




H2020-JTI-EuroHPC-2019-1

Project no. 956748

# ADAPTIVE MULTI-TIER INTELLIGENT DATA MANAGER FOR EXASCALE

# D4.2
# Software to support I/O scheduling policies

Version 1.2

*Date:* July 28, 2023

*Type:* Deliverable
*WP number:* WP4

*Editor:* Alberto Miranda
*Institution:* BSC

| **Project co-funded by the European Union Horizon 2020 JTI-EuroHPC research and innovation programme and Spain, Germany, France, Italy, Poland, and Sweden** | | |
|---|---|---|
| **Dissemination Level** | | |
| **PU** | Public | √ |
| **PP** | Restricted to other programme participants (including the Commission Services) | |
| **RE** | Restricted to a group specified by the consortium (including the Commission Services) | |
| **CO** | Confidential, only for members of the consortium (including the Commission Services) | |

# Change Log

| Rev. | Date | Who | Site | What |
|---|---|---|---|---|
| 1 | 16/05/23 | Jesus Carretero | UC3M | Document creation. |
| 2 | 18/06/23 | Alberto Miranda | BSC | Establish document structure. |
| 3 | 23/06/23 | Alberto Miranda | BSC | Add Executive summary. |
| 4 | 27/06/23 | Alberto Miranda | BSC | Add Introduction. |
| 5 | 03/07/23 | Alberto Miranda | BSC | Add I/O Scheduler: Design. |
| 6 | 05/07/23 | Alberto Miranda | BSC | Restructure document. Add initial text for I/O Scheduler: Implementation. |
| 7 | 10/07/23 | Luan Teylo | INRIA | Add IO-Sets in Section 4 |
| 8 | 10/07/23 | Alberto Miranda | BSC | Minor rewrites of Introduction and Design chapters. Text for Slurm extensions. |
| 9 | 11/07/23 | Alberto Miranda | BSC | Finalise text for Slurm extensions. |
| 10 | 12/07/23 | Alberto Miranda | BSC | Add text for scord service in Section 3. |
| 11 | 12/07/23 | Maysam Rahmanpour | JGU | Add initial results for Cargo data transfers in Section 3.3.4. |
| 12 | 12/07/23 | Ramon Nou | BSC | Add text QoS Control in for scord service in Section 4. |
| 13 | 13/07/23 | Alberto Miranda | BSC | Revise and finalise Section 3.1. Initial text for Section 3.3,Minor improvements to several sections. Finalise description of the Cargo service. Add scord API examples. Add Cargo API examples. |
| 14 | 14/07/23 | Ahmad Tarraf | TUDA | Added FTIO description in Section 4.3 |
| 15 | 14/07/23 | Marc-André Vef | JGU | Revise and finalise Section 3.3.4. |
| 16 | 17/07/23 | Alberto Miranda | BSC | Add API examples. Add figure for *I/O Scheduler* design components. Add figure for *I/O Scheduler* software components. |
| 17 | 18/07/23 | Francieli Boito | INRIA | Add to the integration of IO-Sets with QoS control in Section 4.1.3; complete the description of FTIO and elaborate on its integration with other components in Section 4.3; add links to the IO-Sets and FTIO pre-prints. |
| 18 | 18/07/23 | Alberto Miranda | BSC | General revision. Minor changes. |
| 19 | 20/07/23 | Clément Barthelemy | INRIA | Add details about `IO-Sets` in `BeeGFS`. |
| 20 | 25/07/23 | Hamid Fard | TUDA | Full review. |
| 21 | 27/07/23 | Jean-Baptiste Besnard | PARA | Document review. |
| 22 | 27/07/23 | Alberto Miranda | BSC | Integration of comments from reviewers. |

# Executive Summary

Within ADMIRE, *ad-hoc storage systems* are dynamic, ephemeral services that are deployed per job and that take advantage of node-local storage devices (such as SSDs) to accelerate the I/O performance of scientific applications. This performance benefit emerges, firstly, as a result of forcing applications to use node-local storage as much as possible and, secondly, by carefully managing and orchestrating any eventual accesses to the HPC cluster's parallel file system (PFS). To facilitate ad-hoc storage systems as first-class citizens of a service-oriented HPC architecture, the ADMIRE framework defined a core component, the *I/O Scheduler*, as the entity responsible for orchestrating the tasks required to achieve this goal. As such, the *I/O Scheduler* is responsible for coordinating with Slurm to deploy ad-hoc storage services on demand, automatically capture information from users about I/O requirements, and decide when data should be staged into (or out of) an already deployed service.

