UNIVERSITY OF CALIFORNIA
SANTA CRUZ

OPTIMIZING ACCESS TO SCIENTIFIC DATA FOR STORAGE, ANALYSIS AND VISUALIZATION

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Latchesar Ionkov

March 2018

The Dissertation of Latchesar Ionkov
is approved:

Carlos Maltzahn, Chair

Scott Brandt

Katia Obrazcka

Maya Gokhale

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Optimizing Access to Scientific Data for Storage, Analysis and Visualization

by

Latchesar Ionkov

Scientific workflows contain an increasing number of interacting applications, often with big disparity between the formats of data being produced and consumed by different applications. This mismatch can result in performance degradation as data retrieval causes multiple read operations (often to a remote storage system) in order to convert the data. In recent years, with the large increase in the amount of data and computational power available there is demand for applications to support data access in-situ, or close-to simulation to provide application steering, analytics and visualization.

Although some parallel filesystems and middleware libraries attempt to identify access patterns and optimize data retrieval, they frequently fail if the patterns are complex. It is evident that more knowledge of the structure of the datasets at the storage systems level will provide many opportunities for further performance improvements.

For most developers of scientific applications, storing the application data, and its particular format on disk, is not an essential part of the application. Although they acknowledge the importance of the I/O performance, their expertise lies mostly in numerical simulations and the particular models their application simulates. Most of their efforts are spent of ensuring that the it produces correct numerical results. Ideally, they would like to be able to have a library call that reads a subset of the data from storage (no matter what its format is), and place it in the data structures the simulation defines in the computer memory. Since the data needs to be analyzed and visualized, and the data has to be accessible from third-party tools, the scientists are forced to know more about the data formats.

In this dissertation we investigate multiple techniques for utilizing dataset description for improving performance and overall data availability for HPC applications. We introduce a declarative data description language that can be used to define the complete dataset as well as parts of it. These descriptions are used to generate transformation rules that allow data to be converted between different physical layouts on storage and in memory.

First, we define the DRepl dataset description language and use it to implement divergent data views and replicas as POSIX files. We evaluate the performance for this approach and demonstrate its advantages both because of the transparent application use, and combined performance when the application is combined with analytics and/or visualization code that reads the data in different format. DRepl decouples the data producers and consumers and the data layouts they use from the way the data is stored on the storage system. DRepl has shown up to 2x for cumulative performance when data is accessed using optimized replicas.

Second, we extend the previous approach to the parallel environment used in HPC. Instead of using POSIX files, the new method allows data to be accessed in larger chunks (fragments) in the way it will be laid out in memory. The developers can define what data structures they have in the process' memory and the overall format of the dataset on storage, and the runtime will automatically take care of transforming the data between the two. Both the formats in memory and on disk are described with the DRepl language. Replacing the ability for reading the data as an array of bytes with operations that use descriptions of the data structure, provides better opportunities for the storage system to optimize the access to the persistent data. The integration of this technique in Ceph demonstrates the potential advantages for this approach. The experiments show performance improvements up to 5 times for writes and 10 times for reads, compared to collective MPI I/O.

Third, we explore the future directions of extending the DRepl language to support more complex datasets. The additions would allow scientists to use different resolutions for different parts of a multi-dimensional spaces, and define how to transform the data between resolutions. The changes would also allow completely abstract definitions of datasets not only for continuums, but also for primitive types like real and integer numbers. The fragments of the dataset that are present in memory or disk will have concrete types that are compatible with the abstract types used in the dataset.

Finally, we provide foundations on how to extend the previous functionality to the most complicated data structures used in scientific applications – unstructured meshes.

I dedicate this thesis to my family for their support and patience.

Acknowledgments

The text of this dissertation includes material that was previously published in technical reports [41], at 29th Symposium on Mass Storage Systems and Technologies (MSST) [40], and 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems [42]. Carlos Maltzahn and Scott Brandt directed and supervised the research presented the dissertation.

Thanks to Carlos Maltzahn, Scott Brandt, Michael Lang, and Noah Watkins for their advice and collaboration. Thanks to my co-workers at LANL as well as the leadership of the HPC and CCS divisions for their support and patience.

Chapter 1

Introduction

Scientific models are defined using concepts that can't be represented in digital computers. They employ multidimensional continuums and real numbers with infinite precision. Creating meaningful computer simulations for these models created a significant field of computing known as numerical analysis. The variety of approaches for minimizing numerical errors introduced a variety of data structures for describing multidimensional continuums. Their popularity depends on the overall computing environment: the programming languages used, the processing elements, the available memory, and the number of nodes. Although saving data on persistent storage is a known bottleneck for HPC environments, the task for defining its format on disk is largely left to the application developers.

1.1 Traditional Data Storage for HPC

Traditionally storage systems work with two main types of data structures. The most popular (as available on any computer) way is dealing with streams of bytes organized in hierarchical namespaces. A common interface for this approach is POSIX I/O. The application developers are responsible for creating routines for serializing and deserializig the application data structures to one or many byte arrays. Another method is using relational databases where data with static structure is stored in tables with multiple fields, and a declarative language is used to describe relations between the tables, as well as querying and updating the data. This approach works well with data that consists of millions of uniform records and clear relations, but is not optimal for hierarchical or graph-like data structures.

System configuration in High Performance Computing differs substantially from other popular setups. Figure 1.1 shows a typical organization of an HPC cluster. There are two main subsystems: the compute cluster and the parallel file system. The compute cluster consists of head node(s), compute nodes (CN), and I/O forwarding nodes (IO). They are connected by a high-bandwidth, low-latency network. The compute nodes don't have any local storage. The applications access the parallel file system through the I/O forwarding layer. In the general case, the parallel file system is connected to a different network. The I/O nodes are connected both to the compute and the file system networks. The users can connect to, and use the cluster by logging in to the cluster's compute node(s).

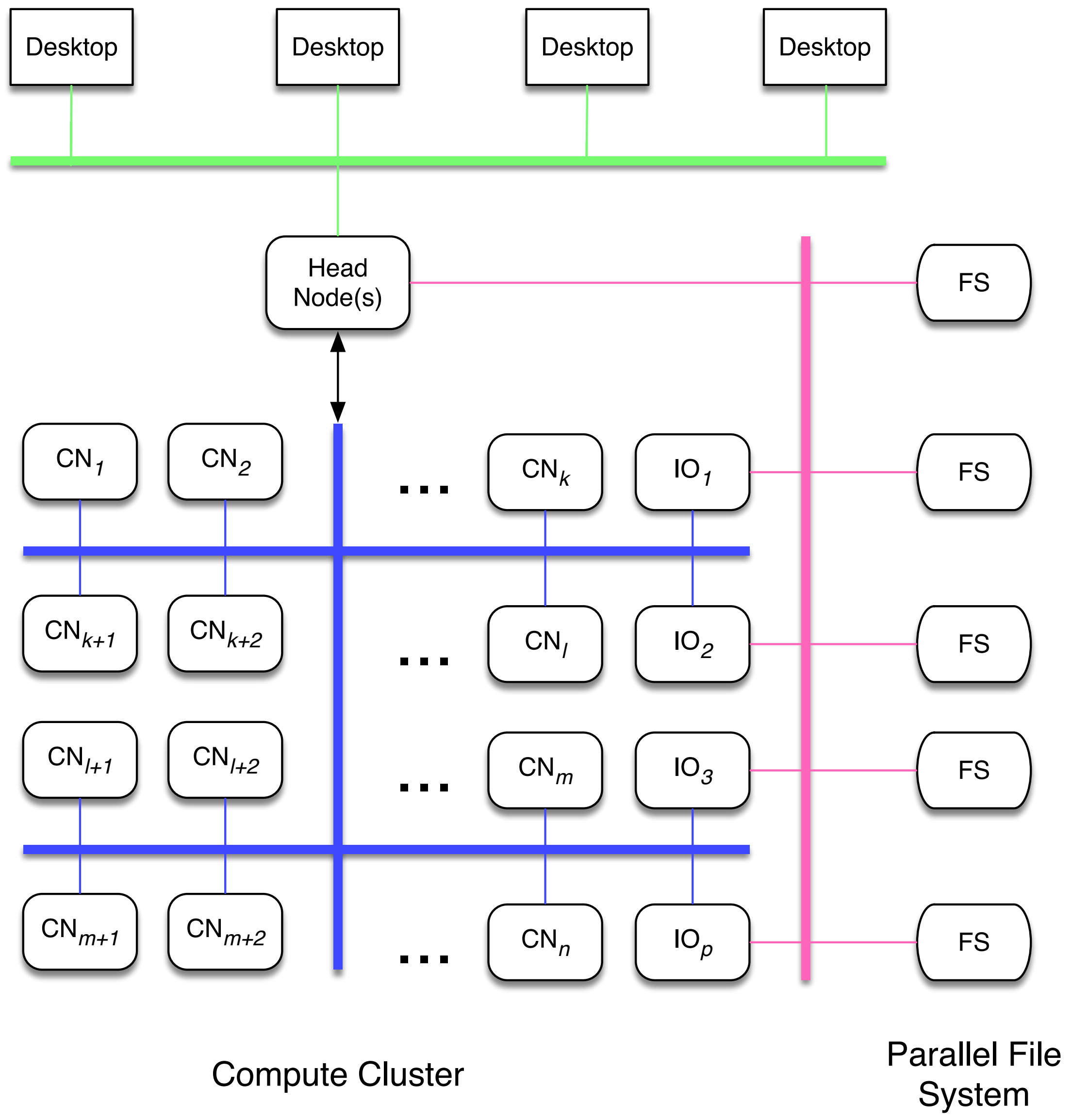


Figure 1.1: Typical HPC cluster. CN depicts the cluster's compute nodes, IO is the I/O forwarding nodes, and FS are the file system storage nodes.

Although there are some advances in using the latter, the vast majority of the scientific applications use the former technique. The dataset semantics and structure are not known to the storage system, as the translation from the memory data structures to a stream of bytes is performed either by the application itself, or by some middleware libraries that it uses. Some of these libraries, for example HDF5, save the structure of the data (also known as metadata) as part of the byte stream, but the storage system cannot tell apart metadata from data and therefore has has no information on the structure of the data.

1.2 New Challenges to the Storage Systems in HPC

The amount of data produced by scientific applications increases with the aggregate memory size of the high-performance supercomputers they run on. Future exascale systems will require hundreds of petabytes storage just to satisfy the need for scratch space [32]. It may be prohibitive to transfer all data produced by exascale simulations outside of the compute cluster. These issues made in-situ and close-to analytics and visualization solutions an important research topic. Analyzing and steering the simulation while it is running can reduce the resources (both computational and storage) used. In most cases the visualization and analytics applications need a small part of the data produced by the scientific application, but because the data layout is optimized to increase the performance of the simulation, finding and reading the required data is slow and may interfere with the data producer.

Another important trend in high performance computing is increased use of multi-physics simulations, where many computational models are bundled together to create more realistic simulation of the studied fields [45]. An example of such simulation is the Energy Exascale Earth System Model (E3SM) [1] that simulates Earth's climate by including models for simulations of the ocean, atmosphere, ice (land and sea), and land. As each of the models within the multi-physics simulation might use different meshing and partitioning techniques, the data formats used by each models are often incompatible. When data has to be shared at the boundaries of the models, it needs to be converted from one representation to another. Although there is increasing focus on sharing data in RAM, when possible, most of the models still use files stored in an "outside" storage system as a way to exchange data with the other models.

Both trends make it harder to define in advance what data format is the best – the one produced by the simulation, or one of the formats that are more suitable for the consumers of the data. The storage system, where the data ultimately reside has a unique opportunity to answer these questions. Unlike the relatively short-running producers and consumers of the data, it can collect statistics on the access patterns during whole lifetime of the datasets, and has opportunities to improve the performance by reorganizing the data or, if space allows, creating replicas of the data in multiple formats.

To achieve that, the storage system needs explicit information on the data semantics and structure. Making that available will allow both better sharing between loosely coupled applications, and flexibility for autonomous data reorganization in the storage system itself.

VPIC [13, 11, 12] is an example of an HPC application that saves its complex datasets on parallel storage. VPIC is a general purpose particle-in-cell simulation that models kinetic plasmas in one, two, or three dimensions. It allows simulations of plasmas with different kinds of particles, with specific masses and electric charges. It uses a second-order, leapfrog algorithm to update the particle positions and velocities in order to solve the relativistic kinetic equations in the plasma. VPIC partitions the space in rectilinear or curvilinear meshes. The cells are partitioned uniformly across the processing elements the simulation is running, and the simulation is run in parallel. After each step VPIC shares data (particles and electromagnetic fields) at the cells' boundaries. VPIC's input deck defines the size of the cells the space is partitioned into, the types and the initial conditions of the particles, as well as the subset of the data produced that should be saved for each simulated entity on storage.

In order to improve performance, VPIC employs N-to-N data storage pattern: each of the simulation ranks saves its data in a separate file. Additionally, the data for each type of particles, as well as for fields and hydrodynamics data is written in a separate file. Although writing to multiple files avoids some of the bottlenecks of parallel filesystems (writing to a single file from hundreds of thousands of nodes greatly affects the storage performance), there are two main issues with this approach. First, parallel filesystems don't perform well when there are hundreds of thousands, or even millions of files in one directory. The metadata storage server, which controls the directory structure, becomes a bottleneck at file creation, as well as when metadata (like file size, or modification time) is changed. Projects like GIGA+ [59] can be employed to improve that problem. Second, the data produced by the simulation is split into arbitrary number of files that reflect the particular configuration of the compute cluster. As the overall storage performance depends on both compute and storage systems configurations, and this approach may lead to inferior results. Also, tying the number of files to particular partitioning of the space makes it harder to restart the simulation later, if it is run on a different compute cluster.

In the examples on how VPIC can utilize the projects described here, we will focus on the storage of fields calculated by the simulation. Table 1.1 shows the data stored for each time step. For each rank, VPIC creates a file and stores the fields for each cell of the local grid. The array allocated in memory has two more elements in each direction in order to receive and use the ghost values calculated from the neighboring ranks.

1.3 Contributions of this Dissertation

This dissertation explores techniques for declarative dataset description that will allow the storage system and/or the applications to utilize automatic data

ex, ey, ez	electric fields in each dimension
div_e_err	div E error
cbx, cby, cbz	magnetic field in each direction
div_b_err	div B error
tcax, tcay, tcaz	TCA fields in each direction
rhob	bound charge density
jfx, jfy, jfz	free current in each direction
rhof	charge density
ematx, ematy, ematz	material at edge centers
fmatx, fmaty, fmatz	material at face centers
nmat, cmat	material at nodes and cell centers

Table 1.1: Important fields data stored by VPIC at each step

transformation and replication. Each of the methods discussed has different application interface, a set of data structures and transformations it allows, and types of scientific data representations that it supports. All methods together support all primitive types (different sized integers and IEEE 754 floating point numbers), composite data structures, as well as multidimensional arrays, multi-resolution multi-dimensional continuum, as well as unstructured grids. These types cover the overwhelming majority of data structures used in scientific applications.

The dissertation explores the extent to which declarative languages can describe scientific datasets and the transformations that can be generated for converting the data to different formats. It further outlines the limitations of using data description languages and proposes possible solutions on how to incorporate transformations where declarative approaches fail. It provides the ability to define application datasets as well as subsets and replicas of their data, as well as accessing the data using the POSIX I/O interface. It then extends the functionality by making it scale better in a HPC environment where data for a dataset is produced and stored in many locations. It extends the use of the language by using it to define the data format in the program's memory in addition to the format in the storage system. Further, the contributions include extensions to the DRepl language that allow support for multi-resolution datasets where the cells of a multi-dimensional continuum can have different sizes for different parts of the space. The runtime also provides a way to define, and execute conversions between the different resolutions.

1.3.1 DRepl: Static Datasets with POSIX I/O Interface

The first contribution is defining a declarative language for describing static datasets for scientific applications and a custom file system that allows the dataset to be accessed in different formats.

DRepl improves the performance of the visualization and analysis tools while keeping the amount of storage and reliability guarantees the same as when using other data replication mechanisms. DRepl is transparent to the applications and doesn't require any modifications of the application's code. It runs as a file server that provides different files for each layout of the data desired. The data layouts (views) can be stored in a file (replica) on the underlying file system (materialized), or can be virtual (non-materialized). If a view is materialized, reading from its file reads the data from the real file on the parallel file system. Reading from non-materialized view uses data from one of the materialized views and converts the data on the fly. When writing data to a view, DRepl updates all replicas. Depending on the concurrency model, the updates can be synchronous or asynchronous.

DRepl defines a language that is used to describe the dataset, views and replicas. The description is employed to generate transformation rules for conversion between the data layouts of each of the views. Given a DRepl description, DRepl creates a user-level file system that can be mounted on the compute nodes. Each of the views is represented as a separate file, while the replicas define how the data is written on the parallel file system. This approach separates the storage data layout from the one used by the applications to access the data. It allows support for legacy scientific applications, without any modifications to their code. As long as one of the views matches the legacy data layout, the application can continue to operate as before, even if the replicas that store the data on disk are in a more portable or convenient layouts, like HDF5 [34] or NetCDF[18].

DRepl allows flexibility that might be used in advanced storage architectures that implement burst-buffer [51] schemes. A burst-buffer is a type of hierarchical file system designed to reduce the number of checkpoints sent to the parallel file systems. Sitting between the compute node and the parallel file system, the burst-buffer is high-speed temporary storage for quick depositing of multiple checkpoints with only a fraction of them being moved to the parallel file system. Burst-buffer nodes can run the DRepl file server, providing transparent in-transit access to the data for the visualization and analytical applications in the appropriate format.

```
dataset {
  const nx = 100
  const ny = 100
  const nz = 100

  type MaterialId int16
  type Field struct {
    ex, ey, ez, div_e_err        float32
    cbx, cby, cbz, div_b_err     float32
    tcax, tcay, tcaz, rhob       float32
    jfx,  jfy,  jfz,  rhof       float32
    ematx, ematy, ematz, nmat    MaterialId
    fmatx, fmaty, fmatz, cmat    MaterialId
  }

  var fields[nx, ny, nz] Field
}
```

Figure 1.2: DRepl dataset description for the fields data.

The results of the experiments show improvements in the cumulative performance when data is stored in multiple replicas and the data is accessed from the replica that has its layout optimized for the particular access pattern.

The chief contribution of DRepl is to provide optimized access from various applications to the same dataset. DRepl decouples the data producers and consumers and the data layouts they use from the way the data is physically stored on the storage system.

When DRepl is used with VPIC, the developers will need to define the abstract dataset description that defines the data produced by the whole simulation. Figure 1.2 shows the part of the description related to the fields' data. The three-dimensional space is split into 10000 cells, 100 in each dimension.

When the simulation is run with N ranks, the user defines two views for each of them. Figure 1.3 shows how the views used by the M^{th} rank are defined. For simplicity, the example assumes that the space is partitioned only across the x axis. The user passes the values for M and N, the rest of the view constants are calculated based on the global constants. The `rM` view is used to read the data into the 3D array in memory. As the array in memory, it has 2 extra elements in each direction to allow for the ghost values. the `wM` view represents only the data that the rank produces and is used to write the data to disk.

The fields data can be stored on disk as an array of structs that contains all the data across all ranks and only for the electric field in two replicas (Figure 1.4).

Dreplfs is started before the simulation. It will create two files on disk – "vpic-fields" and "vpic-electric" for the whole simulation and store the replicas in them. The `default` keyword will instruct DRepl to use that replica as default for accessing data for all views that don't specify a particular replica. It will create $2N$ virtual files that can be used to access the data for each rank:

```
/
  r1
  w1
  r2
  w2
  ..
  rN
  wN
```

```
view "rM" {
  const rank = M
  const ranknum = N
  const lnx = (nx / N)
  const lny = ny
  const lnz = nz
  const ix = M * lnx
  const iy = 0
  const iz = 0

  var lfields[i:lnx+2, j:lny+2, k:lnz+2] =
        fields[i+ix*lnx, j+iy*lny, k+iz*lnz]
}

view "wM" {
  const rank = M
  const ranknum = N
  const lnx = (nx / N)
  const lny = ny
  const lnz = nz
  const ix = M * lnx
  const iy = 0
  const iz = 0

  var lfields[i:lnx, j:lny, k:lnz] =
        fields[i+1+ix*lnx, j+1+iy*lny, k+1+iz*lnz]
}
```

Figure 1.3: DRepl views description for rank M out of N. The first view is used to read the data (including the ghost values), while the second view is used to store the data produced by the rank.

```
replica "vpc-fields" default {
  var rfields = fields
}

replica "vpic-electric" {
  var efields { ex, ey, ez } = fields
}
```

Figure 1.4: DRepl replicas for all data and the electric field.

```
view "vpic-viz" "vpic-electric" {
  var efields { ex, ey, ez } = fields
}
```

Figure 1.5: DRepl view for visualization application that needs only the data for the electric field.

Each rank then uses the pair of files to read or write to the data in the format defined by the appropriate view.

If an visualization application needs only the electric force, it can define it's own view (Figure 1.5). DRepl will use the "vpic-electric" file to read the data for the view, improving the overall performance.

1.3.2 Static Datasets with Scalable Interface for HPC

Although DRepl decouples the storage data format from the format the applications use to access their data, it suffers from some shortcomings in a HPC environment. The fact that the data is accessed using POSIX I/O helps improving the performance of legacy applications, but it also makes the metadata of the dataset is unavailable to the developers. Additionally, the replicas are persisted to the parallel file system using POSIX I/O therefore missing the opportunity to explore the distributed nature of the parallel storage in a more direct way.

ASGARD explores advantages of extending the DRepl functionality by discarding the traditional POSIX I/O interface. Instead of operations for accessing contiguous subset of a file, ASGARD allows the application to access a subset (fragment) of the dataset as a whole, regardless of how the data for that subset is stored. For example, if a data file contains an array of structs, and the application needs to read only one field from each element of the array, it needs to either read a lot of data that it will discard, or issue thousand or millions of short reads. ASGARD allows the field selection to be passed to the storage system and executed on the I/O nodes where the data is stored. It extends DRepl by providing support for dynamic fragment definitions, and functionality for transforming and gathering data from multiple data fragments. While DRepl provides a POSIX API for accessing the data as files, ASGARD avoids the file interface and focuses on the data layout in application memory and how to transform it optimally to the layout on persistent storage.

ASGARD uses the DRepl language to define the dataset and the fragments. Once they are declared, ASGARD provides functionality to query how to convert the content of one fragment to another. ASGARD defines *transformation rules*, which describe the conversion between two fragments. The values required for fragment materialization might not be located in a single fragment, so ASGARD allows the user to get a list of the fragments required to produce a specific fragment.

Unlike DRepl, transformation rules in ASGARD define conversion only between two data layouts. This restriction greatly increases the opportunities for optimizations that can improve the transformation performance.

In order to decrease network usage, ASGARD splits the transformation rules into two parts. *Remote rules* are executed on the server where the source fragment is located and produce a compact representation of the data for the destination fragment. Once the compact representation is received, ASGARD applies *local rules* to copy the parts of the source fragment to the appropriate locations in the destination fragment. ASGARD, in a sense, provides an extension to the standard scatter/gather operations [6]. The transformation rules allow compact definition of complex patterns. The combination of gather (on the side where the data is stored) and scatter (on the side where data is used) provides a powerful tool for transforming data from the format in which it is produced to a format suited for consumption. The fact that the fragments are defined in advance allows an active storage subsystem to perform optimizations that might be too slow or costly if run at the time the operations are executed. For example, the storage subsystem can start transforming and prefetching the data for the fragments that will be read in the future.

In order to evaluate ASGARD's benefits, we modified Ceph's [72] RADOS object store system to support ASGARD's transformation rules for storing and retrieving data to objects. We also created a custom object class that allows developers to define an object that represents an ASGARD dataset and how it is partitioned into stripes on storage. Each of the stripes is defined as ASGARD's fragment. When a client needs to read or write a subset of the data, it can get a list of the stripe objects that contain the data as well as a pair of transformation rules that need to be applied to transform the requested subset to each of the stripes. The operations for each of the stripe objects are executed asynchronously

```
fragment {
  const nx = 100
  const ny = 100
  const nz = 100

  type MaterialId int16
  type Field struct {
    ex, ey, ez, div_e_err        float32
    cbx, cby, cbz, div_b_err     float32
    tcax, tcay, tcaz, rhob       float32
    jfx,  jfy,  jfz,  rhof       float32
    ematx, ematy, ematz, nmat    MaterialId
    fmatx, fmaty, fmatz, cmat    MaterialId
  }

  var fields[nx, ny, nz] Field
}
```

Figure 1.6: ASGARD dataset description for the fields data.

to increase the overall performance. Our experiments show that ASGARD can improve the I/O performance by a factor of seven over both collective and non-collective MPI I/O.

To work with ASGARD, VPIC needs to be more changes. First, the user needs to define the fragment `dsf` that defines the abstract dataset (Figure 1.6) and the stripe fragments it is partitioned in on disk (Figure 1.7).

The creation of the dataset needs to be done only once, usually by the first rank of the simulation:

```
  ds = asgard_dataset_create("fields", dsf, stripes, N);
```

Next, each rank has to define what is the format of the fields data in memory (i.e. when read), and what part of it is actually produced by the rank (i.e. what

```
view "stripeM" {
  const stripe = M
  const total = N
  const lnx = (nx / total)
  const lny = ny
  const lnz = nz
  const ix = stripe * lnx
  const iy = 0
  const iz = 0

  var lfields[i:lnx, j:lny, k:lnz] =
        fields[i+ix*lnx, j+iy*lny, k+iz*lnz]
}
```

Figure 1.7: ASGARD fragment definition for stripe M out of total N.

is stored). Figure 1.8 shows the read definition (`rf`) of the memory fragment and Figure 1.9 defines the write definition (`wf`) of the same fragment.

The memory region is then created as:

```
  mf = asgard_fragment_create(ds, rf, wf);
```

When data needs to be read and written in the buffer `fields`:

```
  asgard_fragment_read(mf, fields);

  asgard_fragment_write(mf, fields);
```

1.3.3 Dynamic and Multi-resolution Datasets

Many scientific codes use multiple resolutions, either as separate runs, or as parts of the same run. There are many reasons for that. Some simulations take too long to run, so the scientists run coarser-grained simulations first, to evaluate if certain phenomena exists and whether it is beneficial to run finer-grained,

```
fragment {
  const rank = M
  const ranknum = N
  const lnx = (nx / ranknum)
  const lny = ny
  const lnz = nz
  const ix = rank * lnx
  const iy = 0
  const iz = 0

  var lfields[i:lnx+2, j:lny+2, k:lnz+2] =
        fields[i+ix*lnx, j+iy*lny, k+iz*lnz]
}
```

Figure 1.8: Read fragment for rank M out of N.

```
fragment {
  const rank = M
  const ranknum = N
  const lnx = (nx / ranknum)
  const lny = ny
  const lnz = nz
  const ix = rank * lnx
  const iy = 0
  const iz = 0

  var lfields[i:lnx, j:lny, k:lnz] =
        fields[i+1+ix*lnx, j+1+iy*lny, k+1+iz*lnz]
}
```

Figure 1.9: Write fragment for rank M out of N.

but longer running computation. For other models, the error produced by the numerical algorithm depends on the data values, so during the execution, some areas of the simulated contiguous space have to be run at higher resolutions than others. The multi-resolution approach balances the use of available computational resources and the overall numerical error.

