.e., a number between 0 and 4,294,967,295 |
| MPI_FLOAT | this is a single precision, 32-bit long floating point number |
| MPI_DOUBLE | this is a double precision, 64-bit long floating point number |
| MPI_LONG_DOUBLE | this is a quadruple precision, 128-bit long floating point number |
| MPI_C_COMPLEX | this is a complex float |
| MPI_C_FLOAT_COMPLEX | same as MPI_C_COMPLEX |
| MPI_C_DOUBLE_COMPLEX | this is a complex double |
| MPI_C_LONG_DOUBLE_COMPLEX | this is a long double complex |
| MPI_C_BOOL | this is a _Bool |
| MPI_INT8_T | this is a 8-bit integer |
| MPI_INT16_T | this is a 16-bit integer |

| | |
|---|---|
| MPI_INT32_T | this is a 32-bit integer |
| MPI_INT64_T | this is a 64-bit integer |
| MPI_UINT8_T | this is a 8-bit unsigned integer |
| MPI_UINT16_T | this is a 16-bit unsigned integer |
| MPI_UINT32_T | this is a 32-bit unsigned integer |
| MPI_UINT64_T | this is a 64-bit unsigned integer |
| MPI_BYTE | this is an 8-bit positive integer |
| MPI_PACKED | - |

Table 2.2: MPI Datatypes

Datatypes from Table 2.2 can be used in combination with the constructor APIs shown in Table 2.3 to build more complex derived datatypes.

| Datatype Constructor | Description |
|---|---|
| MPI_Type_create_hindexed | create an indexed datatype with displacement in bytes |
| MPI_Type_create_hindexed_block | create an hindexed datatype with constant-sized blocks |
| MPI_Type_create_indexed_block | create an indexed datatype with constant-sized blocks |
| MPI_Type_create_keyval | create an attribute keyval for MPI datatypes |
| MPI_Type_create_hvector | create a datatype with constant stride given in bytes |
| MPI_Type_create_struct | create a MPI datatype from a general set of datatypes, displacements and block sizes |
| MPI_Type_create_darray | create a datatype representing a distributed array |
| MPI_Type_create_resized | create a datatype with a new lower bound and extent from an existing datatype |
| MPI_Type_create_subarray | create a datatype for a subarray of a regular, multidimensional array |
| MPI_Type_contiguous | create a contiguous datatype |

Table 2.3: MPI Derived Datatypes Constructors

Before they can be actually used, MPI derived datatypes (created using the constructors in Table 2.3) have to be committed to memory using the `MPI_Type_commit` interface. Similarly, when no longer needed, derived datatypes can be freed using the `MPI_Type_free` interface. Unlike data format libraries, MPI does not provide any permanent data representation (MPI-IO can only read/write binary data), therefore derived datatypes are not used to store any specific data format on stable storage and are instead used only for data transfers or file layout descriptions.

An example code for defining a derived data structure for a structure is shown in Listing 2.2. The structure is defined in Lines 5-9. The function in Lines 12-22 registers this datatype in MPI. This requires to define the beginning and end of each array, its type and size. Once a datatype is defined, it can be used as memory type in subsequent operations. In this example, on process sends this datatype to another process (Line 38 and Line 45).

Since MPI datatypes were initially designed for computation and, thus, to define memory

regions, they do not offer a way to name the data structures.

Listing 2.2: Example construction of an MPI datatype for a structure

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>

typedef struct student_t_s {
    int id[2];
    float grade[5];
    char name[20];
} student_t;

/* create a type for the struct student_t */
void create_student_datatype(MPI_Datatype * mpi_student_type){
    int   blocklengths[3] = {2, 5, 20};
    MPI_Datatype types[3] = {MPI_INT, MPI_FLOAT, MPI_CHAR};
    MPI_Aint     offsets[3];

    offsets[0] = offsetof(student_t, id) ;
    offsets[1] = offsetof(student_t, grade);
    offsets[2] = offsetof(student_t, name);
    MPI_Type_create_struct(3, blocklengths, offsets, types,
    ↪ mpi_student_type);
    MPI_Type_commit(mpi_student_type);
}

int main(int argc, char **argv) {
    const int tag = 4711;
    int size, rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Datatype mpi_student_type;
    create_student_datatype(& mpi_student_type);

