ISAVS: Interactive Scalable Analysis and Visualization System

Conference Paper · November 2017

DOI: 10.1145/3139295.3139299

8 authors, including:

Steve Petruzza
University of Rome Tor Vergata
5 PUBLICATIONS **11** CITATIONS
SEE PROFILE

Attila Gyulassy
University of Utah
54 PUBLICATIONS **947** CITATIONS
SEE PROFILE

Giorgio Scorzelli
University of Utah
40 PUBLICATIONS **423** CITATIONS
SEE PROFILE

Frederick Federer
University of Utah
18 PUBLICATIONS **160** CITATIONS
SEE PROFILE

ISAVS: Interactive Scalable Analysis and Visualization System

Steve Petruzza*†
Aniketh Venkat
Attila Gyulassy
Giorgio Scorzelli
SCI Institute - University of Utah

Frederick Federer
Alessandra Angelucci
University of Utah

Valerio Pascucci
SCI Institute - University of Utah

Peer-Timo Bremer
Lawrence Livermore National Lab

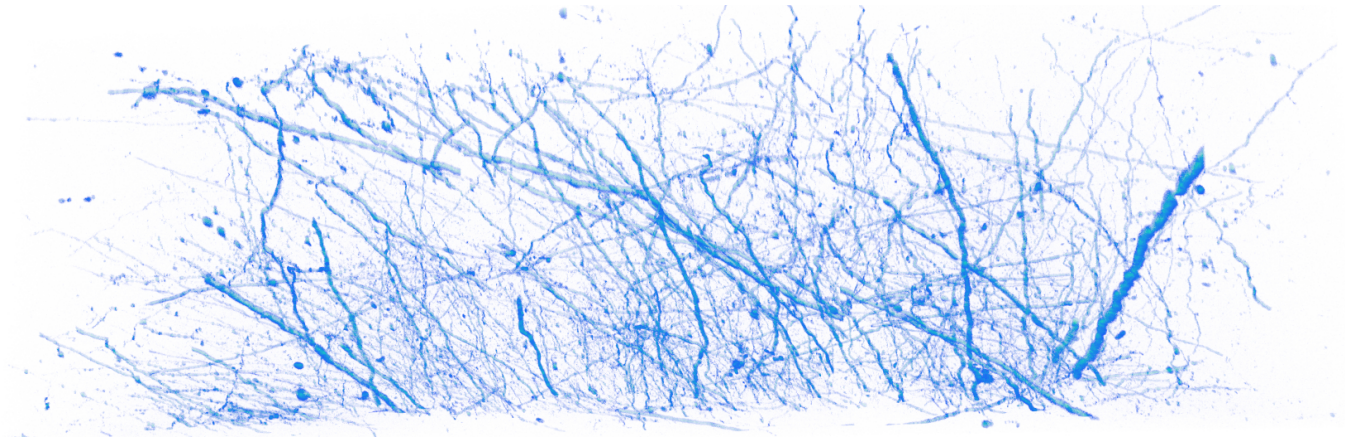

Figure 1: Interactive visualization of large-scale non-human primate brain data (i.e., 6 volumes, each of size 2048x2048x2575). Rendered with OSPRay [Wald et al. 2017]

ABSTRACT

Modern science is inundated with ever increasing data sizes as computational capabilities and image acquisition techniques continue to improve. For example, simulations are tackling ever larger domains with higher fidelity, and high-throughput microscopy techniques generate larger data that are fundamental to gather biologically and medically relevant insights. As the image sizes exceed memory, and even sometimes local disk space, each step in a scientific workflow is impacted. Current software solutions enable data exploration with limited interactivity for visualization and analytic tasks. Furthermore analysis on HPC systems often require complex hand-written parallel implementations of algorithms that suffer from poor portability and maintainability.

We present a software infrastructure that simplifies end-to-end visualization and analysis of massive data. First, a hierarchical streaming data access layer enables interactive exploration of remote data, with fast data fetching to test analytics on subsets of the data. Second, a library simplifies the process of developing new analytics algorithms, allowing users to rapidly prototype new approaches and deploy them in an HPC setting. Third, a scalable runtime system automates mapping analysis algorithms to whatever computational hardware is available, reducing the complexity of developing scaling algorithms. We demonstrate the usability and performance of our system using a use case from neuroscience: filtering, registration, and visualization of tera-scale microscopy data. We evaluate the performance of our system using a leadership-class supercomputer, Shaheen II.

*Also with University of Rome "Tor Vergata", Italy.
†Corresponding author email: spetruzza@sci.utah.edu

SA '17 Symposium on Visualization, November 27-30, 2017, Bangkok, Thailand

ACM ISBN 978-1-4503-5411-0/17/11.
https://doi.org/10.1145/3139295.3139299

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; **Parallel programming languages**; • **Software and its engineering** → **Development frameworks and environments**; **Integrated and visual development environments**;

KEYWORDS

Interactive visualization and analysis, parallel custom analysis workflows, algorithms scalability, microscopy

1 INTRODUCTION

High resolution microscopy systems and large-scale simulations generate an increasing amount of data that are often used for visualization and analysis tasks. Yet, being able to visualize and perform analysis remains a major challenge largely due to the lack of algorithmic and computational solutions to handle, analyze and reconstruct the increasing amount of data that are being collected. Several software tools are available for researchers, from free and open-source solutions, such as Fiji (ImageJ) [Schindelin et al. 2012], Vaa3D [Peng et al. 2010], and KNOSSOS [Helmstaedter et al. 2011], to commercial solutions such as Imaris [Bitplane], Neurolucida [Glaser and Glaser 1990], Amira [FEI 2016], and Volocity [PerkinElmer 2014]. While each offers varying degrees of functionality in terms of visualization, annotation, and custom analytics, scaling to large data size remains a challenge for most. Furthermore, while built-in analytics is generally limited to smaller tasks such as filtering, adding custom analytics requires rigorous coding requirements that may not be user-friendly.

