PaDaWAn: a Python Infrastructure for Loosely Coupled In Situ Workflows

Julien Capul, Sébastien Morais, Jacques-Bernard Lekien

PaDaWAn: a Python Infrastructure for Loosely Coupled In Situ Workflows

Julien Capul, Sébastien Morais, Jacques-Bernard Lekien
CEA, DAM, DIF
F-91297 Arpajon, France
{julien.capul,sebastien.morais,jacques-bernard.lekien}@cea.fr

ABSTRACT

This paper presents PaDaWAn, an infrastructure written in Python to provide loosely coupled in situ capabilities to accelerate file-based simulation workflows. It provides services for in-memory data exchange between applications and a simple configuration model to switch from a file-based workflow to a loosely coupled in situ workflow. The infrastructure is currently based on CEA-DAM Hercule parallel I/O library by providing an ABI-compatible library to intercept simulation data in a transparent way and to facilitate integration into existing simulation codes and tools. PaDaWAn implements a producer-consumer pattern with buffering of data in an in-memory staging service with automatic memory management and running on dedicated resources. We describe the key design decisions and main architectural features, and share the lessons learned from the development of the infrastructure and from setting up test runs on two production-like workflow cases. We conclude on the perspectives for our infrastructure.

CCS CONCEPTS

• **Software and its engineering** → **Publish-subscribe / event-based architectures**; **Software libraries and repositories**; • **Human-centered computing** → *Scientific visualization*; *Visualization theory, concepts and paradigms*;

KEYWORDS

PaDaWAn, In situ, In transit, I/O, Workflow, Python, HPC, Simulation, Coupling

ACM Reference Format:
Julien Capul, Sébastien Morais, Jacques-Bernard Lekien. 2018. PaDaWAn: a Python Infrastructure for Loosely Coupled In Situ Workflows. In *ISAV: In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV '18), November 12, 2018, Dallas, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3281464.3281470

1 INTRODUCTION

In this paper we describe PaDaWAn (for Parallel Data Workflow for Analysis), an experimental infrastructure written in Python to provide loosely coupled in situ workflow capabilities.

The project was initiated as a software engineering experimentation to address the I/O bottleneck challenges expected on future classes of supercomputers. Two specific use cases were identified for driving the design of the solution: 1) asynchronous I/O and high resolution movie images processing executed in parallel with an AMR hydrodynamic simulation, 2) in-memory buffering of data produced by a multi-physics simulation and consumed by many parametric instances of a second simulation scheduled in parallel.

The objectives were to 1) alleviate the burden on the I/O subsystem, 2) increase the output data frequency, 3) limit the persistent storage space used and 4) accelerate studies by reducing the time spent in I/O operations and by running all applications in parallel.

PaDaWAn infrastructure features an in-memory data staging service running on a set of dedicated resources, a controller service managing queues and synchronization between data producer and consumer applications, and a workflow launching command-line utility based on a simple configuration model. The infrastructure client is meant to be lightweight and to integrate seamlessly into our simulation codes. The whole infrastructure code base is written in Python with client bindings in C, C++ and Fortran.

The main contribution of this paper besides providing an overview of the infrastructure design, architecture and implementation is to share the lessons learned from setting up in practice an in situ workflow, to reflect on the design choices made and to expose the challenges of effectively integrating legacy applications into in situ workflows in general.

The paper is organized as follows: Section 2 describes our design decisions and the related works. Section 3 provides an architectural overview and some implementation details. Section 4 presents the results obtained. Section 5 details the lessons learned. Finally Section 6 provides a conclusion and perspectives for our infrastructure.

2 DESIGN DECISIONS AND RELATED WORK

2.1 Lightweight loosely coupled in situ

The use cases described in Introduction involve coupling heterogeneous applications possibly written in different languages and with differing needs in terms of resources, environment and execution runtime. Considering the flexibility required to accommodate these constraints we adopted a **lightweight loosely coupled in situ approach**. Loosely coupled in situ as defined in [10] (also referred as *in transit*) consists in extracting data produced by the simulation and sending it to separate resources for further processing. This approach is used in DataSpaces [6], GLEAN [18], Damaris [7] and Decaf [8]. The lightweight aspect comes from the infrastructure client library which single purpose is to extract and send simulation data to the staging service as fast as possible without performing any complex processing.

