# Key technologies for fast data access and retrieval (D7.2.1 – SGA2)


**Figure 1: Pilot system JURON**

**The pilot system JURON has been used as testbed for most of the research and developments described in this Deliverable. JURON has been developed by IBM and NVIDIA as part of a Pre-Commercial Procurement during the HBP Ramp-up Phase. It is located at Jülich Supercomputing Centre (JUELICH-JSC).**

| Project Number: | 785907 | Project Title: | Human Brain Project SGA2 |
|---|---|---|---|

| | |
|---|---|
| Document Title: | Key technologies for fast data access and retrieval |
| Document Filename: | D7.2.1 (D43.1 D82) SGA2 M12 ACCEPTED 200731.docx |
| Deliverable Number: | SGA2 D7.2.1 (D43.1, D82) |
| Deliverable Type: | Report |
| Work Packages: | WP7.2 |
| Dissemination Level: | PU = Public |
| Planned Delivery Date: | SGA2 M12 / 31 Mar 2019, Request for revision: 22 Jul 2019 |
| Actual Delivery Date: | Submitted: SGA2 M12 / 08 Mar 2019; Resubmitted: SGA2 M19 / 24 Oct 2019; Accepted: 31 Jul 2020 |
| Author(s): | Thorsten HATER, JUELICH (P20) |
| Compiled by: | Thorsten HATER, JUELICH (P20)<br>Anna LÜHRS, JUELICH (P20), SP Manager (also SP7-internal review) |
| Contributor(s): | Cesare CUGNASCO, BSC (P5)<br>Thorsten HATER, JUELICH (P20)<br>Stefanie LACKNER, TUDA (P85)<br>Lena ODEN, JUELICH (P20)<br>Pol SANTAMARIA, BSC (P5) |
| SciTechCoord Review: | |
| Editorial Review: | Guy WILLIS, EPFL (P1) |
| Description in GA: | This report summarises key technologies to support fast data access and retrieval for interactive analysis and visualisation, which have been identified and tested. This Deliverable does not yet comprise the final implementation of these technologies. |
| Abstract: | This Deliverable represents the current status of the ongoing work in Task T7.2.3 of HBP SGA2, focusing on I/O behaviour and optimisation of applications in the context of the HBP. We present the use cases and technologies used in our work, then proceed to the analysis and conclude with our plans for the next phase of SGA2. |
| Keywords: | High-Performance Analytics and Computing Platform, High-Performance Computing, Interactive Analysis, Fast Data Access, HDF5, Hecuba, Distributed Shared Storage, Universal Data Junction, Conduit, Slurm, JURON |
| Target Users/Readers: | Computational neuroscience community, neuroimaging community, HPC community, HPC industry |

## Table of Contents



## Table of Tables



## Table of Figures

## History of Changes made to this Deliverable (post Submission)

| Date | Change Requested / Change Made / Other Action |
|---|---|
| 8 Mar 2019 | Deliverable submitted to EC |
| 22 Jul 2019 | Resubmission with specified changes requested in Review Report<br>Main changes requested:<br>• Change 1 (use cases): "The two chosen image segmentation use-cases are only partially reflecting the complexity of methods and tasks of the HBP. The generality and transferability of the insights from these use-cases in respect to other tasks should be clarified: the selected cell segmentation usecase is not convincing as it is implementing a very simple and unreliable method – generalization of the parallelization and I/O strategy to more complex methods remains unaddressed. The label propagation use-case is insufficiently described: it remains unclear if the use-case refers to the network training or image labelling or both."<br>• Change 2 (HW infrastructure): "It is unclear what underlying HW infrastructure is used to perform deep learning."<br>• Change 3 (deep learning SoA): "A comparison with or usage of state of the art deep learning libraries and their data I/O support is not provided. In particular, Tensorflow provides already extensive parallelization and data I/O strategies for both GPU and CPU (and mixed) functionality." |
| 17 Oct 2019 | Revised draft sent by SP/CDP to PCO.<br>Main changes made, with indication where each change was made:<br>Change 1: see Section 2<br>Change 2: see (new) Section 4 instead of the old Annex 2<br>Change 3: see Section 2.2.2 |
| 24 Oct 2019 | Revised version resubmitted to EC by PCO via SyGMa |

# 1. Introduction

This Deliverable represents the current status of the ongoing work in Task T7.2.3 of HBP SGA2, focusing on I/O behaviour and optimisation of applications in the context of the HBP. We present the use cases and technologies used in our work, then proceed to the analysis and conclude with our plans for the next phase of SGA2.

# 2. Use cases

We selected use cases from the HBP that were mature enough and already in production use at the beginning of SGA2, while still posing interesting challenges regarding their I/O behaviour. We did (and do) not consider changes to algorithms and methods, unless relevant for data access. These choices result in a set of applications weighted towards image processing, especially concerning large-scale files resulting from microscopic scans of brain sections. The results obtained from working with these image-processing applications can be translated into the following three general access patterns:

1) regular access of fixed size, e.g. two-dimensional tiles of an image,
2) random access of fixed size, including selecting overlapping units, and
3) coordinated writes of variable sized blocks.

These patterns cover a large portion of the design space, especially concerning machine learning, which tends to be balanced towards reading data.

In addition, we chose NEST as a demonstrator for in-transit analysis of simulation data. This particular area requires processing of streaming data for interactive visualisation and analysis.

## 2.1 NEST

*Collaboration with JUELICH-INM-6, JUELICH-JSC and NMBU in SP6 and SP7*

NEST is a spiking neural network simulator for models based on point neurons. Simulation proceeds in a series of discrete time steps, during which the neurons evolve independently. Due to the complexity of the emerging behaviours, there is an interest in interactive steering of simulations. This allows researchers to explore interesting phenomena in situ. Furthermore, multi-scale simulations are considered, where NEST provides input to more detailed models and/or receives inputs from more coarse-grained simulations.

We investigated Universal Data Junction (UDJ) as a technology for this scenario. The test case uses a setup of $N$ neurons and $0.1 \cdot N$ synapses per neuron or $0.1 \cdot N^2$ synapses in total. We expect roughly $f \cdot \Delta t$ spikes per simulation step, where $\Delta t = 0.1s$ and $f = 10s^{-1}$. Voltage measurements are produced at the same rate per recorder. This results in small transfers, and implementations will have to be optimised for this case.

## 2.2 HBP Brain Atlas

The HBP Human Brain Atlas aims to offer an interactive, annotated virtual map of the human brain. The main type of data ingested is in the form of microscopic scans of preserved post-mortem human brains at a resolution of 1 µm, called a "section". This results in a data volume of about 50 TB per fully scanned brain, which is then further processed and enriched by annotations.