    if (rank == 0) {
        student_t send = {{1, 2}, {1.0, 2.0, 1.7, 2.0, 1.7}, "Nina
    ↪ Musterfrau"};
        const int target_rank = 1;
        MPI_Send(&send,   1, mpi_student_type, target_rank, tag,
    ↪ MPI_COMM_WORLD);
    }
    if (rank == 1) {
        MPI_Status status;
        const int src=0;
        student_t recv;
        memset(& recv, 0, sizeof(student_t));
        MPI_Recv(&recv,   1, mpi_student_type, src, tag, MPI_COMM_WORLD,
    ↪  &status);
        printf("Rank %d: Received: id = %d grade = %f student = %s\n",
    ↪ rank, recv.id[0], recv.grade[0], recv.name);
    }

    MPI_Type_free(&mpi_student_type);
    MPI_Finalize();

    return 0;
}
```

### 2.3.2 HDF5

HDF5 is a data model, library, and file format for storing and managing data. It supports an unlimited variety of datatypes, and is designed for flexible and efficient I/O and for high volume and complex data. HDF5 is portable and is extensible, allowing applications to evolve in their use of HDF5. The HDF5 Technology suite includes tools and applications for managing, manipulating, viewing, and analyzing data in the HDF5 format. Like MPI, HDF5

also supports its own basic (native) datatypes reported in Table 2.4.

| Datatype | Corresponding C Type |
|---|---|
| H5_NATIVE_CHAR | char |
| H5_NATIVE_SCHAR | signed char |
| H5_NATIVE_UCHAR | unsigned char |
| H5_NATIVE_SHORT | short |
| H5_NATIVE_USHORT | unsigned short |
| H5_NATIVE_INT | int |
| H5_NATIVE_UINT | unsigned int |
| H5_NATIVE_LONG | long |
| H5_NATIVE_ULONG | unsigned long |
| H5_NATIVE_LLONG | long long |
| H5_NATIVE_ULLONG | unsigned long long |
| H5_NATIVE_FLOAT | float |
| H5_NATIVE_DOUBLE | double |
| H5_NATIVE_LDOUBLE | long double |
| H5_NATIVE_B8 | 8-bit unsigned integer or 8-bit buffer in memory |
| H5_NATIVE_B16 | 16-bit unsigned integer or 16-bit buffer in memory |
| H5_NATIVE_B32 | 32-bit unsigned integer or 32-bit buffer in memory |
| H5_NATIVE_B64 | 64-bit unsigned integer or 64-bit buffer in memory |
| H5_NATIVE_HADDR | haddr_t |
| H5_NATIVE_HSIZE | hsize_t |
| H5_NATIVE_HSSIZE | hssize_t |
| H5_NATIVE_HERR | herr_t |
| H5_NATIVE_HBOOL | hbool_t |

Table 2.4: HDF5 Native Datatypes

Besides the native datatypes, the library also provides so called standard datatypes, architecture specific datatypes (e.g., for i386), IEEE floating point datatypes, and others. Datatypes can be built or modified starting from the native set of datatypes using the constructors listed:

| Datatype Constructor | Description |
|---|---|
| H5Tcreate | Creates a new datatype of the specified class with the specified number of bytes. This function is used only with the following datatype classes: H5T_COMPOUND, H5T_OPAQUE, H5T_ENUM, H5T_STRING. Other datatypes, including integer and floating-point datatypes, are typically created by using H5Tcopy to copy and modify a predefined datatype |

| | |
|---|---|
| H5Tvlen_create | Creates a new one-dimensional array datatype of variable-length (VL) with the base datatype. The base type specified for the VL datatype can be any HDF5 datatype, including another VL datatype, a compound datatype, or an atomic datatype |
| H5Tarray_create | Creates a new multidimensional array datatype object |
| H5Tenum_create | Creates a new enumeration datatype based on the specified base datatype, dtype_id, which must be an integer datatype |
| H5Tcopy | Copies an existing datatype. The returned type is always transient and unlocked. A native datatype can be copied and modified using other APIs (e.g. changing the precision) |
| H5Tset_precision | Sets the precision of an atomic datatype. The precision is the number of significant bits |
| H5Tset_sign | Sets the sign property for an integer type. The sign can be unsigned or two's complement |
| H5Tset_size | Sets the total size in bytes for a datatype |
| H5Tset_order | Sets the byte order of a datatype (big endian or little endian) |