This work introduces the first end-to-end software framework (see Fig. 2) for large-scale interactive visualization and scalable analysis that also simplifies the definition and execution of scientific workflows on HPC systems. The system exploits properties of the IDX format [Pascucci and Frank 2001] that enables access to multiple levels of resolution with low latency suitable for interactive exploration of tera-voxel datasets. Data are produced in, or are converted to the IDX format using the high performance I/O library PIDX [Kumar et al. 2011] that bridges the gap between large-scale writes and analytics-appropriate reads. Interactive exploration is achieved via the ViSUS framework [Pascucci et al. 2012] that enables seamless streaming visualization capabilities on devices ranging from iPhones to high-end visualization machines, to multi-display powerwalls, with the data being hosted anywhere from USB hard drives to remote HPC file systems. Finally, using a new python software layer we interface the ViSUS framework with a set of user-defined processing components that can perform operations on the data either interactively or on dedicated computational resources (i.e. local or remote).

In this framework, we introduce a parallel library that allows fast and simple implementation of visualization and analytic workflows. To achieve computational scalability, the library builds upon an existing runtime (i.e. Charm++), a C++ abstraction layer that allows a user to define the processing algorithms as a set of idempotent tasks connected in a dataflow. This abstraction layer introduces two main improvements:

- the user does not need knowledge of the underlying communication or threading managed by the runtime;
- the same task graph can be executed both locally and remotely via a common interface for interactive processing on smaller desktop class systems and also at scale on remote HPC systems for larger data or compute intensive tasks.

Our key contributions are:

- An end-to-end software framework for interactive streaming visualization and analysis of large-scale scientific datasets;
- Highly concurrent parallel processing abstraction layer that supports arbitrary analytics and visualization workflows;
- Strong scalability performance for a large set of common post-processing tasks such as filtering, alignment, stitching and visualization.
- Improve user productivity, code re-usability and portability by designing common workflows that execute on both local desktop PCs and at scale on remote HPC systems;

We demonstrate the usage of this system in a neuroscience setting, enabling interactive visualization and analysis of large-scale non-human primate (NHP) brain 3D microscopy data. We present three algorithms that are experimented at scale on the supercomputer, Shaheen II:

- a parallel filter for denoising data;
- automatic alignment and stitching of multiple 3D volumes;
- interactive volume rendering and image composition.

The parallel analysis workflow is configured and launched through a scripting interface and the output is then used to update the visualization interactively.

2 BACKGROUND

Structuring and efficiently accessing large-scale data have always been the aim of several high level I/O libraries. The most prominent examples are HDF5 [Folk et al. 1999], Parallel NetCDF (PnetCDF) [Li et al. 2003], and ADIOS [Lofstead et al. 2008]. These libraries typically store data in traditional row-major blocks. While they are fast, general-purpose and robust when it comes to data writes, they significantly lag during reads, a critical requirement for interactive, exploratory analysis of large datasets. Parallel IDX (PIDX) [Kumar et al. 2014, 2012; Pascucci et al. 2012], is an I/O library that writes data directly in IDX, a hierarchical, cache-oblivious, multi-resolution data format that can be easily leveraged for fast reads required for interactive visualization and analysis.

On the visualization side, VisIt [Childs et al. 2012] and ParaView [Ahrens et al. 2005] are well-known distributed parallel visualization and analysis applications. They are typically executed in parallel, coordinating visualization and analysis tasks for massive data. The data are typically loaded at full resolution, requiring large amounts of system memory and proximity with data source. Moreover, their use of parallel computing resources is limited to mostly data-parallel tasks and a separate modality that is not well blended with data sources external to parallel computing environment. Both packages, though, utilize a plugin-based architecture, so many formats are supported and have the potential of being extended. The ViSUS application framework [Pascucci et al. 2012] is designed around a hierarchical streaming data model available through the PIDX open library, and enables interactive visualization and on-the-fly processing [Christensen et al. 2016] in a hardware-agnostic manner.

The traditional model for parallel computing has been to manually couple sections of serial computing distributed amongst processors with the exchange of messages between them. The most common implementation of this model is the Message Passing Interface (MPI) [Gropp et al. 1999] which is supported on virtually all medium and large-scale computing resources. However, as the level of parallelism grows it is becoming increasingly difficult, even for expert users, to develop efficient solutions in this rather low-level environment. Instead, more recently, so called, task-based