These augmentations are bundled with the original data and presented to the user interactively, using a web interface. To match the production rate of raw data, processing has to be largely automatic and high-performance. Analyses may build on-top of prior steps, effectively forming a directed graph.

The Brain Atlas analyses pose an interesting challenge, due to the focus on extremely large images which have to be processed in small tiles. This makes high-performance I/O and extensive use of caching a necessity. We selected two of these for our analysis: Cell Segmentation and Label Propagation. The former is quite simple, while exhibiting an important prototypical I/O pattern, which allows rapid exploration of I/O strategies and technologies. The latter use case is more complex and accesses data in randomly selected tiles with a high degree of concurrency.

### 2.2.1 Cell Segmentation

*Collaboration with JUELICH-INM-1 in SP2 and SP5*

The Cell Segmentation code identifies cells in large-scale brain images using classical image processing. The output comprises the boundaries, centroids and surface area of the recognised cells. Subsequent steps in the workflow may filter the cells in a region, based on their bounding box, to compute the cell density in particular areas of interest.

A watershed algorithm, provided by the well-known OpenCV [1] library, is utilised for the segmentation. While considered the standard for image processing, OpenCV offers no facilities for parallelisation across processes or specialised (in particular: caching I/O). Consequently, the algorithm has been manually parallelised using the Message Passing Interface (MPI). The image is split into square tiles, which are then processed independently. The tiles are extended with an additional halo of 100 pixels to ensure that all cells are recognised. The list of tiles is distributed, round-robin fashion, between the available parallel tasks. Input images are stored in the TIFF format, with the bitmaps sliced in tiles of 256×256 pixels. Thus, reading the image results in many small read accesses.

The Cell Segmentation application was studied with a focus on I/O performance optimisation, utilising HDF5 and Hecuba as back ends. We plan on adding DSS as a further back end, but the timeframe depends on the availability of Dynamic Shared Storage (DSS) on the testbed (see Section 4).

### 2.2.2 Label Propagation

*Collaboration with JUELICH-INM-1 in SP2 and SP5*

For the Brain Atlas, sections will be annotated with the area, e.g. the primary visual cortex V1. This annotation is done by human experts, which is a major bottleneck in the process. Therefore, machine learning, in the form of deep convolutional neural networks (CNN), is used to speed-up this process. The expert annotates a subset of the generated images, the resulting labels are learned by the network, and propagated to adjacent sections. The ratio of annotated to non-annotated sections is envisaged to be 1:120. In this document, we present studies on one configuration only, namely the reading of two brain sections. We plan on analysing a larger input set in the coming months.

In this use case, we are concerned solely with training the CNN, which is both the time-consuming and I/O-intensive part, when compared to inference. The application consumes images in the form of chunked HDF5 files, which are generated from the original high-resolution 1 µm TIFF images. In this step, the resolution is reduced to 2 µm in order to allow a single GPU to process the full cortical depth.

The training uses a distributed architecture based on Horovod [2], a library for MPI-based parallel deep learning commonly used on HPC clusters. Horovod handles aggregation of weights, after each mini-batch, into the global weights. One training process per GPU is used, called the master. An additional number of tasks is configured to prepare batches of image tiles to be sent to their master processes. This includes reading the raw tiles and annotations — corresponding to the labels to be learned — and performing augmentation of these tiles. As this step includes rotation, more data is read from

[1] https://opencv.org/
[2] https://github.com/horovod/horovod

disk than for a non-rotated tile. The chunk size of the HDF5 files is chosen such that at most four chunks need to be read to fill a tile. The final model is stored as a HDF5 file on the global file system.

In contrast to most deep learning applications, which process large databases of small images, this use case requires the processing of a small set of extremely large images. Small tiles are extracted from these large images, augmented on the fly and fed to the training process. This shifts the focus more towards the I/O part of the application, including augmentation.

# 3. Technologies

## 3.1 Hierarchical Data Format v5 (HDF5)

HDF5 is a well-known library for file I/O in high-performance computing, data analysis and machine learning. It is developed and provided open source by the HDF group[3]. At a very high level, it offers functionality similar to a "file system in a file" in order to facilitate self-contained, self-describing datasets.

We use HDF5 as an interface to parallel file systems, mainly driven from distributed applications using MPI. HDF5 offers drivers aware of and optimised for MPI applications. Therefore, HDF5 replaces and enhances the POSIX API as a means of writing and reading files in these applications. As the API is quite large, specific details of HDF5 will be discussed in the use case analyses where these different options were used. Further, HDF5 is extensible via the Virtual Object Layer (VOL). This has given rise to interest in using HDF5 as an API for other storage protocols, such as SWIFT and DSS.

## 3.2 Hecuba

Hecuba[4] is a programming model for interaction with distributed storage developed at the Barcelona Supercomputing Center (BSC). Hecuba stores the actual data in a distributed database. However, it can provide a performance gain over the default database connectors. Currently, Hecuba is available as a Python library and provided under an open source license (Apache version 2.0). So far, support for Cassandra and ScyllaDB has been built-in, enabling manipulation of distributed datasets transparently. Through Hecuba, Python applications access persistent data such as regular objects stored in memory. To develop an application, the user describes the data model by extending a Hecuba class and instantiating as many objects as needed. The user can make the in-memory objects persistent or retrieve persistent data either by instantiating objects with an identifier or by calling their make_persistent methods.

## 3.3 Distributed Shared Storage (DSS)

As part of the HBP Pre-Commercial Procurement during the Ramp-up Phase, IBM developed Distributed Shared Storage (DSS), an InfiniBand (IB) Verbs provider to expose Linux blocks to remote and local clients. It is currently not publicly available, but it is expected to be made available as open source. DSS allows an application to treat such block devices as if they were shared memory regions and perform Remote Direct Memory Access (RDMA) via the well-known verbs interface. Furthermore, this interface is implemented as part of the Linux kernel in the form of the rdma_cm library. In particular, the aim is to utilise verbs as an access layer for Storage Class Memory (SCM) and Non-Volatile Memory (NVM). The testbed nodes have NVMe SSDs attached, which are addressable through DSS; see Section 4 for more information. By leveraging IB verbs, well-understood semantics for RDMA are offered to developers as an alternative to POSIX I/O, backed by a specific API that is part of the kernel. All operations are asynchronous; the developer is required to ensure data integrity, where necessary. Currently, DSS is still under active development and therefore not yet

[3] https://www.hdfgroup.org/
[4] https://github.com/bsc-dd/hecuba

publicly available. We plan on implementing at least the I/O part of the Cell Segmentation use case when DSS becomes publicly available. Further work will be guided by our first results. As the interface is quite low-level, we are thinking about offering more easily accessible abstractions on top of DSS, including bindings for the Python programming language.