While, conceptually, the *I/O Scheduler* is a single component of the ADMIRE framework, from an implementation point of view it is composed of several sub-components. Thus, while D4.1 focused on describing the overall interactions between the different ADMIRE components and the *I/O Scheduler*, this prototype deliverable, D4.2, describes the software developed (and under development) to actually enable the ADMIRE ecosystem of I/O microservices while supporting those interactions and enabling I/O scheduling policies. The work described in this report was developed in the context of T4.2 and up to M28 of the project.

# Contents

# 1. Introduction

The design of large-scale HPC infrastructures traditionally focused on maximising parallel processing power but the advent of data-intensive computing applications, such as high-performance data analytics (HPDA) and deep learning (DL), is changing this compute-centric approach. As a result, there is a growing agreement in the HPC community that large-scale supercomputers will act as key data processing nodes in the emerging distributed computing environment integrating HPC and data-intensive computing resources [1, 9].

This growing demand for data processing is accompanied by the disruptive technological progress of the underlying storage technologies. As a result, upcoming exascale HPC systems are transitioning from a simple HPC storage architecture, consisting of a parallel backend file system and archives often based on tapes, towards a multi-tier storage hierarchy that includes node-local non-volatile main memory (NVMM) with a performance close to DRAM, NVMe-based SSDs inside compute nodes with a bandwidth of many GBytes/s, SSDs on I/O nodes, parallel file systems, campaign storage, and archival storage (see Figure 1.1). Unfortunately, there is a significant lack of interfaces for managing this I/O stack, and application access to the different I/O tiers is not considered in any scheduling decision. Data transfers between storage tiers are, therefore, managed explicitly by users. This often leads to uncoordinated I/O accesses, redundant data movement, increased energy consumption, and delayed end-to-end performance.

To address these issues, ADMIRE proposes an architecture where applications do not access the shared PFS directly but rather rely on dedicated *ad-hoc storage systems* for all their I/O needs. Ad-hoc storage systems are I/O microservices that can dynamically virtualise an application's on-node local storage by aggregating it to construct extremely fast burst buffer storage systems for a single application [4]. Since ad-hoc storage systems can be rapidly deployed on demand, they allow the efficient and transparent implementation of per-job storage tiering in an HPC cluster. By

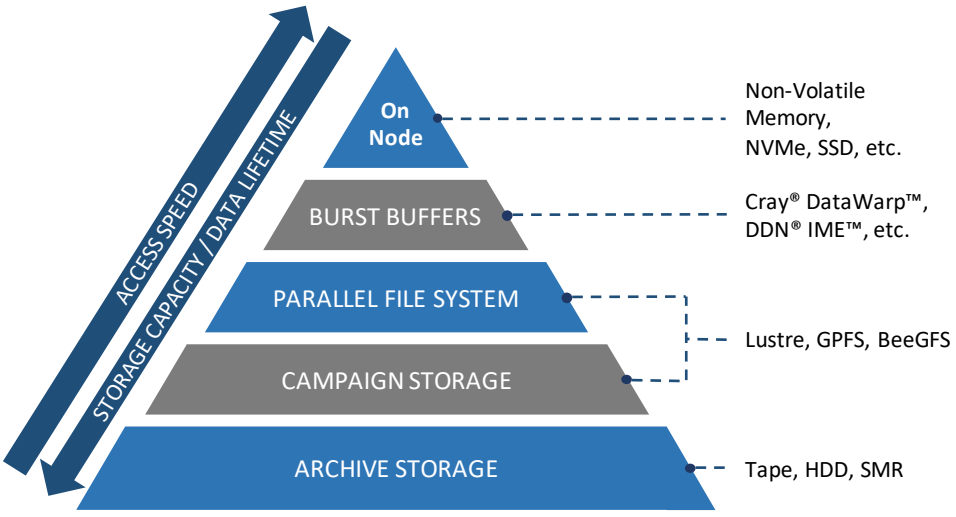



Figure 1.1: The complexity of the storage hierarchy in modern HPC systems.