As advocated by authors of [10], this approach presents several advantages over the tightly coupled in situ alternative used by

ParaView Catalyst [2], VisIt Libsim [11] or ALPINE Ascent [13] in which data processing runs synchronously with the simulation in the same process. In our case, the key advantages that have driven this decision are:

Limited impact on the simulation: as data consumer applications run on separate processes and potentially on distinct nodes, the impact on the simulation is limited to the infrastructure client library the simulation links to. When a data array is passed to the client library, the simulation blocks and a minimum-copy transfer to the staging area is immediately performed, hence limiting the amount of share memory used. This allows to provide a lightweight client library reducing the risk of introducing new potential failure points into the simulation code.

Resources flexibility: the loosely coupled in situ approach provides the flexibility to adapt efficiently the resources required by the different applications in the workflow. In particular, data consuming applications can run a wider range of in situ processing algorithms including those requiring persistent state (e.g. multiple time steps, see Melissa [16]). It also provides the flexibility to accommodate the future trend of architectural and platform heterogeneity (e.g. mixing compute, high-memory, GPU and I/O nodes in the same workflow execution).

Facilitated integration: the infrastructure client library is a lightweight library to link to. It does not embed any routine execution runtime thus limiting the risk of dependencies inconsistencies with the simulation.

2.2 Garbage-collected in-memory data staging service

Our in situ workflow model can be characterized as a *semi-dynamic batch workflow* in which the number of producers and consumers is statically defined at startup but these can be spawn dynamically and connect to PaDaWAn infrastructure at any time. To accommodate this model and handle late-joining consumers, we adopted an intermediate in-memory staging area to buffer data streams. Memory in the staging area is automatically managed through a simple garbage collection scheme that keeps count of the number of remaining consumers to process buffered data. This is made possible by knowing ahead the number of consumers and setting the maximum count at startup.

We also opted to have this data staging service as a single-user service co-scheduled with the user workflow applications rather than being a permanent facility-wide service. The objectives were to simplify the development and to allow the user to fine-tune the staging memory required through an option in the configuration system, see section 3.4.

Similar staging service is used in DataSpaces when combined with the DART [5] communication library. Memory space in the staging area is managed differently though, relying on a configurable number of data versions with automatic eviction in FIFO manner and API calls for applications to explicitly manage locks on data. GLEAN also relies on staging nodes however they are only used for asynchronous I/O to the file systems.

2.3 Socket-based data transport

Different data transport mechanisms have been used in related projects. One is based on MPI and uses a global communicator shared between all applications in the workflow. These applications are launched either within a single MPI program like Damaris or through MPMD launching capabilities of MPI like Decaf. The DART communication library used in DataSpaces is based on RDMA and rely on low level access to supercomputer interconnect to seek raw data movement performances.

Similar to GLEAN and Melissa, our data transport infrastructure is based on TCP socket communications instead. This decision allows to simplify the development of distributed applications by leveraging external libraries to implement reliable patterns and by the capacity to easily handle connection / disconnection and manage failure of one task without compromising the whole system (unlike most MPI implementations). Performance wise, this decision was backed by the fact that we expected fairly good performances using high-speed interconnect through the IP over interconnect layer (e.g. IPoIB).

Using sockets also presents the advantage of portability and can ease the integration of non-MPI or non-HPC tools and services to use the infrastructure with the potential of crossing the barriers of a compute cluster or a computing facility. This is the case for GLEAN which staging service is hosted on a separate cluster that can only communicate with the compute cluster through sockets.

2.4 High level I/O layer coupling

The coupling strategy is based on intercepting and exchanging data produced by the simulation at the parallel I/O library level. We have developed a library ABI-compatible with Hercule [4], a contract-based high performance parallel I/O library developed at CEA-DAM and used by our main simulation codes and post-processing tools. This ABI-compatible library captures and transfers simulation data to PaDaWAn infrastructure. This is intended to facilitate integration into existing simulation codes and processing tools that use Hercule but required the development of specific mappings between Hercule API and PaDaWAn internal API.