## 3.4 Universal Data Junction (UDJ)

Cray's Universal Data Junction (UDJ) project aims to provide an interface between two, or potentially more, applications. The library is not yet publicly available, but will be made open source in the near future. It abstracts the concrete exchange method and handles consensus between participants. Similar functionality can be achieved with the MPI3 Open_port methods family, but with less flexibility regarding the transport layer. For our purposes, UDJ is a convenience layer on top of MPI3.

## 3.5 OpenStack SWIFT

SWIFT is part of the OpenStack ecosystem, providing a REST API to the object storage services of OpenStack[5]. Object stores are interesting alternatives to traditional POSIX file systems, including parallel and distributed file systems, as a different set of trade-offs has been chosen. Further, SWIFT is used as the archival storage solution in the ICEI project[6] and installations exist or are planned at HBP/ICEI partner sites. SWIFT objects are organised in *containers* belonging to an account; below that, little to no structure is enforced. Access to accounts, containers, and objects is offered through the standard HTTP verbs. Objects can be segmented into — not necessarily uniform — chunks, that are catalogued using *manifests*. Overall, this allows for mappings between commonly used file access patterns and SWIFT containers and objects.

We plan to investigate the use of SWIFT for direct data access in data processing. This includes performance analysis, optimisation of the application and performance exploration of basic SWIFT operations. Furthermore, a tiered caching mechanism is planned, that takes care of prefetching data from SWIFT to local storage and collecting local output, which is then written back to SWIFT objects. So far, basic testing on functionality and performance has been conducted.

## 3.6 Conduit

Conduit[7] is a library developed by Lawrence Livermore National Laboratories for data exchange in multi-scale simulations. The library is distributed as open source (BSD-style license). Its primary focus is the description of data in a self-contained format. These descriptions are stored as trees of nodes, with leaves describing an array of primitive datatypes by a tuple of base address, offset relative to base address, stride between elements and size per element. This enables transparent handling of C-style structures. Data can be owned by a node; in which case, it is copied or external, thus eliding allocations and copies. Some additional functionality is offered by the Conduit library, such as exchange through MPI. Conduit does not include facilities to concatenate nodes, which is a common requirement in our use cases. We provide a small library[8] that adds concatenation, which is not entirely trivial as for example structured data must be handled properly.

[5] https://www.openstack.org/
[6] Interactive Computing E-Infrastructure (HBP SGA ICEI); this project implements the federated Fenix infrastructure (https://fenix-ri.eu/) that the HPAC Platform will base its services on.
[7] https://github.com/LLNL/conduit
[8] Available upon request, please contact hpac-support@humanbrainproject.eu

## 3.7 Slurm

Slurm[9] is a well-known resource manager used on the majority of clusters in the TOP500[10] list and is also used on many systems integrated in the HPAC Platform and in the ICEI project. The contributing Task T7.2.1 focusses on scheduling algorithms for data-intensive applications, specifically on methods for the co-allocation of compute and storage resources and allocating storage on distributed file systems, including support for data staging. It contributes by providing plugins to support the use cases investigated in Task T7.2.3. The main development starts by incorporating these design decisions into the implementation of a first prototype for a Slurm plugin. One of the main goals of the Task will be to continue to collaborate closely with Task T7.2.3 to fulfil the requirements relevant to the HBP use cases. A simulation environment was set-up, based on Docker[11], and related technologies were evaluated. For the implementation, we are using a system which uses a global parallel file system such as GPFS, and a distributed file system, which makes the node-local storage resources accessible, such as BeeGFS[12]. For the next steps, we will focus on the main development of the scheduler extensions.

The relevant use cases for the scheduler extensions are Cell Segmentation and Label Propagation. In both cases, the data is staged from GPFS to BeeGFS, processed and drained afterwards, and outputs are stored persistently. Finally, the storage allocation is released. However, this can happen in multiple ways; for example, the input data may be scaled down and stored in HDF5 files. We plan to enable the user to select a storage allocation, referred to as containers, by IDs which are used to identify locations on the fast storage to connect jobs and define workflows with data dependencies across job boundaries. For the implementation in Slurm, an extension of the generic burst buffer plugin is being considered for use in persistent burst buffer mode. Slurm provides only two implementations of burst buffers, whereas the plugin for Cray systems provides more features. Therefore, we will need to port essential features from the Cray plugin to the generic burst buffers as part of the work in Task T7.2.1.

# 4. Hardware Testbed: JURON

Our main testbed is the JURON[13] cluster, one of the two HBP pilot systems that were developed in the context of a Pre-Commercial Procurement in the HBP Ramp-Up Phase. All experiments discussed in the following were performed on JURON.

We summarise the hardware specifications in summary form below. The node topology is shown in Figure 2. One peculiarity of the setup is the connection to the site-global parallel file system. It is mounted on the login nodes and re-exported to the compute nodes via NFS. This results in a less-than-optimal performance. JURON comprises 20 IBM S822LC nodes, which are dual socket POWER8' systems. Each socket features 10 physical cores with 8-way symmetric multi-threading and is clocked at 2-4 GHz. 256 GB of RAM per node are configured. In addition, each node has 2 NVIDIA Tesla P100 GPUs attached via NVLINK and a 1.1 TB NVMe SSD Disk. These SSDs are tied into a cluster-wide file system using BeeGFS.

The nodes are connected through 4-channel EDR InfiniBand adapters (Mellanox ConnectX-4).

