Bredala: Semantic Data Redistribution for In Situ Applications

Matthieu Dreher, Tom Peterka

Bredala: Semantic Data Redistribution for In Situ Applications

Matthieu Dreher
Argonne National Laboratory
9700 S. Cass Ave.
Lemont IL 60439 USA
Email: mdreher@anl.gov

Tom Peterka
Argonne National Laboratory
9700 S. Cass Ave.
Lemont IL 60439 USA
Email: tpeterka@mcs.anl.gov

***Abstract*—In situ processing is a promising solution to the problem of imbalance between computational capabilities and I/O bandwidth in current and future supercomputers. Initially designed for staging I/O, in situ middleware now can support a wide range of domains such as visualization, machine learning, filtering, and feature tracking. Doing so requires in situ middleware to manage complex heterogeneous codes using different data structures. Data need to be transformed and reorganized along the data path to fit the analysis needs. However, redistributing complex data structures is difficult. In many cases, arbitrarily splitting the arrays of a data structure destroys the semantic integrity of the data. We present Bredala, a lightweight library to annotate a data model with enough information to preserve the semantic integrity of the data during a redistribution. Bredala allows developers to describe how to split and merge a data model safely, operations usually done by in situ middleware. We evaluate the cost and performance of our library in a molecular dynamics application. We show that our data model can simplify the workflow graph of large-scale applications, improve the reusability of tasks, and offer an efficient alternative to redistribute the data.**

I. Introduction

In each new generation of top 500 supercomputers, the gap between compute capabilities and I/O bandwidth grows. Extrapolating this trend to exascale, researchers estimate that less than 1% of data produced by simulation will be stored to disk [25]. At this point, data analysis done in postprocessing mode will not be practical anymore because of a significant loss of data and poor I/O bandwidth. Moreover, processing this amount of data may require a similar amount of computational power as was used to generate it.

In situ data analysis is a promising solution to this problem. Treating data while they are still in memory avoids the I/O bottleneck, uses the supercomputer to perform analysis, and shortens the overall time to discovery.

Initially designed for staging I/O, in situ middleware manages workflows that merge simulations and analysis. For example cosmology simulations produce a large set of particles [27] that can be tessellated in order to compute a density estimation of the initial particles. At each of these steps, the original input data are transformed and possibly redistributed across different resources. That is the case when a producer hosted on N processes sends data to a consumer hosted on M processes. Scientists expect that these data transformations will become more common as the workflows become more complex.

Several in situ middleware infrastructures have been proposed. Some come from the I/O community [22], others from the visualization community [7], [6]. Simulation data generally are composed of large arrays representing a set of points or cells with their associated properties such as speed or temperature. I/O interfaces such as NetCDF [4] or HDF5 [30] are usually well suited to represent these data. The redistribution of such data can be done by simply splitting the arrays in equal parts.

However, the data structures used in analysis can be more diverse than those of simulations. Graphs, meshes, trees, or structures with multiple levels of indirection are frequent outside the simulation world. Such structures raise two problems for in situ middleware: (1) It can be difficult to express these structures with current data interfaces inherited from traditional I/O interfaces, and (2) Redistributing such data models without breaking the semantics of the data is specific to each data structure.

Some systems allow transforming the data along the in situ pipeline [10], [5] as part of their communication functionalities. For most in situ middleware, the redistribution is part of the system itself. Therefore, redistributing specific data structures that require complex data manipulation can necessitate the developer to modify the in situ middleware itself for each specific data structure. In the end, it becomes difficult for in situ middleware to redistribute all the possible data models.

We present Bredala[1], a lightweight library designed to build a data model[2] with enough information to preserve the semantics of the data during a redistribution. The key design decision of our library is to include split and merge operations inside the data model. For each field of a data structure, the developer can specify how to split a field into chunks or merge two or more chunks of the same data field. Setting the split/merge rules at the data model level instead of by the underlying in situ middleware provides more flexibility for the user to define complex data models. It also allows the same model to be used in different in situ middleware infrastructures. Bredala also provides redistribution components for common strategies such as round-robin or contiguous strategies that can be used

[1]The traditional cookie of the lead author's home and a favorite of both authors

[2]Abstraction of a data structure

on top of existing in situ middleware or in the middleware itself.

We evaluate Bredala inside FlowVR, middleware designed to create asynchronous in situ workflows. We improved a previous analysis algorithm called Quicksurf, a distributed isosurface algorithm designed for molecular surface visualization. In particular, we show that Bredala allows simplifying the workflow graph of the FlowVR especially at large scale, automates the redistribution of the data, gains performance during the redistribution, and improves the reusability of the workflow tasks.

The reminder of this article is organized as follows. Section II reviews related work. Section III showcases some data structures that are difficult to redistribute with current in situ systems. Section IV describes our data model, and Section V describes our redistribution components. Section VI describes Quicksurf and discusses our implementations. Section VII evaluates the performance and usability of our library. Section VIII summerizes our work and briefly describes future work.

II. Related Work

We briefly survey common interfaces to describe data and transform them in common in situ middleware and I/O systems.

One of the first examples of in situ middleware is visualization. VisIt [6] and Paraview [7] are widely adopted tools to visualize and analyze datasets. Both contain an in situ library, respectively libsim [32] and Catalyst [18], to connect simulations to their visualization pipeline. In both cases, the library serves as an adapter to convert the data format of the simulation to the VTK format [29] at the basis of both tools.

ADIOS [22] is a common interface for several in situ infrastructures [34], [35], [5], [11]. Data are described in a separate XML file with a name, a type, and a layout. Data can be organized by groups with a transport method for each group. The transport method can be an I/O method such as MPI-IO or an in situ system. Some of these systems enable communications from N to M processes. Several systems [33], [5], [10], [23] also allow transforming data along the I/O path.

DataSpaces [12] provides a distributed shared-memory space to act as a staging area between applications. Producers push data into the shared space. The data are then indexed, and the system determines where to store them. Data can be organized following a space-filling curve, but other policies are also possible. Clients make a request for a subpart of the global data. Tong and al. [20] build on top of DataSpaces a framework staging data in memory and on disk.