| H5Tset_offset | Sets the bit offset of the first significant bit |
| H5Tset_fields | Sets the locations and sizes of the various floating-point bit fields. The field positions are bit positions in the significant region of the datatype. Bits are numbered with the least significant bit number zero |

Table 2.5: HDF5 Datatypes Constructors

HDF5 constructs allow the user a fine grained definition of arbitrary datatypes. Indeed, HDF5 allows the user to build a user-defined datatype starting from a native datatype (by copying the native type) and then change datatype characteristics like sign, precision, etc, using the supported datatype constructor API. However, since these user defined data types have often no direct representation on available hardware, this can lead to performance issues.

### 2.3.3 NetCDF

NetCDF as important alternative is popular within the climate community. NetCDF provides a set of software libraries and self-describing, machine-independent, data formats that support the creation, access, and sharing of array-oriented scientific data. In the version 4 of the library (NetCDF4), the used binary file representation is HDF5. Like MPI and HDF5, NetCDF also defines its own set of atomic datatypes as shown in Table 2.6.

Similarly to HDF5 and MPI, in addition to the atomic types the user can define its own types. NetCDF supports four different user defined types:

| Datatype | Description |
|---|---|
| NC_BYTE | 8-bit signed integer |
| NC_UBYTE | 8-bit unsigned integer |
| NC_CHAR | 8-bit character byte |
| NC_SHORT | 16-bit signed integer |
| NC_USHORT | 16-bit unsigned integer |
| NC_INT | 32-bit signed integer |
| NC_UINT | 32-bit unsigned integer |
| NC_INT64 | 64-bit signed integer |
| NC_UINT64 | 64-bit unsigned integer |
| NC_FLOAT | 32-bit floating point |
| NC_DOUBLE | 64-bit floating point |
| NC_STRING | variable length character string |

Table 2.6: netCDF Atomic External Datatypes

1. **Compound**: are a collection of types (either user defined or atomic)
2. **Variable Length Arrays**: are used to store non-uniform arrays
3. **Opaque**: only contain the size of each element and no datatype information
4. **Enum**: like an enumeration in C

Once types are constructed, variables of the new type can be instantiated with nc_def_var. Data can be written to the new variable using nc_put_var1, nc_put_var, nc_put_vara, or nc_put_vars. Data can be read from the new variable with nc_get_var1, nc_get_var, nc_get_vara, or nc_get_vars. Finally, new attributes can be added to the variable using nc_put_att and existing attributes can be accessed from the variable using nc_get_att.

Table 2.7 shows the constructors provided to build user defined datatypes.

| Type | Constructor | Description |
|---|---|---|
| Compound | nc_def_compound | create a compound datatype |
| | nc_insert_compound | insert a name field into a compound datatype |
| | nc_insert_array_compound | insert an array field into a compound datatype |
| | nc_inq_{compound,name,...} | learn information about a compound datatype |
| Variable Length Array | nc_def_vlen | create a variable length array |
| | nc_inq_vlen | learn about a variable length array |
| | nc_free_vlen | release memory from a variable length array |
| Opaque | nc_def_opaque | create an opaque datatype |
| | nc_inq_opaque | learn about an opaque datatype |
| Enum | nc_def_enum | create an enum datatype |
| | nc_insert_enum | insert a named member into an enum datatype |
| | nc_inq_{enum,...} | learn information about an enum datatype |

Table 2.7: NetCDF Datatypes Constructors