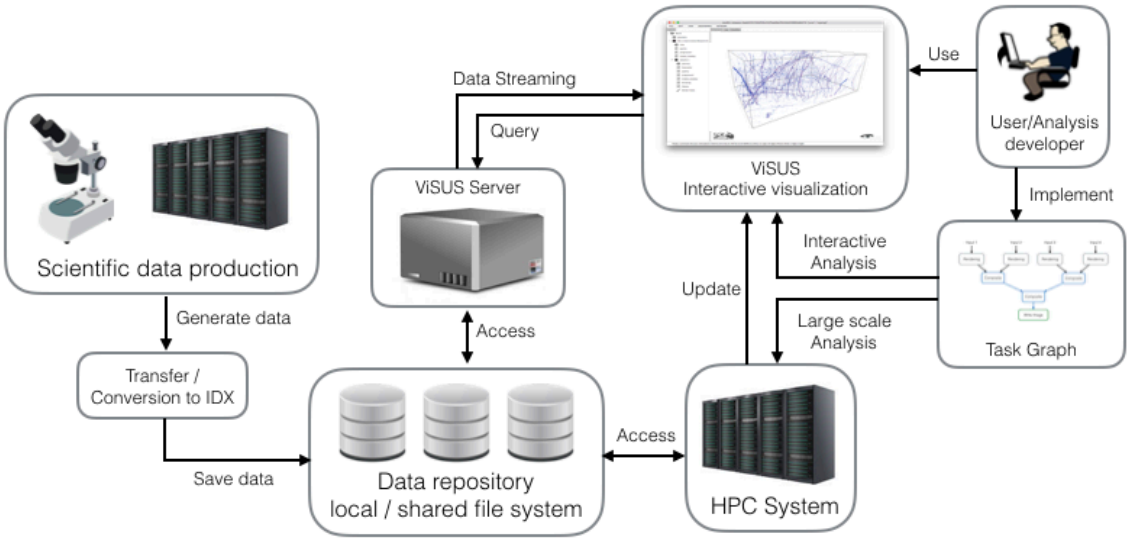

Figure 2: System overview: (i) Data produced by simulations or high-resolution acquisitions are either generated in or converted to the IDX format and stored locally or on shared filesystems. (ii) The ViSUS client enables remote interactive visualization via the ViSUS server that provides fast read access to multi-resolution data queries.(iii) A user/developer can implement visualization/analysis algorithms using the parallel library provided by the framework interactively through the ViSUS client or at scale on an remote HPC systems that will eventually update the visualization for further explorations.

programming models such as Charm++ [Kale and Zheng 2009] have been proposed. Rather than requiring users to directly manage work distribution and message passing, these systems decompose the processing into a workflow of explicit tasks, which are automatically mapped to physical computational resources and executed in a parallel environment. While task-based approaches are more flexible than MPI, both require expert developers to create efficient solutions and often needs to be individually optimized for specific hardware infrastructures.

3 THE FRAMEWORK

Our approach to achieving interactivity in all aspects of the software infrastructure is (1) to use a data model that enables both remote data access to be visualized in a coarse-to-fine progressive manner while maintaining high-performance I/O for data stored locally, (2) to map processing workflows to HPC resources with immediate progressive availability of results, and (3) to enable the execution of identical workflows on both desktop PCs and at scale on HPC systems. The scientific visualization pipeline begins with the production of data from scientific applications (e.g. simulations) or microscopy scans that are stored on local disks or large shared file system.

3.1 Data Streaming Infrastructure

The data access layer of the IDX format employs a hierarchical variant of a Lebesgue space filling curve, commonly referred to as HZ order. Traditionally, space filling curves have been used successfully to develop a static indexing scheme that generates a data layout suitable for hierarchical traversal. The layout instills both spatial and hierarchical locality. Conceptually, the scheme acts as a storage pyramid with each level of resolution laid out in Z-order. Because of these properties, IDX supports progressive reads of low-resolution (sub-sampled) data, which is very suitable for streaming based remote visualization. By converting initially to the hierarchical space-filling IDX file format, the progressive streaming model enables this data to be visualized interactively by the ViSUS framework no matter where it is stored.

The ViSUS framework allows interactive visualization and exploration of large-scale datasets, with the ability to add filters or simple scripting operations to the visualization pipeline. Our new python interface allows interactive design of data processing workflows that receives an input array of data to perform custom analytics either locally or at scale remotely. In both the cases, our framework allows data to be streamed in a coarse-to-fine manner for interactive visualization. The user can create multiple processing nodes for different workflows to be executed in the same environment, configure number of nodes and processors for remote HPC sessions, provide additional parameters to inform the analysis application about the current viewer settings (e.g. datasets loaded, active filters) and operate on multiple datasets in a single workflow.

3.2 Analytics design library

Applying any algorithm, no matter how efficient, to terabytes of data will require massively parallel compute resources both in terms

of available storage and processing capabilities. The traditional approach of solving this challenge is to re-implement the corresponding algorithms in MPI or one of the more recent task-based runtimes. Not only does this require significant expertise in these programming models, but it also leads to (at least) two separate implementations that must be maintained. Instead, we propose to utilize the components of the interactive workflows discussed above to build large-scale parallel workflows. More specifically, our system provides a simple library for developers of analytics algorithms to describe their parallel dataflow as a graph of tasks, each of which either directly uses the interactive components or minor variations thereof. To execute a task graph, we create a backend that interprets a given graph and instantiates at runtime a task based application based on Charm++. This shields users from the complexities of the runtime, provides performance portability across multiple computing environments and produces efficient and easily maintainable systems.