Legion [8] is a data-centric programming model based on logical regions, typed collections that can be further partitioned recursively. The developer also can map where the data are placed in memory. Bauer et al. [9] introduce the notion of a field in the logical region data model that allows slicing fields from the data structure.

Damaris [13] allocates dedicated cores or nodes to perform I/O or analytics. Data are described similar to ADIOS with a separate XML file. Data are gathered on the dedicated resources; however, the user merges the data. Damaris/Viz [14] extends the XML to describe meshes and automatically transfers the meshes to Visit and Paraview for visualization.

```
struct t_state {
    int nbParticles;
    float* pos;      // nbParticles * 3
    float* vel;      // nbParticles * 3
    int* globalId;   // nbParticles
};
```

Fig. 1: Subset of the data structure of Gromacs. The structure contains the number of particles, the position, velocity, and the global ID of each particle.

FlowVR [15] creates a graph between parallel applications. FlowVR does not provide any support for data typing; the developer writes data into untyped buffers. Special components called filters can be used to modify and route the data between tasks.

FFS [16] is a serialization library for HPC applications. An FFS data type is closely linked to a C structure. Similar to an MPI_Datatype, each field of the data structure is described in a map with a name, a size, a type, and a memory offset in the data structure. Additionally, the data types can be imported/exported to XML to allow more flexibility. EVPath [17] is designed to be an event transport middleware layer. It creates a network of modules to compute, transform, and route the data. EVPath uses FFS to manage the data and enables data transformation at runtime.

HDF5 [30] and NetCDF [4], as well as their parallel interfaces [31], [3], are two common data formats for scientific applications. Both formats are based on multidimensional arrays. Data are described in a hierarchical manner with the declaration of groups and subgroups. The user can read and write data with a path representing the placement of the data set in the hierarchy. The interfaces can be specialized for specific application domains. For example, H5hut [19] is designed to facilitate the manipulation of particle data sets.

III. Examples of problematic data structures

We describe in this section two different data structures from simulation and analysis. We show in particular why traditional array-splitting methods do not preserve the semantics of the data during a redistribution operation. In both cases, the goal is to redistribute the data from N to M processes following a block redistribution. We assume that N processes have data within a global spatial domain. The redistribution splits the global domain into M subdomains (or blocks) and sends data to their corresponding subdomain. After the redistribution, each of the M processes has a subset (or chunk) of the initial particles.

A. Particles structure in a molecular dynamics simulation

Figure 1 shows a subset of the global data structure from Gromacs, a well-known parallel molecular dynamics simulation package [28].

In this structure, a particle is composed of three floating-point values for the particle position, three floating-point values for the velocity, and one integer for the particle ID. A valid redistribution preserving the semantics requires the

```
struct tet_t {
  int verts[4];  /*indices of the vertices*/
  int tets[4];   /*indices of the neighbors*/
};
```

```
struct tess_t{
  int num_particles;
  float* particles;
  int num_tets;
  struct tet_t* tets;
};
```

Fig. 2: Subset of the data structure of Tess. The structure contains the number of particles, the positions, the number of tetrahedras generated by the tessellation, and the array of tetrahedra

following guarantees: (1) the position or velocity array can be split only every three floating-point values; (2) the association between individual positions, velocities, and particle IDs must be preserved; and (3) `nbParticles` must represent the current number of particles in the structure. Four steps are required to split this structure properly:

- Identify the field `pos` as the field representing the particle position in space.
- Split the atom position so that particles of the same block are in the same chunk.
- Apply the same partitioning as the atom position on the fields `vel` and `globalId`.
- For each chunk update the field `nbParticles` with the number of particles present in the chunk.

These four operations cannot be done by simply spliting each array independently at the granularity of its elements. Splitting requires semantic information and coordinated manipulations on the arrays. By following the above procedure, however, splitting the array in N chunks does actually preserve the semantics because it does not split within a position or velocity and it does not separate atom positions from their respective velocities and global ID.

B. Tetrahedral mesh in a Delaunay tesselation

Another example is a data structure extracted from a tessellation analysis code [27] (Figure 2).

In this example, `tets` contains offsets to the array `particles` and neighboring tetrahedra. Therefore, simply splitting all arrays into M chunks breaks the semantics of the data structure because the offsets of the tetrehedra become invalid.

Depending on the application, we can redistribute by tetrahedra or by particles. The splitting key is the field that is first redistributed, and the rest of the fields follow the split of the key. Considering the particles as the splitting key, a proper redistribution requires three steps:

- Split the particles in M chunks according to their position.
- For each tetrahedron, assign it to a subdomain only if its four vertices are assigned to the same subdomain, and adjust the offsets of the vertices.
- In a second pass over the tetrahedra, adjust the offsets of the neighboring tetrahedra.

No generic in situ middleware can properly redistribute this data structure. The developer needs an interface to describe to the middleware how to split and merge this structure properly.

IV. Bredala Data model description

We describe our library, Bredala, for data redistribution. It allows the programmer to describe complex data structures with enough information to guarantee the semantic integrity of the data when it is redistributed.

We chose to integrate splitting and merging functions inside the data model instead of an underlying in situ infrastructure. The library provides common splitting and merging strategies and can be extended by the user to fit specific needs.

A. Definition of a semantic item

Bredala's goal is to protect data model semantics during a split or merge. Data structures are often a form of container for smaller pieces of data called semantic items. We define a semantic item as the smallest subset of data that contains all the fields of the original data structure and still preserves its semantics. For instance the structure in Figure 1 is a collection of particles. A semantic item in this structure is a particle. It is the combination of the number of particles (equal to one), three floating-point values for the position, three for the velocity, and one integer for the particle ID. Bredala provides a safe way to access, extract, and merge semantic items within a data model. Note that in some cases a semantic item in a data structure can be interpreted differently depending on the application. In Figure 2, both particles and tetrahedra are valid semantic items.

B. Building a data model

We define a data model as a collection of fields, where each field has a name, a data type, a layout, and split/merge policy. A field is the association of a storage container and an elementary data type. Currently, a storage container can be a singleton (only one object allowed), an array (1D, 2D, or 3D), or an STL vector. An elementary data type is a type that will not be divided during a split. It can be a basic type such as an integer or a complete data structure. We create each field with enough information to identify the subset of data for each semantic item. For instance, in Figure 1 a position is composed of three `floats` that form one indivisible element for this field. The split and merge policies will be described in the next section.