Keeping grids' resolution the same for the whole dataset, or for all time steps of it, is often unfeasible. Additionally, analytics and visualization applications usually need the data at different resolution than the one produced by the application. There are opportunities for performance improvements if the resolution conversions are executed close to where the data is, i.e. within the storage system. For example, there is no need to transfer the data in its standard resolution if the visualization software needs it at much coarser one.

This work investigates the changes required in DRepl and ASGARD in order to provide multi-resolution grids for abstract datasets. Extensions to the DRepl language allow definition of more abstract data structures that are closer to the concepts used in the scientific models. The application developers can define abstract datasets using abstract types like real numbers, and multidimensional continuous spaces. They can then express the concrete types used in their numerical codes as *concretization* of the abstract data types, providing details about the format of the real numbers (for example 64-bit IEEE 754) or the resolution of the cells of the multidimensional space.

The runtime or the transformation engine provide support for converting data

```
fragment ds {
  type MaterialId int
  type Field struct {
    ex, ey, ez, div_e_err        real@approx(proportional)
    cbx, cby, cbz, div_b_err     real@approx(proportional)
    tcax, tcay, tcaz, rhob       real@approx(proportional)
    jfx,  jfy,  jfz,  rhof       real@approx(proportional)
    ematx, ematy, ematz, nmat    MaterialId@approx(error)
    fmatx, fmaty, fmatz, cmat    MaterialId@approx(error)
  }

  var fields[,,] Field
}
```

Figure 1.10: Abstract dataset description for the fields data.

from one resolution to another, using hints provided by the user. Integrating the runtime into the application code (as a library) and the storage system, allows balanced execution model that performs the optimal conversions close to where the data resides, and thus improves the overall performance of the HPC system.

We describe the design and the current implementation of the parser and the transformation engine. This project is a work in progress and requires further effort in order to integrate with an existing storage system as well as evaluate its performance and overall applicability with porting an existing application.

Figure 1.10 show the definition of the abstract VPIC dataset. It defines the `Field` structure as holding real numbers without specifying the format and the size of the values. It also defines how each of the fields to be calculated when the data is converted between different resolutions. The description defines a 3D space that contains the field values.

Figure 1.11 defines a fragment of the dataset that accesses the data in a grid

```
fragment f1 : ds {
  const x0 = 0.0
  const x1 = 1.0
  const dx = 0.3
  const y0 = 0.0
  const y1 = 1.0
  const dy = 0.3
  const z0 = 0.0
  const z1 = 1.0
  const dz = 0.3

  var grid[x0:x1:dx, y0:y1:dx, z0:z1:dx] = fields
}
```

Figure 1.11: Concrete fragment description.

with cell size of 0.3 in each dimension.

1.3.4 Dynamic Datasets with Unstructured Grids

Unstructured grids are the most general case for partitioning multidimensional spaces. The disadvantages of the maintenance, overhead and complexity often overcome the advantages in flexibility of the unstructured grid topologies. Nevertheless there are some classes of numerical algorithms that benefit from the use unstructured grids.

We explored the effort necessary for adding support for unstructured grids into the storage system. The biggest required change is abandoning the declarative dataset language. Instead of compact and space bound definition of a language description, the metadata describing the grid topology requires much bigger metadata structure that depicts the relations between cells in the unstructured grid.

We provide initial design and a plan on how to further tackle the problem of storing and querying topological unstructured grids.

1.4 Dissertation Outline

The organization of the dissertation is as follows. Chapter 2 provides background and an overview of the related work in the area. Chapter 3 presents the work on static datasets and the custom POSIX file systems used to access the data. Chapter 4 presents the extensions of the static datasets to distributed storage systems and parallel scientific applications. Chapter 5 outlines the initial design and the future work for supporting abstract datasets, multi-resolution, and unstructured grids. Chapter 6 provides a conclusion with some additional directions for future work.

Chapter 2

Background

2.1 Simulations and Data Discretization

Major part of the High Performance Computing applications are numerical simulations of scientific models. Scientific theories employ mathematical models that use real numbers with infinite precision and continuous multidimensional spaces. Using computers to solve the equations that describe these models introduce many challenges that are studied by the vast field of numerical algorithm research.

The equations, and variables, used in scientific theories can describe discrete/monolithic objects (like particles), or can be properties of continuous spaces. The spaces can be multidimensional, with most models using three dimensions. Depending on the phenomena they describe and the coordinate systems used, the dimensions can be open (span from $-\infty$ to ∞), periodic (angles in the polar coordinate system), or have specific finite shapes (for example, simulation of a finite object like a piston).

In computer memory, the infinitely precise real numbers are usually represented as 32- or 64-bit floating point numbers [4]. The calculations with them are inherently imprecise and introduce rounding errors to the simulations. There are some efforts in the recent years to introduce new number formats and algorithms that keep track and asses these errors. For example IEEE 1788-2015 [5] defines a standard for interval arithmetic. Unum [36] and Posit [35] provide alternative real number formats that allow variable precision.

2.1.1 Space Tessellation

Numerical algorithms partition the continuous spaces into grids. Splitting the space into cells that cover it without gaps and overlapping is known as tessellation. The shape and the size of the grid's cells depends on the particular numerical methods used for solving the mathematical equations. In most cases, the simulation of the model can be performed in parallel for each cell, requiring only knowledge of the data from the neighboring cells. This property is the foundation of parallel computation in scientific applications.

There are two main types of grids [50]: structured and unstructured. If the cell's position, size, and shape is defined by a general rule applied for the whole space, the grid is considered structured. If the number of neighboring cells, the position, or the shape of the cells are not uniform, the grid is unstructured.

Structured Grids

Structured grids are applicable for uniform spaces with regular shapes. The simplest type of structured grid is when the space is partitioned evenly for each dimension, making rectangularly shaped cells. Other commonly used shapes are triangles and hexagons in 2 dimensions as well as various pyramids, hexagonal prisms, octahedrons, and dodecahedrons in 3 dimensions.

Structured grids can be represented in computer memory as N-dimensional matrices. More generally, a structured grid can be defined as transformation where space position $x = \Big(x_1, x_2, \ldots x + n\Big)$ where $x_i \in \mathbb{R}$ is transformed into a matrix index $a = \Big(a_1, a_2, \ldots a_n\Big)$ where $a_n \in \mathbb{N}$.

Unstructured Grids

For unstructured grids, the cells can have different shapes and sizes, and more importantly, have different number of neighboring cells. Although some definitions of unstructured grids allow overlapping or enclosing cells, they are rarely used in scientific models.

Unstructured grids are used for multi-resolution simulations where in order to contain the numerical errors, some parts of the space need to employ finer-grained partitioning than other parts. They can also be used for tessellating more complex shapes. For example, there is no polyhedral tessellation of a sphere that uses only one type of polyhedron, so ocean simulations use a mix of hexagon and pentagon prisms for representing the Earth [26].

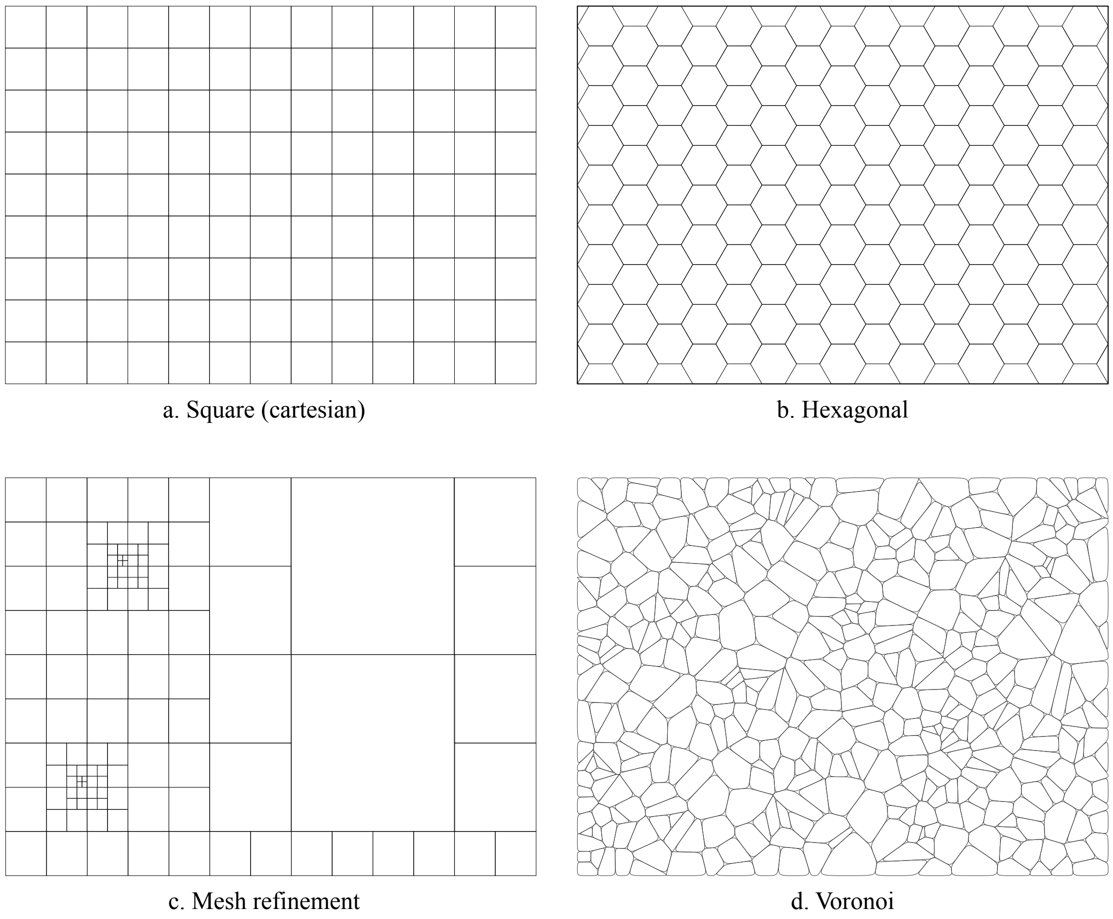


Figure 2.1: Examples of structured (a and b) and unstructured (c and d) grids.

In the general case, unstructured grids can be represented as graphs, with each cell being as a vertex, and the relation "neighbor" depicted as an edge. Although, in theory, the unstructured grids can produce very complicated graphs, in practice the physics theories, the numerical algorithms, and the methods used to generate the grids, lead to certain patterns that are commonly used. For example, the Adaptive Mesh Refining (AMR) algorithms generally have rules that restrict the refinement rate, so if the size of a cell is N, the sizes of its neighboring cells are either N, $N/2$, or $2N$, thus limiting the graph's rank. The fact that the unstructured grid tessellates a contiguous space enforces certain locality of the graph that describes it. For example, unstructured grids for 2 dimensional spaces are planar graphs.

An important subset of the unstructured grids are the block-structured grids. They consist of coarse unstructured grids, where each cell of that grid is further partitioned into a finer structured grid.

In computer memory unstructured grids are represented similarly to graphs. Usually there is a list that defines all cells. The cells' neighbors can be stored in that list for each cell, or there can be a separate list that define the cell's "neighbor" relation. Because the cells partition continuous space, often the partitioning is defined as a list of vertices: points in the space where the cells touch. In that case the unstructured grid is represented by a list of vertices, and each cell is defined as a subset of that list. Finding the neighboring cells for a given cell requires construction of the dual graph of the vertex graph.

Figure 2.1 shows examples of structured and unstructured grids.

2.2 Scientific Data Formats

Traditionally scientific datasets are stored in persistent memory in files using custom formats. In most cases the data layout is not stored together with the data. Instead, in the best case scenario, it is described in the application's documentation, and in the worst case, the interested parties have to uncover it by reading the source code. To make matters even worse, the performance degradation caused by writing to a single file from a distributed applications at scale usually force the developers to partition the data into multiple files.

There are few standard scientific format that are not application specific, and store the data description together with the data. The most popular of them is HDF5 [34]. HDF5 is a library that provides high-level API for storing complex datasets. The data can be organized in hierarchical groups, similar to directories on the file system. Datasets are named entities that contain the actual data values. They are similar to files in a POSIX file system. Unlike files, the datasets also include metadata that describes the data structure. Each dataset has a datatype. The datatypes can be atomic (i.e. 32-bit floating point number), compound (i.e. a list of named records, each with its own datatype), multi-dimensional arrays with constant size, or variable-length one-dimensional arrays. Each data object (group or a dataset) can have arbitrary number of attributes assigned to it. Attribute values belong to one of the defined datatypes. The generality of the HDF5 format leads to a very complex file format. The library practically implements a file system within a file, with inodes, list of free blocks, etc. The complexity makes it hard to achieve good performance for parallel applications, even though there are some solutions that try to alleviate that. The modular implementation of HDF5 makes it easier to incorporate other approaches to HPC storage, like DAOS [53]. The fact that HDF5 is a library that is linked to the application makes it harder to optimize the data layout once the application completes its execution.

Another popular scientific format is netCDF [62]. It allows storage of collection of multi-dimensional arrays of simple data types as well as attributes for them. There are parallel versions [48] of netCDF that improve the performance in HPC environment. Version 4 of the format allows use of the HDF5 format.

The Adaptive IO System (ADIOS) [54] provides a storage abstraction that describes the data produced by the application. The application dataset is defined in an XML file outside of the application source code. ADIOS can automatically generate code snippets for reading and writing the datasets in Fortran and C. In addition to the data description, the XML file provides information on the methods for accessing the data. In addition to the standard POSIX I/O, ADIOS provides support for MPI individual or collective I/O, as well as other methods. like routines that are optimized for specific storage systems. The datasets supported by ADIOS are somewhat limited and supports only simple data types – integers, floating point numbers, as well as multi-dimensional arrays of them. The arrays dimensions have to be variables that are also defined in the XML file and can be modified from the application code. ADIOS doesn't support arrays of compound types. For example, if an application like particle-in-cell simulation keeps multiple values for each particle (position, velocity, energy, etc.), with ADIOS the application developers need to create multiple arrays, one for each property of the particle. This approach works well with languages like Fortran, but is not well suited for the recommended techniques for modern programming languages like C++. ADIOS provides support for both structured and unstructured grids. There is also support for in-situ sharing of the data among multiple applications. Although some modification of the data format can be changed only within the XML file, the fact that the arrays' dimensions and offsets need to be provided both by the application and defined in the XML file, forces the developers to change both the source code and the XML file. Additional work with ADIOS [70] has improved performance by using space filling curves for multi-dimensional array element ordering in long term storage formats. Further developments of ADIOS provide the ability to replace the XML dataset definition with function calls in the application code.

2.3 Scientific Data in HPC Environment

High Performance Computing environments introduce many challenges to data storage. A typical cluster consists of hundreds to hundreds of thousands compute nodes that run the scientific code. They don't have local storage and use parallel file systems to retrieve and save the relevant data. Most scientific applications have a specific synchronous computational pattern where each processing element performs calculations for a certain period of time, then boundary data between the elements is exchanged, and optionally (at certain number of steps) the data is saved to persistent storage. Although this design greatly simplifies the numerical algorithms, it presents many challenges to the storage subsystem, as thousands of nodes try to use it at the same time.

The most popular parallel file system is NFS [56]. One of the oldest network file systems, it was designed to be shared by a relatively small number (up to hundreds) of Unix workstations, instead of compute nodes of a HPC cluster. There are some extensions to it, for example the Parallel NFS [29] that try to improve its performance in HPC environment. The popularity of NFS still makes it ubiquitous for sharing read-only portions of the file system, like standard Unix binaries, libraries and compilers.

There are many parallel file systems that are designed specifically for the HPC environments. PVFS [63], and later PVFS2 [68] is an open source project developed by NASA, Argonne National Laboratory and others. Panasas [73], Lustre [14], GPFS [64], and Ceph [72] are other examples of widely used parallel file systems.

Most of the modern parallel filesystems use a common architecture. They split the file operations into two main groups – data operations and metadata operations, and create two separate services that handle them. Each of the services can be deployed on multiple servers that handle parts of the I/O workload. Typical parallel storage uses tens or hundreds of data storage servers that operate close to independently. Due to the hierarchical nature of POSIX files, partitioning the metadata service is more complicated and requires more coordination. The clients (i.e. compute nodes) are aware of the distributed architecture of the file system and communicate directly to the data and metadata instances, decreasing the bottlenecks of the storage subsystem. There are still issues with scalability when operations are aimed at the same entity, if thousands of nodes access the same file, or try to create files in the same directory.

MPI [27] is a widely used communication library for parallel scientific applications. Since version 2 of the library, it includes MPI I/O – an API for accessing scientific data on storage. It uses MPI data types to define file's format. MPI I/O allows users to define memory and file data layouts, and uses collective I/O operations that improve the performance by coalescing the access across multiple MPI ranks before the requests are sent to the storage system. The collective I/O operations help with some non-contiguous local data access [21, 23, 24], that combined over all ranks ends up being contiguous. But in the general case the globally non-contiguous data access still doesn't scale well. Additionally, MPI I/O performance requires all ranks to progress synchronously, and that mode of operation is unlikely to be continued in the future HPC configurations at exascale, where the expected node failure rates will make it more likely that the codes will adopt more asynchronous techniques.

There is a long tradition of using synthetic files that are indirectly backed by data stored in different format. The original Unix used /dev and /proc [28] file systems to represent diverse type of devices and concepts as files. Plan 9 [61] extended the idea even further, representing all kernel resources as files, including the network and displays. Plan9 also introduced the concept of user-level file systems, allowing user applications to provide functionality by reading and writing to virtual files, instead of defining custom protocols. Services like DNS, as well as network protocols like TCP/IP were implemented as user-level file systems. The SFS semantic file system [30] was a layer on top of NFS that would create files based on user defined transducers. The transducers would allow retrieval of pieces of files, although this work was not focused on large application data. The ATTIC [17] system allowed transparent access to compressed files.

In HPC there are number of middleware libraries that provide file interface for accessing data that is in different formats. PLFS [10] is a transparent layer optimized for writing of parallel application checkpoints. It allows each process of a parallel application to believe it is writing to the a single file while the PLFS middle-ware separates these writes to disjoint files. Although the approach works well for writes, its read performance is often poor. Extensions to this work are focused on increasing read performance. GIGA+ [59] uses similar technique to improve the performance of the metadata operations. DeltaFS [76] allows the creation and the tracking of billions of small files that are backed by data residing in a database. Long distance visualization [7] uses multi-resolution data views to allow reasonable response times. In this scenario, when looking for an area of interest the resolution is sub-sampled to allow fast scanning, and when an area of interest is selected the high resolution data is then streamed in. None of the related work combines multiple semantic views with replicas to provide the configurability and resilience of our proposed solution.

There is not much work done in replicating the data content in different formats, or transforming the data transparently to the application. There is some work on using erasure codes [74] [60] for providing better availability of data in distributed environment.

The intrinsic performance bottlenecks of the POSIX file systems, and the rising popularity of non-traditional databases like Bigtable [19] is increasing the interest of research into how to use them for storing scientific data. SciQL [44] and ArrayQL [49] provide SQL-like languages for accessing scientific data. SciDB [15] can store very large arrays of scientific data. MDHIM [31] is a distributed key-value store designed for parallel applications. SciHadoop [16] is a plugin for Hadoop that allows storing, and optimized access to scientific data in the Hadoop data environment.

There is work done on creating adaptive layout strategies based on the data access patterns[67]. These techniques improve the I/O performance but don't help if the data needs to be read in-transit by visualization or analytical tools.

Although there has been a lot of work on taking some of the processing to the storage system [66, 20], recent work has been done on extending the programmability of the storage system. Malacology [65] exposes some of the standard internal interfaces available on most storage systems, so they can be used outside of the storage system. DeclStore [71] lays the groundwork for using a temporal logic language to declaratively specify distributed storage services and to enable automatic optimizations of their implementations.

Chapter 3

DRepl for Static Datasets

The goal of DRepl is to provide mechanisms for storing and transforming scientific datasets in semantically consistent way, so tools with different access patterns can use a custom replica that allows the fastest access. In order to do that, DRepl needs semantic knowledge of the data read or written by the applications.

Some of the existing libraries allow developers to provide partial, or full description of the scientific dataset. For example, developers using MPI-IO [69] can specify the MPI types of the data accessed. HDF5 goes even further, allowing description of the whole data model, where the data elements have names and can be organized in groups. The drawback of using these libraries is that they require significant changes in the code base. Furthermore, describing dataset with their API is often quite verbose and not easily readable. ADIOS uses an external XML file to define the data format.

DRepl tries a different approach. Instead of defining a single dataset format, its goal is to have the data layout decoupled from the data producer and/or consumers, providing more flexibility at the middleware and storage layers. DRepl allows separation of the abstract dataset, detaching its content from the data layout used in application's memory or the format used on persistent storage.

Instead of providing an API for defining the dataset (or adopting an existing one), DRepl uses a declarative language that allows definition of a dataset, views (subsets) of the dataset, as well as, replicas that store the dataset on persistent storage. Using specific language allows expressive and easy to comprehend definition of datasets. In order to ensure familiarity, we chose syntax similar to the syntax of type and data declarations shared by many of the popular programming languages like C, C++, Java, etc.

We distinguish three major entities related to data and the way it is used and stored. *Dataset* is an abstract definition that describes the data types and data objects that are of interest of any application, regardless of whether it is a producer or consumer of the data. *View* is a subset of the dataset that defines the parts of the data that are of interest of particular application, or set of applications. If the data is accessed as file(s) from the file system, the view may also define the order the data objects are laid out in the files. *Replica* is a subset of the dataset that is persistently stored on a storage system. A replica is *full* if it contains all the data from the dataset, or *partial* if it contains only a fragment of it. If stored on a file system, the replica also defines the file format of the data.

Using a separate language allows us to decouple the dataset definition from the way various tools and applications see and access the data. The dataset specifies the abstract data model which can span across data generated by multiple simulations and sources. Each application can have its own private view of the data, or subsets of the data. The actual format as stored on the file system can be decoupled from some, or all, applications. Legacy applications can still read and write data in the format they use even if the data is stored in HDF5 or other standard scientific formats.

The DRepl language allows definition of datasets, views and replicas. In order to allow legacy data access, DRepl can act as virtual file system that contains separate files for each of the defined views. The data from the dataset is backed to the specified replicas. The applications access the data using I/O operations on one of the synthetic view files on the virtual file system. The DRepl runtime translates these operations to I/O operations on the actual files that store the data replicas.

Figure 3.1 shows an example of a dataset produced by two simulations, Sim1 and Sim2 and accessed by two other applications, Viz1 and A1. The data is stored in three replicas, with the A, B, C, and D parts having two copies each. DRepl is managing the three replicas to provide the custom views the four applications have defined. As you can see DRepl has a lot of freedom in producing the views and can optimize for load balancing replicas, to prioritize the view of an application or to maximize data resilience, for example.

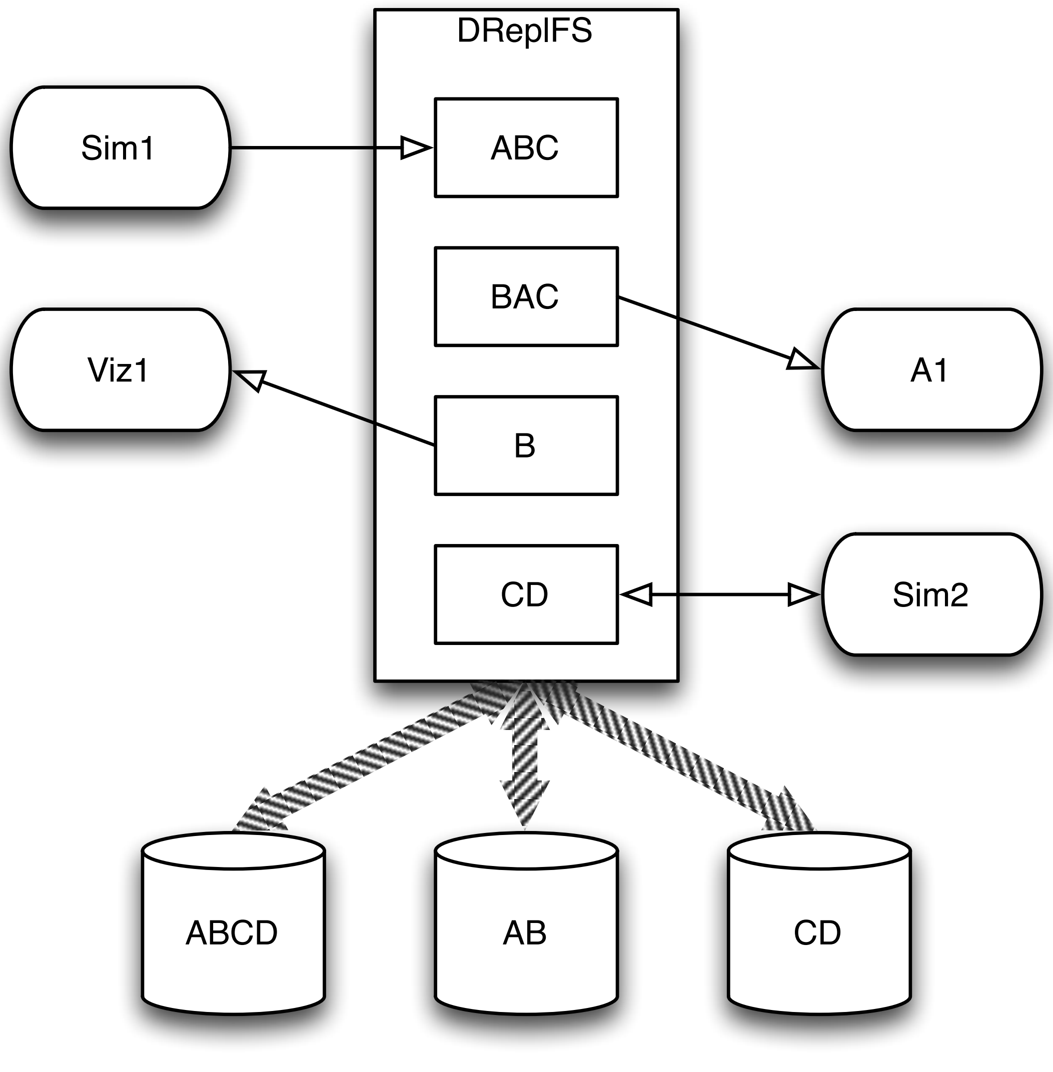


Figure 3.1: Example of DRepl being used by four applications and storing the data in three replicas

3.1 DRepl Language

The DRepl language includes definition of datasets, views and replicas. It allows declaration of custom data types based on a set of primitive types, as well as composite types like arrays and structs. Its syntax is loosely based on the type and variable definition in the Go [33] programming language, which is similar to the type and variable definitions in C, C++ and Java. The DRepl language is designed to allow representation of native application datasets with complex views in order to enable visualization and analytics optimized access to data of interest. NetCDF [48], HDF5 [34] and the data formats from local large-scale HPC applications were investigated to ensure good representation of real datasets.