The key insight enabling our system is the fact that much of the complexity in dealing with parallel implementations comes from the extreme generality of existing environments. Not only must existing runtimes support a wide range of communication and synchronization primitives, but they must do so dynamically without any assumption on the underlying algorithm. Instead, for the type of large scale data processing of interest here, one typically deals with a static task graph that is known a priori. The graph may change in size based on which data must be processed or how many resources are available, but the basic structure, including a partial order of operations and all communication steps, is well known. Therefore, we provide users with a simple interface to describe their computation explicitly as a graph in which each task has a predefined set of inputs and outputs and executes a given function. Each task can be implemented either as a standalone function, that can be used also in the streaming computations, or even exploiting existing threading libraries such as OpenMP [Dagum and Menon 1998]. This provides a highly modular yet so far entirely serial implementation of the algorithm of interest. The library dynamically maps this graph into the execution data model of Charm++, which will handle the task mapping (i.e. which task is executed on which processor) as well as the communication. The additional layer of abstraction provides a number of benefits: First, developing, maintaining and debugging is significantly simpler as user level code effectively executes in serial and there exists an explicit (offline) description of the task graph to study. Second, the system to a large extent becomes independent of the underlying hardware and system software stack. Third, our task graph easily integrates with existing solutions at all levels of abstraction. It is important also to note that each task can use arbitrary existing libraries or tools virtually unmodified since each task is simply a serial execution of some operation.

An algorithm developer using this library needs to perform three basic steps: first, implement all tasks used in the algorithm; second, provide de-/serialization routines for the objects that are exchanged between tasks; and third, extend the `TaskGraph` class to describe the dataflow. The first two are generic and are required in some form for any implementation. The third, represents a procedural description of the task graph. Tasks are defined by Task objects that contain a set of input and output ids (i.e. the tasks that will be communicating with the given task) and a callback id that defines which function will be executed at runtime by the task. To define a `TaskGraph` the user needs to implement a function (i.e. task()) that given a task id returns a Task object.

Listing 1 showcases an example implementation of a `TaskGraph` for a k-way reduction dataflow with an additional wrap-up step useful for saving the final result of the reduction. This `TaskGraph` together with the definition of the callback function can be used to perform a volume rendering and composite workflow as illustrated in Listing 2. The instantiation of the `ReductionPlusOne` graph in the example requires to provide the domain decomposition (i.e. the block_decomp parameter indicates how many blocks the domain should be divided into in three dimensions) and a reduction factor (i.e. the valence parameter). After defining the `TaskGraph`, the `DataFlowController` is initialized by adding the `TaskGraph`, registering the callbacks and providing the initial inputs. In most cases the initial inputs simply inform the leaf nodes which part of the domain they are suppose to operate. Finally, the `DataFlowController` will be responsible for the execution of the `TaskGraph` on the available resources.

Listing 1: TaskGraph implementation for a k-way reduction dataflow. For simplicity we assume there are k^d many leafs

```
1   // Constructor to set the valence and number of leafs
2  ReductionPlusOne::ReductionPlusOne(int leafs, int valence) {
3    d = log(leafs,valence);
4    k = valence;
5    // Assuming k^d leafs
6    total = (std::pow(k,(d+1)) -1) / (k-1)
7
8    // Add task types (i.e., callback ids)
9    callback_ids.push_back(LEAF_CB);
10   callback_ids.push_back(REDUCE_CB);
11   callback_ids.push_back(ROOT_CB);
12 }
13
14 // Get callbacks Ids
15 vector<int> ReductionPlusOne::callbacks() {
16   return callback_ids;
17 }
18
19 // Create a logical task from an id
20 Task ReductionPlusOne::task(int task_id) {
21   Task t;
22   t.id = task_id;
23
24   // Assign the input for a leaf
25   if (task_id >= (total - k^d))
26       t.type = callback_ids[0];
27   else { // Assign inputs for other tasks
28       incoming.resize(k);
29       for (int i=0; i < k; i++)
30         t.incoming[i] = task_id*k+i+1;
31   }
32
33   // Assign the output for the root task
34   if (task_id == 0)
35       t.type = callback_ids[2];
36   else {  // Assign the output for the other tasks
37       t.type = callback_ids[1];
38       t.outgoing.resize(1);
39       t.outgoing[0].resize(1);
40       t.outgoing[0][0] = (task_id-1)/k;
```

```
41    }
42
43    return t;
44  }
```

3.3 Dynamic Runtime System

The Charm++ runtime is used as backend for the dynamic execution of the tasks. In our implementation, we represent the tasks as *chares* [Kale and Zheng 2009]: migratable-objects that represent the basic unit of parallel computation in Charm++. The tasks in the task graph are mapped to a collection of chares called a *chare* array. The runtime can launch a large number of chares simultaneously and can periodically balance the load by migrating chares when necessary. The implementation creates a single *chare* array that holds all tasks needed throughout the execution of the task graph and based on the task id, each chare is able to determine the callback to use (i.e. using the function *task(id)*).

The communication between chares are done using remote procedure calls. This means that the chares containing the input data can asynchronously start the dataflow by simply sending the data to the corresponding chare of the dataflow. Similarly, for each outgoing Payload each task simply adds an input to the corresponding downstream task identified through the task id. As defined at the end of Listing 2, the controller launches the execution of the leaf tasks (i.e. run function) that will asynchronously trigger execution of the other tasks in the graph.