We build a data model for a particular data structure in two steps. First, similar to MPI data types or FFS [16], we describe each field of the data structure (see Figure 3). However, we use a higher-level interface that simplifies the description of the data structure compared with MPI data types. For instance, we do not need to indicate the offset address of a field in a C data structure.

Bredala provides classes for simple types (`float`, `double`, `int`, etc.); structures; and 1D, 2D, and 3D arrays. Future versions of Bredala will also provide support for recursive data types.

Once all the fields are declared, the developer inserts them into a handler `ConstructData` that represents the data model. When inserting a field, the user provides (among others) a name for the field, a semantic flag, and a split and merge policy. The handler can be seen as a map of fields with the field name as key and with additional information per field.

```
struct t_state state;

// Generate the fields
shared_ptr<ArrayConstructData<float>> pos =
                make_shared<ArrayConstructData<float>>(
                state->pos, state->nbParticles * 3, 3, false);
// Similar for remaining fields

// Create the container and push the fields
ConstructData container;

container.appendData("pos", pos, ZCURVEKEY, PRIVATE, SPLIT_DEFAULT,
        MERGE_APPEND_VALUES);
// Similar for remaning fields
```

Fig. 3: Code snapshot to build the data model corresponding to the data structure of Figure 1. First, an object is created for each field. Then the objects are pushed in the container representing the data model. Each field is associated with a name, optionally a semantic flag (ZCURVEKEY for instance), and a splitting/merging policy.

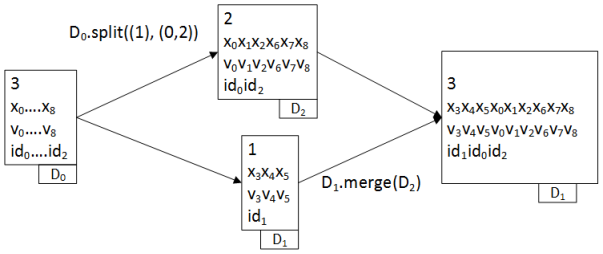


Fig. 4: Initial data model D_0 containing three particles. We split D_0 in two sub data models D_1 and D_2. D_1 contains the particle 1 and D_2 contains the particles 0 and 2. D_0 remains unmodified. We then merge in place D_2 into D_1, but the destination could be another data model as well.

The semantic flag allows the user to indicate that a field has a particular meaning in the data model that can be used during a split/merge. We will show an example in Section V-C. We describe the split and merge policies in the next section.

Since our data model is not tightly linked to a C/C++ structure as with MPI datatypes, one can construct a model incrementally at runtime. The developer can, for instance, merge several C structures into one data model. Moreover, the map-based data model enables a form of introspection. When reading a data model, the developer does not have to know its exact structure, only the names of the needed fields and their data types. The map-based representation also enables slicing: the user can simply remove a field by its name. The user receives a pointer to the removed data field, which can be inserted into another data model or destroyed. The user can insert the same field into multiple data models. In that case, the data models have access to the same data. Bredala manages the fields with shared pointers. A field will be destroyed only if there are no more references to the field. This approach allows the memory footprint to remain as low as possible.

C. Splitting and merging the data model

The handler provides an interface to split and merge the data model safely. The data model can be split by range of items, list of individual items, or spatially. Splitting the data model results in N subdata models. A subdata model has the same fields as the original data model, but each field has only a subset of data from its original field. Merging N identical data models, that is, models with the same field names, aggregates the data from each field.

Splitting and merging are performed field by field by using the selected policies. We implemented several common split and merge policies for each collection of elementary data types. For instance, in Figure 3 the policy SPLIT_DEFAULT, used for the field `pos`, will divide the array of positions while guaranteeing that three consecutive floating-point values will never be separated. The policy MERGE_APPEND_VALUES will concatenate the two arrays of position. Figure 4 gives an example of split and merge of a set of particles.

D. Extending the data model

Developers might want to extend our data model to (a) add or modify a split or merge policy to an existing elementary data type or (b) to support a new elementary data type.

Adding a split or merge policy to an existing elementary data type simply requires adding the new constant name and the split or merge code into the appropriate elementary data type class.

Modifying an existing split or merge policy is done by extending the elementary data type class and overloading the split or merge function. This can be desired when creating specific data types with particular behaviors. For example, Bredala has a bounding box data type built on top of the array data type. Both the split and merge functions are overloaded to modify the behavior of the default array split and merge policies.

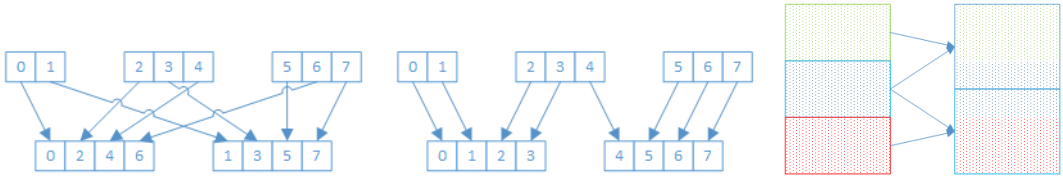
The developer can add the support for new data types in two ways. The first is to extend an existing elementary data type as we did to create the bounding box type; this requires little effort. The second is to extend the abstract base class used for every elementary data type. We deliberately kept its interface simple, with only five functions to overload, plus the constructor.

E. Serialization

We built Bredala as a bridge between parallel codes involving network communication. The handler and all the elementary data types implement a `serialize()` and `deserialize()` function. Our current implementation uses Boost [2] to serialize the data model. This choice allows Bredala to not rely on any communication library such as MPI for its serialization and facilitates its integration into a wide range of infrastructures.

V. Data redistribution components

The core of the problem addressed by Bredala is to safely distribute a data model from N to M processes without breaking the semantics of the data model. Bredala provides redistribution components for three common strategies: round-robin, contiguous, and block (Figure 5). The round-robin component distributes the data item by item in a cyclic manner. The contiguous component redistributes all the items while preserving their order. The block component divides an n-dimensional global domain into subdomains that are contiguous in each dimension and assigns each item to a particular subdomain.