The content of a DRepl definition file can be divided into three main sections: dataset definition, view definitions, and replica definitions.

3.1.1 Dataset Section

The dataset section defines the types that are used in the dataset as well as the variables that make up the dataset. The dataset section is an abstract description of the dataset. Although it defines the sizes of the data types, it doesn't define the endianness or way the elements in multidimensional arrays are laid out (row-major, row-minor, z-order, etc.). These specifics are defined when views are described based on the dataset.

int8	1-byte signed integer
int16	2-byte signed integer
int32	4-byte signed integer
int64	8-byte signed integer
float32	single-precision floating point number
float64	double-precision floating point number
string*[0-9]+*	variable size string of characters

Table 3.1: DRepl primitive types

Primitive Types

DRepl defines 7 primitive types (Table 3.1). The suffix of the string type defines the maximum size of the string that can be stored in variables of that type.

```
PrimitiveType = "int8" | "int16" |
   "int32" | "int64" | "float32" |
   "float64" | "string"
```

Structs

DRepl structs are similar to the struct type in Go and C. Multiple elements of different data types can be arranged in a struct. A name needs to be assigned to each of the elements.

```
StructType = "struct" "{" { FieldDecl ";" } "}"
FieldDecl = IdentifierList Type
IdentifierList = identifier { "," identifier }
```

The example below defines a struct with three fields: a, b, and c:

```
struct {
   a        float64
   b, c     float32
}
```

Arrays

An array is a numbered sequence of elements of the same type. Arrays can be single- or multi-dimensional, with fixed size in each dimension.

```
ArrayType = "[" ArrayLengths "]" Type
ArrayLengths = Expression { "," Expression }
```

The `Expression` in `ArrayLengths` can be an arithmetic expression containing integer constants (named or unnamed).

In the example below, `b` is a two-dimensional 5×5 array, and `c` is one-dimensional array of size $N + 1$ (where N is a constant defined in the dataset section).

```
 var b    [5,5]float64
 var c    [N+1]float32
```

3.1.2 Named Types

Similarly to Go (keyword `type`) and C (keyword `typedef`), the developers can assign names to the defined custom types. The names allow usage of "shortcuts" when a type is referenced often.

```
TypeDecl = "type" identifier Type
```

For example the struct from 3.1.1 can be assigned a unique named to be used later to define variables of that type:

```
type ABC struct {
   a        float64
   b, c     float32
}
```

3.1.3 Types

```
Type = identifier | CustomType
CustomType = ArrayType | StructType
```

Unlike most languages, DRepl allows types to be referenced before they are defined, so there is no need for forward declaration mechanisms.

3.1.4 Variables and Constants

A variable is a named instance of a type. The names of the variables need to be unique.

Constants are named values that can't change. They don't use storage space and are replaced with their value while the DRepl description is parsed.

```
VarDecl = "var" IdentifierList Type
IdentifierList = identifier { "," identifier }
```

```
ConstDecl = "const" identifier "=" Expression
```

Similarly to Go, DRepl constants don't have intrinsic type and are converted to the type required when they are being used.

3.1.5 Dataset

A dataset is the top-level construct in a dataset section. Dataset is a collection of types, variables and constants.

```
Dataset = "dataset" "{" { TypeDecl | VarDecl | ConstDecl } "}"
```

The example below defines a one-dimensional array whose elements have three 32-bit float values:

```
 dataset {
     const N = 1000000
     type Point struct {
         a, b, c    float32
     }
     var data [N]Point
 }
```

As mentioned earlier, DRepl doesn't require types or constants to be defined before they are used. The example below is a valid dataset definition and is identical to the previous example:

```
dataset {
    var data [N]Point
    const N = 1000000
    type Point struct {
        a, b, c    float32
    }
}
```

3.1.6 View Section

Views define subsets of the dataset. Data in a view can be accessed by read and write operations on the virtual file provided by the DRepl file server. If a view is stored as part of a replica, the view is *materialized*. Reading data from materialized views is fast, but there is performance penalty when the data is modified due to the need to update replicas that are of different format.

No new types can be defined in the view section, unless they are sub-types of the types defined in the dataset section. The user can define *substructs*, i.e. structs that contain only some of the fields of a dataset struct, or *slices* – parts of an array type defined in the dataset section.

The variables defined in the view are also based on dataset variables, providing full, or partial content of the dataset variable. Each view variable is of the same type as the variable it is based on, or a compatible sub-type.

3.1.7 View Substruct

```
ViewSubstruct = "{" { ViewFieldDecl ";" } "}"

ViewFieldDecl = IdentifierList ViewType
```

Example of usage of a view substruct:

```
var b [] {

    b

} = data
```

The view variable `b` defines an array with the same size as the dataset array `data` (from the previous section), but each element of the array contains only the field `b` from the original `data` elements.

3.1.8 View Slice

The view slice contains only subset of the elements of an original dataset array.

```
ViewSlice = "[" SliceLengths "]"

SliceLengths = Expression { "," Expression }
```

The `Expression` in the slice length definition can be an arithmetic expression containing temporary variable names that are used to express which elements from the array are included in the slice. For example, the snippet below defines a view variable `d` that contains every 5^{th} element of the `data` array.

```
var d[i] = data[i*5]
```

The temporary variables can't be used outside of the slice definition, and their names can be reused. For example, this is valid DRepl definition:

```
var d[i] = data[i*5]

var a[i] = data[i]
```

3.1.9 Named View Types

As in the dataset section, the user can assign names to any defined view types, in order to simplify the view definition. The example below shows the definition of type `Subpoint` that is based on dataset type `Point` but contains only fields `a` and `c`.

```
type Subpoint {

    a, c

} Point
```

3.1.10 More Complex Examples

A view variable `d` that contains only the `b` fields of every 3^{rd} element of `data`:

```
 var d [i] {

     b

} = data[i*3]
```

The same result with defining view type:

```
type PointB { b } Point

var d [i]PointB = data[i*3]
```

The temporary variables used for defining slices don't have to be used for the same dimension across the slice definition. For example, in order to reverse the column and row order in a two dimensional array, one can use:

```
var d [i,j] = a[j,i]
```

3.1.11 View Declaration

The view declaration defines a view. Each view has a name that corresponds to the name of the virtual file that allows access to the data in the layout defined in the view definition body. Additionally, a number of flags can be specified that control the element order in arrays as well as if the view can be used to update the dataset. The data in the dataset can't be updated via read-only views.

```
ViewDecl = "view" identifier ReadOnlyFlag
         ElementOrderFlag "{" { ViewTypeDecl | VarTypeDecl } "}"
ViewDecl = "view" FileName
ReadOnlyFlag = "read-only" | <nothing>
ElementOrderFlag = "rowmajor" | "rowminor" | "default"
```

Views can be defined in the same file as the dataset, or they can be defined in separate files and the file name can be specified to include to content of the file while parsing the dataset definition.

In future, the view declaration will also allow flags that allow control over the endianness of the data in the view.

3.1.12 Replica Section

Replica sections define how the data from the views is stored on the underlying storage system. Replica is a sequence of one or more views. Each replica has a name that corresponds to a filename on the file system. The replicas are regular files that are defined on the file system name space level, and can be accessed and manipulated by all tools that work on regular files. This design choice greatly simplifies the implementation without impacting the generality of the supported datasets.

Each materialized view has to belong to at least one replica. The views that are not part of any replica are *n*on-materialized. When data from non-materialized view is accessed, it is transformed on-the-fly from the default view. On writes, all replicas are updated irregardless if the view is materialized or non-materialized.

```
ReplicaDecl = "replica" identifier
              "{" {ReplicaView} "}"
ReplicaDecl = "replica" FileName
ReplicaView = "view" identifier
```

3.2 Implementation

The DRepl implementation consists of three main modules: DRepl language parser, replication engine and a file server (dreplfs).

3.2.1 Language Parser

The DRepl language parser receives a description of the dataset, views and the replicas, and produces internal representation of the types and variables defined in each, as well as the relations between them.

The parser includes rudimentary preprocessor that allows the DRepl definition to be split into multiple files for easier maintenance. For example the main file of the definition can contain only includes for each view, replica, or the dataset itself:

```
include "dataset.drepl"


include "view1.drepl"

include "view2.drepl"


include "replica1.drepl"

include "replica2.drepl''
```

In addition to receiving all elements at the same time, the parser allows addition and removal of view and replica definitions at later times. That allows changing the replicas and views as needed.

The parser's internal representation includes information required for parsing, but unnecessary for the actual data access. The parser can produce *transformation rules* that are used by the replication engine to provide DRepl's core functionality. They are described in detail in section 3.3.

While processing the dataset description, for each variable in the dataset, the DRepl parser locates all related variables in the view. For each pair of related variables it produces a map describing how to transform data from one variable to the other. If a view is marked as read-only, the maps describing how to transform its variables to others are not created.

3.3 Transformation Rules

The transformation rules are compact representation of the views and the replicas, and what conversions are required to transform the data from one view/replica to another.

3.3.1 Blocks

The internal representation of a view consists of a list of contiguous byte regions called *Blocks*. A list of top-level Blocks defines the overall layout of the view. There is one top-level Block for each variable defined in the view. The rest of the Blocks describe the structure of the top-level blocks in finer detail. For example, a Block that describes an array slice references a block that defines the element of the array.

Each block has defined an offset from the beginning of the view, size, source Block (if part of an unmaterialized view) and a list of destination Blocks that describe regions in other replicas that need to be updated upon the Block's update.

```
type Block struct {
    offset int64
    size   int64
    source Block
    dests  list of Block
}
```

DRepl currently defines three distinct types of Blocks.

The simplest block, *SBlock*, defines region that is always read or replicated as a whole entity. All primitive types are described as SBlocks, but in some cases an optimization code can coalesce multiple adjacent SBlocks into a single, bigger SBlock.

A *TBlock* is a collection of other blocks. It corresponds to a `struct` composite type in the DRepl language. In addition to the default Block properties, it contains a list of Blocks that define each of the fields it contains. The offsets of these Blocks are relative to the beginning of the TBlock. The size of the TBlock can't be smaller than the maximum of the $offset + size$ of any of the Blocks it includes.

An *ABlock* defines a multidimensional array of identical blocks. In addition to the default Block properties, it contains the dimensions of the array, and a reference to a Block that describes the array's element. The offset of the element Block is always zero.

The ABlock also has a field defining the element order of the array, with currently supported orders of row-major or column-major. The element order defines how to calculate the offset of an array element. For example, in row-major of a $n+1$ dimensional array, the offset D of an element with index $x = \left(x_0, x_1, \ldots x_n\right)$ in an array with dimensions $d = \left(d_0, d_1, \ldots d_n\right)$ is:

$$D(x) = \sum_{i=0}^{n} \Big(\prod_{j=0}^{i-1} d_j \Big) x_i \tag{3.1}$$

Calculating the transformation of ABlocks also requires some additional information about the source and dests blocks in other views. The flexibility of DRepl slice definition allows both changing the order of the elements within a dimension as well as across dimensions. For each dimension i of an ABlock, the source and destination descriptions contain five integer values $(a_i, b_i, c_i, d_i, idx_i)$. The element with index $(x_0, x_1, \ldots, x_n)$ in the original ABlock corresponds to an element $(y_0, y_1, ...y_n)$ in the source/destination ABlock, where

$$y_{idx_i} = \frac{a_i x_i + b_i}{c_i x_i + d_i} \tag{3.2}$$

If the value of the calculation is not a whole number, the element doesn't have a corresponding element in the source/destination block. Once the y_j values are calculated, the offset of the element is calculated using the element order for the source/destination ABlock.

3.3.2 Example

Figure 3.2 shows an example the conversion map for the dataset description below:

```
dataset {
   var p struct {
      a, b, c float32
   }
}


view default {
   var p = p
}


view viz {
   var pa { a } = p
   var pba { b, a } = p
}
```

In this figure you can see that the *default* view is in the lower box, with a *TBlock* that holds the variables a,b,c. The top box is the *viz* view. Variable *a* in

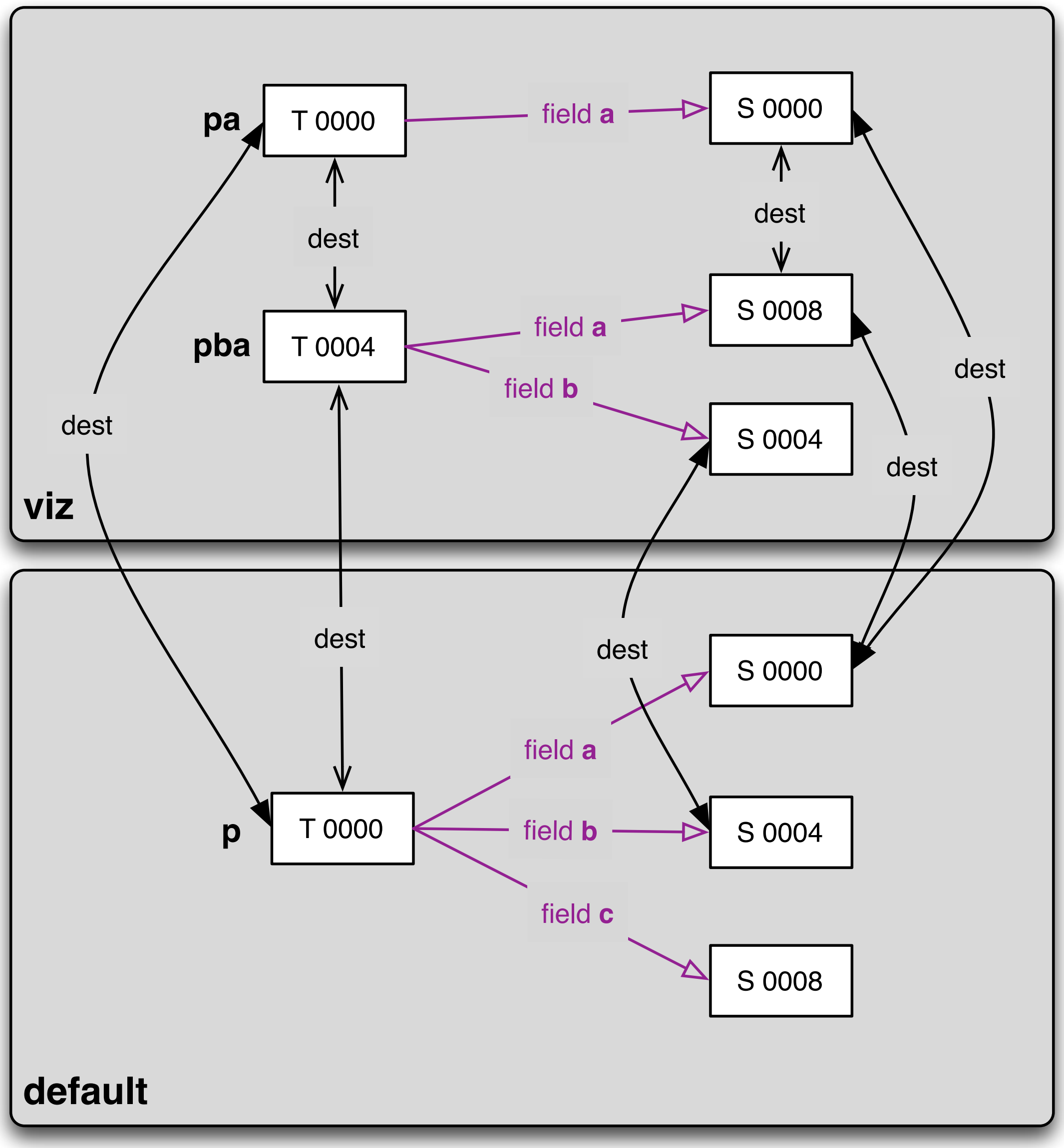


Figure 3.2: Example of a conversion table of two views

the *default* view gets two references from the viz view, while element b gets one reference and element c gets no references from the *viz* view.

The conversion map for a replica is constructed by concatenating the conversion maps for all the views that belong to it.

3.4 Replication Engine

The replication engine uses the transformation rules to implement the core of DRepl's functionality. Internally the replication engine keeps similar information for views and replicas.

3.4.1 Updating Data

The dataset can be updated via a `write` operation to one of the views. The `write` operations receive as arguments the file offset, the number of bytes written and the data. The replication engine finds all Blocks that belong to the specified range and applies the conversion rules specified in the blocks. These rules cause writes to each of the destination Blocks from the Block's description. If the writes belong to a replica, instead of a view, they are executed as writes to the underlying file that contains the replica.

The replication engine ensures that the write operations don't modify only parts of a Block (except for ABlock). Operations that break this rule are not executed and an error is returned to the user. The replication engine doesn't guarantee atomicity of the updates. If two applications are modifying the same data simultaneously, the content of the replicas can be a mix of the two updates.

The replication engine also allows transformation of whole view to another view.

3.4.2 Reading Data

Read operations to materialized views are passed through to reads from the underlying file that contains the replica's data.

Non-materialized, read-write views use the transformation rules created by the parser. Using the (offset, count) pair, the appropriate blocks are found from the conversion map and the data is copied from one or more of the materialized views.

3.5 File Servers

There are two implementations of the DRepl file server – a user-space and a user/kernel space combined solution. The first implementation (Fig. 3.3) is written in Go language and uses the 9P file protocol to create a file server in user-space. It combines the parser, the replication engine, and the file server code in a single binary.

The second implementation (Fig. 3.4) runs the parser in user-space, but the replication engine and the file system are written as a Linux kernel file system and run in kernel mode. The user-space implementation can run on more operating systems, but has some performance disadvantages.

For the kernel implementation the conversion maps are serialized in a compact format to a region of memory that is passed to the kernel file system. That format is used as a basis for the local and remote transformation rules described in Chapter 4.

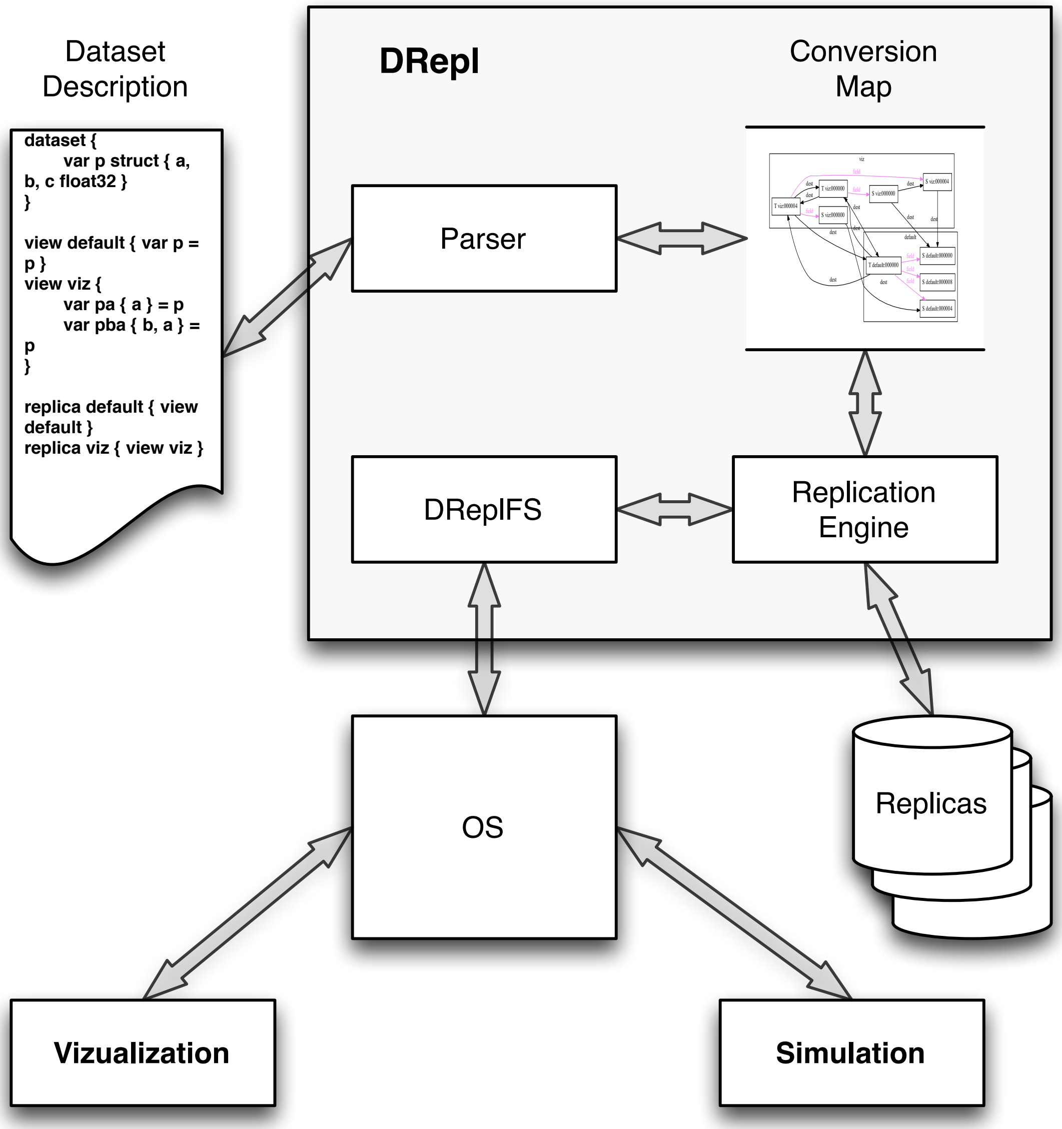
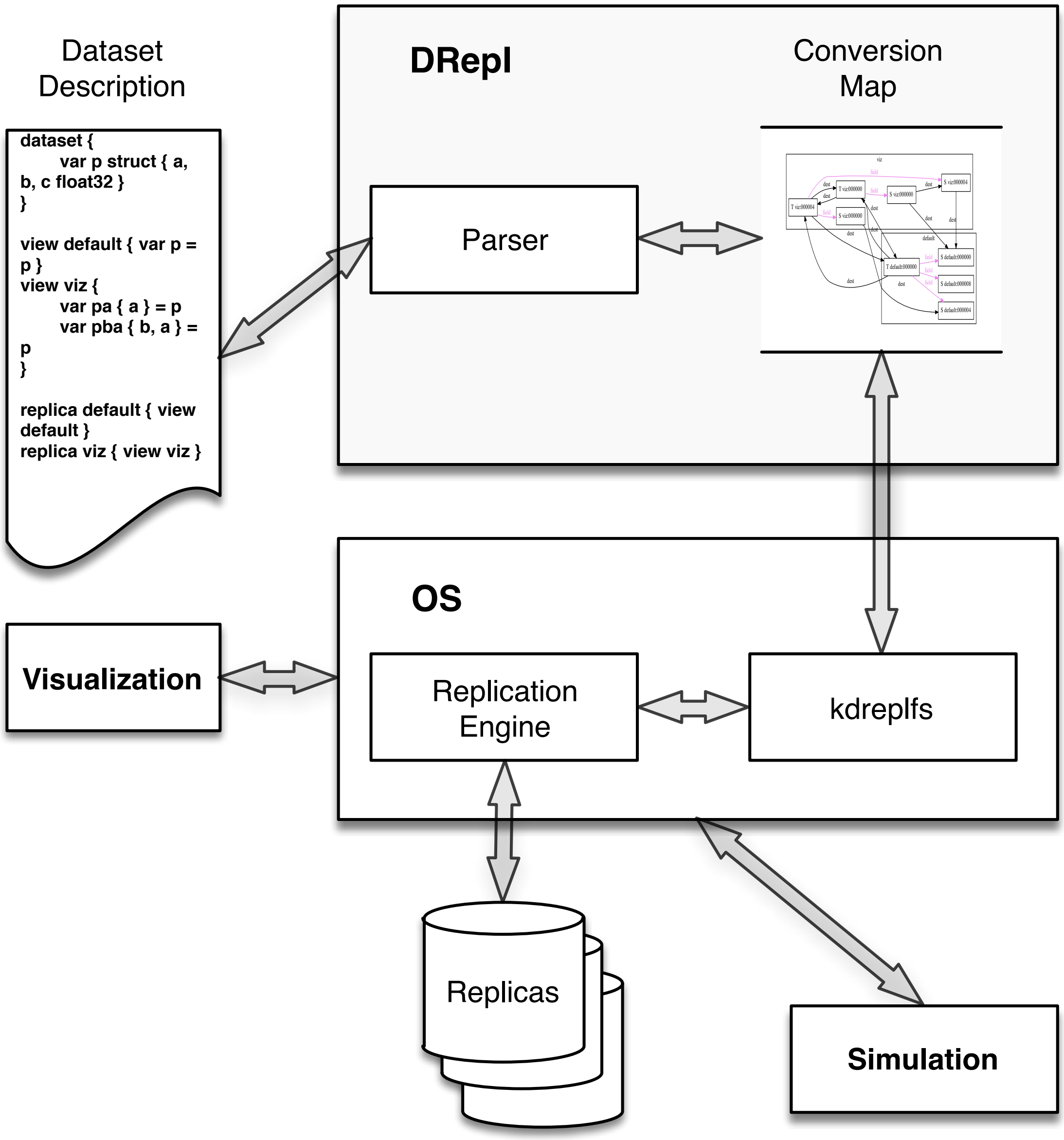


Figure 3.3: DRepl user-space implementation

3.6 Replica Layout

Each replica is a separate file, or directory, with name specified when the replica is declared. If any of the views of the replica contains variable-sized data, the replica is a directory with multiple files in it. Otherwise, data for all views is



```
dataset {
        var p struct { a,
b, c float32 }
}

view default { var p =
p }
view viz {
        var pa { a } = p
        var pba { b, a } =
p
}

replica default { view
default }
replica viz { view viz }
```

Figure 3.4: DRepl hybrid implementation

stored in the same file.

The data from the views are laid out in the order they are declared in the replica definition. Each view starts at offset divisible by 8, and padding is added between the views if the previous view's size is not divisible by 8.

type	alignment	size
int8	1	1
int16	2	2
int32	4	4
int64	8	8
float32	4	4
float64	8	8
string$N$	2	$2 + N$

Table 3.2: Alignment and size of primitive types

3.6.1 Primitive Types

Each of the primitive types starts at offset that naturally aligns to its type. Table 3.2 shows their alignment and size. Strings can contain up to N bytes. Their content is prefixed with a 16-bit value of their actual size.

3.6.2 Structs

The fields in a struct are laid out sequentially without any explicit padding between them, or at the end of the struct. The alignment rules for the type of the first field define the alignment requirements for the struct. The rest of the fields are placed based on their alignment requirements.