[9] https://slurm.schedmd.com/
[10] https://www.top500.org/
[11] https://www.docker.com/
[12] https://www.beegfs.io/
[13] https://hbp-hpc-platform.fz-juelich.de/?page_id=1073

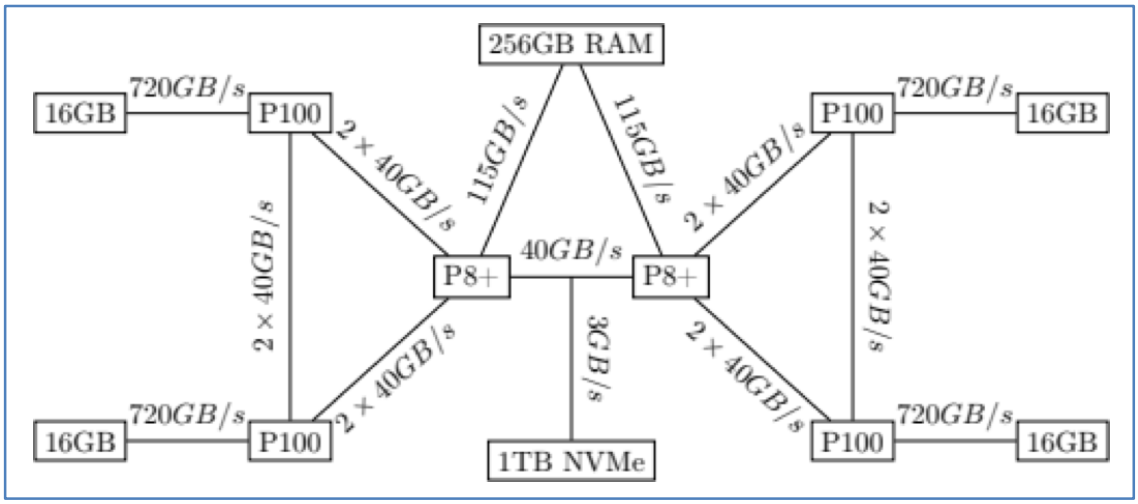


**Figure 2: Topology of a JURON SL822LC node**

# 5. Analysis of Use Cases

In the following, we summarise our analyses of I/O behaviour the use cases presented above. We investigate avenues for optimisation and the application of different technologies. All studies below were conducted on the Power8 HBP pilot system JURON (see above).

## 5.1 Cell Segmentation

### *5.1.1 HDF5*

The output is written into a single HDF5 file in parallel via the HDF5 MPI-IO driver. As the test system possesses a number of local NVMe disks, we explored two options for storing the output, first to the site-global GPFS installation and second to the local BeeGFS, backed by the fast NVMe disks. Input images are stored in the TIFF format, which results in an internal representation consisting of small tiles with 256×256 pixels each. Thus, reading the image requires many small read accesses. First, an analysis of the original application was done. Here, the input is read from GPFS. Figure 2 shows the distribution of the read times over the complete run time using a tile-size of 512×512. With growing numbers of processes, the median read time increases and a greater number of large outliers occur. We conclude that the parallel accesses to the parallel file system interfere. We further analysed the total runtime as a function of the number of MPI tasks and the output destination, see Table 1.

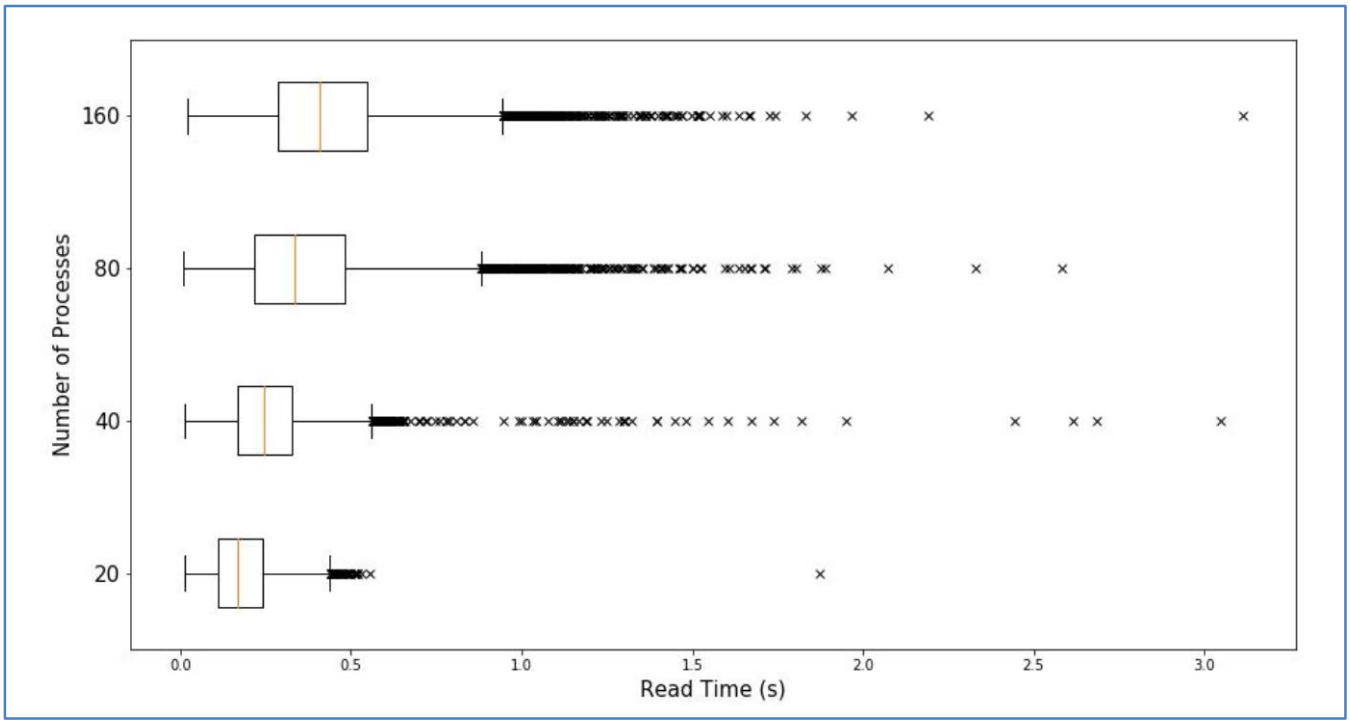


**Figure 3: Latencies of read accesses to GPFS in the original application**

As only a single image is used, we staged the full image into shared memory before entering the actual algorithm, using MPI3-shared memory windows. This results in less accesses to the file system, as only one task in a shared memory domain reads the image.