With this strategy it is transparent to the simulation whether data are written on disks or sent through the network. This is expected to simplify the setup of an in situ workflow by first testing the corresponding file-based workflow. Then PaDaWAn aims at providing a simple configuration and launching utility (see Section 3) to execute the workflow in situ without the need to modify the simulation code.

This strategy also permits to capture and leverage high-level and metadata-rich description of data arrays (as provided by our I/O library interface) to allow "smarter" filtering, dispatching and processing along the data path.

This approach is at the core of ADIOS [14] which decouples the I/O interface from the data transport implementation. GLEAN also uses pNetCDF and HDF5 ABI-compatible libraries to facilitate integration into existing codes.

2.5 Python as main development language

The key distinguishing design decision was to chose Python as main development language. The primary objective was to accelerate development time and facilitate experimentation of various implementations by leveraging the large ecosystem and community surrounding the language.

We also wanted to demonstrate the ability of the language to run efficiently in an HPC production environment, notably the capability of embedding Python through its C API and the ability of Numpy to wrap arrays without copy. As PaDaWAn is also essentially a data transfer infrastructure coordinating network I/O operations and does not perform any complex computation on data nor provide a routine execution runtime, Python seemed a reasonable choice in terms of performance.

Last, having an infrastructure natively written in Python (and using Numpy) allows to deliver more rapidly to a scientific user community heavily relying on Python/Numpy for accessing and processing simulation data.

3 ARCHITECTURE AND IMPLEMENTATION DETAILS

3.1 Data containers and API

The Hercule library uses the concepts of *collections* and *records* as data containers. Simulation codes may produce different collections (e.g. post-processing output, checkpoints, coupling data) consisting in a set of records indexed by the MPI rank of the process emitting the record and by the current simulation time step. Each record is self-describing and contains data arrays and metadata (properties, relationships and contract-based semantic information). Hercule API can be summarized as: *open/close_collection*, *open/close_record*, plus several methods to fill the record with data arrays and metadata.

In addition to this API, the provided Hercule ABI-compatible client library includes a record iterator method for consumer applications. This additional service requires some modifications in existing codes when executed as consumers.

This client library is written in C and uses the Python C API to maps Hercule methods to PaDaWAn Python methods. C++ and Fortran bindings are also available.

3.2 Data staging service

The data staging service is a distributed in-memory key-value store running on a dedicated set of cores or nodes. The client and server codes are written in Python and use *ZeroMQ* [19] library to handle TCP communications. The server stores data in memory in a Python dictionary and features a thread-pool executor to handle multiple connections. Compression is available in option with *blosc* [3]. A composite client can manage multiple server instances by simply dispatching/fetching record data based on the record index (the MPI rank of the record producer is used to distribute records evenly among the server instances). Overall, the implementation requires less than 500 lines of Python code.

3.3 Producers/consumers coordination

Data exchange and synchronization between producers and consumers is illustrated on Figure 1.

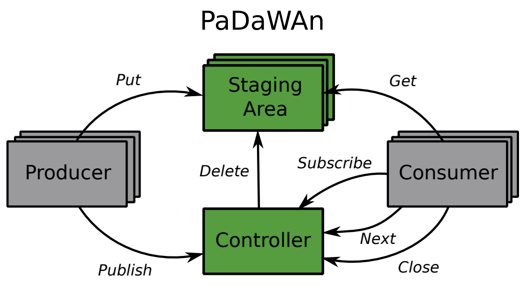

Figure 1: PaDaWAn components coordination

It implements a publisher/subscriber pattern that manages late-joining subscribers by buffering raw data in the staging area and managing record queues (one for each subscriber) in a *controller* process.

During an I/O step of a particular collection, the simulation opens a record and fills it with data arrays and metadata. The PaDaWAn client transfers immediately these data arrays into the staging area and replaces the array in the record structure by its key in the staging area. When the simulation closes the record, the client publishes the record ID to the controller which pushes it in each consumer's queue.

On consumer side, applications subscribe to a collection and retrieve records one by one through the use of an iterator. Data arrays within the record are only fetched from the staging area if requested. When processing a record is completed, a call to the *close* method notifies the controller which then decrements the consumer reference count for this record. When this reference count reaches zero, the controller triggers deletion of all its data arrays in the staging area.