(a) (b) (c)

Fig. 5: Redistribution components available in Bredala. (a) The round-robin component distributes the data item by item in a cyclic manner. (b) The contiguous component redistributes all the items while preserving their order. (c) The block component divides a global domain into subdomains and attributes each item to a particular subdomain.

A. General structure

The redistribution components have three main functions: assigning a destination process for each item, transporting the split data models from N to M processes, and merging the received data models on the M processes. Each component is composed of sources and receivers of data. Sources and receivers can be the same processes or can be separated.

We adopted an algorithm similar to the one used by FlexIO [35] to redistribute the data. Each component implements four functions: `computeGlobal()`, `split()`, `redistribute()`, and `merge()`. The `computeGlobal()` function computes the global parameters necessary to determine how to split the data. It is executed on the source side only. The `split()` function assigns a destination for each item, splits the local data model into subdata models, and serializes them. The `redistribute()` function transports the serialized data models. The transport is done in five steps:

1) Each source stores in an array the ranks of all the destinations it has to send a message.
2) The arrays of all the sources are gathered at rank 0 of the sources.
3) The aggregated array is sent to the root of the destinations.
4) The array is scattered among all the destination. At this point, all the destinations know how many messages they will receive.
5) The data messages are exchanged point to point.

The components are implemented with MPI, but they could be implemented with any communication library. The only collective communications occur within the sources (MPI_Gather, step 2) and within the destination (MPI_Scatter, step 4) but not between the sources and destinations. We also have a point-to-point implementation available that requires no collectives. By default, steps 1 to 4 are performed at each iteration. The developer can skip these steps on demand if the data distribution has not changed between two consecutive iterations.

The `merge()` function aggregates all the received subdata models into one data model. It is called every time a destination receives a subdata model. It has two implementations. The first one directly deserializes and merges the received data model with the accumulated data model. The second implementation deserializes the received data model and stores it. When all the subdata models are received, they are merged all at once. The first implementation has a lower memory footprint, but it can require multiple memory reallocations when a new data model is received. On the other hand, merging all the sub data models at once requires one memory reallocation. Therefore, the second implementation runs faster at the cost of a higher memory footprint. The user must select the desired implementation.

B. Round-robin and contiguous redistribution components

The round-robin component does not require any global computations. Each semantic item is distributed according to its rank in the local data model.

The contiguous component requires two global pieces of information to perform the redistribution: the total number of semantic items and the global rank of the first local semantic item, that is, the rank of the semantic item when considering all the semantic items in all the sources. The component then computes the destination for each semantic item.

C. Block redistribution component

The block component redistributes items into subdomains with optional ghost regions. The component assumes that all the data on the sources belong to a global spatial domain. The component then assigns to each receiver a subdomain of the global domain and sends each item to their corresponding subdomains.

The basic structure to describe a domain in Bredala is a block. A block can be seen as a hexahedral region in a 3D grid. A block contains three pieces of information.

- *Global* grid, that is, the size of the grid of the complete space of the data
- *Local* grid, that is, the size of the grid managed by the local data model including ghost regions, and
- *Own* grid, that is, the size of the grid managed by the local data model excluding ghost regions.

The size of the regular grid cell is stored as well.

The `computeGlobal()` function reads the field `domain_block` added by the user from the data model and computes the subblocks for the destinations; that is, it generates new blocks and fills their local and own grid information. The `split()` function determines for each semantic item in which subblock(s) it belongs based on its position (semantic flag ZCURVE). The `redistribute()` and `merge()` functions follow the same pattern as do the other redistribution components.

VI. Application to the Quicksurf algorithm

We tested our data model with an in situ pipeline developed in previous work [15], based on the Quicksurf algorithm [21]. Quicksurf is an isosurface extraction algorithm. We integrated Bredala with FlowVR and adapted the Quicksurf pipeline to use our data model and redistribution methods. Our goal is to evaluate how our data model can be integrated into an in situ middleware infrastructure to improve its usability and performance.

A. FlowVR

The FlowVR middleware is designed to build in situ workflows. A FlowVR workflow is a graph where nodes are parallel programs (called modules) performing computations and edges are communication channels. Communication channels are created between input and output ports; each module can have several input and output ports. The developer describes each communication channel in a Python script that allows creating virtually any communication patterns. However, FlowVR provides no support for data types. Instead, the developer is given a buffer to data that must be serialized manually.

The same problem arises with data redistribution. The developer has to provide the necessary redistribution components that are usually hard coded for certain types or structures. A common practice with FlowVR to mitigate this problem is to separate different data structures into individual communication channels. For example, a structure composed of several arrays is sent through a different communication channel for each array, resulting in a more complex graph.

B. Quicksurf algorithm

The Quicksurf algorithm is a common rendering method in biology to visualize the surface of a molecule. It is a variation of the isosurface extraction method [24]. Quicksurf is composed of three steps.

a All the atoms are inserted into a regular 3D grid according to the position of their centers.
b For each grid cell, we compute a density value based on the atoms present in the cell and those in its neighbors.
c We perform a marching cube algorithm on the density grid.

Gromacs uses an irregular domain decomposition to distribute atom positions across multiple processes. However, the Quicksurf algorithm cannot use directly the same redistribution because Quicksurf works on a regular domain decomposition and the isosurface requires ghost regions to compute the density values at the borders of the blocks. Consequently, atom positions are exchanged between steps (a) and (b), or density values are exchanged between steps (b) and (c) to transition between the distribution from Gromacs to the distribution necessary for the isosurface.

C. Pipeline improvement with Bredala

In [15], the Quicksurf pipeline was implemented with FlowVR. Each step was a separate module. The first module computed a Morton code [26][3] for each atom and sorted the atom positions by their Morton codes. The second module computed the grid density, and the third module performed the marching cubes. The full pipeline includes three data streams: positions, Morton codes, and grid density. A specialized redistribution module was also designed to redistribute the atom positions or grid density. The module is composed of two submodules: routers and mergers. The routers are the equivalent of the sources of the redistribution components, and the mergers are the receivers. After the redistribution, each merger has all the atom positions or density values from a subdomain. The redistribution communications are performed through FlowVR using an NxM pattern. This pattern is reproduced for each data stream.