### 2.3.4 GRIB

GRIB is a library and data format widely used in weather applications. It differs from previously described libraries in the sense that it does not define datatypes that can be used to store a wide range of different data. Instead, GRIB very clearly defines the sections of every – so called – message which is the unit sent across a network link or written to permanent storage. Every message in GRIB has different sections, each of which contains some information. GRIB defines the units that can be represented in every message and thus does not specifically need datatypes to represent them. But a library supporting these formats needs to know the mapping from a message type to the contained data fields and their data types. In that sense, GRIB is not a self-describing file format but requires code to define the standardized content. GRIB messages contain 32-bit integers that can be scaled using a predefined data packing schema. The scaling factor is stored along with the data inside the message.

## 2.4 Storage Components

**TODO**: This section was originally about "big data" storage, need to refocus to be a description of key elements of the storage components we currently care about

In the context of Big Data, there are many (typically Java based) technologies that address storing and processing of large quantities of data.

The Hadoop File System (HDFS) is a distributed file system that is designed to work with commodity hardware. It provides fault tolerance via data replication and self healing. One limitation of its design is its consistency semantics which allows concurrent reads of multiple processes but only a single writer. Its suboptimal performance on high-performance storage and assumption to work on cheap hardware makes it no optimal choice for HPC environments. Therefore, many vendors support HDFS adapters on top of high-performance parallel file systems such as GPFS and Lustre.

Hive utilizes data stored on HDFS and other Big Data solutions and offers a convenient SQL interface to this data. This allows users to explore data using SQL by accessing data stored on the file system. The advantage is that no extract, transform, load (ETL) process is necessary; simply move the data into the file system, create a scheme on the existing files.

Drill [4] also provides an SQL interface to existing data. Similar to Hive existing data can be adjusted, but in the case of Drill, data may be stored on various storage backends such as simple JSON file, on Amazon S3, or MongoDB.

Alluxio [5] offers a scalable in-memory file system. An interesting feature is that one can attach (mount) data from multiple (even remote) endpoints such as S3 into the hierarchical in-memory namespace. It provides control to the in-memory data, for example, to trigger a flush of dirty data to the storage backend and an interface for pinning data in memory (similar to burst buffer functionality). Data stored on Alluxio can be used on various big data tools.

[4] `https://drill.apache.org`
[5] `https://www.alluxio.com/docs/community/1.3/en/`

### 2.4.1 MongoDB

The MongoDB[6] is an open-source document database. Its architecture is high-performant and horizontally scalable for cluster systems. MongoDB offers a rich set of interfaces, e.g., RESTful access, C, Python, Java.

The data model of MongoDB provides three levels:

- Database: follows our typical notion; permissions are defined on the database level.
- Document: This is a BSON object (binary JSON) – consisting of subdocuments with data. An example as JSON is shown in Listing 2.3. Each document has the primary key field: _id. The field must be either manually set or it will be automatically filled.
- Collection: this is like a table of documents in a database. Documents can have individual schemas. It supports indices on fields (and compound fields).

To access data, one has to know the name of a database (potentially secured with a username and password), collection name. All documents within the collection can be searched or manipulated with one operation.

In the example of Listing 2.3, it would also be possible to create one document for each person and use the *_id* field with a self-defined unique ID such as a tax number.

Listing 2.3: Example MongoDB JSON document