A simple back-pressure mechanism is implemented through a configurable fixed-size queue in the controller which blocks the producer if the maximum number of elements in the queue is reached.

The controller is written in Python and uses a single-threaded coroutine-based mechanism provided by the Tornado library [17].

3.4 Workflow configuration and launching

PaDaWAn provides a simple in situ workflow configuration and launching mechanism. The workflow is described in a configuration file as illustrated in the example script on Figure 2.

In this illustrating example the configuration file is embedded in a SLURM sbatch script which concludes by running the PaDaWAn workflow launcher utility. This utility first launches the staging service instances and the controller then executes all application commands.

Note that the collection is uniquely identified by a file system path even though no file is written during the execution of the in situ workflow. This way no changes are required in the application code, its input data or its execution command when run in situ.

```
#!/bin/sh
#SBATCH -N 13
#SBATCH --exclusive

cat > workflow.cfg << EOD
[padawan]
staging_instances = 2
collections = output
applications = simulation, processing, writer

[output]
path = /path/to/output.herc
nb_consumers = 2

[simulation]
cmd = /path/to/run_simulation.sh -N 8 -n 256 --output /path/to/output.herc

[processing]
cmd = srun -N2 -n64 /path/to/processing.sh --file /path/to/output.herc

[writer]
cmd = /path/to/hercule_writer --hercule_path /path/to/output.herc
EOD

module load padawan
padawan_launcher workflow.cfg
```

Figure 2: Example of a PaDaWAn workflow script

4 EARLY RESULTS

The two use cases described in Introduction served as early evaluation test runs of PaDaWAn infrastructure. In the following, we provide details about the first use case that may hint at the potential of the presented solution. As the second use case was only run on a low scale for functional testing purpose, we deemed the results not relevant for the present paper. Even tough these test runs did not represent large scale runs and we acknowledge that a thorough scalability study remains to be performed, the process of setting up these two cases proved to be very valuable in terms of lessons learned which is one of the main purpose of this paper.

The first use case consisted in executing successive runs of an in situ workflow composed of 1) a shockwave propagation simulation using the hydrodynamic AMR code HERA [9], 2) a parallel graphic processing of the simulation output data to extract iso-surfaces and produce scripted movie images, and 3) the asynchronous writing of simulation checkpoint data. The image processing was performed by LOVE [1], a domain-oriented parallel visualization tool based on VTK/ParaView developed at CEA-DAM. The writing of checkpoint data was done by a parallel writer specifically developed to interface with PaDaWAn infrastructure and use Hercule file format. The objective was to assess the capability of the infrastructure to withstand an order of magnitude increase in the simulation output frequency compared to the same file-based workflow, thus producing a movie with a much higher resolution and saving a significant amount of disk space by not persisting intermediate data.

The in situ workflow was executed on Tera-1000 and used a total of 97 nodes, including 80 nodes for the simulation, 8 nodes for LOVE, 1 node for the Hercule writers and 8 nodes for PaDaWAn in-memory staging (totaling 1TB of aggregated staging memory). The AMR mesh size grew from approx. 200 million cells to almost 1.3 billion cells at the end of the simulation. At each I/O step, the simulation normally writes several dozens of field quantities over the entire mesh.

Compared to the file-based approach, we increased the output frequency by a factor of 11, totaling 447 saved time steps and about 50TB of intermediate data that were streamed through PaDaWAn in-memory infrastructure instead of being written on disks. In terms of output data, only 1.5GB of final images were written on disks. We also observed a x2.5 speed up on average for the time spent in I/O by the simulation. This can be explained by faster data transfers to the distributed staging area than parallel writes in a small set of files that necessarily incurs additional synchronization and communications between simulation processes.

Even though this workflow was not optimized and filtering on the simulation side could save a large amount of data transfer, this test run demonstrated the potential of the infrastructure to sustain a fairly decent load.

5 LESSONS LEARNED AND DISCUSSION

5.1 Design decisions

Python as main development language. The use of Python and its large and versatile ecosystem proved to be very convenient for development and allowed to deliver functional solutions quickly. A good example is the distributed in-memory staging service developed with less than 500 lines of code and which sustained almost 1TB of data in distributed Python dictionaries. In terms of performance, as the infrastructure only manages data transport, staging and synchronization (no computation) the overhead due to interpretation had no noticeable impact in our cases.