Listing 2: Example of volume rendering and compositing dataflow.

```
1  int volume_render(vector<Payload>& in,
2                    vector<Payload>& out, TaskId id);
3  int composite(    vector<Payload>& in,
4                    vector<Payload>& out, TaskId id);
5  int write_image(  vector<Payload>& in ,
6                    vector<Payload>& out, TaskId id);
7
8  // Reduction tree + additional wrap-up task
9  ReductionPlusOne graph(block_decomp, valence);
10
11 // Initialize the library runtime controller
12 DataFlowController c;
13 c.initialize(graph);
14
15 // Register the callbacks
16 vector<CallbackId> avail_cid = graph.callbacks();
17 // Leaf task will volume render the local data
18 c.registerCallback(avail_cid[0], volume_render);
19 // Internal nodes will composite the image
20 c.registerCallback(avail_cid[1], composite);
21 // The wrap-up task will write the image
22 c.registerCallback(avail_cid[2], write_image);
23
24 // Set initial inputs and start execution
25 map<TaskId,Payload> initial_inputs;
26 c.run(initial_inputs);
```

In order to assist users in the implementation of their algorithms, the library also provides prototypical implementations of common task graphs, such as, reductions and broadcasts for users to use or modify.

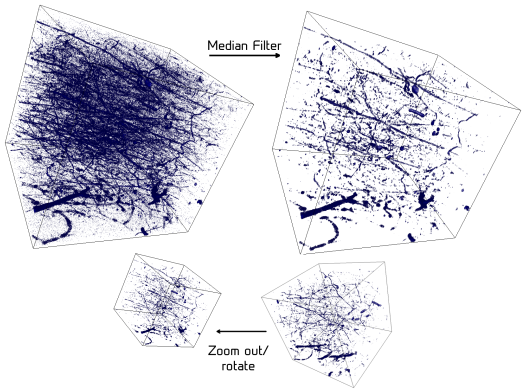

Figure 3: Interactive median filtering on a volume of size 1024x1024x1024. Rendering at 4fps 1024x1024 framebuffer.

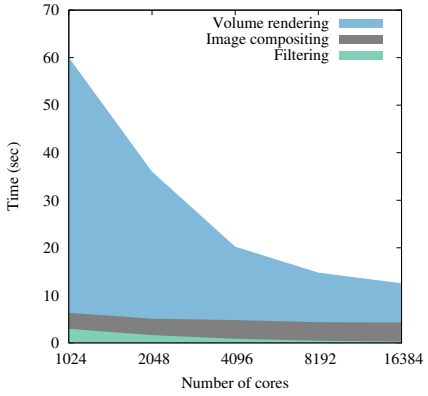

Figure 4: Scaling performance for median-filter, parallel volume rendering and image compositing using a k-way reduction dataflow with a dataset of size 2048x2048x2575 voxels.

4 USE CASE: NEUROSCIENCE

Usability and performance of our system is demonstrated through common neuroscience tasks such as filtering, registration and visualization of large scale NHP microscopy data. We have acquired 2P images of axons labeled with GFP (through intracortical injections of AAV9-GFP) and blood vessels labeled with Alexa594-conjugated tomato lectin through transparent Clarity-treated blocks (~60 mm^3) of macaque V1. Typically, an injection site of 1mm diameter in V2 produces a 7x4mm field of GFP-labeled axon terminals in V1 at several cortical depths, totaling a volume of about 60 mm^3. Imaging even a small fraction of this volume, i.e. 5 mm^3 at 0.5 μm z-resolution, takes several hours of continuous acquisition, generating approximately a terabyte of data. All the experiments presented in this section were performed using a leadership class supercomputer, Shaheen II, a Cray XC40 system with 6,174 dual socket compute nodes based on 16 core Intel Haswell processors with Aries Dragonfly connectivity to capture the parallel.

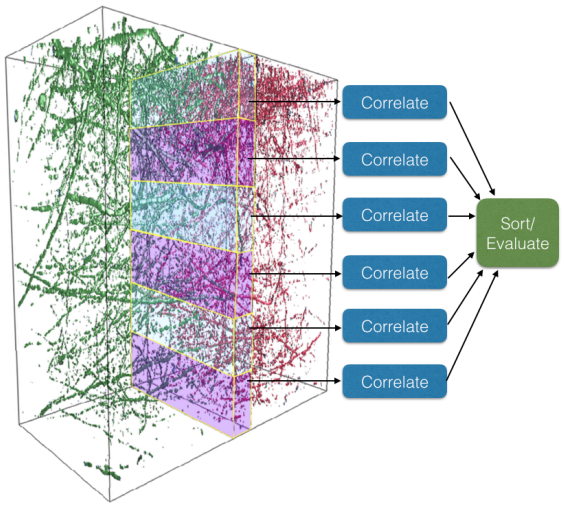

Figure 5: Decomposition of the overlapping sub-volumes used to register two adjacent volumes.