3.6.3 Arrays

Elements of an array are laid out sequentially in the order defined by the element order value. There is no padding between them or at the end of the array. The alignment rules of the element type define the alignment requirements for the array type.

3.7 Experimental Study

While evaluating the user-space prototype, we wanted to make sure we isolate implementation choices (like programming language, file protocol, etc.) from the performance inherent to our design choices.

We created a simple dataset and tested the bandwidth when reading and writing from it. We wrote a description of the dataset and three views in the DRepl language and measured the performance using DReplfs to access the data. Additionally, we created a hand-optimized file system written in the same programming language and file protocol that implements the same dataset. Comparing DReplfs with the optimized implementation allows the evaluation of the performance penalty due to the complexity of the DRepl language and any inefficiencies in the replication engine implementation. Lastly, we compared the performance to using a proxy file system, implemented again using the same programming language and file protocol, but directly translating the access to the files it serves to the underlying file system. This comparison allows us to isolate the performance of the chosen programming language and file protocol from the overhead added from maintaining views and replicas in DRepl. The results demonstrate that even though the penalty for maintaining multiple replicas with divergent data formats is considerable, the combined performance of an application that stores its data in one replica of a certain format, and a consumer that reads the data from a replica with a different format have overall better performance than if the consumer uses the original format.

3.7.1 Dataset

We used a simple dataset in which the scientific application stores three values for each "point" of the simulation. The number of "points" is configurable. We ran our experiments with 168 million points.

```
struct Point {
    a    float
    b    float
    c    float
}


Point    data[N]
```

3.7.2 Views

We define three views (data layouts) of the dataset:

Array of Structures (aos)

```
    Point    data[N]
```

Structure of Arrays (SOA)

Most of the legacy applications, especially the ones written in FORTRAN, store separate arrays for each value.

```
    float    a[N]
    float    b[N]

    float    c[N]
```

Visualization (Partial)

In most cases the visualization requires only some of the values. We chose a visualization view that contains only an array of the b values.

```
    float    b[N]
```

3.7.3 File servers that provide multiple data views

When the file servers are mounted, they provide access to three views of the data:

SOA Data in legacy format (structure of arrays). First $4 * N$ bytes contain the data for the array a, followed by $4 * N$ bytes for array b, and then $4 * N$ bytes for array c.

aos Data in natural format (array of structures). Contains N elements, each 12 bytes long with the values of a, b and c for that element.

b Data in the visualization (partial) format. Contains N elements, each 4 bytes long with values only of b.

We ran sets of experiments varying the following parameters:

Replica number

We tried three different combinations of replicas: one replica for each view, two replicas containing SOA and b views (aos view is unmaterialized), and one replica containing the SOA view (aos and b views unmaterialized).

Synchrony of updates

We tried synchronous vs. asynchronous updates. At least one of the replicas is updated synchronously before the write operation completes. Reading performance is not affected by the synchrony parameter.

Read and write operations access all data in the view sequentially.

The experiments were performed on 4 socket, 4 core servers (total 16 cores) with 32GB of RAM. We used separate SSD disks for each of the replica files. The OS buffer cache was cleaned between every experiments. We also performed the tests on rotational disks and the results were very similar.

3.7.4 User-space file servers

We compared performance of three user-space file servers that provide access to the data in the three different views:

dreplfs

Our prototype that receives a dataset, view and replica definitions in the DRepl language, converts it to the internally used conversion tables, provides access to the views as virtual files, storing the data in the specified replica files. DReplfs is implemented in Go and uses the 9P2000 file protocol (go9p [39] library).

dsfs

> A hand-optimized file server that provides access to the three views mentioned as virtual files and allows storage of the data to one, two or three replicas, each containing one of the views. Dsfs is implemented in Go and uses the 9P2000 file protocol (go9p library).

ufs

> A "proxy" file server that provides access to the files on the file system over the 9P2000 protocol. Ufs is implemented in go and uses the 9P2000 file protocol (go9p library).

DReplfs and dsfs have an option that allows the transformations to the replicas to be performed asynchronously. Ufs doesn't allow replication.

Converting data from one view to another can be very inefficient. For example, if a program writes to the first 400 bytes of file `SOA`, updating the first 100 values of a, the operation needs to be converted to 100 writes to file `aos`, each writing 4 bytes with stride 12 bytes. Even using functions like `writev` can be slow, because of the time and resources required to prepare the data for the call. Using processors close to the data can improve the performance somewhat. In order to improve it even further, we use `mmap` to map the content of the materialized views to memory. The conversion between different views then is equivalent to conversion of data in memory. All three implementations use `mmap` to read and write to the files, instead of the standard POSIX I/O calls.

Raw Performance

Figure 3.5 shows reading performance. The read performance when reading from materialized replica is roughly equal for all tested file servers. Reading from unmaterialized replica is about 15 times slower for dreplfs and three times slower for dsfs. The conversion from materialized to unmaterialized view needs to be done before the read operation can return, so the asynchronous mode doesn't provide any performance increases.

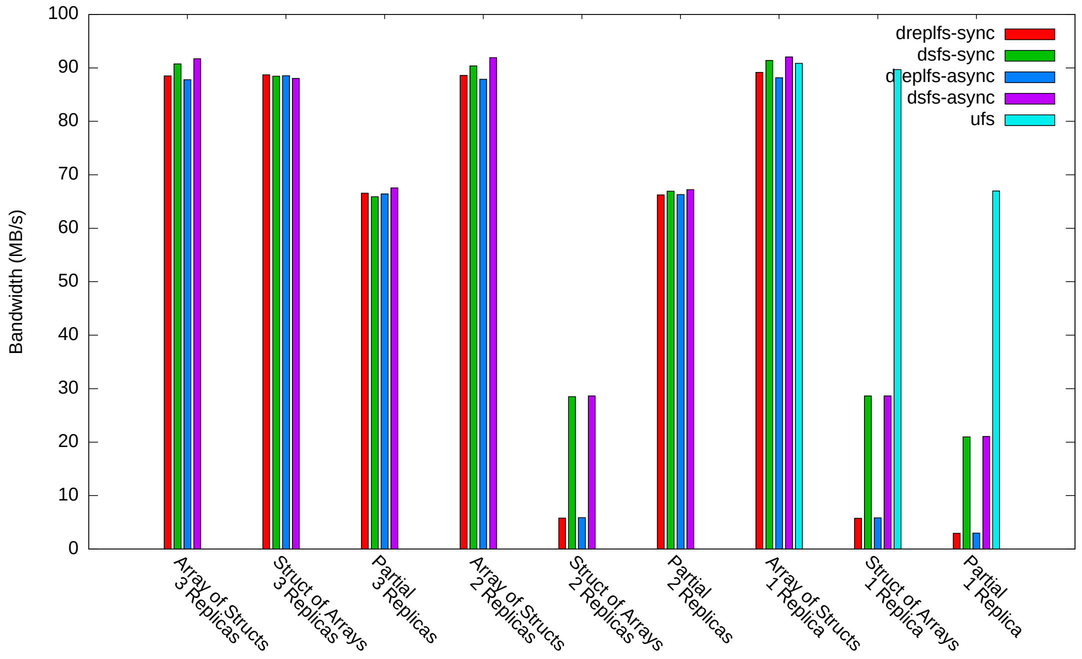


Figure 3.5: Read performance for each of the file servers for various access patterns.

Figure 3.6 shows writing performance. The hand-optimized file server is about three times faster than dreplfs. There is big advantage in running the replication asynchronously. The penalty of maintaining more than one replicas is pronounced more in dreplfs than the hand-optimized file server.

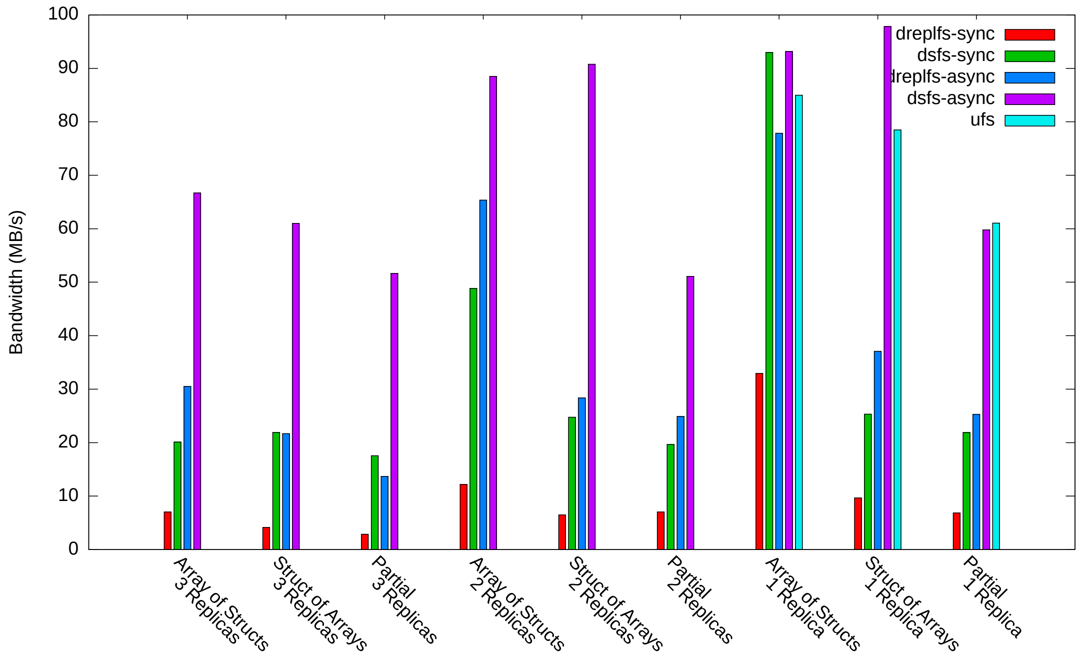


Figure 3.6: Write performance for each of the file servers, for various access patterns.

Combined Performance

In order to evaluate the performance advantage of creating multiple replicas and reading from the most optimal, we emulate two separate cases:

single replica

The simulation application writing the data to a single replica (containing the AOS view) and the visualization application reading the partial view from it;

two replicas

Replicas containing the AOS and the b views are created, the simulation application reads the data from the AOS replica, the visualization application is reading from the partial replica.

Figure 3.7 shows cumulative bandwidth for accessing the dataset from one or two replicas. Even though the write performance is decreased when maintaining two replicas, the cumulative performance is improved by the fact that the read operations use the optimized replica.

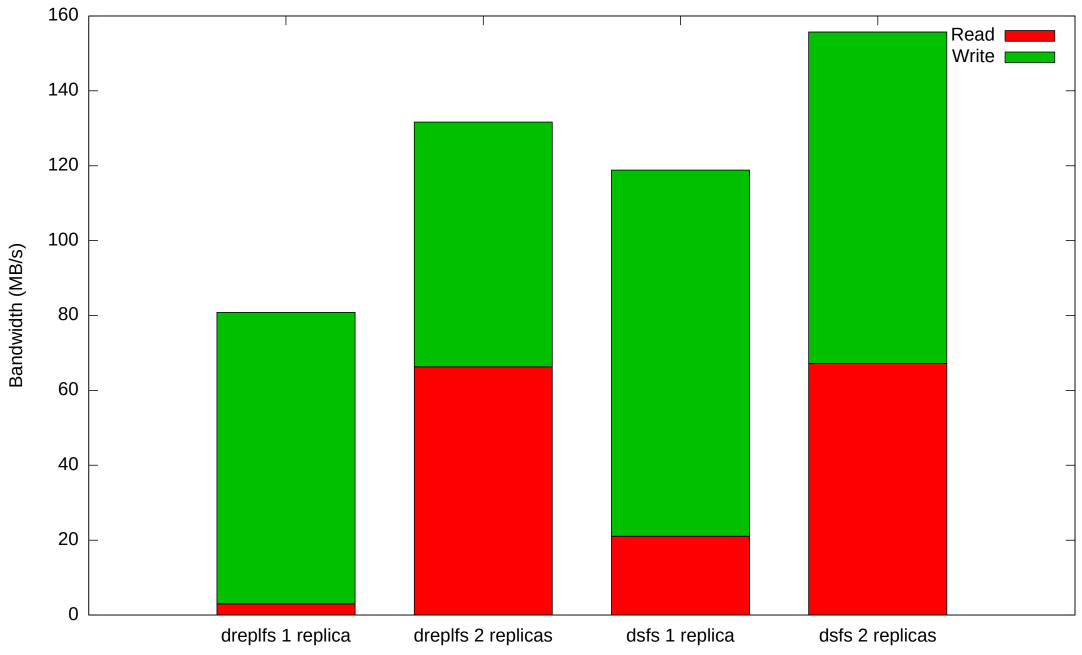


Figure 3.7: Cumulative bandwidth for accessing one and two replicas

3.7.5 Kernel-space file server

We compared performance of the our kernel implementation of the DRepl file system (kdreplfs) with conventional POSIX file system.

Raw Performance

Figure 3.8 shows reading performance. Our tests have shown that there is no difference between asynchronous and synchronous mode (in both cases the conversion needs to finish before we return the data), so the figure doesn't have the asynchronous results. Reading from materialized view is about nine times faster than reading from unmaterialized view. Further optimizations of the replication engine should might decrease the difference between them.

The POSIX read performance is shown only in comparison to the 3 replicas configurations, because in these configurations and the POSIX case the data is read directly from the filesystem and no unmaterialized view processing is required.

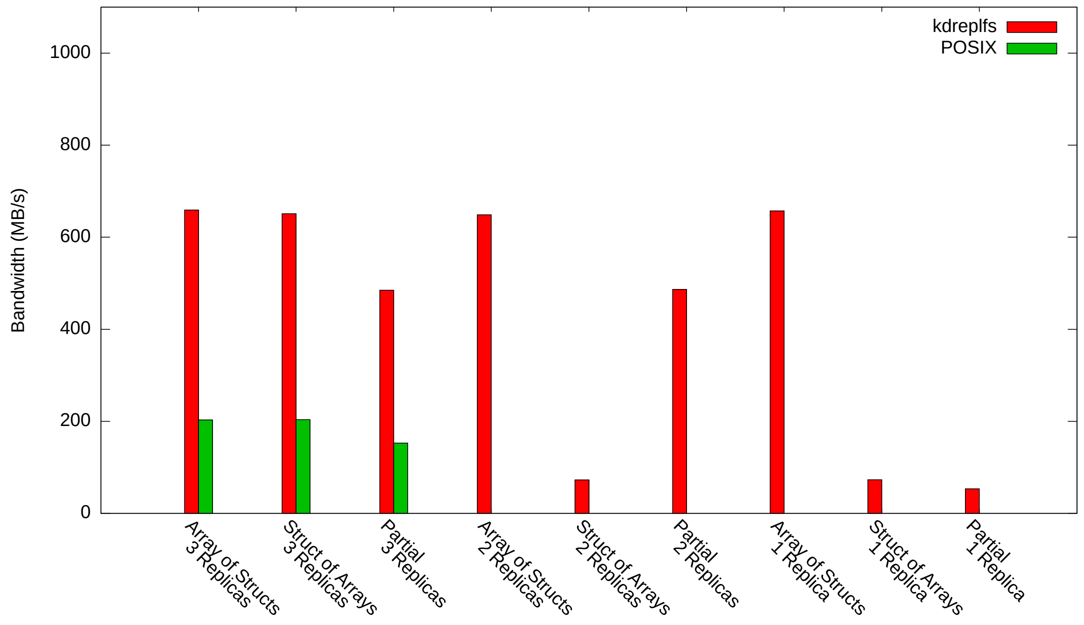


Figure 3.8: Kdreplfs read performance

Figure 3.9 shows writing performance. The asynchronous mode is usually faster than the synchronous one, although there are cases when that's not the case. In asynchronous mode, the kdreplfs needs to allocate additional memory in the kernel and copy the data buffer so the replication engine can continue to work even after the write call returns to the user process. The memory allocation adds some overhead. Our implementation makes sure that at least one replica has the data committed before the execution is returned to the user space. That may slow down the asynchronous mode even further.

The POSIX write performance is shown only in comparison to the one replica configuration, because in these configurations and the POSIX case the data is written directly to the file system and no replication of the data is required.

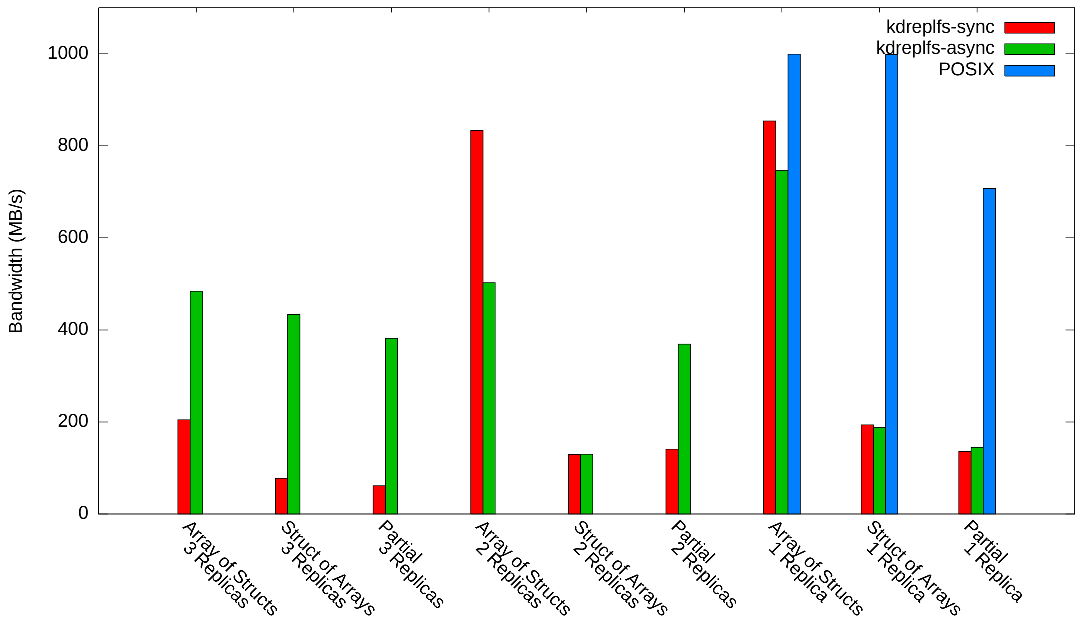


Figure 3.9: Kdreplfs write performance

Combined Performance

Figure 3.10 shows cumulative bandwidth for accessing the dataset from one or two replicas. Even though the write performance is decreased when maintaining two replicas, the cumulative performance is improved by the fact that the read operations use the optimized replica. The graph also shows the POSIX performance when the data is written in the SOA format and then read in chunks and converted to the AOS format in memory (the chunk size is 524288 points).

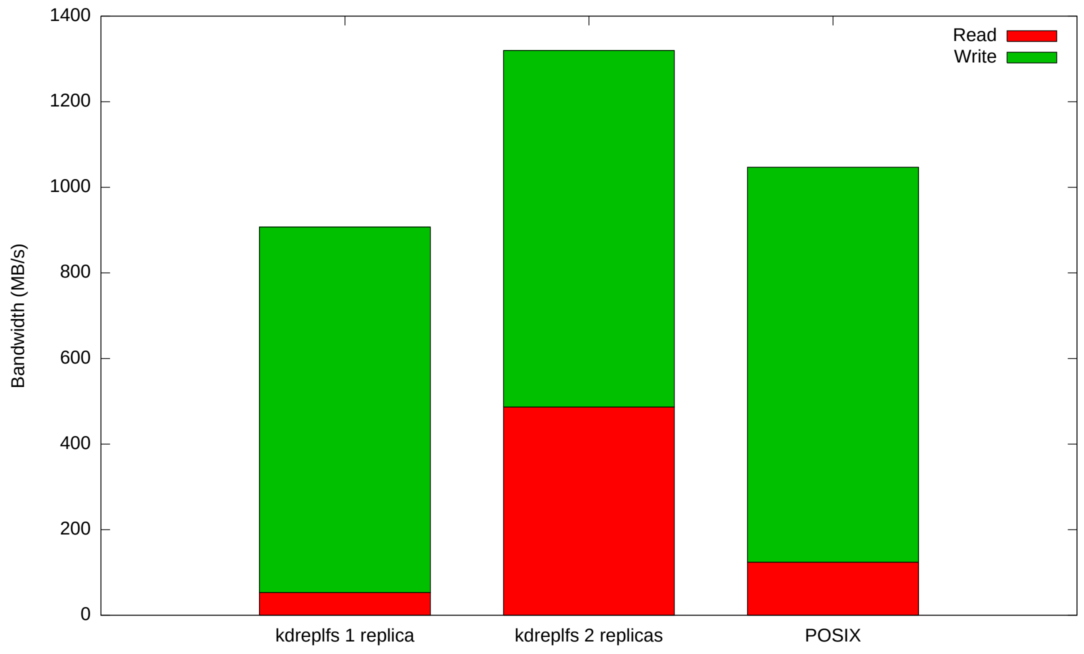


Figure 3.10: Cumulative bandwidth for accessing one and two replicas

3.8 Discussion

DRepl provides a novel method for optimized access to application datasets that are read and written with multiple contrasting patterns. DRepl decouples the data producers from the data consumers and additionally from the physical data layouts used on the storage system.

DRepl provides a language for describing binary data saved in files. Combined with custom file system, this approach allows legacy applications to continue working unmodified, while the data is being transformed to different formats in persistent storage. It also allows the data to be accessed as if written in other non-native formats. The data can be replicated on-the-fly.

The experimental studies have shown increased performance in both read and writes on various physical media. A prototype file system was implemented in both user and kernel-space and the language was designed and specified to allow construction of multiple dataset views. These views are displayed as separate but consistently updated files from the DRepl file server.

DRepl's approach allows flexibility on where the conversion between replicas is being performed. The file server can be run locally on the node that runs the scientific application, on the parallel file system nodes, or on nodes that perform I/O aggregation and forwarding. It works well with legacy scientific applications without imposing changes in their code.

Using multiple complete replicas of the data increases the reliability of the storage system. Also in a realistic work flow where multiple readers and writers are access the data, DRepl will improve the cumulative performance due to the performance gained from the read operations accessing the optimized replica.

DRepl ideas were used as basis by other researchers to explore similar approaches in other fields. For example, DRE [52] describes an FPGA implementation of a transformation engine that accelerates irregular and data-intensive applications.

The language has some limitations. The dataset description is static, with all variables and array sizes predefined at startup. Although this approach supports many scientific simulations, some like Adaptive Mesh Refinement, do not fit in this model. Although DRepl supports the most used array element orders, additional ones like Z-order [55] and Herbert curve [38] would be useful.

The file server implementations also have limitations, most notably the fact that they don't allow dynamic addition and removal of replicas and views.

Chapter 4

ASGARD: Extending DRepl for Distributed Storage

The initial work on DRepl provided mechanisms for accessing scientific data as binary files. Although this approach is useful, as it allows running unmodified legacy applications, it doesn't fully address the challenges of running HPC codes as well as the architecture of the modern storage systems for HPC. Recent adoptions of additional storage layers, like burst buffer, make HPC storage systems are increasingly more hierarchical and harder to represent as standard Unix file systems.

ASGARD is a more advanced approach on employing DRepl's ideas in a distributed environment. It abandons the idea of accessing the data using read and write operations to a file of a specific format. Instead, it provides a way for the developer to define how the data needs to laid out in the application's memory.

ASGARD uses that description, and the description of the data format on storage on storage to create a plan on how the data, possibly from multiple storage sources, needs to be retrieved, and what transformations are required to convert it in the required layout. This approach fits well with hierarchical storage systems that can include local persistent storage like NVRAM, burst buffer nodes, and parallel file system consisting of thousands of object store devices. Based on how data is partitioned across those active elements, ASGARD can generate transformation rules that can be executed remotely on the storage devices and extract the necessary subset regardless on the format the data is stored as.

ASGARD retains the language for defining of datasets from the original DRepl. It discards the concepts of views and replicas, and introduces a new entity that combines both. A $f$ragment of the dataset contains a subset of its data. Each array defined in the application rank can be a fragment. Or, all data, needed by the rank can be a fragment. Fragment definitions can be quickly created and are not necessarily kept in a centralized location like in DRepl.

The only information kept in one location is the collection of definitions of fragments that contain actual data. They are similar to the replicas from the original DRepl, but they don't contain all-to-all conversion maps.

Instead of offset and size, the read and write operations in ASGARD receive a fragment description, and a buffer in RAM where the data that needs to be written, or where the data that needs to be read should be placed. ASGARD provides functionality to query how to convert the content of one fragment to another. It provides transformation rules that describe the conversion between the two fragments. ASGARD can also calculate the coverage of fragments, providing a list of fragments required to produce the data for specific fragment.

In order to decrease the network utilization, ASGARD splits the transformation rules into two parts. The *remote rules* are executed on the server where the data of the source fragment is located. They produce a compact representation of the data that discard all unnecessary parts of the source fragment. That data is sent over the network to the node that requested the data. Once the compact representation is received, ASGARD applies the *local rules* to copy the parts of the source fragment to the appropriate locations in the destination fragment. ASGARD, in a sense, provides an extension to the standard scatter/gather operations. The transformation rules allow compact definition of complex patterns. The combination of gather (on the side where the data is stored) and scatter (on the side where data is used) provides a powerful tool for transforming data from the format in which it is produced to a format suited for consumption. The fact that the fragments are defined in advance allows an active storage subsystem to perform optimizations that might be too slow or costly if run at the time the operations are executed. A list of possible optimizations is described in section 4.2.4.

In order to evaluate ASGARD's benefits, we modified Ceph's [72] RADOS object store system to support ASGARD's transformation rules for storing and retrieving data to RADOS objects. We also created a custom object class that allows developers to define an object that represents an ASGARD dataset and specify how it is partitioned into “stripes” on storage. Each of the stripes is defined as ASGARD’s fragment. When a client needs to read or write a subset of the data, it can get a list of the stripe objects that contain the data as well as a pair of transformation rules that need to be applied to transform the requested subset to each of the stripes. The operations for each of the stripe objects are executed asynchronously to increase the overall performance. Our experiments show that ASGARD can improve the I/O performance by a factor of seven over both collective and non-collective MPI I/O.

4.1 Design

In many cases there is not a single “right” data format that is best for storing application data. The goal of this work is to allow developers to easily define the subsets of the data that each process needs, and to provide support for transforming the existing data into those subsets. Such functionality allows the storage subsystem to continually optimize the layout of the data. To support this ASGARD needs semantic knowledge of the application data. ASGARD doesn’t try to be an end-to-end solution; instead, it provides functionality that can be integrated into client libraries and storage systems. It doesn’t cover actual data storage or decisions on where and how data should be replicated in order to optimize I/O performance, but does provide the semantic information so these decisions can be made intelligently.