**Table 1: Total runtimes of Cell Segmentation with varying numbers of MPI tasks and output destinations**

| Processes | Original | | Input | | | Input and Output | |
|---|---|---|---|---|---|---|---|
| | GPFS [s] | BeeGFS [s] | Loading [s] | GPFS [s] | BeeGFS [s] | GPFS [s] | BeeGFS [s] |
| 20 | 3,521 | 1,861 | 32 | 3,493 | 1,751 | 1,538 | 1,615 |
| 40 | 1,984 | 1,158 | 38 | 2,998 | 1,228 | 795 | 899 |
| 80 | 2,162 | 967 | 42 | 2,356 | 755 | 543 | 645 |
| 160 | - | - | 77 | 1,858 | 687 | 220 | 280 |

In the original version, the output is written to a sequential HDF5 file. The access pattern leads to many small and non-continuous writes to the parallel file system. To optimise the output phases, we use chunked HDF5 files instead. Figure 3 shows the impact on the read and write accesses. HDF5 employs a read-modify-write pattern for chunked files, thus each write has a corresponding read access in the analysis. The final application runtimes are shown in Table 1. Remarkably, now it is faster to write directly to GPFS than to BeeGFS. The reason for this effect is not known yet.

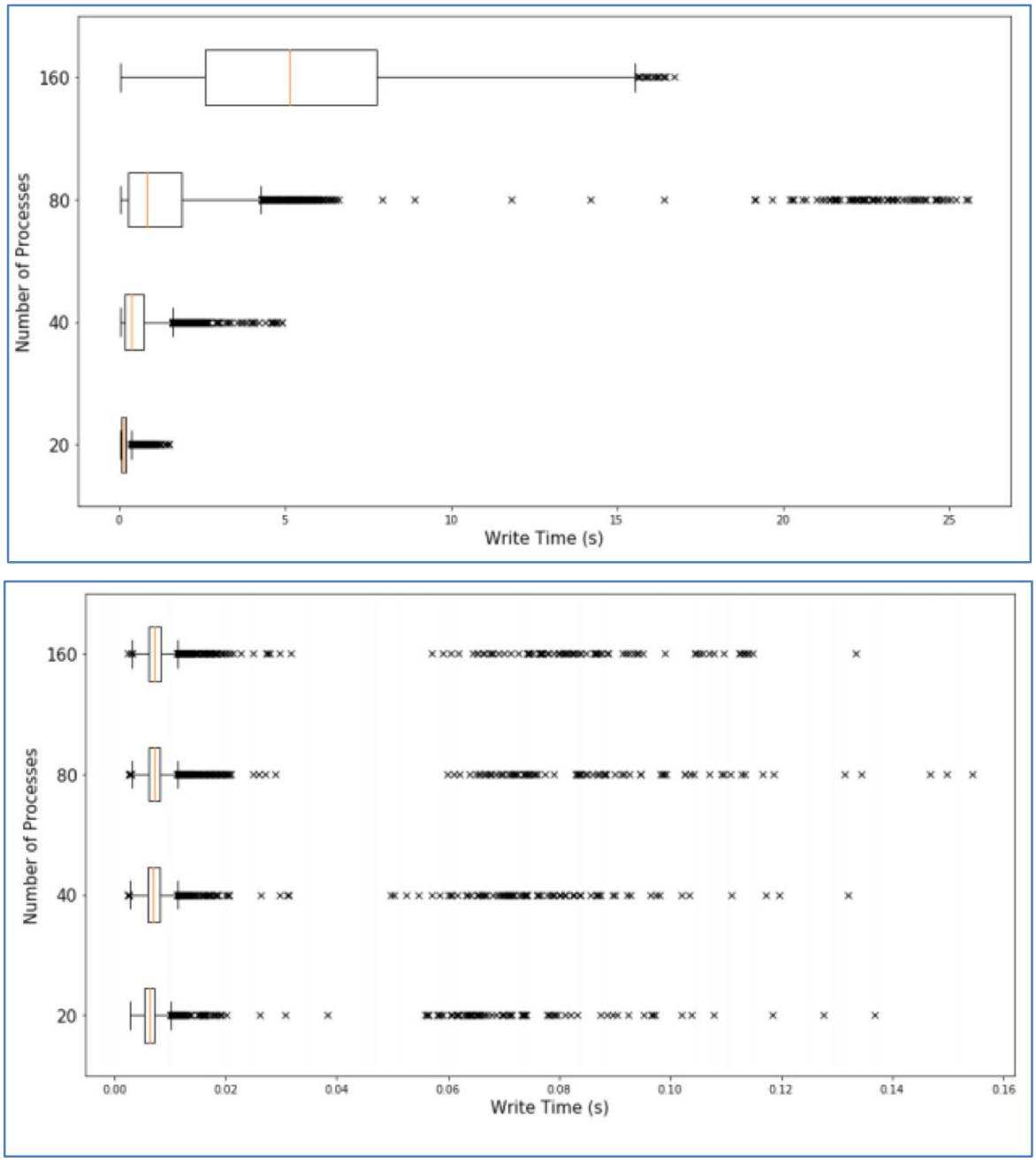


**Figure 4: Latencies of write requests to BeeGFS in the final version with sequential (upper panel) and chunked (lower panel) HDF files**

### *5.1.2 Hecuba*

We evaluated the impact of saving the images in distributed storage and considered data-location aware policies. Furthermore, we explored and evaluated the impact of different data models and technologies when performing queries on the results. We were able to adapt the application and compare the impact of storing the dataset on HFD5 files on GPFS versus Cassandra or Scylla. We also identified that GPFS was subject to high performance variability, whereas both Cassandra and Scylla suffered from little variance and scaled when adding nodes. In this aspect, Scylla delivered a higher throughput than Cassandra, resulting in slightly faster execution times. These tests were conducted on copies of the image on each local NVMe disk, to avoid the variability in performance introduced by GPFS. Early tests based on HDF5 over GPFS were conducted with segment sizes of 5000×5000 pixels, which allowed a smaller memory footprint and better performance on GPFS than with smaller segments. Furthermore, having small blocks translated to an increase in the time spent reconciling blocks. Consequently, more reconciliation tasks were necessary and the number of pixels evaluated in this phase grew with smaller segments, resulting in notably slower bounds reconciliation.

Therefore, we kept working with 5,000×5,000 segment sizes. In Figure 4 the performance of the application after disabling the output can be observed. We used these values as a baseline for understand the I/O impact of the different storage solutions. In Figure 4, we also compare the baseline execution runtime with the runtime when using different back ends, to observe their impact. Nonetheless, having large segments resulted in an imbalance, since the amount of time required to process a block is directly related to the number of cells identified. The principal cause that prevents the application to scale when running on four or more nodes in parallel is the imbalance due to large blocks and a static distribution of work. In the cell segmentation application, NumPy arrays are used to store the results. Hecuba stores them at runtime without knowing their shape or data type in advance. The arrays are stored each in a separate table, which distributes the data among the nodes. This data model will be reviewed and evaluated in the future, to assess under which circumstances it is adequate.