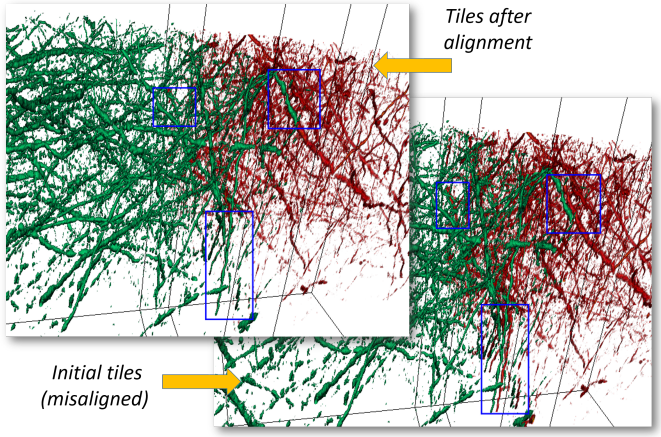

Figure 6: Slab of NHP neuronal data before and after alignment

4.1 Filtering

Often, algorithms can be characterized by data-parallel stencil operations, that is, operations that can be completed independently for each voxel given a small neighborhood. Common filtering algorithms that fit this model are minimum, maximum, average, blur, sharpen, edge detection, and deconvolution. Designing the right set of filters to use, their sequence, and parameters, is usually achieved through by trial-and-error in an interactive exploratory setting. Our software infrastructure enables a user to interactively test different filters available in the ViSUS framework or custom developed through the python interface with results instantaneously available for visualization in the ViSUS viewer. Fig. 3) illustrates the median-filter in action on a dataset containing a billion voxels.

Within our framework, defining a dataflow that contains a median filter in combination with a k-way reduction dataflow (i.e. a variant of the listed code in Listing 1 and 2) to execute volume rendering and reduction image composition at scale is extremely simple. The processing library automatically decomposes the input domain among the available tasks (i.e. depending on the number of cores and nodes requested) where each task reads only the sub-volume of interest for the computation. Note that due to the particular data layout, making queries with spatial locality in IDX is more efficient compared to others (e.g. row major order layout).

The rendering task uses the VTK [Schroeder et al. 2006] volume rendering (i.e. SmartVolumeRendering) to render a sub-volume of the data and the composition of the images done via a simple front-to-back ordering. Results in Fig. 4 show good scaling of the dataflow applied to a microscopy dataset of size 2048x2048x2575 voxels to produce a rendered image of size 2048x2048.

4.2 Registration

Image volumes of cleared brain tissue are created as a stack of 2D images taken at regular intervals (e.g. every 0.5 micron) on the z-axis. A single volume is acquired through the depth of the tissue at a given X,Y coordinate within the larger region of interest containing labeled cells. The scan then moves to the next X,Y coordinate (maintaining a 15% overlap with adjacent volumes to aid alignment in later steps) until the entire region of interest has been imaged. As the microscope finishes scanning one field of view and moves on to the next position, a range of movements causes the data to often be mis-aligned. To create a single 3D dataset encompassing the entire region of interest, each individual X,Y volume needs to be aligned using the overlapping fluorescent blood vessels between adjacent volumes (see Fig. 5).

To compute pairwise relative positions between adjacent volumes, we use 3D Normalized Cross Correlation (NCC) as a similarity metric. The alignment process is done in three steps:

(i) Decompose the volume into slabs along the Z-direction
(ii) For each corresponding slab of the two adjacent volumes, load the data in the overlapping region and compute pairwise relative position using 3D NCC
(iii) Choose the most reliable displacement corresponding to the largest NCC peak value and smallest NCC shape width of the peak

In order to scale to large volumes, contrary to sliding window based spatial correlation, we transform the data from spatial to frequency domain using the open source FFTW library [Frigo 1999] and compute the correlation in the frequency domain. This results in good speed up which is further exploited in the parallel setup. Finally, for an unbiased NCC, we use summed area tables to compute the local mean square energy required to normalize the correlation coefficient. Fig. 6 shows a slab before and after the alignment process. Seemingly, this strategy to align volumes makes it an ideal candidate for parallelization. We describe the parallel dataflow implementation next and provide details for optimal global positioning based on minimum spanning tree of the undirected weighted graph as a simple post processing step.

To perform the registration of multiple 3D volumes in parallel, we define a dedicated dataflow that uses 2D neighbour communication pattern (see Fig. 9) where individual alignment for a pair of

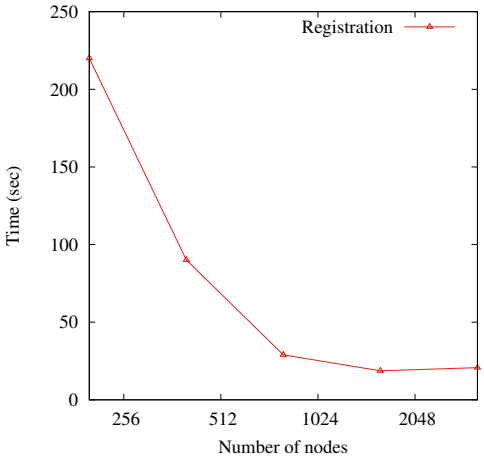

Figure 7: Scaling performance of registering 25 3D microscopy volumes, each of size 1024x1024x1024 voxels.

slabs is evaluated first. The results are then collected in an another task where the alignment that report the best correlation values will be used to compute the final global positions. To compute the optimal global positions, we find the minimum spanning tree of an undirected weighted graph where the nodes correspond to the volumes and the edges represent the pairwise relative positions with the best correlation value chosen during the previous step. We use the inverse of the correlation co-efficient (i.e. higher correlation co-efficient corresponds to lower weight of the edge in the graph) as the weights for the spanning tree computation. This way the resulting spanning tree will maximize the correlation factor among all the pairs in the graph. Fig. 8 shows the alignment results for a configuration of 4 volumes. In this particular example, the minimum spanning tree is given by the edges 0-2-3 that correspond to largest correlation factor. Finally the optimal global positions are used by the framework to update the position of each volume in the viewer.