```
1 "_id" : ObjectId("43459bc2341bc14b1b41b124"),
2 "people" : [ # subdocuments:
3     { "name" : "Max", "id" : 4711, "birth" : ISODate("2000-10-01")},
4     { "name" : "Lena", "id" : 4712, "birth", ... }
5   ]
```

MongoDB's architecture uses sharding of document keys to partition data across different servers. Servers can be grouped into replica sets to provide high availability and fault tolerance.

**Query documents** A query document is a BSON document that is used to search all documents of a collection for data that matches the defined query. The example in Listing 2.4 specifies documents that contain the subdocument people with an id field that is bigger than 4711. Complex queries can be defined. In combination with indices on fields, MongoDB can search large quantities of documents quickly.

Listing 2.4: Example MongoDB Query document

```
1 { "people.id" : { $gt : 4711 } }
```

[6] https://docs.mongodb.com/

# 3 Requirements

The goal of this section is to provide high level requirements that provide a generic view of what the system needs to do and how it relates to dependencies.

## 3.1 Functional Requirements

1. Import/Export of data in or out of the ESD system. Concomitant requirements include
   - A defined list of "ESD supported" file formats.
   - Conventions for the internal metadata of those files and/or for the content and format of any "supplementary" metadata needed to meet any internal ESD metadata requirements.
2. Discover data within the ESD system.
3. Browse available data within ESD system.
4. List available data within the ESD system.
5. Create, Retrieve, Update, Delete data ... (including Append where that concept makes sense).
6. Handling of scientific/structural metadata as first class citizen
7. Exposing data via (at least):
   a) a new ESD interface, suitable for use in the high performance simulation environments to be expected on next generation machines,
   b) a POSIX file system interface,
   c) an HDF interface suitable to be bound into client HDF libraries deployed both locally (LAN) and remotely (WAN),
   d) a GridFTP interface,
   e) a suitable (new) REST interface, and
   f) a suitable web HTTP catalogue interface.

   Other libraries and tools as required and effort permits.
8. Appropriate access control (to be defined).
9. System wide configuration of available storage resources.
10. Notifications, Publish, Subscribe.

Data access presumes the ability to:

11. identify the file or object which contains interesting data, possibly by consulting metadata catalogues, and eventually obtaining an identifier for the object and an endpoint through which it can be accessed;
12. to ask to have the object made available

- This step may require obtaining permissions to access the data, and/or acceptance of a licence;
- The object may need to be brought up from lower tier storage (see section 3.3), and
- For some more advanced uses of data, endpoints may also be dynamically allocated.

13. either transfer the whole file, or,
14. identify sections of interest in the file/object and remotely access those sections.

### 3.1.1 Additional Non-Functional Requirements

A set of over-arching non-functional requirements were introduced in Section 1.1.1, the need to be:

1. Performant — coping with volume/velocity and delivering adequate bandwidth and latency.
2. Cost-Effective — affordable in both financial and environmental terms at exascale.
3. Reliable — storage is durable, data-corruption in transit is detected and corrected,
4. Transparent — hides specifics of the storage landscape and does not ***require*** users to change parameters specifically to a given system.
5. Portable — should work in different environments.
6. Standards based — using interfaces, formats and standards which maximise re-usability.

In addition, we can add

7. Versatile — able to utilise heterogeneous storage, including, but not limited to
   - existing traditional storage systems such as tape, object store, and parallel disk,
   - existing non-traditional storage systems such as the HDFS, and
   - potential new storage systems such as burst buffers, and new variants of hyper-convergence.
8. Fault Tolerant — able to detect and repair hardware and software failures.
9. Highly-available — meeting expectations of users similar to that they have of conventional disk sub-systems.

## 3.2 Roles and Actions

**TODO**: This section probably better belongs in the Logical View

### Actors

There are three actors who can interact with the ESIWACE storage resource:

- Unprivileged Users
- Priviliged Users
- Administrators

In the following, we use the term object to refer primarily to something with equivalent semantics to a file. A more fine grained object access will also be available via any APIs exposed by the service.

### Unprivileged User

An unprivileged user is someone who has only readonly access. These users can:

- navigate content, using faceted browse against public tags,
- list all (public) tags to which they have access,
- given a tag, list all tags carried by objects with the first tag,
- given a list of tags, list all tags carried by objects with all members of that list,