We integrated Bredala into this pipeline. First, each data structure is reassembled into a single data model. Consequently, only one data stream is necessary between the modules. In terms of code complexity, both implementations require the same number of lines of code to create a data model: two instructions for each data field.

Besides reducing the number of communication channels, another improvement is the management of the metadata. Along the pipeline, we maintain metadata about the domain decomposition such as the global domain of the simulation, the local domain, and the own domain. With FlowVR, such metadata are managed through a different API from the Message API, adding complexity. With Bredala, the metadata are simply another field pushed into the data model.

We also reimplemented the redistribution module using the block redistribution component from Bredala. The FlowVR implementation required approximately 300 lines of code, two separate modules, and an NxM communication pattern for each data stream that is hard coded for the particular data structures used in the pipeline. The Bredala implementation requires 25 lines of code (see Figure 6) with only one module that hides all the costly communications from FlowVR. The communications are directly performed through MPI inside the component instead of the FlowVR communication channels. The module is fully generic and can be used with any data model that can be expressed with Bredala. We evaluate the performance of these implementation in Section VII.

D. Extensibility

A possible extension to Quicksurf is to color the surface based on the speed of the atoms. This requires adding the atom speed into the pipeline and modifying the grid density to incorporate a color component in addition to the density value.

Most of the pipeline is hard coded with particular data types in the FlowVR-only version. To add the atom speeds, we have to modify the graph of the workflow to add communication channels for the speed and modify the code in all the modules to forward the atom speeds along the pipeline. Extra modifications are also required in the density grid module to compute the color of a cell.

[3] A Morton code is a hash of a cell coordinate.

```
if(isSource || isDest)
    component = new RedistBlockMPI(rankSource, nbSource, rankDest, nbDest,
            MPI_COMM_WORLD);


while(Module->wait()){
    if(isSource){
        Message msgIn;
        shared_ptr<ConstructData> container = get(Module, msgIn, in);

        component->process(container, REDIST_SOURCE);
    }

    if(isDest){
        shared_ptr<ConstructData> result = make_shared<ConstructData>();
        component->process(result, REDIST_DEST);

        MessageWrite msgOut;
        containerToMsg(Module, msgOut, result);

        Module->put(out, msgOut);
    }

    if(isSource || isDest)
        component->flush();
}
```

Fig. 6: Code snapshot to build the redistribution module with Bredala.

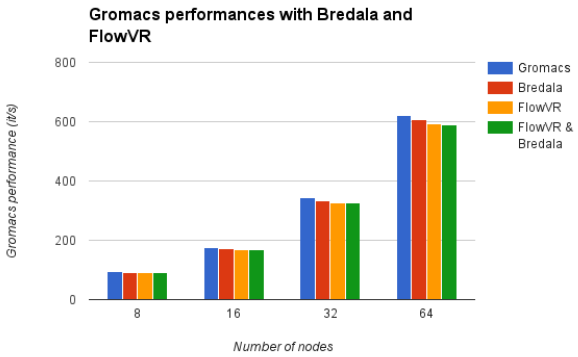


Fig. 7: Performance measurements of Gromacs with several instrumentation methods.

The FlowVR with Bredala version does not need to modify the graph workflow. The speed is simply pushed inside the data model and forwarded across all the modules automatically. The same applies for the redistribution component. We only need to modify the module computing the density grid to compute the color of a cell.

Thus, Bredala simplified the graph of the workflow, improved the reusability of the pipeline, and simplified code of the modules.

VII. Performance evaluation

In this section we evaluate the performance of Bredala when building and redistributing a data model, and we compare the results the performance of with FlowVR. We used Gromacs as our application driver.

A. Experimental context

The experiments ran on Froggy, a 190 compute node cluster from the Ciment infrastructure.[4] Each compute node has 2 eight-core processors, Sandy Bridge-EP E5-2670 at 2.6 GHz, with 64 GB of memory. Nodes are interconnected through a FDR InfiniBand network. FlowVR 2.1 and Gromacs 4.6.7 are compiled with OpenMPI 1.8.3 (no OpenMP). For all experiments using Gromacs, we run a Martini simulation with a patch of 54,000 lipids representing about 2,100,000 particles in coarse grain [1].

B. Data model cost

We modified the code of Gromacs to output data from the simulation using three different methods. In all cases, we extracted the atom positions and the atom IDs from the simulation. The first configuration uses only Bredala to measure the cost of building a data model. We built a data model composed of one array of the form $[x_1, y_1, z_1, id_1, ..., x_N, y_N, z_N, id_N]$ that is then serialized. The second configuration uses only FlowVR to isolate the cost of our in situ middleware. We created the same array as the Bredala configuration and copied it in a FlowVR message. The third setup uses both Bredala and FlowVR. This evaluates the cost of both building the data model and the cost of the in situ middleware and compares these costs with the cost of the two previous configurations. We built the same data model as in the Bredala setup, serialized it, and copied the serial buffer in a FlowVR message. We measured the performance of these three configurations and compared the results with the performance of the unmodified version of Gromacs (Figure 7). We used 16 cores per node for the simulation. Data were extracted every 100 iterations of the simulation.

Molecular dynamic simulations have a very high frequency, commonly reaching 1000 Hz. At this rate, the slightest perturbation on the simulation can have a noticeable impact on the simulation performance. In this context, extracting data from the simulation while impacting as little as possible is challenging. Our goal with these experiments is to evaluate the additional cost of Bredala when combined with an in situ middleware.

At all scales, Bredala is the less costly setup, with a maximum cost of 2.9% on the simulation performance at 32 nodes while FlowVR has an impact of 5.5%. This difference of cost is explained by the synchronizations that each MPI process of the simulation has to make with the FlowVR daemon every output iteration. In all the cases, Bredala adds a limited extra overhead when combined with FlowVR. In the worst case, the FlowVR and Bredala configuration adds an extra 0.4% of performance drop compared with the FlowVR-only setup on 32 nodes. These results show that Bredala can be used in conjunction with another in situ middleware framework for a slight additional cost compared with the cost of the in situ middleware itself. Note also that adding Bredala with FlowVR does not introduce more variability in performance compare to FlowVR alone.