Due to the high memory requirements of the correlation task, we restricted the number of cores per node to only 4 out of the 32 available cores. Figure 7 shows the results for up to 3200 nodes where the parallel execution exhibits strong scaling. It is important to notice that for the given domain decomposition, even at 3200 nodes, each registration task will correlate only 2 slices per slab. This means that at this scale the problem is over-decomposed and the workload of the registration algorithm becomes very small vis-a-vis the communication and runtime overhead which has a higher impact on the overall performance.

5 CONCLUSIONS

Massive amounts of scientific data are an increasing challenge for scientists and engineers. With rapidly growing data sizes, generation, distribution, analysis and visualization of the data requires specialized software infrastructures that (1) enables interactive visualization and exploration, (2) enables designing complex workflows in an interactive setting, and (3) scales the computation of those workflows to full-scale data efficiently utilizing HPC resources.

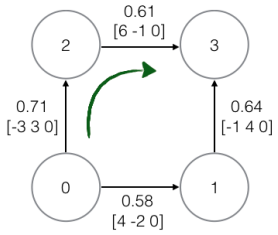

Figure 8: Undirected weighted graph of initial displacement for 4 1024x1024x1024 volumes. The nodes correspond to the volumes and the edges report the highest correlation factor and the displacement for each pair (in pixels).

Current solutions do not offer straight forward support for the definition and prototyping of visualization and analysis workflow that can be executed interactively or at scale. Furthermore, the user who intend to develop an algorithm for execution at scale is forced to deal with the complexity of parallel programming (i.e. communication, scheduling, resource management, portability, etc.) on HPC systems. Here, we present the first end-to-end software framework that simplifies interactive visualization and analysis of tera-scale datasets.

To enable interactive exploration of the data, the framework takes advantage of the multi-resolution IDX data format and the ViSUS streaming infrastructure. In this environment, we introduce a new library that creates an abstraction layer while separating the definition of the algorithm from actual implementation and execution. This library further allows the user to define visualization and analysis workflows as task graphs that can be executed on local resources for interactive analysis or at scale on HPC systems.

Our new approach enables developers to implement their algorithms without the knowledge of underlying communication primitives and resource allocation on different architectures or at different scales. Furthermore, this component provides flexibility to easily implement an algorithm, test it interactively on local resources using the data streaming infrastructure (i.e. ViSUS) or at scale on HPC resources using the full-scale resolution of the data.

We demonstrated how our infrastructure scales and simplifies three algorithms applied to large-scale neuroscience problems: rendering, de-noise filtering and 3D image registration showing strong scaling on the leadership class supercomputer, Shaheen II. The simplicity of use, choice of operating either locally or remotely on HPC systems, fast multi-resolution streaming infrastructure, parallel custom analytics and a simple python interface for rapid prototyping and analysis makes our infrastructure an ideal choice for research labs and engineering firms with massive data aiding in improving the user productivity and supporting scientists to find new insights and breakthroughs in their data.

ACKNOWLEDGMENTS

This work is supported in part by NSF: CGV: Award:1314896, NSF:IIP Award :1602127 NSF:OAC Office of Advanced Cyberinfrastructure (OAC): Award 1649923, DOE/SciDAC DESC0007446, CCMSC DE-NA0002375, and PIPER: ER26142 DE-SC0010498. This material is

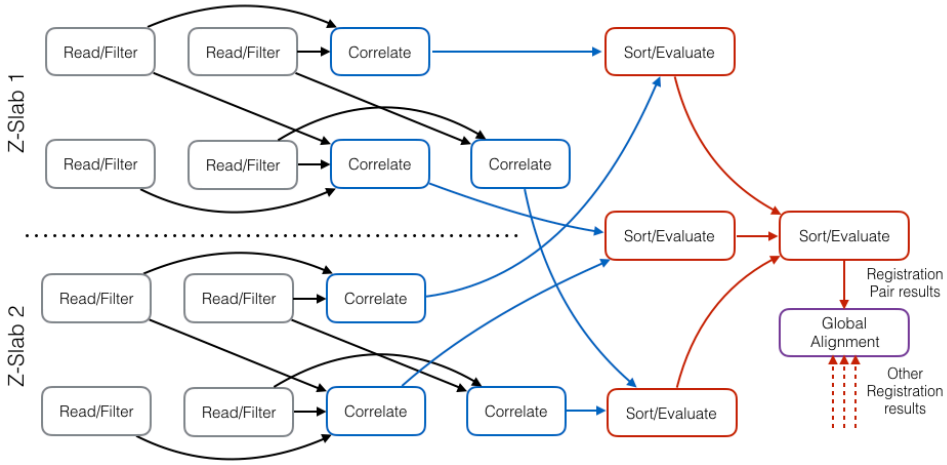

Figure 9: Registration dataflow: for each input volume a set of tasks reads one or more z-slabs in the overlapping region. These slabs (i.e., potentially filtered to remove noise) are then sent to the correlation tasks to perform the registration. The results of the registration are collected by another set of tasks (i.e., sort/evaluate) that will evaluate the final global position for each volume.

based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375. For computer time this research used the resources of the Supercomputing Laboratory at King Abdullah University of Science and Technology (KAUST) in Thuwal, Saudi Arabia. Thanks to Will Usher for the teaser image.