- given a tag list, list all objects with the union set of all those tags,
- retrieve any visible object from the list of objects presented by any tag list,
- interact with any visible object via limited read only operations.

### Privileged Users

A privileged user is someone who has CRUD access to (their) content within the archive as well as all the abilities of an unprivileged user applied to their own content. They can:

- create, retrieve, update, and delete content within prescribed quotas,
- control access to their objects (see below),
- assign tags to objects,
- navigate both public or (own) private tags.

Controlling access:

- Users can create “group” identifiers, and associate user identifiers with that group.
- They must themselves be a member of any group they create.
- They can add/remove any other user identifiers known to them to that group.
- How users find the identifiers of other users is not defined here.
- If they use the identifier “public” for a group, then users in this group (who may also include the special identifier ”anonymous”), then users in this group will have readonly access to these objects.

- Users can assign any group identifer of which they are a member to any object they create. In doing so, they make "their" objects into "shared" objects (except for the public group as defined above, where they are simply making the object readonly to that group).
- Any user with "shared" access has the same privileges for that object as the original owner, except that of modifying or removing the group tag.
  - This means they can delete, update, and retreive the object. Of course deleting it will disassociate the group tag.
- Users can list the groups of which they are members, and list the members of any of those groups.

(Note that this usage of group is not identical to the concept of UNIX groups, not least, because users control their definition.)

#### Administrators

Administrators can

- start and Stop the service,
- access all data held by all priviledged users,
- manage privileged users: create, update, delete users,
- allocate Quotas,
- retreive usage information,
- configure the layout of content in the service against available storage resources,
- migrate content within the storage resources (a process that might temporarily disable user access),
- configure any required compute, cache, and network services.

#### Consequences

- API needs to reflect some sort of access token to limit some users to readonly operations.
- All operations need to be logged, in particular, all creations, retrievals, and updates discriminated by users.

## 3.3 Supporting a Storage Hierarchy

Currently most weather and climate centres deploy an HPSS system of some sort in their environment, which is usually very costly. The ESD storage system will need to have at least some sort of internal hierarchy and caching system.

This will need to support

1. Explicit migration, where for example, users explicitly tag their data for a "lower" tier of storage (cheap and/or slow), but the ESD system needs to cache the data enroute to tape.
2. Overflow, where for example a particular deployed ESD system is unable to handle new data stores to disk without flushing old data to tape, and

3. Transparent (and/or non-transparent) data migration (where for example) data migrates from tape to disk in response to full or partial read requests through one of the ESD interfaces.

It might also need to support

4. caching data to and from burst buffers.

## 3.4 Related Work and Dependencies

In this section we discuss the influence of important third party architectures on the ESD requirements.

### 3.4.1 The Highly Scalable Data Storage Project

The HDF group in ...

**TODO**: Update the description of the HSDS project

We here repeat some of the requirements and assumptions from that project, recognising that because we desire that the ESD is as compatible with the HSDS system as possible, most of these assumptoins and requirements apply to the ESD system as well.

**Key Object Storage Assumptions** . The HSDS project makes the following assumpions about objects and the interactions with the object storage system:

1. Files and Objects: Files are shared into objects.
2. Key size: Object keys are limited to 1024 characters.
3. Key names: The first 3-4 characters of the keys should be randomly distributed (to avoid request rate limits due to a single storage system be targeted for systems such as AWS where key names influence object location).
4. Object size needs to follow the "goldilocks principle" needing to be not too small, and not too large: dealing with too many small objects would be prohibitively expensive in a public cloud and the use of very large objects ($>$ 100 MB) would introduce excessive latency.
5. Updates to a storage object will be complete (i.e. the entire object will be overwritten), atomic (i.e. last writer wins), and either succeed or fail — and in the latter case there should be no update to the object.
6. Partial reads could be possible, though.
7. The object storage system will not provide support for "transactions" (i.e. the "all or nothing" update of two or more objects).