C. Redistribution component performance

Next we benchmarked our block redistribution component and compared it with our hard-coded version using FlowVR developed for the Quicksurf pipeline. We reproduced the same data model as FlowVR with Bredala to test our redistribution component. The data model is composed of two arrays. The first one has the same structure as in Section VII-B (atom positions with atom IDs). The second array is composed of a Morton code for each atom. Upon initialization, we generated

[4]https://ciment.ujf-grenoble.fr

x atoms, computed their Morton codes, and sorted the atoms by their Morton code. Atom positions are generated uniformly in a global domain space.

We first benchmarked our component in an in situ scenario. N processes are emitting data (sources) and the same processes receive the data (receivers). The sources emit atoms from a global domain, but the receivers only receive atoms from a subdomain of the global domain. With our uniform distribution of atoms, each source sends atoms to each receiver. To measure the performance of our components, we run an infinite loop redistributing the atoms. Atoms are generated only once at initialization. The components can start the next iteration only when all the receivers have received and merged their data. The component's performance is then given by the number of redistribution they can perform per second.

We measured our component's speed in both weak (Figure 8(a)) and strong scaling (Figure 8(b)). In weak scaling, each source emits 200,000 atoms. In strong scaling, 2,100,000 atoms are distributed among the sources. For each scenario, we placed one source and receiver per node. Each component computes all the necessary information for the redistribution at each iteration.

In the weak-scaling scenario, FlowVR displays the best speed at low scale with 68 Hz on eight nodes compared with 53 Hz with Bredala. At larger scale, however, the gap between both implementations shrinks, and their performance drops to 13 Hz with Bredala and 16 Hz with FlowVR at 64 nodes. With Bredala, the main reason for the drop of performance when scaling is the cost of the `split()` function (Figure 8(c)), which is the most time-consuming operation (58% of the total execution time at 64 nodes). When increasing the number of nodes, we increase the number of subdata models to generate, one subdata model for each receiver. This increases the amount of operations necessary to divide and concatenate a data model. FlowVR also suffers from the increased complexity to split the initial data (Figure 8(e)). However, the time spent doing communication increases significantly as well. At 32 nodes, the split and the communications represent 46% and 40%of the total time. With FlowVR, each communication is described in the workflow graph. Adding a node generates $2xN$ new communication channels with N the number of nodes (two data structures to transport). This increases the amount of work done by the FlowVR daemon and slows the communications managed by the daemon.

In the strong-scaling scenario, FlowVR also displays a better speed at low scale, with 56 Hz on eight nodes against 43 Hz for Bredala. However, the performance of FlowVR starts to drop beyond 16 nodes to reach 29 Hz on 64 nodes. We observe that the time spent in communication increases significantly from 5 ms at 8 nodes to 25 ms at 64 nodes (Figure 8(f)). As with the weak-scaling case, the increase in communication time is explained by the increased complexity of the workflow graph. Even though the time to split the data is decreasing when scaling up, it is not enough to compensate for the increased communications. Bredala is able to scale up to 90 Hz at 32 nodes but drops to 35 Hz at 64 nodes. We observed that both the times of splitting the data and communications are decreasing up to 32 nodes (Figure 8(d)) but then increase at 64 nodes. Future work will improve the manipulation of array to avoid unnecessary copies during split operations.

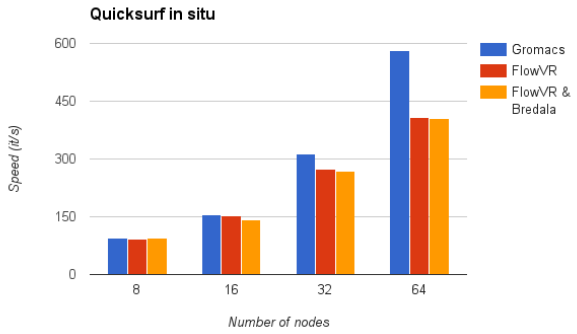


Fig. 9: Performance measurements of Gromacs with the Quicksurf pipeline implemented with FlowVR and Bredala with an in situ strategy. Gromacs performance stand-alone performance is given as reference.

Overall, we see that except at low scale, the Bredala redistribution component scales better than FlowVR while providing a fully generic redistribution pattern.

D. *Quicksurf pipeline*

We also compared the performance of the hard coded version of the Quicksurf algorithm with our implementation using Bredala as described in Section VI-C. Gromacs runs on 15 cores per node, and we use the last core to compute Quicksurf. We triggered the in situ pipeline every 100 iterations of Gromacs. We configured the pipeline so that the redistribution component exchanges atom positions.

Our first experiment runs a full in situ scenario where the complete analysis pipeline is hosted on the simulation nodes and the redistribution is performed between the simulation nodes. Figure 9 compares the performance of both implementations with the performance of Gromacs without modifications. Except at 16 nodes, both implementations have a similar cost for the simulation. At 64 nodes, both implementation costs 30% of the simulation performance.

Our second experiment uses staging nodes to compute the analysis after the redistribution. After the computation of the Morton codes, the data are routed to the staging nodes to compute the grid density and the marching cubes. Results are summarized in Figure 10. We allocate one node as a staging node for every 32 simulation nodes. As previously, the performance of both implementations is close, with the Bredala implementation slightly less costly in all the cases. At 64 nodes (62 simulations, 2 staging nodes), the FlowVR implementation costs 23% of the simulation performance while the Bredala implementation costs 16%.