REFERENCES

James Ahrens, Berk Geveci, and Charles Law. 2005. Paraview: An end-user tool for large data visualization. *The Visualization Handbook* 717 (2005).

Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Cyrus Harrison, Gunther H. Weber, Hari Krishnan, Thomas Fogal, Allen Sanderson, Christoph Garth, E. Wes Bethel, David Camp, Oliver Rübel, Marc Durant, Jean M. Favre, and Paul Navrátil. 2012. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*. 357–372.

Cameron Christensen, Ji-Woo Lee, Shusen Liu, Peer-Timo Bremer, Giorgio Scorzelli, and Valerio Pascucci. 2016. Embedded domain-specific language and runtime system for progressive spatiotemporal data analysis and visualization. In *Large Data Analysis and Visualization (LDAV), 2016 IEEE 6th Symposium on*. IEEE, 1–10.

Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering* 5, 1 (1998), 46–55.

Amira FEI. 2016. 3D Software for Life Sciences. (2016).

Mike Folk, Albert Cheng, and Kim Yates. 1999. HDF5: A file format and I/O library for high performance computing applications. In *Proceedings of Supercomputing*, Vol. 99. 5–33.

Matteo Frigo. 1999. A fast Fourier transform compiler. In *Acm sigplan notices*, Vol. 34. ACM, 169–180.

Jacob R Glaser and Edmund M Glaser. 1990. Neuron imaging with Neurolucidaa PC-based system for image combining microscopy. *Computerized Medical Imaging and Graphics* 14, 5 (1990), 307–317.

William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press.

Moritz Helmstaedter, Kevin L Briggman, and Winfried Denk. 2011. High-accuracy neurite reconstruction for high-throughput neuroanatomy. *Nature neuroscience* 14, 8 (2011), 1081–1088.

Laxmikant V Kale and Gengbin Zheng. 2009. Charm++ and AMPI: Adaptive runtime strategies via migratable objects. *Advanced Computational Infrastructures for Parallel and Distributed Applications* (2009), 265–282.

Sidharth Kumar, Cameron Christensen, JohnA. Schmidt, Peer-Timo Bremer, Eric Brugger, Venkatram Vishwanath, Philip Carns, Hemanth Kolla, Ray Grout, Jacqueline Chen, Martin Berzins, Giorgio Scorzelli, and Valerio Pascucci. 2014. Fast Multiresolution Reads of Massive Simulation Datasets. In *Supercomputing*, Julian-Martin Kunkel, Thomas Ludwig, and HansWerner Meuer (Eds.). Lecture Notes in Computer Science, Vol. 8488. Springer International Publishing, 314–330. DOI: https://doi.org/10.1007/978-3-319-07518-1_20

S. Kumar, V. Vishwanath, P. Carns, J.A Levine, R. Latham, G. Scorzelli, H. Kolla, R. Grout, R. Ross, M.E. Papka, J. Chen, and V. Pascucci. 2012. Efficient data restructuring and aggregation for I/O acceleration in PIDX. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. 1–11. DOI: https://doi.org/10.1109/SC.2012.54

Sidharth Kumar, Venkatram Vishwanath, Philip Carns, Brian Summa, Giorgio Scorzelli, Valerio Pascucci, Robert Ross, Jacqueline Chen, Hemanth Kolla, and Ray Grout. 2011. PIDX: Efficient Parallel I/O for Multi-resolution Multi-dimensional Scientific Datasets. In *IEEE International Conference on Cluster Computing*.

Jianwei Li, Wei-Keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. 2003. Parallel netCDF: A High-Performance Scientific I/O Interface. In *Proceedings of SC2003: High Performance Networking and Computing*. IEEE Computer Society Press, Phoenix, AZ.

J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. 2008. 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, CLADE '08*. ACM, New York, 15–24.

Valerio Pascucci and Randall J Frank. 2001. Global static indexing for real-time exploration of very large regular grids. In *Supercomputing, ACM/IEEE 2001 Conference*. IEEE, 45–45.

V. Pascucci, G. Scorzelli, B. Summa, P.-T. Bremer, A. Gyulassy, C. Christensen, S. Philip, and S. Kumar. 2012. The ViSUS Visualization Framework. In *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, E Wes Bethel, Hank Childs, and Charles Hansen (Eds.). CRC Press.

Hanchuan Peng, Zongcai Ruan, Fuhui Long, Julie H Simpson, and Eugene W Myers. 2010. V3D enables real-time 3D visualization and quantitative analysis of large-scale biological image data sets. *Nature biotechnology* 28, 4 (2010), 348–353.

PerkinElmer. 1998–2014. Volocity. http://www.perkinelmer.com/. (1998–2014).

Johannes Schindelin, Ignacio Arganda-Carreras, Erwin Frise, Verena Kaynig, Mark Longair, Tobias Pietzsch, Stephan Preibisch, Curtis Rueden, Stephan Saalfeld, Benjamin Schmid, and others. 2012. Fiji: an open-source platform for biological-image analysis. *Nature methods* 9, 7 (2012), 676–682.

Will Schroeder, Ken Martin, and Bill Lorensen. 2006. The Visualization Toolkit 4th Edition. (2006).

I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navratil. 2017. OSPRay - A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan 2017), 931–940. DOI: https://doi.org/10.1109/TVCG.2016.2599041