8. The storage system will not be read-write consistent.
9. Metadata associated with each object will be at least 1024 bytes long.
10. All objects managed by any service instance will exist in one "bucket" for which that service will need read-write authority.

With these assumptions, we might expect that the HSDS (and hence ESD services) would need to perform a number of data management functions that are not provided by the underlying object storage paradigm:

11. Listing of files, and file permissions, will need to be managed by a service (listing keys will generally be inefficient (and would not work well with randomly distributed keys), and file permissions are notions that belong in HSDS/ESD metadata space, not in the storage system).

12. The reverse lookup between an object and the parent file will need to be explicilty managed by a service (an object will not itself know of its parent).

13. All object creation and update needs to be managed through a service interface. Object retrieval may be either via a service which aggregates objects on the server side, or via a client which aggregates objects after interrogating a service to find appropriate objects.

These functions are naturally provided by traditional (POSIX) file systems.

# 4 Architecture: Viewpoints

We begin a discussion of the architecture by providing an overview of the ESD middleware before discussing each of the perspectives in more detail.

## 4.1 Architectural Overview and Development View

**Problem Summary** The ESD middleware has been designed to deal with the fact that existing data libraries for standardized data description and optimized I/O such as NetCDF, HDF5 and GRIB do not have suitable performance portable optimisations which reflect current data intensive system architectures and deliver cost-effective, acceptable data access bandwidth, latency and data durability.

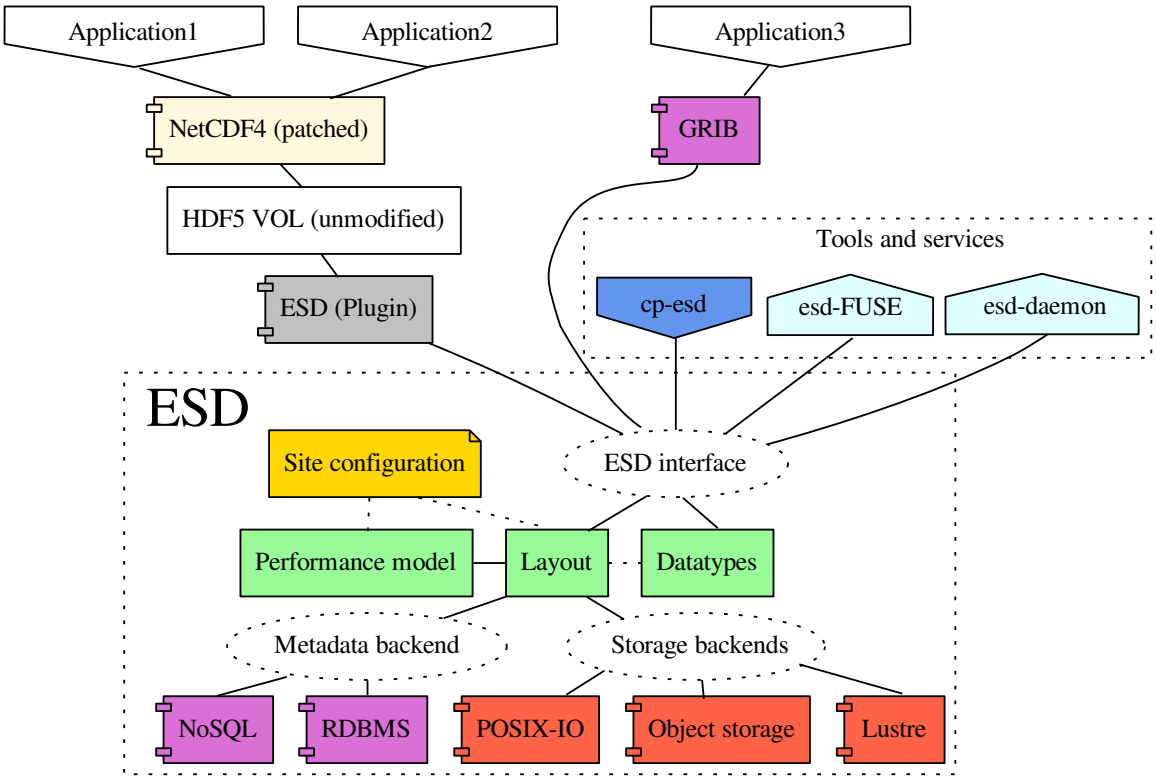



Figure 4.1: Summary overview of ESD architecture and relationship to application and storage systems.

**The ESD Middleware** To address these issues of performance portability, and exploiting exiting shared, interoperable interfaces, based on open standards, we have designed the *Earth System Data (ESD)* middleware, which:

1. understands application data structures and scientific metadata, which lets us expose the same data via different APIs;
2. maps data structures to storage backends with different performance characteristics based on site specific configuration informed by a performance model;

3. yields best write performance via optimized data layout schemes that utilize elements from log-structured file systems;

4. provides relaxed access semantics, tailored to scientific data generation for independent writes, and;

5. includes a FUSE module which will provide backwards compatibility through existing file formats with a configurable namespace based on scientific metadata.

Together these allow storing small and frequently accessed data on node-local storage, while serializing multi-dimensional data onto multiple storage backends – providing fault-tolerance and performance benefits for various access patterns at the same time. Compact-on-read instead of garbage collection will additionally optimize and replicate the data layout during reads via a background service. Additional tools allow data import/export for exchange between sites and external archives.

The ESD aids the interests of stakeholders: developers have less burden to provide system specific optimizations and can access their data in various ways. Data centers can utilize storage of different characteristics.

**ESD Interface** This represents the API exposed to other libraries and users. The API will be independent from the specific I/O backend used to store the data and will support structured queries to perform complex data selections in the variables. The API will be able to support the complex workflows of future applications, while backward compatibility with legacy codes will be provided through a FUSE module.

**ESD Datatype Component** The datatype component will provide native ESD middleware datatypes that can be used by users or other libraries to describe data points inside variables. We follow the approach pursued by the HDF5 library, that is, we provide a set of native datatypes and a basic set of datatype constructors that can be used to build custom derived datatypes.

**ESD Layout Component** The layout component allows the middleware to store pieces of data on different backends depending on specific site configuration contained in the performance model. The layout component in this case takes responsibility for generating additional technical metadata describing data placement and for storing it in the appropriate metadata backend (i.e. MongoDB). A more detailed description of what technical metadata is, is given in the rest of this section.

**Incomplete Document**

This version of the ESIWACE middleware architecture does not include the remainder of the material because at this time (March 9, 2017), much material has either been removed for re-ordering, or it is subject to reconsideration as a consequence of the first deliverable review.