In addition to DRepl dataset language, ASGARD provides a C API for dataset definition. Although using an expressive language makes it easier to communicate and share data, there are cases when embedding a DRepl parser is too cumbersome and slow. For example, incorporating a parser in a storage system can unnecessarily affect its performance and memory footprint, while adding little value.

ASGARD uses two major entities related to data and the way it is used and stored. *Dataset* is an abstract definition that describes the data types and data objects that are of interest of any application, regardless of whether it is the producer or consumer of the data. For scientific applications, datasets are usually big and consist of all data produced by simulations run on thousands of processors. *Fragment* is a subset of the dataset and defines the parts of the data that are of interest to a particular set of applications, an application, or one of the application's ranks.

For ASGARD's design, the following assumption were used:

- Processes create and use a subset of the application data;
- A process' fragment(s) don't change often and don't depend on dataset values;
- There is no single "right" data layout. Data might be stored in the format in which it is produced or consumed, or any other intermediate format;
- Storage systems can optimize performance by using information about the structure of data and the patterns by which data is accessed;

- The data is likely to be stored far from where it is used.

Although the main goal of ASGARD is to improve I/O performance for HPC workloads, its design is general enough so it can also improve other common use cases. For example, it can be used to access a 2D region an image file, or specific rows from a database table. This wider applicability can allow ASGARD to be included in a broader range of storage systems.

4.1.1 Changes to the Language Definition

The DRepl language is was designed to provide virtual files that can be read and written partially. This approach does not require the size of the array slices to be restricted, because the user can restrict the amount of data they require at access time. ASGARD doesn't allow partial access to fragments, therefore the arrays size needs to be restricted as part of the fragment definition. ASGARD extends the DRepl language by allowing the user to define the size of the array slice in addition to the expression that moves its elements. The example below creates a 50x50 slice `b` from the original `data` array:

```
var b[i:50,j:50] = data[i, j]
```

4.1.2 Dataset and Fragment Representation

Unlike DRepl, ASGARD doesn't generate global conversion map that provides transformations from each view and replica to any other view or replica. Each fragment is defined as a list of Blocks, with a subset of them being top-level Blocks.

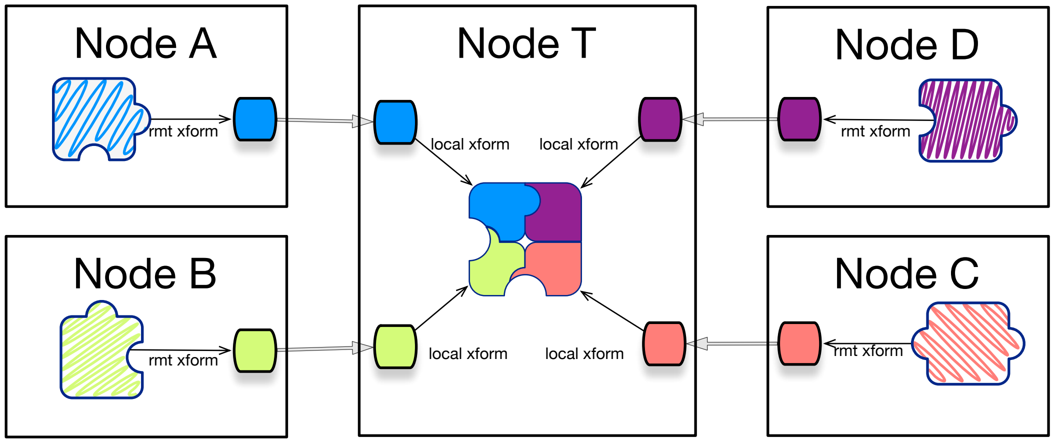


Figure 4.1: Construction of fragment's data from remote content

Each block that is not part of the top-level list can be referenced only once as a field of a TBlock, or an element of an ABlock. ASGARD doesn't use the source Block field of each Block. This approach makes it easier to generate, and optimize the transformation rules for conversions between two fragments.

The dataset is a special case of a fragment. Blocks in normal fragments have a single entry in their destination list – the Block in the dataset. The dataset Blocks keep track of the Blocks in each fragment they correspond to.

4.1.3 Fragment Source Coverage

Due to the fact that data for a dataset is stored in multiple fragments on storage, materializing a particular fragment into the application's memory generally requires pulling data from a subset of the other fragments (Figure 4.1). ASGARD provides a mechanism to retrieve all possible combinations and what percentage of the desired fragment they cover. Section 4.2 has details on the

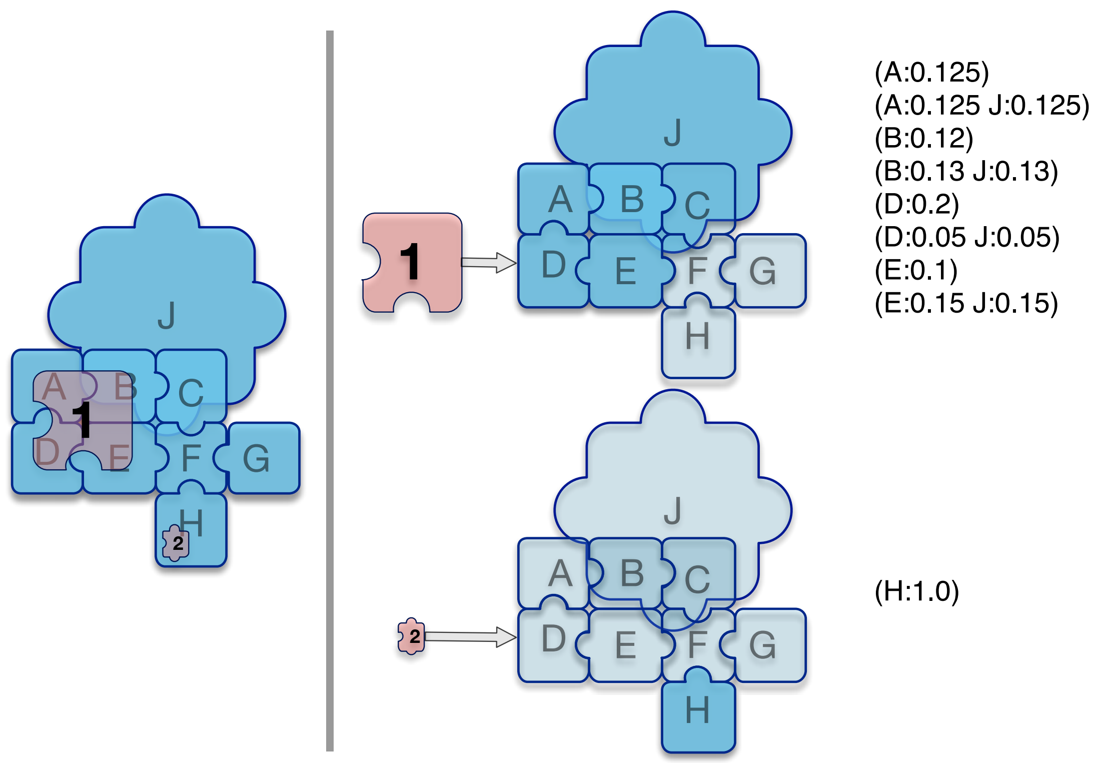


Figure 4.2: Example of coverage lists for two fragments

implementation specifics. In a complex hierarchical storage system, there might be multiple fragments at different locations, that contain subsets of the required data. Providing all possible combinations gives an opportunity to the application, a runtime, or the storage system, to select a combination that achieves the best overall performance.

The source coverage of a fragment is described by a list of lists of sources. Each of the sources contains a name of a fragment and the percent of coverage it provides. The inner lists provide interchangeable options of fragments, that when combined with other options within the outer list consist of the fullest possible coverage.

Figure 4.2 shows an example for source coverage for fragments `1` and `2` from fragments `A`, `B`, `C`, `D`, `E`, `F`, `G`, `H`, and `J`. Fragment `2` is 100% covered by fragment `H`, while 12.5% of `1` is covered solely by `A`, 12.5% is covered by both `A` and `J`, 12% is covered solely by `B`, 13% is covered by both `B` and `J` and so forth.

4.1.4 Fragment Transformation

Once a combination of fragments as sources is chosen, the user can generate transformation rules for converting data from each pair of source and destination fragments.

The transformation rules are generated for two fragments: a source and a destination fragment. They define what data from the source fragment is needed, and how to convert it to the format of the destination fragment. They are symmetric and can be also be used to convert data in the opposite direction.

In complex datasets, the elements of an ABlock can be TBlocks, which can in turn contain fields that are ABlocks, with TBlock elements, and so forth. Two fragments of the dataset can contain different subsets of elements and fields for each of the blocks, and ASGARD will recursively apply the correct transformations to convert one fragment to the other.

Figure 4.4 shows the internal representation defined for the abstract dataset, as well as the two fragments **`frag0`** and **`frag1`** defined in Figure 4.3.

The dataset has two blocks defined – an ABlock with two dimensions with sizes $[80000, 80000]$, and an element of the array which is a SBlock. Fragment **`frag1`**

```
dataset {
   var data[80000, 80000] float64
}

fragment frag0 {
   var ds[i:1000, j:1000] = data[i, j]
}

fragment frag1 {
   var a [i:5000, j:6000] = data[i+500, j+300]
}
```

Figure 4.3: Example of a dataset and two fragments

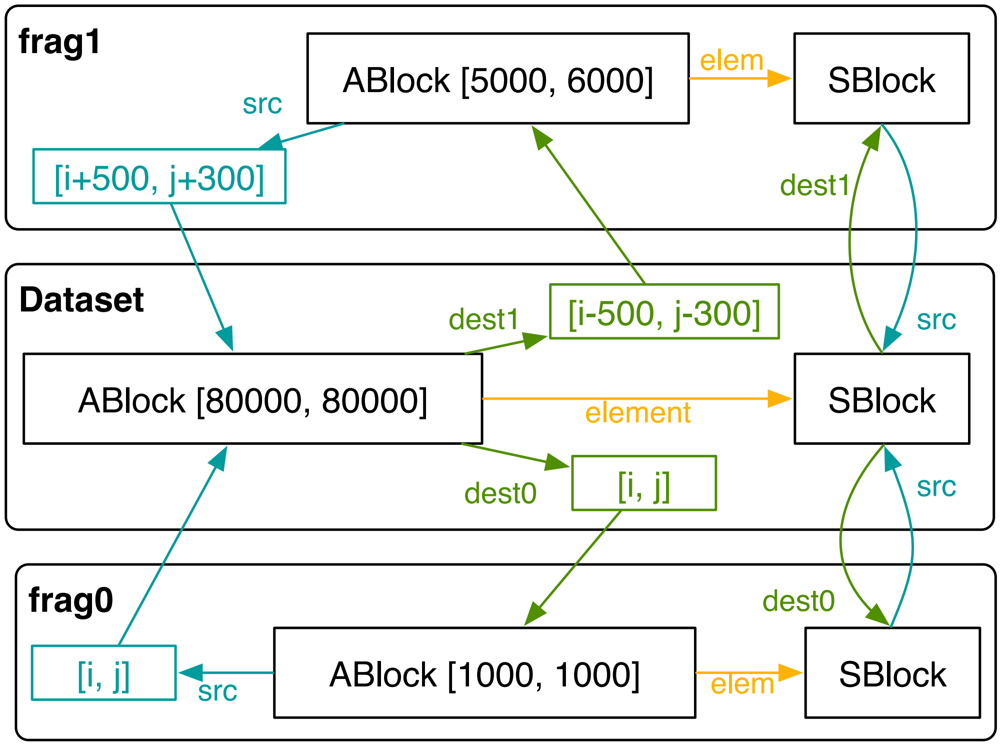


Figure 4.4: Example of a conversion map of two fragments

contains an ABlock with 2 dimensions $[5000, 6000]$. The ABlock is connected to corresponding ABlock in the dataset, where element $[i, j]$ from it corresponds to element $[i + 500, j + 300]$ in the dataset ABlock. Similarly, `frag0` defines a smaller ABlock with 2 dimensions: $[1000, 1000]$ and its element $[i, j]$ corresponds to the element with the same address in the dataset ABlock.

In distributed environments, where communication is expensive, it is beneficial to split the transformation rules into two parts. The *remote transformation rules* extract the necessary data from the source fragment and convert it into a compact intermediate buffer that can be efficiently transmitted over the network to where the data is needed. The *local transformation rules* use the data from the intermediate buffer and place it in the right format in the destination fragment.

The blocks in the fragments have the appropriate dataset blocks as sources, and accordingly the dataset blocks have the fragments' blocks as destinations. Figure 4.5 shows the remote and local transformations for materializing fragment `frag0` from fragment `frag1`.

4.1.5 Memory vs. Storage Layouts

In some cases, the in-memory data layout is different than the content of the data that needs to be written. For example, the numerical algorithms require the data for a cell, and the ghost values from the neighboring cells to be kept together in the same multidimensional array (Fig. 4.6). The two-dimensional 8x8 array A updates the central 6x6 elements that represent fragment F_{22}, while the rest are

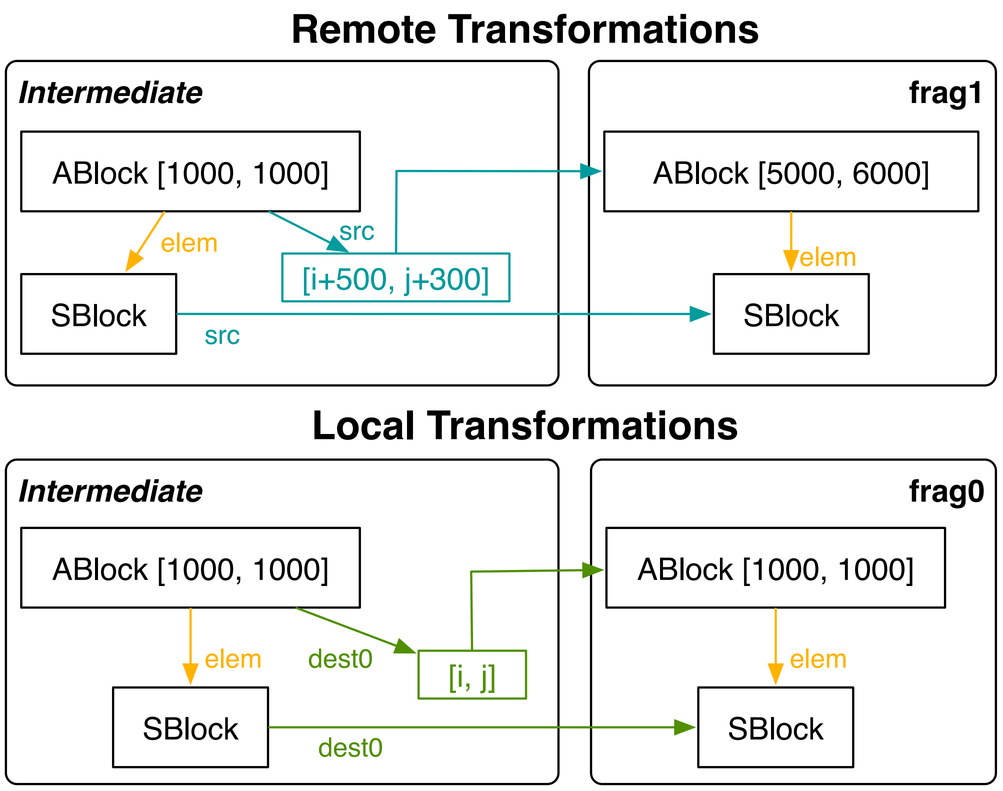


Figure 4.5: Example of local and remote transformation rules

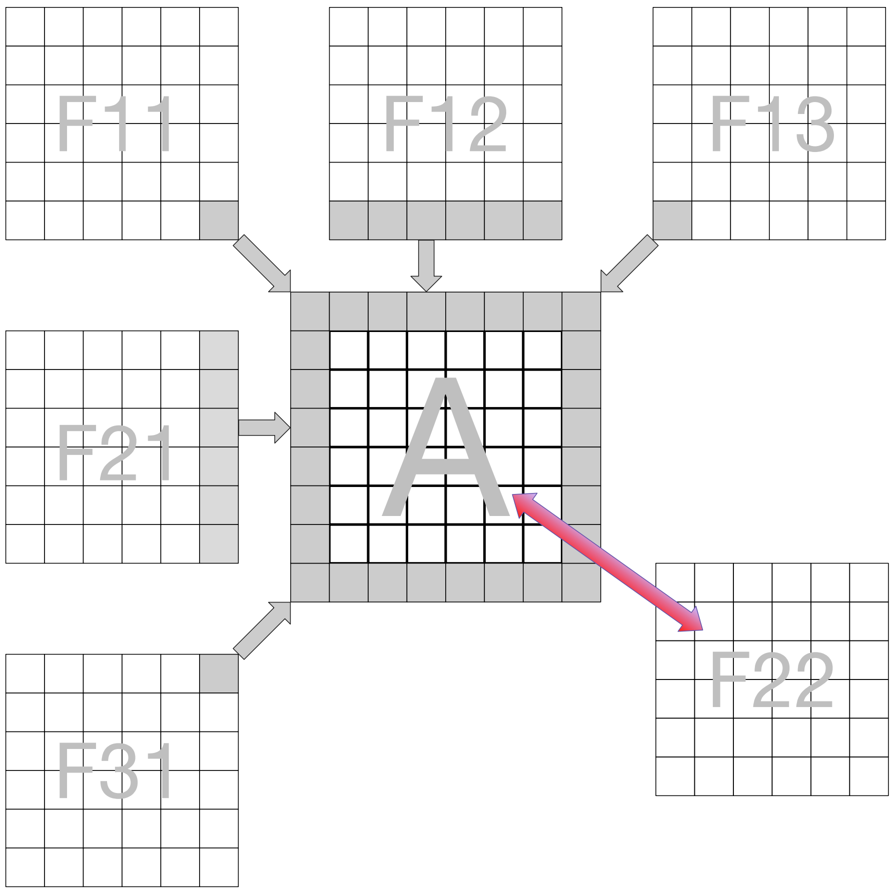


Figure 4.6: Array A with ghost cells from fragments F_{11}, F_{12}, F_{13}, F_{21}, A_{31}. The middle cells of A represents the data for fragment F_{22}.

ghost values from the neighboring fragments F_{ij}. When the rank needs to save its data, only F_{22} needs to be saved, and the ghost cell values have to be ignored, since they are updated by other ranks.

ASGARD provides support for combining more than one set transformation rules together. The application can use that functionality to transform the fragment representing A to F_{22} with the local transformation rules that transform F_{22} to the fragments saved on persistent storage.

4.2 Implementation

ASGARD's implementation is based on DRepl's replication engine. The read and write functionality was replaced by code that transforms data from one fragment, in its entirety into another. The functionality was greatly extended by providing support for finding the fragment's coverage, generating split transformation rules for a pair of fragment, and improving the transformation by optimizing the transformation rules description.

DRepl's language parser, once integral part of the system, is no longer the only way to construct fragments. It is also the only part that is still implemented in Go only. The rest of the functionality has both Go and C implementation. ASGARD's C implementation is only three thousand lines of code, and the fact that it is in C makes it easy to integrate it into most storage systems.

4.2.1 Transformation Engine

The most essential part of ASGARD is the transformation engine. It receives the transformation rules (i.e. two fragments, with their Blocks connected), and a buffer containing data formatted as defined by one of the fragments. The engine transforms the data into the format, defined by the other fragment.

The transformation is performed iteratively for each top-level Block from the source fragment (Algorithm 4.2.1). For compound blocks, like TBlock and ABlock, the transformation is run recursively for each of their elements.

Algorithm 4.2.1: TRANSFORM$(source, dest, srcbuf, destbuf)$

```
for each $block \in source.toplevel$
  do { for each $dblock \in block.destinations$ and $dblock \in dest$
         do TRANSFORMBLOCK$(block, dest, buffer)$

procedure TRANSFORMBLOCK$(sb, db, sbuf, dbuf)$
 $dest \leftarrow db.fragment$
 switch $sb.type$
   case $SBlock$
     do { $copy(dbuf + db.offset, sbuf + sbuf.offset)$
   case $TBlock$
     do { for each $fld \in sb.fields$
            do for each $dfld \in fld.destinations$ and $dfld \in dest$
                 do { $so \leftarrow fld.offset$
                      $do \leftarrow dfld.offset$
                      TRANSFORMBLOCK$(fld, dfld, sbuf + so, dbuf + do)$
   case $ABlock$
     do { for each $idx \in sb.dimensions$
            do { $so \leftarrow$ IDXTOOFFS$(sb.elo, sb.dimensions, idx)$
                 $didx \leftarrow$ CONVERTIDX$(sb.elo, idx, db.elo)$
                 $do \leftarrow$ IDXTOOFFS$(db.elo, db.dimensions, didx)$
                 TRANSFORMBLOCK$(sb.el, db.el, sbuf + so, dbuf + do)$
```

4.2.2 Fragment Source Coverage

The coverage value is defined between two fragments – a source and a destination. It defines how much data both fragments share. Coverage of 1 means that all data for the destination fragment is available in the source fragment. Similarly, coverage of 0 means that the fragments don't share any data.

Identifying the list of fragments needed to materialize a given fragment is not a trivial task. Fragments may (and usually do) overlap, so data might be pulled from many combinations of remote fragments. The ASGARD runtime doesn't have the knowledge on which combination of fragments will provide the best overall performance. Therefore, it provides the functionality of finding all variants and let's the application, or the storage system, using additional information, to pick the correct choice.

The API returns a two-level list. The elements of the outer list are lists of tuples of the format `(percentCoverage, fragmentName)`. Each of the elements of the inner list are interchangeable and cover the same area of the fragment. The user needs to pick only one of them in order to cover that area. The maximum coverage of the fragment is achieved by iterating the outer list, picking an entry from each inner list and combining them together.

Internally, each inner list is represented by a structure called `OrSet`. An `OrSet` is a list of `(percentCoverage, fragmentName)` tuples called `Source`. There are two operations defined for the `OrSet`. Adding a `Source` to the set might add an additional entry in the list if the fragment is not part of the set. If the tuple fragment is already part of the set, the coverage is updated to be the maximum from the original and new value. Scaling an `OrSet` multiplies the coverage values of all tuples by the number specified.

The outer list is represented by a structure called `AndSet`. It is a list of `OrSet` entries. Adding an `OrSet` to an `AndSet` first checks if there are entries with exactly the same list of fragments. If so, for each fragment in the set it updates the coverage value by adding the new one. Otherwise, it adds a new `OrSet` entry. The union operation combines two `AndSet`s into one by iteratively adding the entries from one to the other. Scaling an `AndSet` scales each of the `OrSet` entries it contains.

Discovering coverage for a SBlock or a TBlock is relatively straightforward. The coverage of an SBlock is a `OrSet` with all fragments that contain the same SBlock and coverage value of 1. Coverage of a TBlock is computed recursively by finding the coverage for each of the fields, and scaling it by the size of the field divided by the size of the whole TBlock.

Finding coverage for an ABlock is more convoluted, as described in Algorithm 4.2.2. For each dimension of the multidimensional array, the coordinates of the vertices for each fragment are collected, and sorted. Then they are used to generate all combinations of multidimensional cubes. Each cube is associated with one `OrSet`. The algorithm checks if each of the fragments overlaps with a cube, and if so it adds an entry for it to the appropriate `OrSet`. To make things even more complicated, the elements of the slices in fragments may only contain the subset of the data that the element of the array we are covering, so the algorithm is run recursively to find the coverage of the element before we can create the appropriate entry. Luckily, we don't have to check for full coverage of the element Block, since we only care the percent of coverage for the already selected source fragment. Lastly, we combine the cubes coverage into one `AndSet` that covers as much of the original ABlock as possible.

Figure 4.7 shows and example of the four fragments covering part of the same array. The black dots show all combinations of vertices that we use to generate the coverage cubes (in 2D case, rectangles). Each of the dashed rectangles is checked against the slices areas. Rectangle 1 is not covered by any fragment, so it doesn't produce any entries. Rectangle 2 is covered by one fragment, so it produces one `OrSet` with a single entry. Rectangles 3 and 4 are covered by 2 and 3 fragments respectively, and produce `OrSets` with 2 and 3 entries.

4.2.3 Split Transformation Rules

In distributed environment, the data for a fragment can be on a remote node, and sending it over the network is both slow and expensive, in energy terms. In the general case, both the source and the destination fragments are big, but the size of the data they share is relatively small. Transferring the whole data of the source or the destination fragments to the other location and performing the transformations there is ineffective.

ASGARD provides functionality for creating a new fragment that is an intersection of the source (i.e. remote) and the destination (i.e. local) fragments. The transformation rules for converting the destination to the intermediate fragment are executed remotely. The rules for converting the intermediate fragment to the destination are executed locally.

The procedure for creating the intermediate fragment is relatively straightforward. It iterates through all top-level blocks in the source and creates their intersection block with the destination. If there is no intersection, no block is added to the intermediate fragment. If a SBlock from the destination is connected to a block from the source, the destination is an SBlock with the same size. If a TBlock is connected to a block from the source, the procedure is executed recursively for each of its fields, and the result is a TBlock that includes all fields that intersect. The intersection of an ABlock has all elements that belong both to the source and the destination ABlocks.

4.2.4 Optimizer

The investigations on how to improve the performance of the original DRepl implementation by optimizing the conversion map proved mostly futile. The fact that the conversion map contains transformation rules from multiple views and replicas restricts the opportunities for optimization. In ASGARD the transformation rules define the conversion between two fragments increasing the chances for optimization opportunities.

As the transformation rules can be complex, ASGARD's performance in the general case can be sub-optimal. There are many common cases when the transformation rules can be optimized further. We implemented some optimizations that cover the most frequently used patterns.

The optimization procedure is given two fragments and produces two new fragments that have identical data layout, but might contain fewer blocks and simple transformation rules between them. The algorithm again iterates through the top-level blocks and tries to optimize each of them individually. For the compound blocks (i.e. TBlocks and ABlocks), it first tries recursively to optimize their fields/elements.

Figure 4.8 shows two of the optimizations that are currently performed on TBlocks. If sequential SBlock fields within a TBlock are copied to sequential fields in a destination TBlock, they can be replaced by a single SBlock with their combined size. If all fields from a TBlock are copied, the whole TBlock is replaced with an SBlock.

ABlock optimizations (Figure 4.9) are currently performed only if both source and destination blocks are of the same element type. They include replacing a dimension of the ABlock with a TBlock if the required elements for the dimension are sequential. For example, if the transformation rules of a 3D floating-point matrix with size $(100, 100, 100)$ copy the hyper-slab $(10 \ldots 50, 10 \ldots 50, 10 \ldots 50)$, the original ABlock is replaced by a 2D matrix with size $(100, 100)$ containing TBlocks with three fields: a hole of size `float[10]`, a SBlock of size `float[40]`, and another hole of size `float[50]`. Thus applying the transformation rules will be executed for ten thousand (slightly more complicated) elements, instead of one million. As with TBlocks, if all elements from an ABlock are copied, the optimizer replaces it with a SBlock with the same size.