However, the use of Python in this context presents several downsides. For larger scale workflows, performance issues may arise at startup when many processes hit the NFS at the same time to import Python modules, several solutions exists though (e.g. see [12]). A single Python server may also not scale for larger simulations (expected 10,000+ clients) without re-designing (e.g. tiering). On client side, the use of Python adds a dependency that may interfere with other Python versions that the simulation code may use. Also, embedding Python in a compiled library makes debugging harder.

Socket-based data transport. The use of TCP sockets through the ZeroMQ library and using IPoIB proved to be an efficient solution in terms of development time over performance ratio. The observed speed up on I/O steps was satisfying enough to retain this solution for our use cases. This choice would also allow to manage disconnections and failures of distributed components though developments remain to be done in PaDaWAn to implement these specific fault tolerance capabilities.

5.2 Architecture

One may argue that separating staging resources from processing resources incurs data movements that may not be sustainable at exascale era. This architecture choice is a trade-off to have the flexibility required to accommodate any kind of application, not restricted by the staging service runtime or its allocated resources. We plan to limit data movement upstream by implementing "smart" filtering on producer-side. This is particularly relevant for batch workflows in which datasets produced by the simulation but not consumed downstream are known a priori and thus can be discarded. Combined with an access to high-level data and metadata, powerful filtering may be introduced.

It would also be possible as an option to request the job scheduler to place consumer tasks on the same nodes as the staging service and implement some form of inter-process or shared-memory communication to exchange data. An alternative could be to attach a pool worker processes to the staging service to localize certain processing tasks with the data. However, the adopted architectural approach also promotes modularity by isolating functionalities as independent services, thus simplifying development, experimentation and deployment. It may also anticipate future trend of mixing heterogeneous hardware in a job (e.g. mixing large memory nodes, compute nodes or GPU nodes).

With respect to scalability, even though the test runs presented before proved to be satisfying and will allow a fair amount of production workflows to be run, a thorough scaling investigation remains to be performed to identify bottlenecks in the architecture and extend PaDaWAn's capabilities to the largest simulation and workflow scales. Some potential contention points have already been identified in our current use of Python for instance. The use of a single controller process to manage the infrastructure metadata may also be a point of concern depending on the types of workload.

5.3 In situ workflows in practice

Although the infrastructure was designed to facilitate integration, not all applications readily fit with the in situ paradigm. In file-based workflows, applications are black boxes with distinct and sequential input-compute-checkpoint-output steps. For in situ workflows which are distributed and concurrent by nature, applications must accommodate data streams and interleave the execution of these steps for efficiency. For instance in our two test cases, the downstream applications (the visualization tool and the second simulation code) had to be refactored to use the record iterator described in section 3.1 to ingest records streams.

Stateful applications must also coordinate their checkpointing to allow globally consistent checkpoints accross the workflow. The basic strategy employed in our second test case (the graphic processing in our first test case was stateless) was to have the upstream simulation to add a checkpoint flag in its output data and have downstream applications react to this flag and trigger checkpointing.

Another set of challenges originated from the inability of the supercomputer job scheduler to co-schedule multiple and isolated jobs. Instead a global allocation must be estimated (adding new source of errors) based on the requirement of each in situ workflow application and PaDaWAn service. These services and applications are then run as steps within this single global job. In our test cases, this generated multiple runtime errors as simulation codes were originally meant to run as the main job application. Modifications were required in some codes and in their execution scripts to isolate their use of shared space on the file system (for instance, to use a step-specific temporary folder rather than muddling in the job-wide temporary folder provisioned by the scheduling service). The placement of tasks on the globally allocated resources also happened to be an issue which required further adaptation of some simulation execution scripts. Debugging these runtime errors was made difficult due to the distributed nature of in situ workflows and required a heavy and hygienic use of logs.