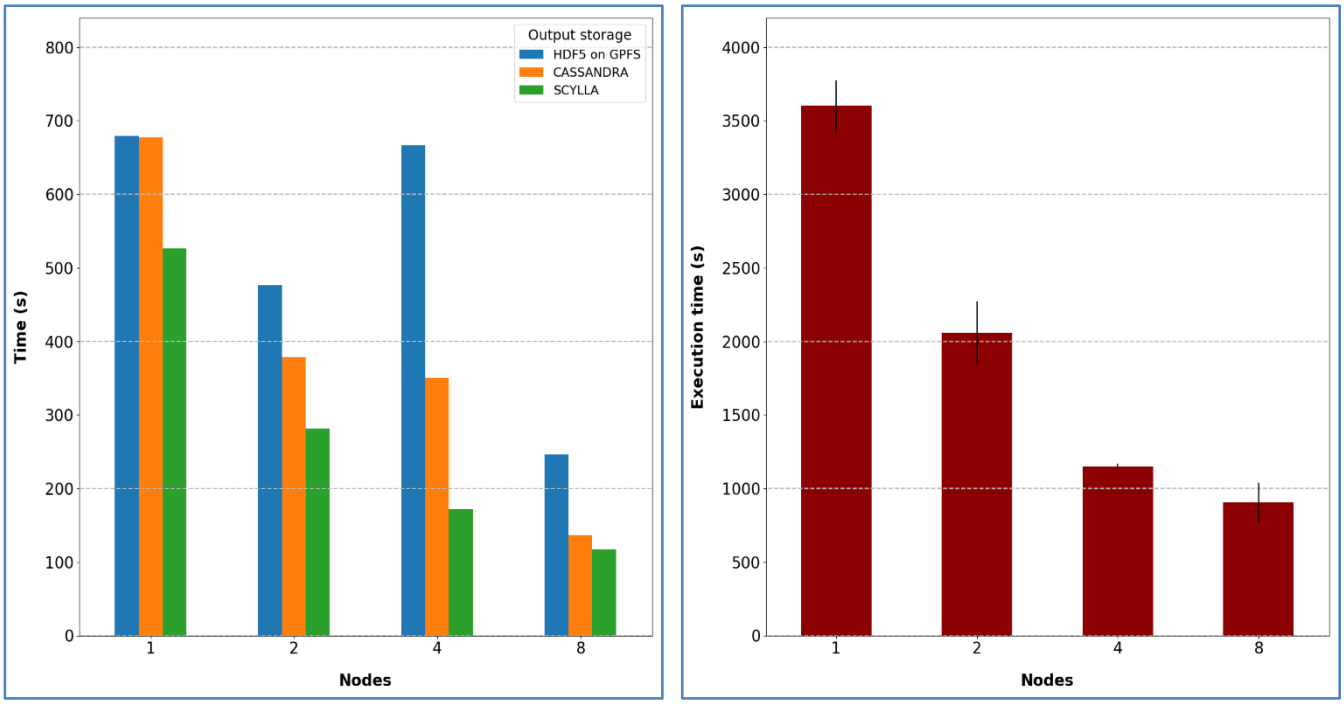


**Figure 5: Effect of different storage back ends at runtime**

**The left graph shows the overhead over the runtime when disabling output completely, shown on the right.**

The application code was greatly simplified by porting it to Hecuba. We eliminated synchronizations and kept the cell characteristics as soon as they were identified, which in HDF5 could not be done without a performance impact. An extract of the original cell segmentation main method and the adaptation to Hecuba can be seen in Table 2.

**Table 2: Comparison of the original cell segmentation code with the adaptation in Hecuba**

| Hecuba | HDF5 |
|---|---|
| | # Persistent objects already instantiated<br>total_table = None<br>for (row, col, indexes) in local_slices:<br>  # ...<br>  total_table =<br>  np.concatenate((total_table,<br>  table), axis=0)<br>  # Write labeled data<br>  out_labeled[row[0]:row[1],<br>  col[0]:col[1]] = labels |
| # Persistent objects already instantiated<br>for (row, col, indexes) in local_slices:<br>  # ...<br>  # Write labeled data (numpy.ndarray)<br>  data_table[rows[0],cols[0]] = labels<br>  # Write cells information (numpy.ndarray)<br>  cell_table[rows[0], cols[0]] = table | table_length =<br>  np.array(total_table.shape[0],<br>  np.uint64)<br># Synchronize to unify the cells \\ information<br>comm.Barrier()<br>len_cells = np.zeros(comm.size,<br>  dtype=np.uint64)<br>comm.Allgather(table_length, len_cells)<br>offsets = [0] + np.cumsum(len_cells).tolist()<br>n_cells = len_cells.sum()<br>n_features = total_table.shape[1]<br># Create and preallocate the cells \\ information<br>out_file.create_dataset("cells",<br>  shape=(n_cells, n_features),<br>  dtype=np.float32)<br># Write cells information<br>out_file["cells"][offsets[comm.rank]:<br>  offsets[comm.rank + 1]] = total_table |

### 5.1.3 Summary

We identified several opportunities for optimisation. First and foremost, relying on POSIX as a storage technology resulted in an I/O bottleneck. A large number of small read and non-sequential write operations result in sub-optimal performance. We partially removed the bottleneck by staging the data to local NVMe disks or MPI3-shared memory. Then, by saving the results, either as an HDF5 file on BeeGFS or using a distributed array in Cassandra, we were able to scale down the I/O steps of the application.

Nevertheless, the imbalance of work items among processes becomes critical when launching the application on four or more nodes in parallel. In this scenario, half of the computation time is wasted by idling processes waiting for the slowest processes to finish. This performance issue can be mitigated by using smaller block sizes, which reduces the imbalance. This introduces a trade-off between imbalance and parallelisation overhead. Porting to Hecuba simplified portions of code required to synchronize metadata for HDF5 files or file creations and to simplify many of the file operations.