4.2.5 Ceph Integration

ASGARD was integrated into the Ceph distributed storage system. The modifications are primarily in Ceph's object store device (OSD) layer – RADOS. A new storage object class that represents a dataset was created. As the default read/write operations of RADOS objects were limited (they allow accessing only one range of sequential data), we introduced two new operations that receive as one of their parameters transformation rules and can atomically extract all the data specified by the transformation. In order to achieve good performance, these operations needed to be extended into the the layer responsible to the actual saving of data on persistent storage – the ObjectStore level.

The Ceph integration heavily utilizes ASGARD's functionality that serializes a collection of fragments, and the connections between their Blocks, into an array of bytes suitable for transmission over a network connection.

The RADOS distributed object store

RADOS (Reliable Autonomic Distributed Object Store) is part of the Ceph project. It provides scalable storage for data entities – objects. The objects can be of arbitrary size. Each object has a name and belongs to an object pool. The pools define certain policies that apply to all objects in the pool. For example, they define the replication level and the mapping rules for distributing the replicas across the storage cluster.

The RADOS storage system consists of multiple object storage devices (OSDs). They can be accessed directly from the clients, decreasing the single points of failure in the system, and increasing the overall performance. The OSDs operate mostly independently from each other, with global cooperation needed only at events like removing (at failure) and adding additional devices.

There are two types of values associated with an object. The first one is a byte array, similar to POSIX files. The user can read or write ranges from the data array that are defined by an offset and range size. The second set of values, usually referred as metadata, is a key/value store that allows the user to associate arbitrary number of values for a key.

RADOS also provides a higher-level interface where storage developers can create object classes. The object classes define a list of methods that can be used from RADOS users. The methods are implemented by using the low-level object interface. The methods are executed on the storage device where the object's data is located, and greatly improves the performance since most of the data access is local.

ASGARD Object Class

An object of the new ASGARD class represents a whole dataset. It doesn't contain any actual data from the dataset, only the metadata required to find out the location of the data. There is similarity between a Ceph object that describes a file on the Ceph filesystem and an ASGARD object. Both contain the metadata common for the whole file, as well as names of RADOS objects that contain the actual data. Unlike files, the ASGARD objects allow the data to be "striped" in any format supported by the ASGARD runtime. For example, a dataset that contains an array of structs can have it stored in objects that have a separate array for each field of the struct instead.

Although all fragments are treated equally by ASGARD, the Ceph integration uses a certain convention. All fragments, except one, are connected only to one other fragment, and that fragment is considered the dataset. The Blocks of the dataset can be connected to Blocks in multiple fragments. A fragment description in Ceph always contains both the fragment and the dataset descriptions.

All data for the ASGARD object (i.e. metadata for the ASGARD dataset) is stored as object properties in the object map.

When an ASGARD object is created, the user provides a description of the dataset, as well as all ASGARD stripes, or *persistent fragments*. The input parameter of the create method is a list of connected fragments. The first fragment in the list is the dataset description, and the rest are descriptions of the persistent fragments. The name of the object, and the fragments' names are used to generate names for the RADOS objects that will contain the actual data for each fragment.

For existing objects, the user can retrieve the ASGARD description of the dataset and the stripes as a whole, or just the description of the dataset.

The most significant operation on ASGARD objects is getting a fragment's coverage. When the user calls the method, it passes a description of the fragment it needs to read or write. The ASGARD class calculates which persistent fragments contain the data for the requested fragment, and returns a description that contains list of fragments: the first fragment in the list is the fragment passed as an input, the second is the dataset fragment. The rest of the fragments are the persistent fragments that contain the data necessary for materializing the input fragment. Based on the information, the user can construct the transformation rules for each persistent fragment, and access the RADOS objects directly and asynchronously.

The ASGARD object class implementation also includes a user library that hides the burden of reading and writing data stored in the persistent fragments. Given a fragment, it queries RADOS for the fragment coverage, and calculates the remote and the local transformation rules for each RADOS object that needs to be accessed. Since the construction of the transformation rules can be slow, and it is likely that the same subset of the data will be used multiple times, the library allows the operation to be performed well in advance and the information to be stored for later use. When a read or a write operation on the fragment is performed, the client library asynchronously issues all requests, waits for the data from each object to arrive, and applies the local transformation rules to convert it to the right format.

New OSD Operations

The second part of the integration in Ceph is introducing two new operations on RADOS objects. The original RADOS only supports accessing sequential regions of an object defined by offset and length. We added two additional operations *dread* and *dwrite*, that use ASGARD transformation rules to specify what data from the object is needed and how to convert it to a compact buffer that is passed to/from the client. The transformations are applied on the OSD instances that contain the data, significantly decreasing the number of operations over the network, increasing the overall throughput of the storage system, and decreasing the I/O latency. The client has an opportunity to execute multiple operations asynchronously, potentially to multiple OSD instances, and therefore utilize the full potential of a Ceph RADOS cluster.

Secondary Storage Fragments

Instead of updating all fragments from the client, the Ceph integration allows the primary persistent fragments to be associated with other RADOS objects called *s*econdary fragments. They can be fragments of another dataset, specially marked fragments of the same dataset, or just replicas of the data that are not formally connected to a dataset. When the client asks for coverage, the secondary fragments associated with the same dataset are not returned. If the client updates one of the primary fragments, the update automatically triggers updates to the secondary targets, initiated by the OSD where the particular primary target is located. The implementation uses the transformation rules between the primary fragments and the associated secondary fragments, and applies the technique mentioned in section 4.1.5 to transform the data received from the client directly to the format appropriate for the secondary fragment.

4.3 Experimental Study

In order to evaluate ASGARD's performance, we set up a small Ceph cluster with 4 OSD servers and 1 metadata server, running on 4 nodes. The nodes have an eight-core Intel Xeon processor and 128GB of RAM. We used Ceph's default settings. Each object store instance was using a single rotational SATA disk with a capacity of 4 TB. There were 8 additional nodes that we used during the tests. All nodes were connected with an Infiniband EDR interconnect. We used OpenMPI as our MPI library.

The goal of our experiments is to demonstrate the benefits of offloading complex data transformations to the storage system. We compared ASGARD to MPI I/O, as both provide similar functionality and are relatively easy to use. The first test we ran was the MPI Tile I/O benchmark from the Parallel I/O Benchmarking Consortium [9]. We modified the code and added support for Ceph ASGARD objects. We ran the code on total 12 nodes, including the 4 that run Ceph. Tile I/O partitions a 2D array of data into multiple MPI ranks. Each rank is responsible for reading or writing a tile of the whole array. We varied the size of the tiles from 512x512 to 4096x4096 elements while running 8 ranks per node (96 total ranks). In order to evaluate the bottlenecks of the setup, we also ran the tests with the same tile size (4096x4096), but varying the number of ranks per node, from 1 to 8 (total 96 ranks).

Figure 4.10 shows the read and write performance while varying the tiles and ranks. ASGARD outperforms both read and write for the non-collective and collective MPI I/O operations. We used Ceph's monitoring infrastructure to inspect the number of operations and read/write bandwidth going to the OSD instances. ASGARD's implementation used the fewest number of operations per second. While both ASGARD and collective MPI I/O had similarly high data size per operation, non-collective MPI I/O used a high number of operations with much lower data size per operation. During write operations, in addition to writing data to the OSDs, non-collective MPI I/O tests were also reading a lot of data. The reason for that is that the Linux file systems operate on blocks, and modifying even small amount of data per block requires reading the whole block, modifying some of the data, and writing the block.

We also evaluated I/O performance using the HPIO benchmark [2]. It is highly customizable and allows I/O workloads that support contiguous/noncontiguous data layout both in memory and in files. For these tests we run the benchmark only on the 8 nodes that were not running Ceph, with 8 ranks per node (total 64).

For the first evaluation we tried to mimic a use case where the dataset consists of a one-dimensional array of structs, containing three 64-bit values. The application is only concerned with one of these values. When the memory or the file layouts are contiguous (C), only the value of interest is stored. When they are non-contiguous (N), all data is stored, but only the value of interest is accessed. Figure 4.11 shows the results for reading, while varying the size of the array. For contiguous reads from storage, non-collective MPI I/O access performs well, as Ceph file system is well-optimized for reading the contiguous file layout and most of the data is prefetched to the local cache. ASGARD's performance is comparable, but can probably be improved with further optimization. When data is read from the file in non-contiguous patterns, both collective and non-collective

MPI I/O degrade significantly, by a factor of 100 for the non-collective case. ASGARD clearly outperforms both MPI I/O modes and is up to ten times better than collective MPI I/O. Figure 4.12 shows similar results for write performance. ASGARD performs noticeably worse for very small writes, because the overhead for sending the transformation rules, as well as encoding and decoding them, becomes the main bottleneck. For non-contiguous file patterns, ASGARD performs up to five times better than collective MPI I/O.

Ceph provides statistics on the read and write bandwidth of the overall cluster, as well as the total number of operations per second. Figure 4.13 shows the performance while running the 96-rank experiment with MPI Tile I/O benchmark. The graphs for ASGARD and Collective MPI I/O are much shorter, because of the higher performance, the experiments completed faster. As expected by the numbers provided by the benchmark itself, ASGARD has the highest write bandwidth (Figure 4.13a). Although the collective and non-collective MPI I/O bandwidths are close, non-collective MPI I/O takes much longer, and writes much more data. The reason for that is the smaller chunks of data written cause Ceph to read-modify-write much more often than with collective MPI I/O. Figure 4.13b partly explains the differences in performance. While ASGARD performs only few operations per second, collective MPI I/O performs close to 40, and non-collective MPI I/O performs close to 500. Although ASGARD's and collective I/O operations have much larger data loads, the overhead of the operations causes huge degradation for non-collective MPI I/O.

4.4 Performance Evaluation with Real Applications

ASGARD was integrated in the UNITY [43] storage system that is being developed jointly by Oak Ridge National Laboratory, Georgia Tech, and Los Alamos National Laboratory. The goal of the project is to unify memory and storage spaces. It tracks the scientific application data from the application's RAM through the multiple locations on storage, including the eventual storage on archival systems. It extends beyond files, and uses the mechanisms developed in ASGARD to describe the data in terms of datasets and fragments both in memory and on storage. ASGARD provides the necessary flexibility so the data can be replicated closer to where it is needed and transformed transparently to the user applications so it can be in formats that improve the overall performance.

The UNITY system performance was evaluated [58] with two applications – SNAP and VPIC. SNAP [75] is a proxy application that represents a deterministic neutral-particle transport simulation. VPIC [13, 11, 12] is a general-purpose particle-in-cell simulation for modeling kinetic plasmas is one, two, or three spatial dimensions.

We include these results with permission from the authors of the evaluation in order to demonstrate that ASGARD performs well with real applications.

Figure 4.14 shows the results comparing UNITY's performance compared to collective MPI I/O for storing SNAP checkpoint data on disk. The experiments use weak scaling, i.e. the problem size per rank is constant when the number of ranks varies. The data was stored on a Lustre filesystem. UNITY outperforms the MPI I/O solutions, while providing greater flexibility on the number of nodes that can be used to restart the application.

Figure 4.15 depicts the relative performance of VPIC while using UNITY for data storage. "VPIC I/O" refers to the original VPIC implementation with the existing I/O functionality activated in the input deck. 'RT/NS" refers to the VPIC implementation which uses the UNITY to allocate data but the data is not saved on disk. "RT/NS I/O" refers to the same runtime data allocation implementation but file I/O is issued through the UNITY interface. The y-axis represents the overhead of the experiments as compared to running VPIC with the same input parameters, but without any I/O being performed. The overhead of using UNITY for just memory allocations and fragment publishing is negligible. Adding I/O to the runtime increases the overhead to about 75% in the largest experiment, while the existing VPIC I/O functions nearly doubled the runtime of the original.

4.5 Discussion

ASGARD provides a simple language for describing complex scientific datasets and subsets. It creates a compact representation of the transformations required for data conversion between fragments. ASGARD is designed for distributed systems and intelligent storage elements. It optimizes the size of the data sent over the network by offloading some of the conversion to the storage system.

The results demonstrate that complex scatter/gather transformation rules allow superior performance without synchronous progress mandated by collective MPI I/O.

The availability of rich semantic information about the dataset as well as the separation between the fragments' declarations and the data conversions provide many opportunities for performance optimizations. Results showed that ASGARD reduced the number of operations going to the OSD per data transferred.

Although the Ceph integration provides support for secondary storage fragments, that functionality hasn't been fully evaluated yet. The Ceph integration supports overlapping primary fragments, but there is currently no logic on how to choose between multiple combinations of overlaps.

Algorithm 4.2.2: $\text{ArraySources}(b, sources)$

```
local $andset, orset, complete, v, idx, cube$
$andset \leftarrow \emptyset$
for each $dim \in b.dimensions$
  do $v_{dim} \leftarrow \emptyset$
for each $d \in sources$ and $b \cap d \neq \emptyset$
  do for each $dim \in b.dimensions$
       do $v_{dim} \leftarrow \text{ADD}(v_{dimm}, d.start_{dim})$
          $v_{dim} \leftarrow \text{ADD}(v_{dimm}, d.end_{dim})$
for each $dim \in b.dimensions$
  do $\text{SORT}(v_{dim})$
for each $dim \in b.dimensions$
  do $idx_{dim} \leftarrow 0$
$complete \leftarrow false$
repeat
  for each $d \in b.dimensions$
    do $cube_{dim}.start \leftarrow v_{dim,idx_{dim}}$
       $cube_{dim}.end \leftarrow v_{dim,idx_{dim}+1}$
  $cov \leftarrow \text{VOLUME}(cube)/\text{SIZE}(cube)$
  $orset \leftarrow \emptyset$
  for each $d \in sources$ and $b \cap cube \neq \emptyset$
    do $dcov \leftarrow \text{SCALE}(\text{BLOCK-COV}(b.el, d.el), cov)$
       $orset \leftarrow \text{ADD}(orset, dcov)$
  if $orset \neq \emptyset$
    then $andset \leftarrow \text{ADD}(andset, orset)$
  $dim \leftarrow 0$
  repeat
    $idx_{dim} \leftarrow idx_{dim} + 1$
    $dim \leftarrow dim + 1$
  until $dim < \text{LEN}(b.dimensions)$ or $idx_{dim-1} < \text{LEN}(v_{dim-1})$
  $complete \leftarrow dim \geq \text{LEN}(b.dimensions)$
until $complete$
return $(andset)$
```

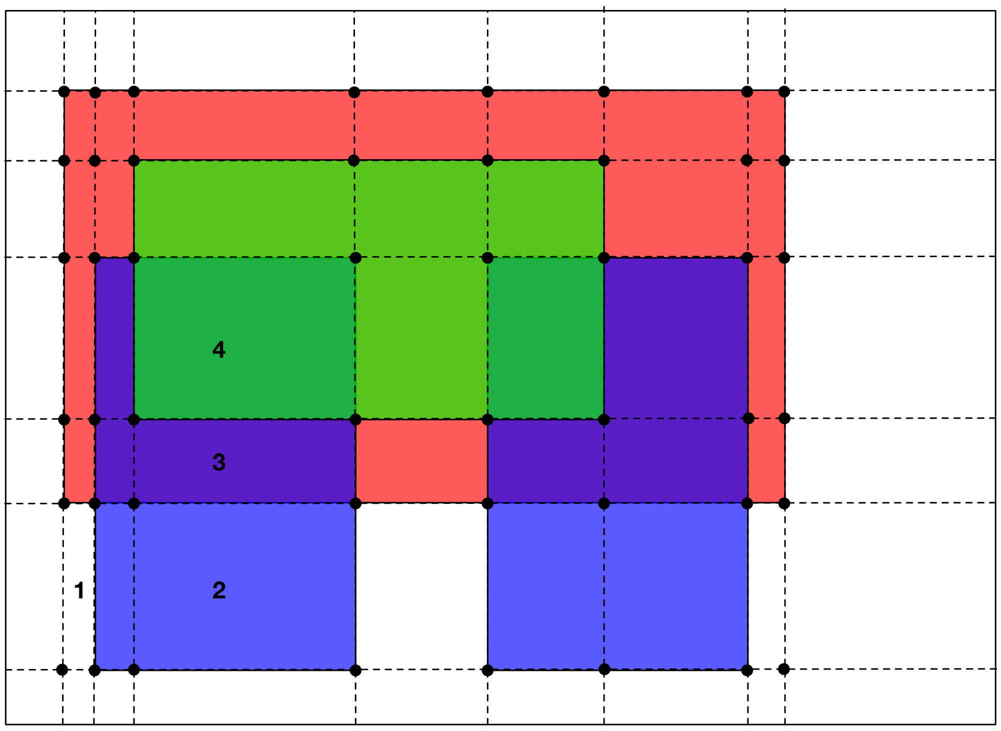
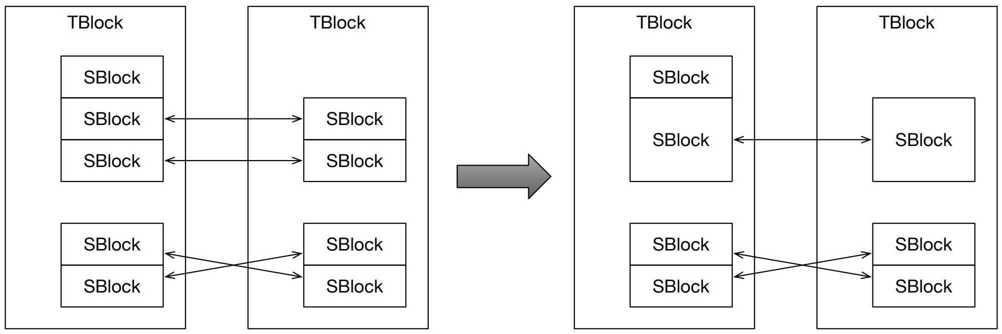
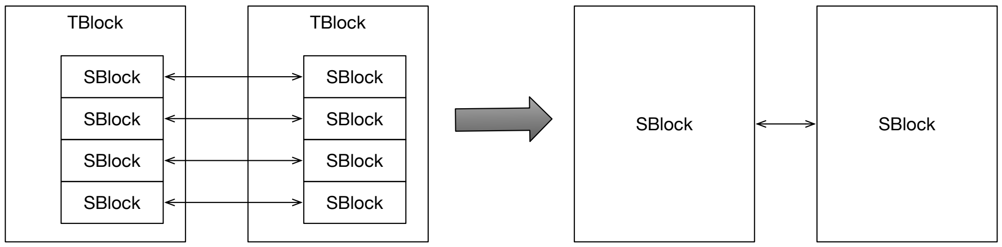


Figure 4.7: Points of interest for a 2D array with four fragments. The green fragment has 100% coverage, with some parts of it covered by more than one other fragment. The black points define rectangles that are checked against each fragment to find how many of them cover it.



a. Merging neighboring fields



b. Replacing TBlock with a SBlock

Figure 4.8: TBlock Optimizations

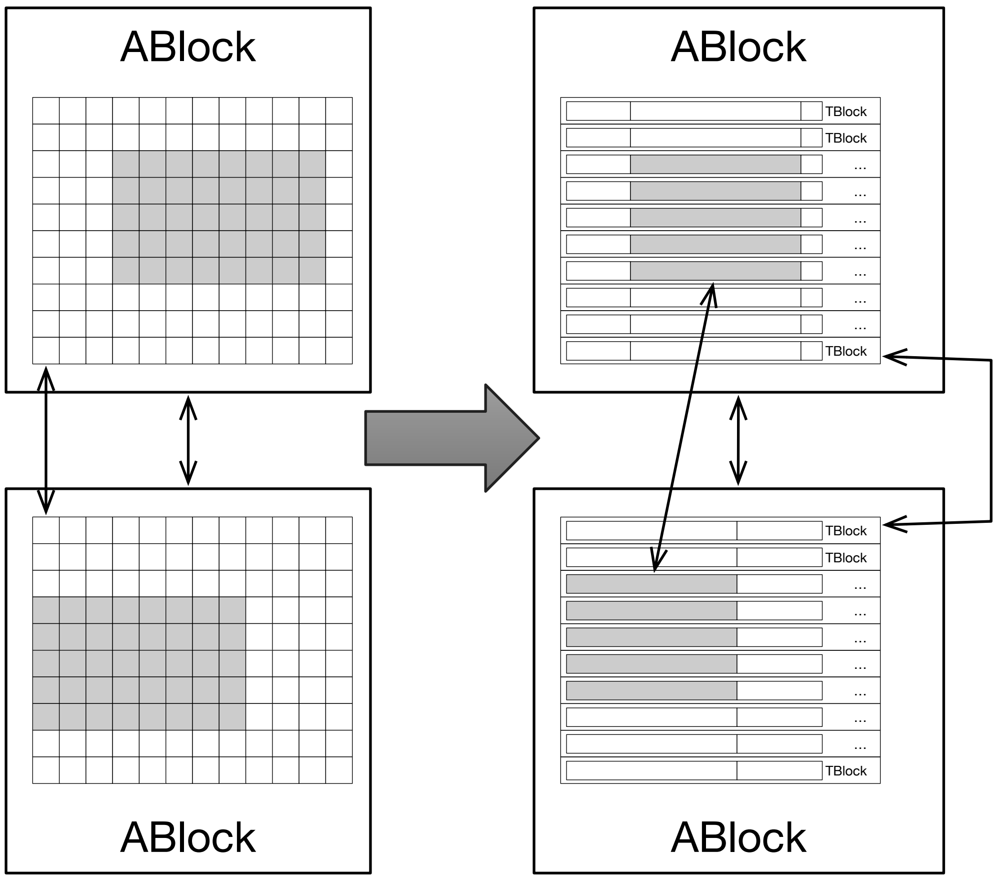


Figure 4.9: ABlock Optimizations. Two-dimensional ABlocks with SBlock elements are replaced with one-dimensional ABlocks with TBlock elements.

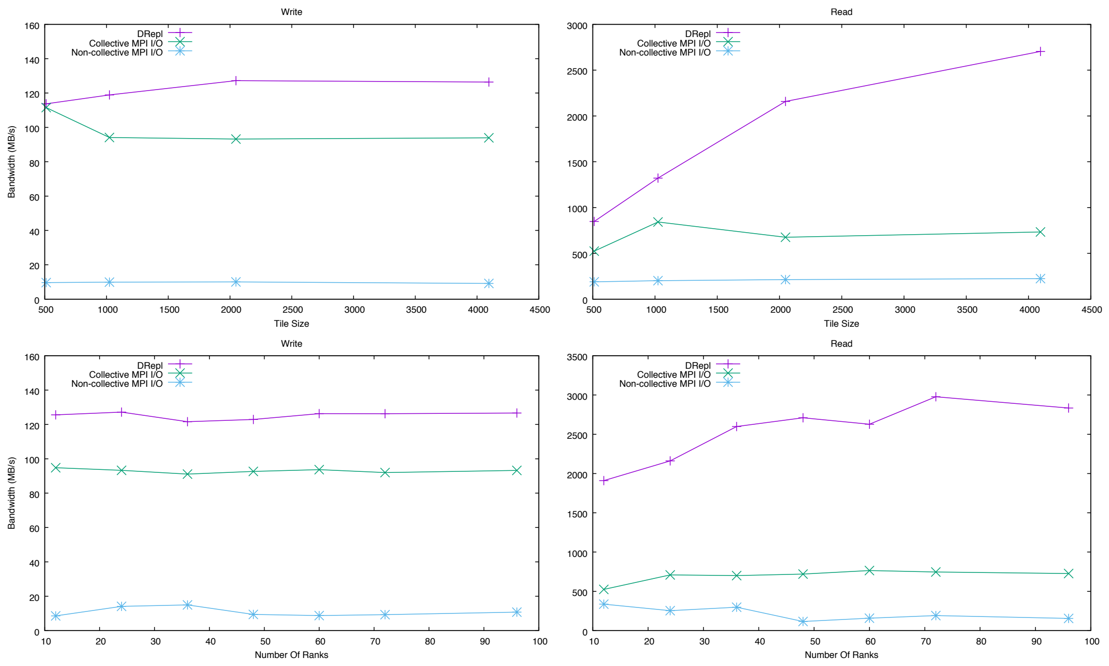


Figure 4.10: MPI Tile I/O performance

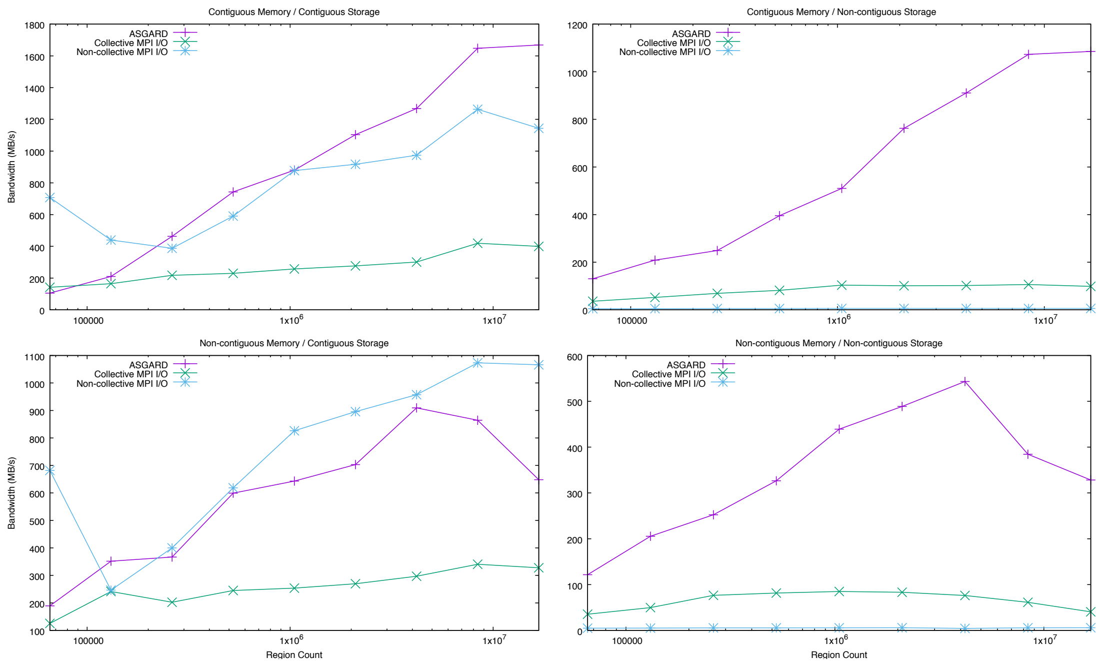


Figure 4.11: HPIO read performance

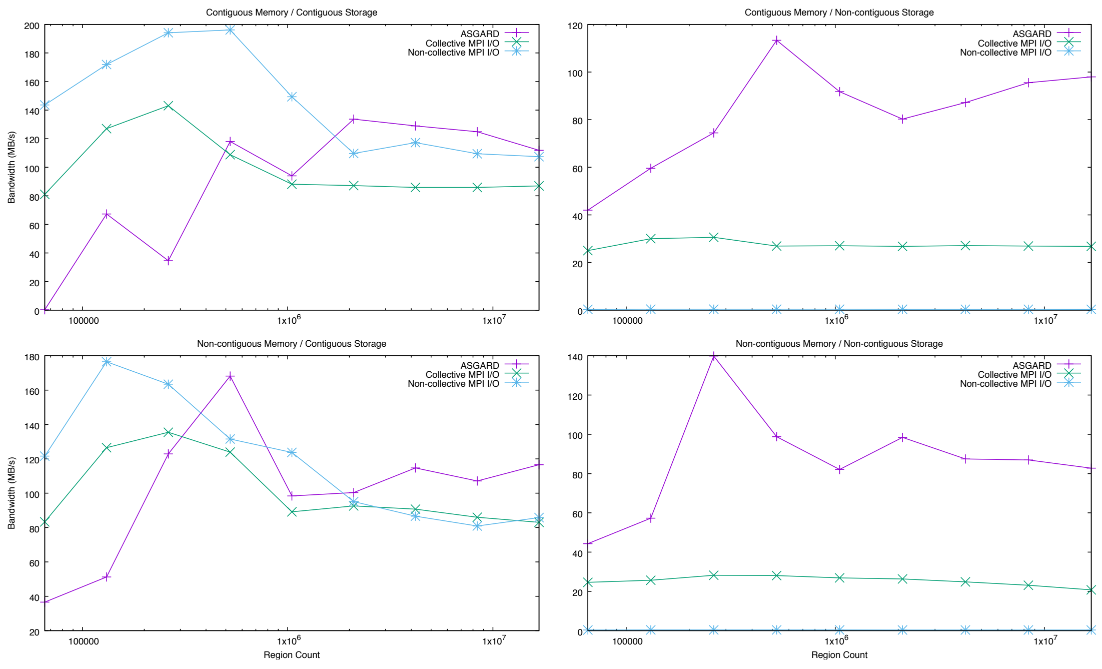
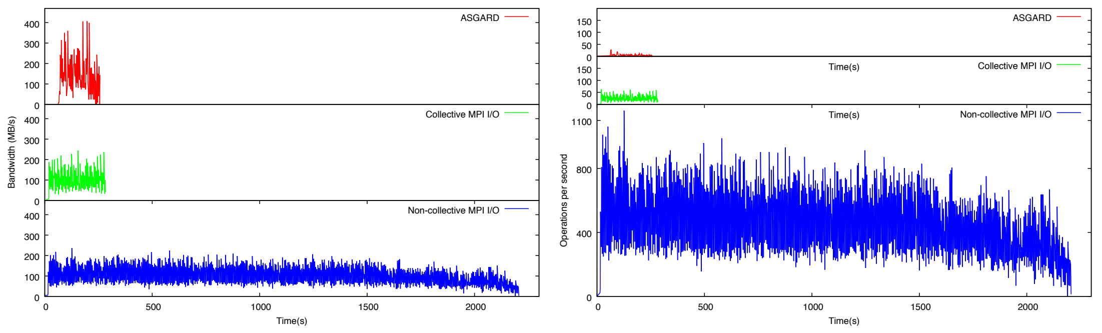


Figure 4.12: HPIO write performance



(a) Write bandwidth **(b)** Total operations per second

Figure 4.13: Ceph performance for the 96 rank Tile I/O experiment.