One takeaway is that, despite the apparent user-friendliness of the approach, branching existing simulation codes and tools to such infrastructure and setting up the first time an in situ workflow may still be a complex task involving some development and debugging skills. However, once this initial phase is achieved, simulation end users would be able to compose themselves their in situ workflows from existing file-based workflows using the simple configuration mechanism provided.

6 CONCLUSIONS AND PERSPECTIVES

PaDaWAn is a software infrastructure providing services to run file-based workflows in a loosely couple in situ way. Its originality is the use of Python as main development language and the simple means provided to configure and switch from a file-based workflow to an in situ workflow. It has been successfully tested on some production-like workflow use cases.

It is still in development stage and requires further evaluation and proofing, in particular with respect to scalability to larger simulation and workflow scales. We also want to explore alternative service implementations, for instance leveraging existing in-memory stores like REDIS [15] for the staging infrastructure. We plan to extend its compatibility with other I/O libraries such as HDF5, NetCDF or ADIOS and develop writers in different file formats. Finally, as PaDaWAn is currently a write-oriented infrastructure (pipeline/streaming mode), we plan to extend it to provide accelerated read capabilities by using the in-memory staging in cache mode and explore "smart" data loading mechanisms.

REFERENCES

[1] D Aguilera, Th Carrard, C Guilbaud, J Schneider, and S Sorbet. 2012. Visualization and post-processing for high performance computing. *Chocs* 41 (2012), 57–67.

[2] Utkarsh Ayachit, Andrew Bauer, Berk Geveci, Patrick O'Leary, Kenneth Moreland, Nathan Fabian, and Jeffrey Mauldin. 2015. Paraview catalyst: Enabling in situ data analysis and visualization. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization.* ACM, 25–29.

[3] blosc. [n. d.]. http://blosc.org

[4] O Bressand, L Colombet, A Fontaine, G Harel, and J-B Lekien. 2012. Hercule: A library of scientific data management for numerical simulation. *Chocs* 41 (2012), 29–37.

[5] Ciprian Docan, Manish Parashar, and Scott Klasky. 2010. Enabling high-speed asynchronous data extraction and transfer using DART. *Concurrency and Computation: Practice and Experience* 22, 9 (2010), 1181–1204.

[6] Ciprian Docan, Manish Parashar, and Scott Klasky. 2012. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing* 15, 2 (2012), 163–181.

[7] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, and Leigh Orf. 2012. Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on.* IEEE, 155–163.

[8] Matthieu Dreher and Tom Peterka. 2017. *Decaf: Decoupled dataflows for in situ high-performance workflows.* Technical Report. Argonne National Lab.(ANL), Argonne, IL (United States).

[9] Hervé Jourdren. 2005. HERA: a hydrodynamic AMR platform for multi-physics simulations. In *Adaptive Mesh Refinement-Theory and Applications.* Springer, 283–294.

[10] James Kress, Scott Klasky, Norbert Podhorszki, Jong Choi, Hank Childs, and David Pugmire. 2015. Loosely Coupled In Situ Visualization: A Perspective on Why It's Here to Stay. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization.* ACM, 1–6.

[11] T Kuhlen, R Pajarola, and K Zhou. 2011. 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).*

[12] Steven H Langer, Brian Spears, J Luc Peterson, John E Field, Ryan Nora, and Scott Brandon. 2016. A HYDRA UQ workflow for NIF ignition experiments. In *In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV), Workshop on.* IEEE, 1–6.

[13] Matthew Larsen, James Ahrens, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus Harrison. 2017. The ALPINE In Situ Infrastructure: Ascending

from the Ashes of Strawman. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*. ACM, 42–46.

[14] Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, et al. 2014. Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks. *Concurrency and Computation: Practice and Experience* 26, 7 (2014), 1453–1473.

[15] Redis. [n. d.]. https://redis.io

[16] Théophile Terraz, Alejandro Ribes, Yvan Fournier, Bertrand Iooss, and Bruno Raffin. 2017. Melissa: large scale in transit sensitivity analysis avoiding intermediate files. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 61.

[17] Tornado. [n. d.]. http://www.tornadoweb.org

[18] Venkatram Vishwanath, Mark Hereld, Vitali Morozov, and Michael E Papka. 2011. Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 19.

[19] ZeroMQ. [n. d.]. http://zeromq.org