## 5.2 NEST

We investigate the in-transit processing of simulation data, based on a prototype scenario: a distributed producer, here NEST, writes data to single consumer process, represented by a simple custom visualisation of synaptic potentials. The transport of data is implemented using UDJ. For describing the data to be exchanged between applications, we leverage Conduit. Also, operations can be requested from the remote side by sending messages through Conduit nodes. We extended NEST with a new back end for the recording infrastructure that packages the voltage signal with metadata into a Conduit node. Each recorder maps to a sub-node. The visualisation is written in C++

using the VTK library in an event driven style. It polls an UDJ endpoint for incoming Conduit nodes and unpacks these into corresponding commands (currently plot and quit) and the actual data portion (see Figure 5). The consumer application processes time series data that might have been generated by different processes on the NEST side. It therefore needs to concatenate data blocks from different producer processes under the same path in the node. For efficiency, we implemented this as a two-step process: the back end in NEST will merge nodes over shared memory domains and only one process per domain initiates the communication. The consumer will thus receive one node per sender and merge these into a single final data node, see Figure 6. We noted that some of the features of Conduit did not work for our use case, namely passing structured external data via MPI. We do not present performance numbers in this document, as this is foremost a demonstration of the concept. UDJ is currently lacking functional support for asynchronous communication, which would allow us to overlap transfer and production of the update. This would effectively decouple the producer from the consumer, unless the communication takes more time than to produce the next step. This capability is planned to become available in future versions of UDJ.

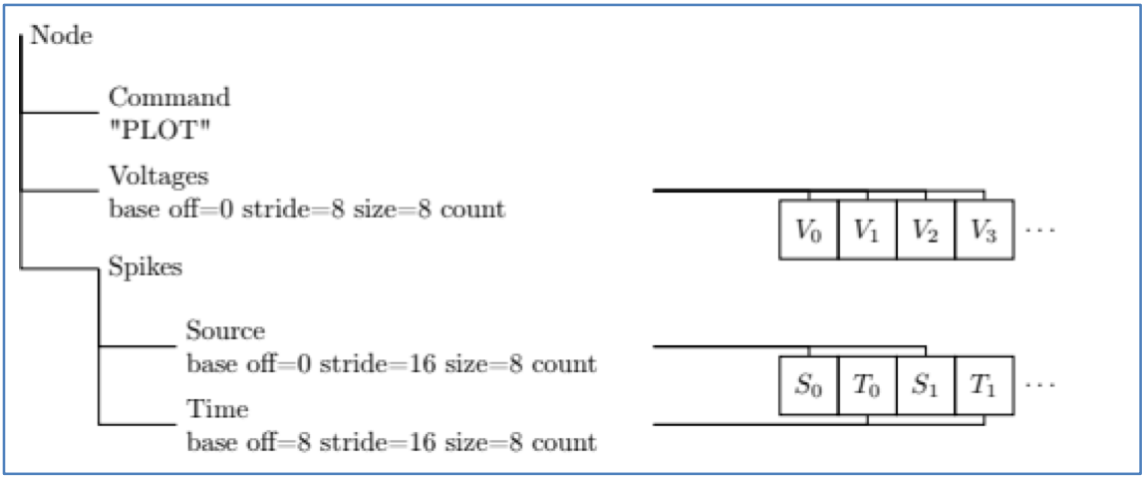


**Figure 6: Logical layout for Conduit nodes transferred from NEST to visualisation**

**Spikes are stored as an array of structures comprising each of an 8B timestamp and an 8B source ID, which is represented in Conduit as two-strided external arrays. Voltages are external contiguous arrays of 8B floating point numbers.**

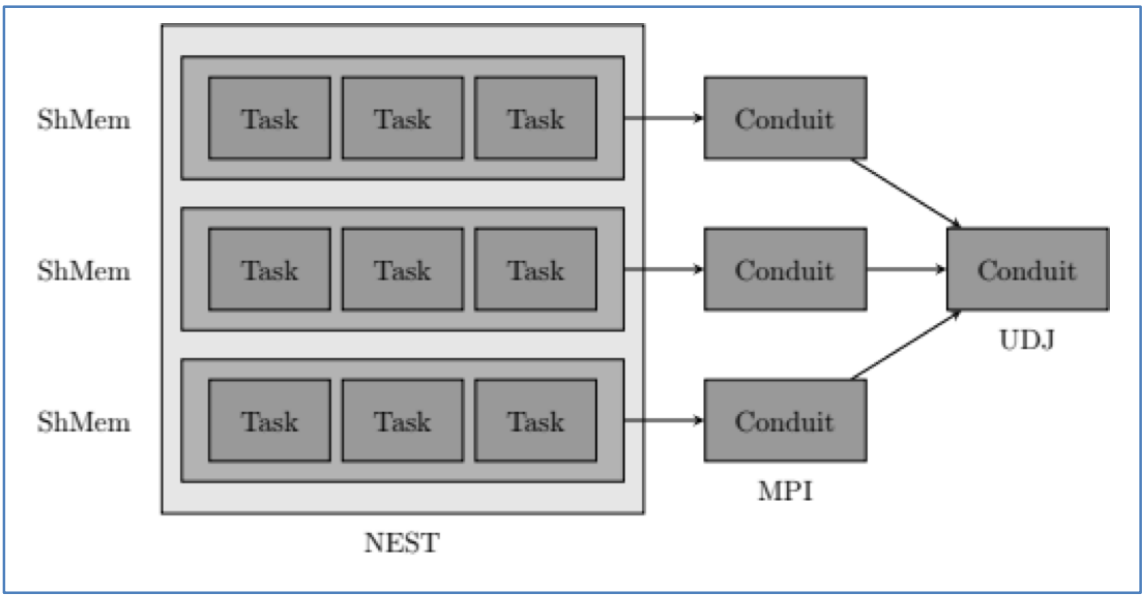


**Figure 7: Conceptual view of the merge and transfer process of a Conduit node from producer to consumer**

**Nodes are generated by each task individually and merged by the recording back end into one node per shared memory domain. Each domain elects a leader which transfers the results through UDJ to the visualisation.**

## 5.3 Label Propagation

The analysis of the label propagation use case is still work in progress. We present here first results with one input set of two brain sections on a conventional parallel file system and explore some options to tune HDF5 file access. Furthermore, we investigate in-memory caching. At the time of writing, the BeeGFS installation on our testbed is not functional, due to a longer maintenance period, but it is expected to become available again within the next weeks. Analysis of the use case with a local, fast file system will be conducted as one of the next steps.

### *5.3.1 HDF5*

We investigate the following setup for the test case: A set of three images is processed using $N_{Task}$ processes and $N_{GPU}$ GPUs, resulting in $N_{GPU}$ master processes and $(N_{Task} - N_{GPU})/N_{GPU}$ slaves per master. The application was profiled using the Darshan I/O characterisation tool. We first collected the total runtime of the application in Table 3 and the data volume read and written in Table 4. The master processes are responsible for writing the finished model and perform no read operations. Note that HDF5 uses a read-modify-write scheme to update data. We restrict the analysis to I/O on HDF5 files, the remainder being logging to console and file — which can be turned off — and reading the source code and configuration. Based on the runtime of the application, which depends on the number of processes, it is obvious that the generation of training batches is a major factor.