The performance improvement brought by Bredala is overall quite small: 7% performance improvement in the best case and less than 2% otherwise. This is surprising considering the performance difference between the two redistribution component alone (Figure 8). Two main facts explain these results. First, most of the impact on the simulation performance comes from the data extraction from the simulation and the extra in situ computations that impact the cache. Second the simulation

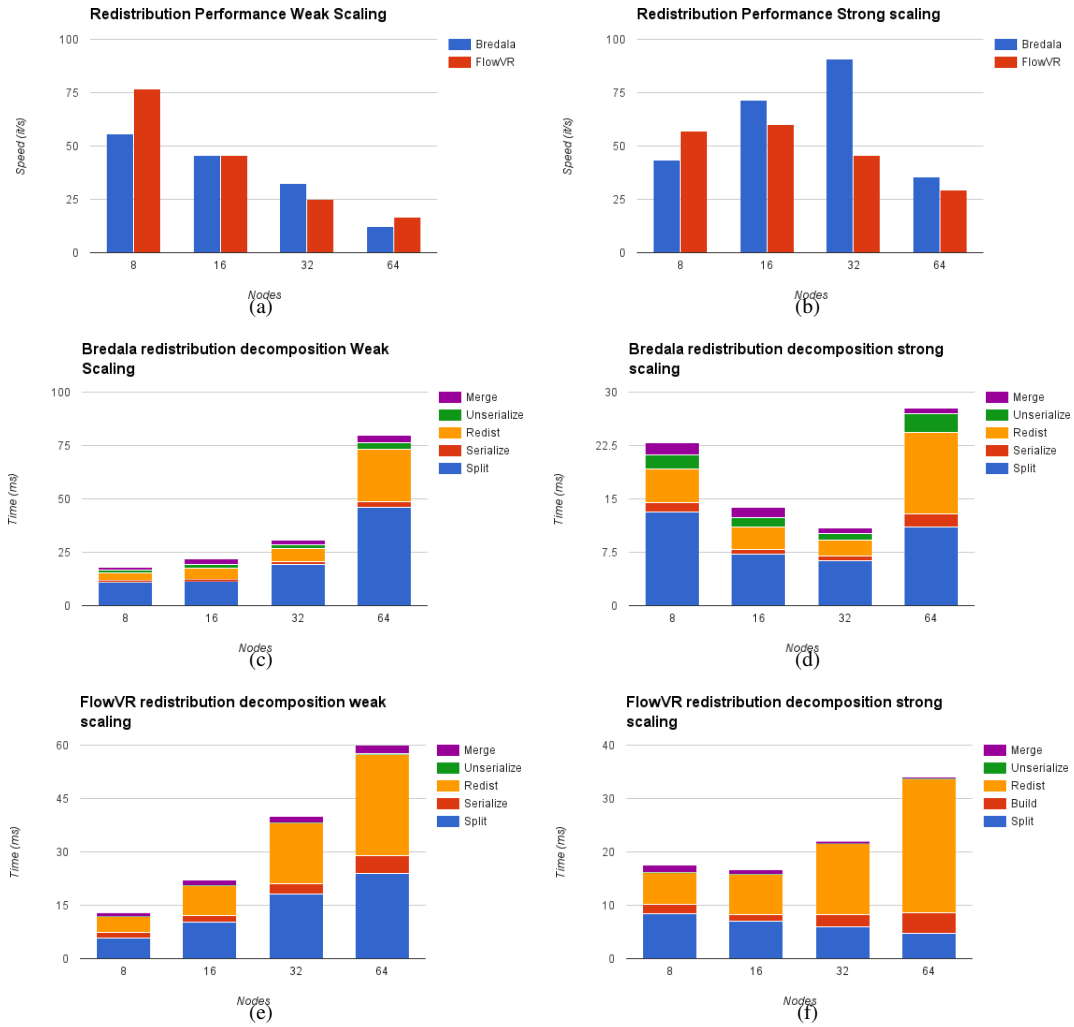


Fig. 8: Redistribution components performance and time decomposition with Bredala and FlowVR.

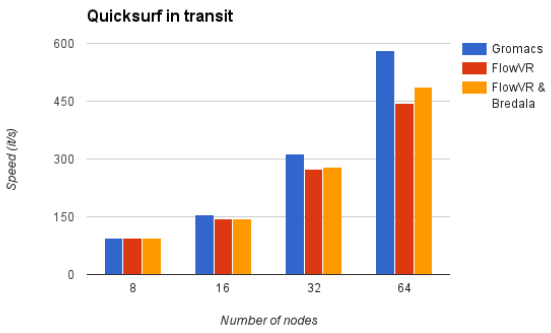


Fig. 10: Performance measurements of Gromacs with the Quicksurf pipeline implemented with FlowVR and Bredala with an in transit strategy. Gromacs performance stand-alone performance is given as reference.

extracts data at only six iteration per second on 64 nodes. That data rate is slower than the peak data rate manageable by the components.

We showed that converting the pipeline to use Bredala does not add a significant cost to the simulation performance compared with what FlowVR already costs. Moreover, Bredala enables more generic and maintainable code that is easier to extend. In some cases, Bredala can even improve the impact on the simulation performance with more efficient data redistribution components.

VIII. Conclusion

We introduced Bredala, a lightweight library designed to protect the semantics of data during parallel redistribution in in situ applications. The user describes a data model field by field and adds extra information to allow the library to isolate individual semantic items and manipulate them during parallel redistribution. Bredala comes with several common redistribution patterns such as spatial redistribution. We compared our library with FlowVR, and we studied the benefits of combining FlowVR with Bredala and applied them to the Quicksurf algorithm. We showed that combining FlowVR with Bredala does not add a significant overhead compared with the overhead of FlowVR itself while improving greatly the reusability of the FlowVR modules.

So far, Bredala uses only the main memory for certain operations such as serialization. Several in situ middlewares

such as FlowVR or Damaris expose specialized allocators to transfer data to the middleware. Future version of Bredala will provide an allocator interface to use directly the allocator provided by in situ middleware and avoid unnecessary copies. We will also extend the current set of basic types for data field to n-dimensional arrays.

ACKNOWLEDGMENT

This material was based upon work supported by Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357, program manager Lucy Nowell. All the computations presented in this paper were performed using the Froggy platform of the CIMENT infrastructure (https://ciment.ujf-grenoble.fr), supported by the Rhône-Alpes region (GRANT CPER07_13 CIRA) and the Equip@Meso project (reference ANR-10-EQPX-29-01) of the programme Investissements d'Avenir supervised by the ANR.