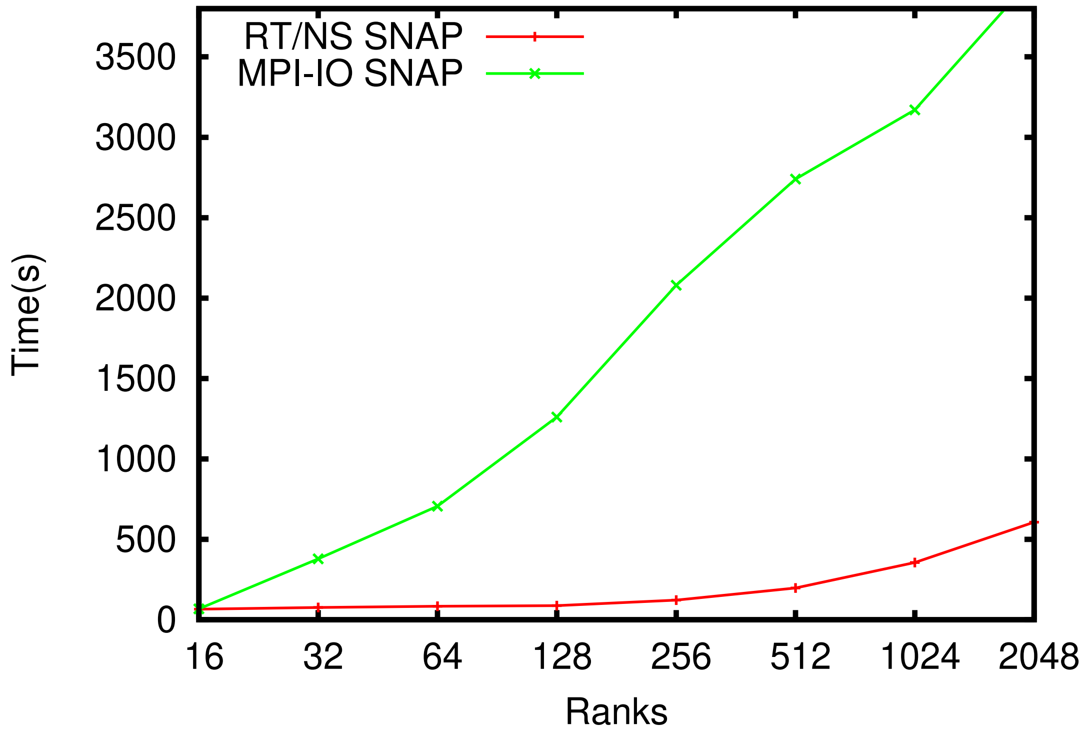


Figure 4.14: Comparison between UNITY's and MPI I/O performance for checkpointing the application data to disk.

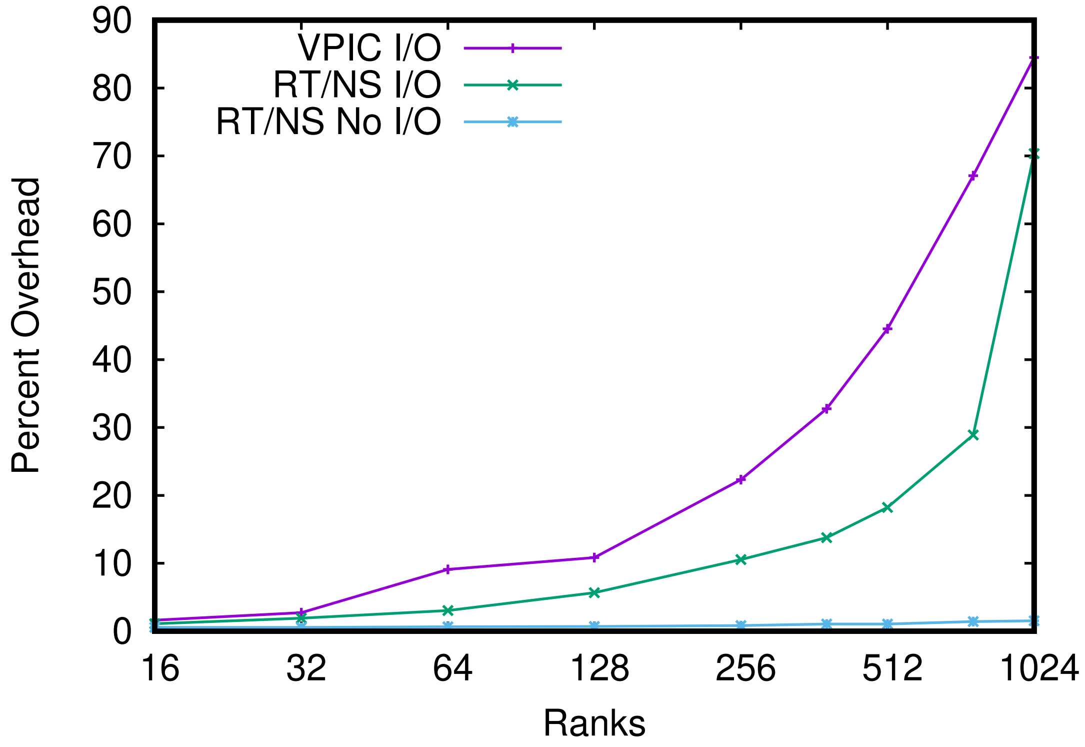


Figure 4.15: Unmodified VPIC comes with its own I/O functionality which it uses to write its output and checkpoints to persistent storage. We compare VPIC's native performance (VPIC I/O) against VPIC implemented to use UNITY with and without persistence activated (RT/NS No I/O and RT/NS I/O respectively)

Chapter 5

Future work: Thoughts on Multi-resolution Datasets and Unstructured Grids

DRepl and ASGARD provide advanced functionality for scientific workloads in HPC environment. Although they work well with many applications with fixed, or bounded datasets, the fact that the dataset is static restricts their use in a number of applications where structure of the data changes during the simulation execution. Examples of such behavior are the Adaptive Mesh Refinement algorithms, as well as some algorithms that periodically repartition unstructured meshes in order to balance the overall performance and precision of the simulation.

This chapter describes the future work on how to extend the DRepl language and the transformation engine to provide support for more dynamic datasets.

Section 5.1 outlines an approach where the structure of the dataset can still be defined with a declarative language. Extending the language to allow more abstract datasets and allowing the developers to defer the definition of the precision, format and size of the values to the fragments' descriptions can greatly extend the applicability of the approach. In addition to support for Adaptive Mesh Refinement techniques, this approach will allow integration in cases where the simulations use a mix of dense and sparse regions of data. Section 5.2 explores ideas on how to handle cases where the structure of the datasets changes so drastically that it can't be defined in advance with declarative languages.

5.1 Dynamic and Multi-resolution Datasets

Although DRepl allows application developers to define an abstract description of the application's data, its design is still too focused on how the data is placed in memory (RAM or storage). Although it allows flexibility on the endianness of the data, or the element order of the arrays, the overall structure of the dataset is still fixed. The variables and types have fixed size. If the developers define an element of array of type `float32`, they can't later decide to use `float64` and still be able to read the old data transparently.

The same deficiencies apply for data stored in arrays. As a rule[1] multidimensional arrays store data from multidimensional contiguous spaces. Once a space

[1]Legacy Fortran code sometimes uses extra dimensions to group properties that are better represented with structs in modern languages. For example, a 4-dimensional array can partition a 3D space as the first 3 dimensions, and have 'pressure', 'temperature', 'electromagnetic force' values as indices 1, 2, and 3 in the fourth dimension.

is tessellated into cells of certain size, it can't be re-partitioned into a grid with different resolution.

This section describes the design and the initial steps in implementing the extended functionality required to support dynamic and multi-resolution dataset. Further effort is needed to integrate it into a storage system, port existing applications, and evaluate its performance and applicability.

5.1.1 Data Model

We modified DRepl and ASGARD data model to extend support for more abstract types. Previously, all data types had to be concrete, i.e. the size that values of the type will take in memory needed to be defined. Although the new data model still requires all variables to be typed, the types don't necessarily need to be concrete. For example, a variable can be defined to hold real numbers (i.e. numbers from $\mathbb{R}$), and its presentation in the computer memory (i.e. whether it is in one of IEEE 754 formats, its size, etc.) can be finalized later.

Similarly, the tessellation of multidimensional spaces into grids can be postponed for a later, more concrete representation of the dataset. The data model allows abstract definition of a space with just the number of dimensions, or further limiting the dimension range, or completely defining the partitioning of the space, including the grid's cell size.

Along with the multiple extensions, there are some features from the previous work that were removed. Previously, the elements that belong to an array slice could be defined in a way that allowed greater flexibility (like “every third element”, etc.) and were formally defined by the equation 3.2. The experience with porting scientific codes to DRepl and ASGARD demonstrated that the feature was not useful in practice. Most of the applications use contiguous hyperslabs of the dataset arrays.

Abstract Types

Abstract types are closer to the mathematical sets used by scientists to define their models. They are detached from the way values are stored in computer memory.

The data model defines three primary abstract types: `integer`, `real`, and `string`. Values of the integer type are negative or positive whole numbers. Values of the real type are any real number. Strings is an array of characters in UTF-8 format.

There are two compound abstract types. A `struct` type is a collection of named *fields*, generally from different types. A `space` type is an N-dimensional continuum, where each point in it is associated with a value of its “element” type. Each dimension can be either open or periodic:

Open The dimension is unlimited, and any value between $-\infty$ and ∞. Each value is associated with separate element.

Periodic The dimension is unlimited, but periodic. Any value between $-\infty$ and ∞ is valid, but multiple values are associated with the same element. A periodic dimension has a *period* P defined. Values $x + nP$ for $n \in \mathbb{Z}$ are all associated with the same element.

Concretization of Abstract Types

Abstract types can be made more concrete by defining some of their properties. There are four properties currently defined for primary types:

Size Size of the value when stored on memory/disk. For `string` type, the number of characters depends on the format and is one, or two bytes less than the defined size.

Endianness Which byte of the value is stored first: most significant one (MSB), or the least significant one (LSB).

Alignment Alignment for a value of that type. When value is stored in memory or disk, the alignment defines the offset of the value. If needed, padding is prepend so the offset of the value matches the value of the property. For example, if a value needs to be aligned to 8 bytes, the offset has to be divisible by 8.

Format For the `string` type, the property defines if it is stored as a C-string (i.e. ending with the `NUL` character), or it has the string size as a 2-byte prefix. Other formats might be added later, including formats for the `real` type, if formats other than IEEE 754 are supported.

In order to concretize a space, it needs to be partitioned into cells. Each of the values of a space is associated with one of the cells. A grid is a subset of the whole space, split into cells of the same size. For each dimension of the grid, there are three properties defined:

Start The smallest value for the dimension. The start is a real value and has to be smaller or equal than the end.

End The largest value for the dimension. The end is a real number is has to be bigger or equal than the start.

Cell Size Size of the cell in that dimension. The size is a positive real number.

Currently the data model postulates that the cell includes the start, but excludes the end, i.e. cell is defined as $(B, E]$, where B is the start, E is the end, and S is the size.

The number of cells in the dimension can be calculated as

$$\frac{E - B}{S}$$

The properties defined for the whole grid are:

Dimension Order Order of which the dimensions are laid in memory.

Approximator Defines how to convert data across different grids of the same space. More about it in 5.1.1

If a type has only some of its properties defined, it is still considered an abstract type. If all properties for a type are defined, it is considered a *concrete* type. For example, an 32-bit integer type is an abstract type, while a 32-bit big-endian integer type is a concrete type.

Approximators

The data model allows definition of grids that represent the same space that have different resolutions, i.e. different cell sizes. In order to support converting data from one grid to another, there must be a way to split or coalesce values that are associated with multiple adjacent cells.

In the general case, a value for a cell might need to be constructed from data stored in cells in different resolution. For example, if a fragment defines a one-dimensional grid with cell size 5, converting the data into a grid with cell size 2 will need to split the value of the first cell into three cells (with the value for the third cell being constructed by both the current cell from the original grid, as well as the next one). The general definition of an approximator function is:

```
type Cell struct {
  size []real // original size of the cell (for each dimension)
  part []real // part of the cell being used
  val  Value  // the value that is associated with the cell
}


func approx(newsize []real, cell []Cell) (var Value, err error)
```

The function receives the size of the new cell, as well as a list of the cells that overlap with it. The `size` defines the size of each cell, while `part` describes the part of it that overlaps with the new cell. The function returns an approximation of the value that will be associated with the new cell. In the example above, the approximator function will be called with `newsize` value of 2. For the first two of the new cells, the list of source cells will have a single entry with `size` 5 and `part` 2, while for the third cell, there will be two entries, with the value for `part` of 1. The approximator can return an error to signal that the transformation between the data in the fragments is impossible.

The data model defines some standard approximators that can be used.

None If this approximator is set, the data will not be approximated. It can be used only for spaces that have exactly the same size of the cells.

Proportional The proportional approximator assumes that the value is equally spread across each of the original cells. It calculates the fraction of the value that belongs to the portion that belongs to the new cell. The value for the cell is the sum of all fractions, i.e.

$$V'_j = \prod_{i=0}^{dim} \frac{S_{ji}}{S^p_{ji}}$$

$$P' = \sum_{i=0}^{k} P_i V'_i$$

where S_{ji} is the size of the i-th dimension of the j-th cell, S^p_{ji} is the size of the part that overlaps with the new cell, and P_i is the value stored in the i-th cell.

The approximator is defined for values of the integer and real abstract types, and all their variants. If the type is integer, the value is rounded to the nearest whole number.

Minimum The minimum approximator returns the smallest of the values of the overlapping cells. The approximator is defined for values of the integer and real abstract types, and all their variants.

Maximum The maximum approximator returns the biggest of the values of the overlapping cells. The approximator is defined for values of the integer and real abstract types, and all their variants.

Truncate The truncate approximator is defined for values of the `string` type and all of its variants. If there is only one original cell, it truncates its value if necessary, in order to fit into the new type. Otherwise it sets the new value to an invalid string (if the type supports it), or an empty string (if it doesn't).

Not-a-Number For the `real` type, and its variants that support a `NaN` value, the not-a-number approximator sets the value to `NaN` to indicate that it is invalid.

For the `string` type and all its variants that support invalid string value, it sets it to that value.

The purpose of the approximator is to indicate that a value is not valid without preventing the new fragment from being materialized.

Error Always returns error if the sizes of the new and the original cells differ, or if there is more than one original cell that overlaps with the new cell. The approximator is defined for all types.

An approximator can be associated with any type defined in a fragment. The property is ignored for types that are not associated with grids. The default approximator for all primary types is Error. The default approximator for a struct type is to iteratively call the approximators of each of its fields. The default approximator of a space/grid is Error.

Datasets and Fragments

While the original DRepl defines three abstractions: dataset, view, and replicas, and ASGARD defines datasets and fragments, the new data model unifies all of them into a single abstraction of scientific data. A fragment is a collection of data constants, types and variables. Fragments can be organized into hierarchical relationships. When a fragment is defined, it can be based, or derived from another fragment. In that case, the derived fragment's types and variables need to be more concrete variants of the types and variables of the parent fragment. The special case of a fragment that doesn't derive from another fragment is known as a *dataset*.

A fragment that contains abstract types and variables is an *abstract fragment*, and one that contains only variables of concrete types is defined as an *concrete fragment*. Data can only be associated with concrete fragments. They are the only ones that can be instantiated in memory, or saved in persistent storage. The abstract fragments exist only as metadata of the dataset definition, and can be used to direct how data of concrete fragments is transformed.

A fragment is a list of *variables* with each of them an instance of a defined type.

5.1.2 Language

The DRepl language was modified to support the new data model. The dataset, view and replica sections are replaced by a fragment section that allows hierarchical definition of fragments. The type and variable definitions are streamlined to support the new hierarchical model.

Types

The fact that the types don't have to be completely specified introduces big changes from the type definition in DRepl. Each type can have certain list of properties attached to it. If the properties for the specific type (either primary or compound) are not fully specified, the type is not concrete.

There are 6 properties: byte order, size, alignment, format, dimension order, and approximator.

```
PropertyName = "byteorder" | "size" | "align" | "format" |
    "elo" | "approx"
Property = PropertyName { "(" Const ")" }
PropertyList = Property { "," Property }*
```

The byte order specifies which byte of the value appears first: `lsb` for least significant byte, or `msb` for most significant byte. It can be specified for primitive types. The default byte order is derived by the fragment's byte order (if defined).

The size property specifies size of the type in bytes. It can be specified for primitive types only. There is no default size.

The alignment specifies how values of the type should be aligned in memory or storage. It is specified in bytes. If necessary, padding is added before the memory location so its offset is divisible by the specified property value. The default alignment is 0.

The format property can be set for `string` types and defines the format of the string. If it is set to `C`, the end of the string is specified by a `NUL` (0) character. If it is set to `P`, the length of the string is specified by the first 2 bytes of the string, in least-significant-byte first byte order.

The elo property can be specified for `space` types and defines the order of the cells in memory. Currently there are two element orders defined: `rowmajor` and `columnmajor`.

The approx property defines how a value of a type that is associated to a cell of a space is converted to cell(s) of different size.

```
Type = { | PrimitiveType | StructType | SpaceType | NamedType}
   { "@" PropertyList } { "=" Type }
```

A type can be primitive, struct, space, or a named type. In some cases, when the type can be implied, the type name can be omitted.

integer	whole numbers
real	real numbers
string	variable sized string of characters in UTF-8 format
int8	1-byte signed integer
int16	2-byte signed integer
int32	4-byte signed integer
int64	8-byte signed integer
float32	single-precision IEEE 754 floating point number
float64	double-precision IEEE 754 floating point number
string$N$	variable size string of characters with length up to N bytes

Table 5.1: Extended list of primitive types

Primitive Types There are ten primitive types defined, as shown in Table 5.1. The first three types are abstract. Based on those, seven more primitive types are defined.

```
PrimitiveType = "integer" | "real" | "string" |
   "int8" | "int16" | "int32" | "int64" | "float32" |
   "float64" | "string"[0-9]*
```

Structs Structs are similar to the struct type in Go and C. Multiple fields of different data types can be arranged in a struct. Each field has a name and a type assigned to it.

```
StructType = "struct" "{" { FieldDecl ";" } "}"
FieldDecl = IdentifierList Type
IdentifierList = identifier { "," identifier }*
```

The example below defines a struct with three fields: a, b, and c:

```
struct {
   a        real
   b, c     float32
}
```

Spaces and Grids A space is a multi-dimensional continuum that has values of a specified type associated with it. If the space is fully, or partially partitioned into cells, it is also known as a grid.

```
ArrayType = "[" Dimension { "," Dimension }* "]" Type
Dimension = { Expression } { ":" Expression { ":" Expression } }
   { "@periodic(" Expression ")" }
```

The `Expression` in `Dimension` is a numeric constant (named or unnamed).

All four expressions associated with each dimension are optional. The first expression is the smallest allowed value. The second is the largest allowed value for that dimension. If the space is partitioned, the third is the size of the cell. Note, that it is allowed for a space type to be partitioned only for some of its dimensions. An optional property of the dimension specifies if it is periodic, and what its period is. It is invalid to have period smaller than the largest value (second parameter).

For example, a three-dimensional space of real numbers is defined as:

```
[,,] real
```

A space in polar coordinates can be defined as:

```
    [,0:360:@periodic(360)] real
```

There are two dimensions, the first is unbounded, while the second one has a range from 0 to 360 and a period of 360.

A concrete two-dimensional grid with cell size of 5.0 can be defined as:

```
    [0:100:5, 0:100:5] float64
```

The definition creates a 400 element array, with 20x20 cells.

Named Types

Similarly to Go (`type`) and C (`typedef`), the language allows association of names to custom types. The names allow usage of “shortcuts” when a type is referenced often.

```
TypeDecl = "type" identifier Type
NamedType = identifier
```

As in DRepl, the extended language allows named types to be referenced before they are defined, so there is no need for forward declaration mechanisms.

Type Concretization

A type can be defined as a concretization of another type. For primary types, concretization consists of defining previously undefined properties. It is an error to change a property that is already defined. For example, type concretization for the integer type is defining its size:

```
@size(64) = integer
```

For structs, in addition to defining new properties, concretization can also include selecting a subset of the base type fields.

```
struct {
    a, b, c integer
} = struct {
    a, b integer
}
```

The concretization can be run recursively for each of the fields. For example, the similarly to the example above, the fields a and b can be defined as concrete types instead of the abstract `integer`.

```
struct {
   a, b, c integer
} = struct {
   a, b int64
}
```

The type that is being concretized is called a *base type*. The relation between a type and its base is significantly different than the relations between a class and its base class commonly found in object oriented languages. While classes usually extend the functionality and contain more data than their base class, the types in the data model are always subsets of their base types.

Variables and Constants

A variable is a named instance of a type. It is associated with real data. If the type is concrete, a variable defined in a fragment takes space in memory when the fragment is materialized. The names of the variables should be unique.

```
VarDecl = "var" IdentifierList Type { "=" IdentifierList }

IdentifierList = identifier { "," identifier }*
```

When a variable is declared, it can be defined to contain subset of the data of a variable defined in a base fragment. In that case, the type of the new variable has to be concretized type of the type of the old variable. For example, if the current fragment is based on a fragment with variables `a` and `b`:

```
var a real

var b [,]real
```

variables `aa` and `bb` can be defined as:

```
var aa @size(64) = a

var bb [0:10:1,0:10:0.1] = b
```

Constants

Constants are named values that can't change. They don't use storage space and are valid only during the parsing stage.

```
ConstDecl = "const" identifier "=" Expression
```

Similarly to Go, DRepl constants don't have intrinsic type and are converted to the appropriate type when they are used.

```
const N = 1000
const M = 4.3
```

Fragment

A fragment is a collection of constants, types and variables. It can be based on another fragment definition, in which case it inherits all constants and types previously defined. It doesn't automatically derive the variables of the base fragment.

Fragments can be viewed as organized in a tree. The root of the tree, a fragment that doesn't have a parent, is known as the *dataset*. It usually consists of abstract types and variables based on them. Each of the other fragments in the tree provides increasing concretization of the types and the variables defined in the dataset. Variables can be freely defined only in the dataset. Variables defined in the other fragments have to be concretizations of the variables defined in the fragment's parent.

```
Fragment = "fragment" identifier { "@" FragmentPropertyList }
    { ":" identifier } "{"
    { TypeDecl | ConstDecl | VarDecl }* "}"


FragmentPropertyName = "byteorder" | "format" | "elo"
```

```
FragmentProperty = FragmentPropertyName { "(" Const ")" }

FragmentPropertyList = Property { "," Property }*
```

A subset of the types properties can be defined for the fragment as a whole. If they are, their values are default of all types defined in the fragment. The user can override these defaults in the type definition.

For example if the user creates a dataset, and two fragments based on it:

```
fragment dataset {
   type Point struct {
      x, y real
   }
   var data [,]Point
}
fragment f1 : dataset {
   var d [0:1:0.1, 1:10:1] = data
}
fragment f2 : dataset@byteorder(msb) {
   var d [0:100:1, -30:10:1] struct {
      x float64
   } = data
}
```

Fragment `f1` is not concrete and can't be materialized in memory. Although the space is concretized, field `x` and `y` are not of concrete type – they lack both byte order and size. Fragment `f2` is a concrete fragment, because all types used in its variables have defined sizes and byte orders.