**Table 3: Runtimes of the application before optimisation**

| Nodes | Cores (SMT) | Processes | GPUs | Time [s] |
|---|---|---|---|---|
| 1 | 10 (1) | 10 | 2 | 7,913 |
| 1 | 20 (1) | 20 | 2 | 3,965 |
| 1 | 20 (2) | 40 | 2 | 2,728 |
| 1 | 20 (1) | 20 | 4 | 9,233 |
| 1 | 20 (2) | 40 | 4 | 5,099 |

**Table 4: Transferred volumes per process before optimisation (10 tasks, 2 GPUs)**

| | Read [MiB] | Write [MiB] |
|---|---|---|
| Master | 227 | 220 |
| Slaves | 1183 | 0 |

The HDF5 library automatically caches accessed chunks in the Raw Data Chunk Cache (RDCC) when using the chunked data format. However, the default parameters are not optimal for large files and chunks. For the smallest configuration (ten tasks, two GPUs), we show the impact of increasing the RDCC size in Table 5. The RDCC provides a measurable benefit, but since it is not shared between processes, data sharing is not exploited. The batch generator accesses image tiles randomly, which could benefit from a shared cache.

There are two options for a shared cache: using a local fast file system or using main memory. The latter is only feasible when a small number of sections is used, as each consumes 1.8 GB. Due to the random-access pattern, we stage the full images into memory and local storage.

Finally, we studied the effect of compressing the on-disk dataset. Chunked HDF5 files can optionally apply a transparent filter pipeline to chunks while reading or writing. One such filter is compression with the GZIP deflate algorithm. Shuffling the data is accomplished by another filter at no measurable performance cost and may help the compression rate. Compression reduces the requirements on disk bandwidth at the cost of increased CPU load during reading, due to decompression. As the former is quite low on the systems under consideration, we expect an overall performance improvement. Compression adds a one-time cost during generation of the dataset. Compression is entirely transparent for the application; no changes are needed to read compressed files. We list the effects of the different settings combinations in Table 5. Since savings in bandwidth are expected to be the main benefit of compression, we did not test the actual application with

higher compression levels. While there is a minor impact on the runtime, the savings in storage are substantial.

**Table 5: Effect of dataset compression, staging, and RDCC on runtime**

| Tasks | GPUs | Filter | Staged | File Size [GiB] | RDDC [MiB] | Runtime [s] |
|---|---|---|---|---|---|---|
| 10 | 2 | None | No | 1.8 | 1 | 7,913 |
| 10 | 2 | None | No | 1.8 | 1,024 | 7,665 |
| 10 | 2 | None | Yes | 1.8 | 1 | 7,752 |
| 10 | 2 | GZIP 3 | No | 1.1 | 1 | 10,454 |
| 10 | 2 | GZIP 3 | No | 1.1 | 1,024 | 7,747 |
| 10 | 2 | GZIP 3 | Yes | 1.1 | 1 | 7,770 |
| 20 | 2 | None | No | 1.8 | 1 | 3,965 |
| 20 | 2 | None | No | 1.8 | 1,024 | 3,792 |
| 20 | 2 | None | Yes | 1.8 | 1 | 3,787 |
| 20 | 2 | GZIP 3 | No | 1.1 | 1 | 5,211 |
| 20 | 2 | GZIP 3 | No | 1.1 | 1,024 | 3,814 |
| 20 | 2 | GZIP 3 | Yes | 1.1 | 1 | 3,812 |

**Higher compression rates could not be achieved with GZIP, even when adding a pre-shuffle filter.**

# 6. Outlook

In the coming months, we plan to extend our results in various directions. We will explore DSS as an alternative to traditional block or object based I/O and analyse feasibility in at least one use case. Due to ongoing developments and legal issues, DSS has not been in productive use so far. Furthermore, we will use our experiences for defining an API based on the requirements of the different use cases. We foresee two possible outcomes; one being an API defined and implemented by members of the HBP and the other could be a set of extensions to a well-known interface, such as HDF5. Both approaches have merits and shortcomings. Extending HDF5 lowers the barrier to entry for users, as HDF5 is an established product, and remove some of the complexities of implementation as only new back ends need to be developed. Offering a locally developed library would allow us to tailor the design specifically to the needs of the Project, but places the full burden of implementation and maintenance on the HBP, along with driving adoption and providing training. Potentially, such a solution could also offer higher performance, as the front end can accommodate some of the specifics of the back ends. Currently, we plan to provide back ends for SWIFT and potentially DSS, and we consider a translation layer between Conduit and our chosen API solution. A third alternative would be to use Hecuba as a surface API. The current public version of Hecuba offers a Python API and Apache Cassandra. BSC plans to explore SWIFT as a back end for Hecuba and to offer Fortran and C++ bindings, which would make Hecuba an interesting choice. In our experience so far, we often found the need to transfer large datasets from one storage tier to another, most commonly in the form of a preparatory staging step, followed by processing and finally persisting the results and cleaning-up the temporary storage. We will formalise this process in the SLURM workflow manager. If time allows, we plan to diversify our set of use cases, which currently has image processing as its main focus. We will further investigate the feasibility of tiered strategies, where data is staged first into local storage and then into memory. This is potentially beneficial, as the access to global storage is provided through a single gateway and thus all compute nodes share the available bandwidth.

# Annex 1: Conduit Merge Library

The Conduit library provides various ways to manipulate tagged tree data structures (Nodes). Notably absent is a method to concatenate leaves. This scenario commonly arises in distributed processing, where each task generates one slice of a larger dataset. From a user's standpoint, this omission is inconvenient, given the utility.

We provide a small library that implements the method Node concat(Node const&lhs, Node const& rhs), which will concatenate arrays addressable via the same path in order. Leaves at different paths will be inserted into the result unaltered. We also offer overloads for handling collections of Nodes. As different paths can refer to memory located in a shared buffer, the implementation is not trivial. Therefore, we make this functionality available as a library.

Note that the implementation can be further optimised, especially with regards to the allocation of the final data structure. Also, the concat operation is a special case of a general higher-order function mergeWith that applies a function Node f(Node const& lhs, Node const& rhs) to leaves at the same path and inserts the returned Node into the result at the same path. This could be further generalised to apply a Node of such functions to its arguments, where a function at a path is applied at all paths below unless explicitly overridden at a more specific path. If the need arises for these generalisations, our library will be extended accordingly.