REFERENCES

[1] http://philipwfowler.wordpress.com/2013/10/23/gromacs-4-6-scaling-of-a-very-large-coarse-grained-system/.

[2] The boost collection. http://boost.org.

[3] Parallel netcdf home page. http://www.unidata.ucar.edu/software/netcdf/.

[4] Unidata netcdf home page. http://trac.mcs.anl.gov/projects/parallel-netcdf.

[5] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan, and S. Klasky. Just in time: Adding value to the io pipelines of high performance applications with jitstaging. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 27–36, New York, NY, USA, 2011. ACM.

[6] S. Ahern, E. Brugger, B. Whitlock, J. S. Meredith, K. Biagas, M. C. Miller, and H. Childs. Visit: Experiences with sustainable software. *arXiv preprint arXiv:1309.1796*, 2013.

[7] J. Ahrens, B. Geveci, and C. Law. 36 paraview: An end-user tool for large-data visualization. *The Visualization Handbook*, page 717, 2005.

[8] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[9] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Structure slicing: Extending logical regions with fields. In *High Performance Computing, Networking, Storage and Analysis (SC14)*, pages 845–856, Nov 2014.

[10] D. Boyuka, S. Lakshminarasimham, X. Zou, Z. Gong, J. Jenkins, E. Schendel, N. Podhorszki, Q. Liu, S. Klasky, and N. Samatova. Transparent I Situ Data Transformations in ADIOS. In *Cluster, Cloud and Grid Computing (CCGrid), 2014*, pages 256–266, May 2014.

[11] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki. Flexpath: Type-based publish/subscribe system for large-scale science analytics. In *Cluster, Cloud and Grid Computing (CCGrid'14)*, pages 246–255, May 2014.

[12] C. Docan, M. Parashar, and S. Klasky. DataSpaces: an Interaction and Coordination Framework for Coupled Simulation Workflows. *Cluster Computing*, 15(2):163–181, 2012.

[13] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf. Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O. In *CLUSTER - IEEE International Conference on Cluster Computing*. IEEE, Sept. 2012.

[14] M. Dorier, R. Sisneros, Roberto, T. Peterka, G. Antoniu, and B. Semeraro, Dave. Damaris/Viz: a Nonintrusive, Adaptable and User-Friendly In Situ Visualization Framework. In *LDAV - IEEE Symposium on Large-Scale Data Analysis and Visualization*, Atlanta, United States, Oct. 2013.

[15] M. Dreher and B. Raffin. A Flexible Framework for Asynchronous In Situ and In Transit Analytics for Scientific Simulations. In *14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Chicago, United States, May 2014.

[16] G. Eisenhauer, M. Wolf, H. Abbasi, S. Klasky, and K. Schwan. A type system for high performance communication and computation. In *Workshop on D3Science associated with e-Science 11*. IEEE Computer Society, 2011.

[17] G. Eisenhauer, M. Wolf, H. Abbasi, and K. Schwan. Event-based systems: Opportunities and challenges at exascale. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 2:1–2:10, New York, NY, USA, 2009. ACM.

[18] N. Fabian, K. Moreland, D. Thompson, A. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. Jansen. The Paraview Coprocessing Library: a Scalable, General Purpose In Situ Visualization Library. In *Large Data Analysis and Visualization (LDAV'11)*, pages 89–96, Oct 2011.

[19] M. Howison, A. Adelmann, E. W. Bethel, A. Gsell, B. Oswald, and Prabhat. H5hut: A High-Performance I/O Library for Particle-Based Simulations. In *Proceedings of Interfaces and Abstractions for Scientific Data Storage (IASDS10)*, Heraklion, Crete, Greece, Sept. 2010.

[20] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Romanus, N. Podhorszki, S. Klasky, H. Kolla, J. Chen, R. Hager, C. S. Chang, and M. Parashar. Exploring data staging across deep memory hierarchies for coupled data intensive simulation workflows. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, May 2015.

[21] M. Krone, J. E. Stone, T. Ertl, and K. Schulten. Fast Visualization of Gaussian Density Surfaces for Molecular Dynamics and Particle System Trajectories. In *EuroVis 2012 Short Papers*, volume 1, 2012.

[22] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, M. Parashar, N. Samatova, K. Schwan, A. Shoshani, M. Wolf, K. Wu, and W. Yu. Hello ADIOS: the Challenges and Lessons of Developing Leadership Class I/O Frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

[23] J. Lofstead and K. Schwan. Xchange: high performance data morphing in distributed applications. 2005.

[24] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA, 1987. ACM.

[25] K. Moreland. Oh, $#! Exascale! The Effect of Emerging Architectures on Scientific Discovery. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 224–231, Nov 2012.

[26] Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical Report Ottawa, Ontario, Canada, 1966.

[27] T. Peterka, D. Morozov, and C. Phillips. High-Performance Computation of Distributed-Memory Parallel 3D Voronoi and Delaunay Tessellation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014.

[28] S. Pronk, S. Pall, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. Kasson, D. van der Spoel, B. Hess, and E. Lindahl. Gromacs 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29(7):845–854, 2013.

[29] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit–An Object-Oriented Approach To 3D Graphics*. Kitware, Inc., fourth edition, 2006.

[30] The HDF Group. Hierarchical Data Format, version 5, 1997-2014. http://www.hdfgroup.org/HDF5/.

[31] The HDF Group. Parallel Hierarchical Data Format, version 5, 1997-2014. http://www.hdfgroup.org/HDF5/PHDF5/.

[32] B. Whitlock, J. M. Favre, and J. S. Meredith. Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System. In *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '11, pages 101–109, Aire-la-Ville, Switzerland, Switzerland, 2011. Eurographics Association.

[33] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, Q. Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. PreDatA - Preparatory Data Analytics on Peta-Scale Machines. In *Parallel Distributed Processing (IPDPS'10)*, pages 1–12, 2010.

[34] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky. GoldRush: Resource Efficient in Situ Scientific Data Analytics Using Fine-grained Interference Aware Execution. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, 2013.

[35] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.-A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu. FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics. In *Parallel Distributed Processing (IPDPS'13)*, pages 320–331, May 2013.