More Complex Examples

The primitive types that are not completely abstract can be defined as:

```
type int8 @size(1),align(1) = integer

type int16 @size(2),align(2) = integer

type int32 @size(4),align(4) = integer

type int64 @size(8),align(8) = integer

type float32 @size(4),align(4) = real

type float64 @size(8),align(8) = real
```

The example below defines an abstract dataset, with two concrete fragments. All data in fragment `f1` is “least significant byte” first, the second in “most significant byte first”. The exception if variable n, which is in “least significant byte” order in both cases:

```
fragment dataset {

   var a, b, c, n int64

}

fragment f1@byteorder(lsb) : dataset {

   var fa, fb, fc, fn = a, b, c, n

}

fragment f2@byteorder(msb) : dataset {
   var fa, fb, fc = a, b, c
   var fn@byteorder(lsb) = a
}
```

Note that although the dataset fragment is mostly abstract, the variable sizes are already limited to 64-bit integers and can't be changed further. Similarly the dataset can limit the range of a space without setting the size of the cells:

```
fragment dataset {
   var a [-100:100]real
}
fragment f1 : dataset {
   var fa [::0.1]float64 = a
}
fragment f2 : dataset {
   var fa [0:20:1.3]float32 = a
}
```

Fields in a structure can have different approximators:

```
fragment dataset {
   type Point struct {
      x real@approx(proportional)
      y real@approx(nan)
   }
```

```
        var data []Point
    }
```

5.1.3 Implementation

The two main parts of the implementation are the language parser and the transformation language. Both of them are based on ASGARD's code, but required major changes and enhancements.

Language Parser

The introduction of the hierarchical fragments in place of the two-level dataset and fragment arrangement in DRepl and ASGARD resulted in major redesign of the code. The space definition streamlined the complex array and slice data structures and requires less code than the previously used view slice definitions. Type and fragment properties are new features that require careful handling when they are inherited by base types and fragments.

The parser keeps its own internal descriptions of the fragments. It generates transformation engine descriptions only for concrete fragments, or abstract fragments that are not based on other fragments.

Overall, the new parser is about 30% smaller than the original DRepl's parser.

Fragment Source Coverage

The source coverage uses the size of the types and can therefore be calculate intersections only for concrete fragments. The coverage for primitive and struct types is similar to the approach used in ASGARD. A major difference is taking into account the approximators when evaluating the coverage. An `Error` approximator, for example, makes the block unsuitable for consideration and its coverage is assumed to be zero. Similarly, if a type's approximator is set to `None`, but it is in grids of different sizes, the coverage is considered to be zero.

In ASGARD an element of an array corresponded to at most one element of an array of another fragment. The coverage between the two arrays could be easily calculated by counting the number of common elements, and dividing it by the number of elements in the destination arrays. The introduction of different grid cell sizes poses new challenges of the fragment source coverage. Counting common elements is no longer applicable. Instead, the runtime uses the volume of the cells for calculating its coverage. The volume is defined as a product of the cell's sizes in each dimension.

In ASGARD, spatially an element of an array is always either completely covered by an element from another array, or completely uncovered. The new system allows a cell to be partially covered by another cell. While in some cases partial coverage might be permissible, in many other cases the numerical algorithms will produce invalid results if they use values calculated on partial coverage.

There are three cases for a cell to be covered by data from other fragments, A cell of a space can be fully covered from a single fragment. They are considered *SF* (single-fragment/fully) covered. There are two ways for a cell to be partially covered. It can be partially covered by every single other fragment, but fully

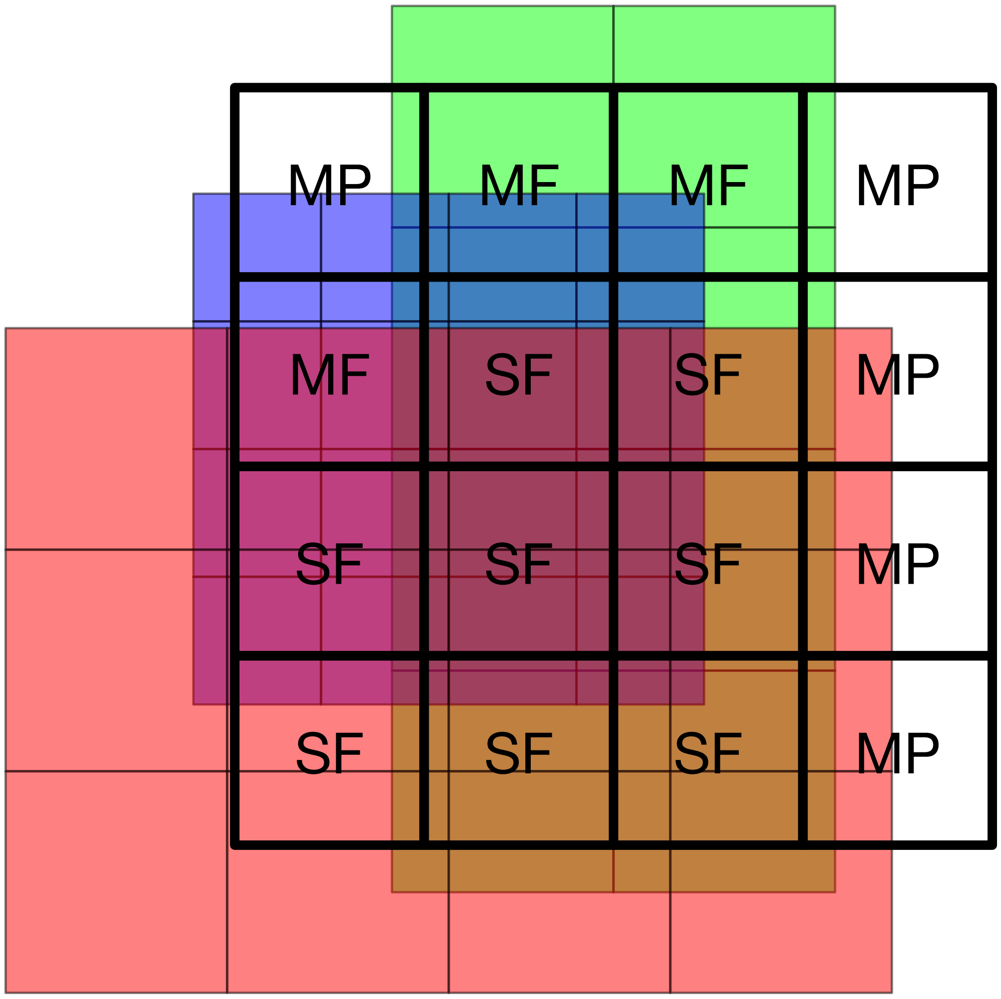


Figure 5.1: Example of cell coverage.

covered, when all the fragments are taken into account. In that case, the cell is *MF* (multiple-fragment/fully) covered. If the cell is partially covered even if all fragments are used, it is considered *MP* (multiple-fragment/partially) covered. Figure 5.1 show examples of cell coverage types.

There are multiple ways to try to alleviate the problem of MP cells. One is to extrapolate the value for the missing part. For the proportional approximator that means calculating the average for the rest of the cell (based on it volume), and using the calculated average for the uncovered volume. There doesn't seem to be similarly feasible approaches for other approximators, and even for the proportional one the solution might be incorrect. Therefore there is no perfect answer whether a MP cells should be considered covered at all. Currently the implementation uses the approximator property of the space to decide on how to return coverage for partially covered cells. If the space approximator is set to proportional, the coverage algorithm considers MP cells as part of the coverage for the whole space (depending on the volume of the cell covered). Otherwise the algorithm considers the cell uncovered.

Transformation Rules

In ASGARD the transformation rules for converting between two fragments were symmetrical and could be used in both directions. The introduction of variable-sized cells and approximators makes it impossible to do that in the general case.

The removal of the flexibility of definition of slices reduced the complexity of ABlock connectivity. The connections between equivalent ABlocks no longer require additional parameters for calculating the equivalent indices in the destination ABlock.

Transformation Engine

Previously, materializing a fragment from multiple sources could be done in completely parallel. Each region in the destination fragment memory was referenced by a single other fragment. Support for MF cells breaks that advantageous rule. The values for these cells need to be calculated once the data from all fragments covering it is available. This creates synchronization points that can affect the overall performance of the transformation.

Supporting variable-sized cells and approximators makes the general case for grid transformation more complex and potentially slower. The values from all source cells need to be collected, the sizes and parts of the cells calculated (as described in 5.1.1) before the approximator can be called. Although there is potential for optimizations (for example the sizes and parts don't matter for the `Maximum` approximator), the current implementation doesn't support any optimizations. They will be added in the future, once the overall design is confirmed and the performance in real applications is evaluated.

5.1.4 Discussion

The work described in this section allows major enhancements to the way scientific datasets are described and used. The scientists can tell the storage system not only what data structures they use in the simulation, but how they relate to the abstract concepts in the scientific models they simulate. It allows data to be produced and consumed at different resolution levels, making it easier to decouple visualization and analytics applications from the simulation. It also allows different parts of the simulation to use different resolutions. This allows support for applications that use block-structured grids and AMR techniques.

During the design phase there were considerations to allow even more flexible data definitions. One such enhancement would be to introduce abstract structs, similarly to abstract spaces. Instead of defining the list of fields, an abstract struct would allow dynamic addition of fields to it in sub-fragments. Although there are cases that this feature could be useful, it was ultimately rejected as too complicated to implement and support.

Although substantial, the list of supported approximators might not cover all possible use cases. Numerical scientists often take extreme care for minimizing the numerical errors, and our implementations might not meet their requirements. In the future, we plan to extend the implementation to support custom approximators. Their definition is going to be outside of the language for fragment description. Instead, we will allow the application developers to mark certain functions in the application's native code as approximators, and as a part of the build process will produce code suitable for shipment to the storage system to act as approximators. One of the approaches currently considered is to use the LLVM [46] framework to translate the native code into LLVM's intermediate representation (IR) or WebAssemply (WA) [37] code. The custom code will be registered with the storage system upon application's startup and can be later used as approximators in the fragment definitions.

Although the new design extends the dataset support and transformation considerably, there are still few areas where it falls short. For example, it doesn't support coordinate system conversion. If an application defines its spaces in spherical coordinates, the system doesn't allow the visualization to read it in cartesian coordinate system.

When fragment coverage is calculated, the current implementation also doesn't give preference to fragments that use the same primitive types' format and byte order. In future, the coverage information will be changed to include not only the percent of the fragment covered, but also the complexity of the transformation. That will give the runtime better knowledge on what alternative to choose if there are multiple fragments that cover the same region.

5.2 Unstructured Grids

The previously described approaches cover cases where dataset's description can be concisely depicted in a declarative language. There is an important class of scientific data, where the structure is so variable, that describing it with a language is difficult. Unstructured meshes provide a great challenge for separating the data values from the dataset description. Because of the variability of the multidimensional grid tessellation, the cells can have variable number of neighbors, making all approaches to introduce uniformity impossible. Another challenge is the fact that the tessellation is not static, or even predictably dynamic. Very often it depends on some of the values calculated during the simulation, and is driven by the goal to achieve a good balance between performance and correctness.

Very often, the grid decomposition information is intermixed with the cell's data. In one of the most popular approaches, the data for all cells are stored in a one-dimensional array (or list), with pointers to the cell's neighbors as part of the cell data.

This section describes some approaches for supporting unstructured data in storage systems. They vary based on the amount of metadata that need to be stored and the operations that the storage system can provide to the application developers.

The approaches still assume that the abstract dataset that the application uses is uniform and can be described in the terms defined by a declarative language. There are multi-dimensional spaces, real and integer values, and they all can be grouped together in structs.

All approaches separate the information on how the unstructured grids are partitioned from the data itself. The goal is for the storage system to be able to perform metadata operations (i.e. operations on the dataset as a whole) without going through all the data. The potentially smaller size of the metadata makes it easier to read and update, and it can be easily replicated for better performance in distributed environment.

5.2.1 Unstructured Grids as Graphs

Some numerical algorithms can be implemented based only on the topology of a grid. In this case, the grid can be represented as a graph. An important assumption for this class of data structures is that the metadata doesn't contain any spatial information, and the querying operations can't answer questions about coordinates in the grid's spaces. There are two popular representations grids that use graphs.

Graph of Cells

For the first, each cell is represented as a vertex, and an there are edges between each neighboring vertices. The metadata describing the grid consists of two lists: list of vertices (cells), and list of edges (cell pairs). Assuming that the edges only connect spatially close cells, the average amount of space required to store the metadata is proportional to the number of cells.

This format allows only one operation. It can return a list of neighboring cells up to a specified distance (in connectivity sense). Distance 1 is the most commonly used operation for retrieving cell's direct neighbors.

Graph of Vertices

The second representation has points as vertices, and cells defined as a list of vertices. It has more information and can be used to derive the first. Additionally, it can also be used to derive the planes and edges where neighboring cells touch. The metadata describing a grid requires two lists: list of vertices, and a list of cells (and each cell is a list of vertices). Depending on the number of vertices that describe a cell, the amount of storage space required to store the metadata is larger than the first approach, and is proportional to both the number of cells, and number of vertices.

This format allows more operations than the previous one. In addition to finding out the neighboring cells, it also allows to find how many vertices the cells share, and therefore the dimensionality of touching surface. For example, cells that share two vertices share an edge (2D shape), cells that share three or more vertices share a 3D surface, and so forth.

5.2.2 Unstructured Grids as Geometric Regions

Assuming that a cell has a convex shape in the N dimensional space, it can be defined by a list of $N-1$ dimensional hyperplanes. Each hyperplane is defined by $N+1$ values. The total number of values, specifying a cell is proportional to N^2. Therefore, the amount of storage space required to store the metadata is proportional to CN^2 where C is the number of cells.

This format allows operations that use coordinates in the space. For example, the user can find a list of cells, that are at a certain distance from a point in space. Or a list of cells that intersect with another cell. Calculating most of these operations precisely can be computationally expensive and can affect the performance of the storage system as a whole. A big advantage of this format is that the space can be partitioned into the operations don't require the complete metadata in order to perform spatially local queries.

5.2.3 Unstructured Grids and Voronoi Diagram

An important subclass of unstructured grids described as geometric regions are the Voronoi tessellations. Given a set of points P_j where $0 < j < n$ in the N

dimensional space, Voronoi's tessellation partitions the space into n cells, where all points in the j-th cell are closest to P_j than any other point.

$$C_j = \{x \in X \mid d(x, P_j) \leqslant d(x, P_k) \quad \forall k \neq j\}$$

The Voronoi tessellation is usually constructed by Delaunay triangulation algorithms [8, 47]. Although there are distributed algorithms [57] for calculating Voronoi diagrams, the algorithms are computationally intensive. A great advantage of this format is that it uses much less storage metadata space than the general description of geometric regions.

5.2.4 Incorporating Unstructured Grids

Although there are multiple advantages of representing the unstructured grid as a graph, there is a distinct challenge of constructing the global graph from fragments. Unlike the geometrical representations, the topological representations lack criteria how to stitch fragments into a whole. The applications that use topological unstructured grids usually have the initial construction outside of the application, or as a serial part where a single process creates the whole graph. Once the graph is created, modifying it is simpler operation that needs only global information only to ensure that newly created cells have unique IDs.

Within the ASGARD framework, the unique ID generation can be associated with the dataset object. As the dataset is abstract and doesn't contain details about space partitioning, it can still be described with a declarative language.

One approach for the grid construction is for a single process to create a fragment that contains the complete graph using the unique ID service. Once the grid is constructed, the distributed application that will use it will create its own fragments of it, and the global graph can be discarded if it takes too much space. Modifications of the grid include adding and removing cells. The operation can be performed on fragments of the grid, again by using the unique ID service.

Chapter 6

Conclusion

Productive storage systems for exascale demand drastic changes to the data organization and the interfaces they provide to the applications. In this dissertation, we explored new approaches of improving the storage system interfaces and performance by introducing more knowledge of the structure of scientific data to the storage subsystem. These techniques enable the storage system to provide optimal performance throughout the full life cycle of the data, by changing its on-disk format, as well as using multiple divergent replicas.

In addition to further exploring the challenges of storing and sharing unstructured grids, there is a number of other directions for improving the work presented here. The data description language still doesn't capture all the metadata for scientific datasets. For example, there is no way to express the coordinate systems used for multidimensional spaces. The lack of the information makes it harder to share data between multi-physics codes that use different coordinate systems.

The visualization software needs extra direction on how to show the data on the computer screen. The problem needs to be further researched and appropriate enhancements to be implemented. Another improvement that can be useful is allowing the application developers to define their own approximators to be used when data is converted from one resolution to another.

Bibliography

[1] Energy exascale earth system model. http://goo.gl/CDsV8w.

[2] Hpio benchmark.

[3] Introduction to the 9p protocol. *Plan 9 Programmer's Manual*, 3, 2000.

[4] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[5] Ieee standard for interval arithmetic. *IEEE Std 1788-2015*, pages 1–97, June 2015.

[6] Standard for information technology–portable operating system interface (posix(r)) base specifications, issue 7. *IEEE Std 1003.1, 2016 Edition (incorporates IEEE Std 1003.1-2008, IEEE Std 1003.1-2008/Cor 1-2013, and IEEE Std 1003.1-2008/Cor 2-2016)*, pages 1–3957, Sept 2016.

[7] James P. Ahrens, Jonathan Woodring, David E. DeMarle, John Patchett, and Mathew Maltrud. Interactive remote large-scale data visualization via prioritized multi-resolution streaming. In *Proceedings of the 2009 Workshop on Ultrascale Visualization*, UltraVis '09, pages 1–10, New York, NY, USA, 2009. ACM.

[8] Franz Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, 23(3):345–405, 1991.

[9] Parallel I/O Benchmarking Consortium. MPI Tile I/O Benchmark.

[10] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 21:1–21:12, New York, NY, USA, 2009. ACM.

[11] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, and D. J. Kerbyson. 0.374 Pflop/s Trillion-particle Kinetic Modeling of Laser Plasma

Interaction on Roadrunner. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 63:1–63:11, Piscataway, NJ, USA, 2008. IEEE Press.

[12] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas*, 15(5):055703, 2008.

[13] K J Bowers, B J Albright, L Yin, W Daughton, V Roytershteyn, B Bergen, and T J T Kwan. Advances in petascale kinetic plasma simulation with vpic and roadrunner. *Journal of Physics: Conference Series*, 180(1):012055, 2009.

[14] Peter J Braam et al. The lustre storage architecture. 2004.

[15] Paul G Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968. ACM, 2010.

[16] Joe B Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt. Scihadoop: array-based query processing in hadoop. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–11. IEEE, 2011.

[17] Vincent Cate and Thomas Gross. Combining the concepts of compression and caching for a two-level filesystem. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IV, pages 200–211, New York, NY, USA, 1991. ACM.

[18] Unidata Program Center. Network common data form.

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[20] Chao Chen and Yong Chen. Dynamic active storage for high performance i/o. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 379–388. IEEE, 2012.

[21] Avery Ching, Alok Choudhary, Kenin Coloma, Wei-keng Liao, Robert Ross, and William Gropp. Noncontiguous i/o accesses through mpi-io. In *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pages 104–111. IEEE, 2003.

[22] Avery Ching, Alok Choudhary, Wei-keng Liao, Rob Ross, and William Gropp. Noncontiguous i/o through pvfs. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 405–414. IEEE, 2002.

[23] Avery Ching, Alok Choudhary, Wei-keng Liao, Lee Ward, and Neil Pundit. Evaluating i/o characteristics and methods for storing structured scientific data. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[24] Kenin Coloma, Avery Ching, Alok Choudhary, Wei-keng Liao, Rob Ross, Rajeev Thakur, and Lee Ward. A new flexible mpi collective i/o implementation. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–10. IEEE, 2006.

[25] Mariano P. Consens, Kleoni Ioannidou, Jeff LeFevre, and Neoklis Polyzotis. Divergent physical design tuning for replicated databases. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 49–60, New York, NY, USA, 2012. ACM.

[26] Sergey Danilov. Ocean modeling on unstructured meshes. *Ocean Modelling*, 69:195–210, 2013.

[27] Jack J Dongarra, Steve W Otto, Marc Snir, and David Walker. An introduction to the mpi standard. *Communications of the ACM*, page 18, 1995.

[28] Roger Faulkner and Ron Gomes. The process file system and process model in unix system v. In *Proceedings of the USENIX Conference*, January 1991.

[29] G Gibson, B Welch, G Goodson, and P Corbett. Parallel nfs requirements and design considerations. *IETF Draft*, 2004.

[30] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. Semantic file systems. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 16–25, New York, NY, USA, 1991. ACM.

[31] Hugh Greenberg, John Bent, and Gary Grider. Mdhim: A parallel key/value framework for hpc. In *HotStorage*, 2015.

[32] Gary Grider. Exa-scale FSIO Can we get there? Can we afford to? In *7th IEEE International Workshop on Storage Network Architecture and Parallel I/Os*, 2011.

[33] Robert Griesemer, Rob Pike, Ken Thompson, et al. The go programming language. *The Go Programming Language*, 2010.

[34] The HDF Group. Hierarchical data format version 5, 2017.

[35] John Gustafson and Isaac Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, 4(2), 2017.

[36] John L Gustafson. *The End of Error: Unum Computing*. CRC Press, 2017.

[37] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 185–200, New York, NY, USA, 2017. ACM.

[38] David Hilbert. On the continuous mapping of a line on a surface part. *Mathematical Annals*, 38(3):459–460, 1891.

[39] Latchesar Ionkov. Package for implementing 9p servers and clients in Go.

[40] Latchesar Ionkov, Michael Lang, and Carlos Maltzahn. Drepl: optimizing access to application data for analysis and visualization. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–11. IEEE, 2013.

[41] Latchesar Ionkov, Michael Lang, and Douglas Otstott. uDRepl: Making Whole Out of Pieces. Technical report, Los Alamos National Laboratory, 08 2016.

[42] Latchesar Ionkov, Carlos Maltzahn, and Michael Lang. Optimized scatter/gather data operations for parallel storage. In *Proceedings of the 2Nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, PDSW-DISCS '17, pages 1–6, New York, NY, USA, 2017. ACM.

[43] Terry Jones, Michael J. Brim, Geoffroy Vallee, Benjamin Mayer, Aaron Welch, Tonglin Li, Michael Lang, Latchesar Ionkov, Douglas Otstott, Ada Gavrilovska, Greg Eisenhauer, Thaleia Doudali, and Pradeep Fernando. Unity: Unified memory and file space. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*, ROSS '17, pages 6:1–6:8, New York, NY, USA, 2017. ACM.

[44] Martin Kersten, Ying Zhang, Milena Ivanova, and Niels Nes. Sciql, a query language for science applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 1–12. ACM, 2011.

[45] D. E. Keyes, Lois Curfman McInnes, C. Woodward, W. D. Gropp, E. Myra, and M. Pernice. Multiphysics simulations: Challenges and opportunities. 10/2012 2012.

[46] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International*

Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[47] Der-Tsai Lee and Bruce J Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980.

[48] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 39–39. IEEE, 2003.

[49] KT Lim, D Maier, J Becla, M Kersten, Y Zhang, and M Stonebraker. Array ql syntax, 2013.

[50] Vladimir D Liseikin. *Grid generation methods*, volume 1. Springer, 1999.

[51] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012.

[52] S. Lloyd and M. Gokhale. In-memory data rearrangement for irregular, data-intensive computing. *Computer*, 48(8):18–25, Aug 2015.

[53] Jay Lofstead, Ivo Jimenez, Carlos Maltzahn, Quincey Koziol, John Bent, and Eric Barton. Daos and friends: a proposal for an exascale storage system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 50. IEEE Press, 2016.

[54] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24. ACM, 2008.

[55] Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing.* International Business Machines Company New York, 1966.

[56] Bill Nowicki. Nfs: Network file system protocol specification. Technical report, 1989.

[57] Yurai Núnez-Rodrıguez, Henry Xiao, Kamrul Islam, and Waleed Alsalih. A distributed algorithm for computing voronoi diagram in the unit disk graph model. In *Proc. 20th Canadian Conference in Computational Geometry (CCCG'08)*, pages 199–202, 2008.

[58] Douglas Otstott, Sean Williams, Latchesar Ionkov, Ming Zhao, and Michael Lang. A foundation for automated placement of data. submitted for publication, 2017.

[59] Swapnil Patil and Garth A Gibson. Scale and concurrency of giga+: File system directories with millions of files. In *FAST*, volume 11, pages 13–13, 2011.

[60] Dimitri Pertin, Sylvain David, Pierre Evenou, Benoît Parrein, and Nicolas Normand. Distributed file system based on erasure coding for i/o-intensive applications. In *CLOSER*, pages 451–456, 2014.

[61] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. 10(3):2–11, Autumn 1990.

[62] Russell K Rew and Glenn P Davis. The unidata netcdf: Software for scientific data access. In *Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*, pages 33–40, 1990.

[63] Robert B Ross, Rajeev Thakur, et al. Pvfs: A parallel file system for linux clusters. In *Proceedings of the 4th annual Linux showcase and conference*, pages 391–430, 2000.

[64] Frank B Schmuck and Roger L Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, 2002.

[65] Michael A Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. Malacology: A programmable storage system. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 175–190. ACM, 2017.

[66] Seung Woo Son, Samuel Lang, Philip Carns, Robert Ross, Rajeev Thakur, Berkin Ozisikyilmaz, Prabhat Kumar, Wei-Keng Liao, and Alok Choudhary. Enabling active storage on parallel i/o software stacks. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–12. IEEE, 2010.

[67] Huaiming Song, Hui Jin, Jun He, Xian-He Sun, and R. Thakur. A server-level adaptive data layout strategy for parallel file systems. In *Parallel and*

Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International, pages 2095 –2103, may 2012.

[68] PVFS2 Developer Team. Parallel virtual file system, version 2, 2003.

[69] Rajeev Thakur, William Gropp, and Ewing Lusk. A case for using MPI's derived datatypes to improve I/O performance. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–10, Washington, DC, USA, 1998. IEEE Computer Society.

[70] Yuan Tian, Scott Klasky, Hasan Abbasi, Jay Lofstead, and Ray Grout. Edo: Improving read performance for scientific applications through elastic data organization. In *Proceedings of the IEEE International Conference on Cluster Computing*, Cluster '11. IEEE, 2011.

[71] Noah Watkins, Michael A Sevilla, Ivo Jimenez, Kathryn Dahlgren, Peter Alvaro, Shel Finkelstein, and Carlos Maltzahn. Declstore: Layering is for the faint of heart. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association, 2017.

[72] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[73] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *FAST*, volume 8, pages 1–17, 2008.

[74] Chao Yin, Jianzong Wang, Changsheng Xie, Jiguang Wan, Changlin Long, and Wenjuan Bi. Robot: An efficient model for big data storage systems based on erasure coding. In *Big Data, 2013 IEEE International Conference on*, pages 163–168. IEEE, 2013.

[75] Joe Zerr and Randal Baker. Snap: Sn (discrete ordinates) application proxy - proxy description, 2013.

[76] Qing Zheng, Kai Ren, Garth Gibson, Bradley W Settlemyer, and Gary Grider. Deltafs: Exascale file systems scale better without dedicated servers. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 1–6. ACM, 2015.