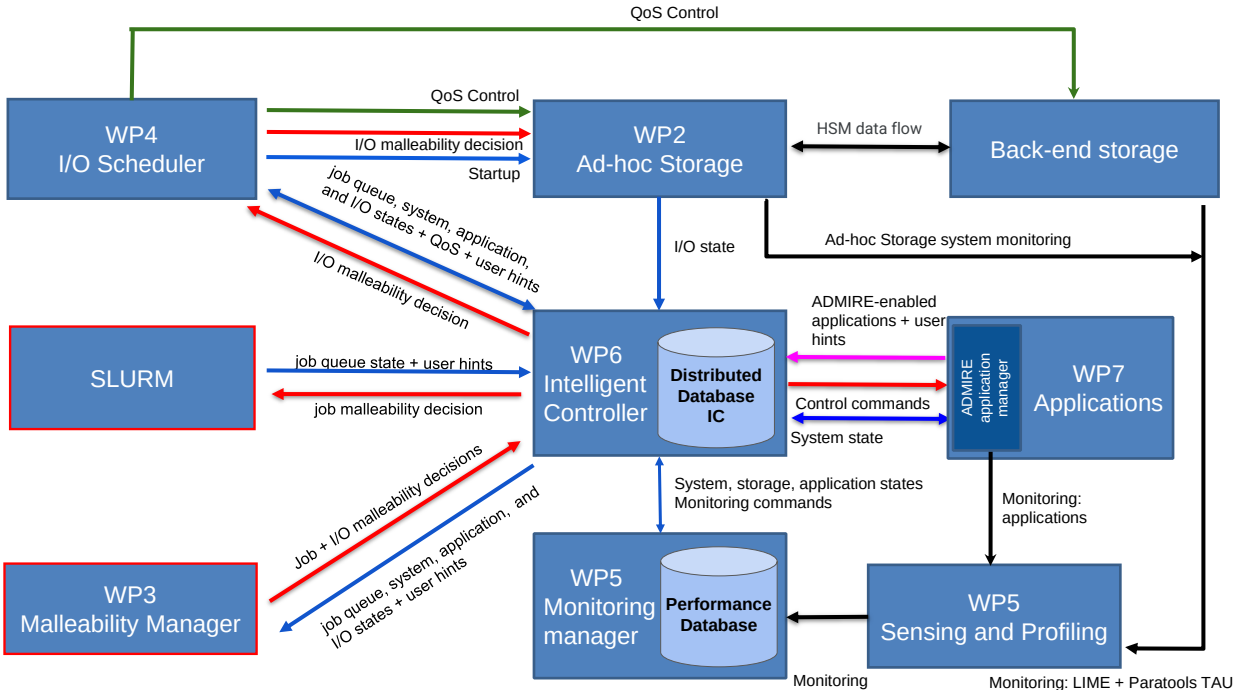



Figure 1.2: Information flow within the ADMIRE framework. The *Intelligent Controller* acts as the nexus point for collecting, processing and distributing all the information required for the proper functioning of the ADMIRE.

moving an application's I/O access from the PFS to an ad-hoc storage service, the application's data locality is improved and the PFS' I/O congestion is reduced.

To support this vision, data movements between the PFS and the ad-hoc storage services need to be carefully orchestrated. To this end, WP4 is developing an *I/O Scheduler* component that will coordinate and schedule the data flows between the shared backend parallel file system and the ad hoc storage systems, executing data transfers as necessary according to the information gathered by other components in the ADMIRE framework.

## 1.1 The ADMIRE architecture

To fulfil its responsibilities, the *I/O Scheduler* must necessarily collaborate with other components in the ADMIRE framework. To clarify these interactions, Figure 1.2 depicts how the consortium currently envisions the flow of information in the framework, and the place the *I/O Scheduler* has in it. The HPC storage subsystem is represented in the upper part of the figure and consists of the Ad-hoc and Backend Storage Systems. The former provides each application with a specialised high-performance storage tier tailored to the application's characteristics, while the latter represents the main parallel file system used by the HPC platform (e.g. Lustre, GPFS, etc.). As mentioned, these storage tiers are coordinated by the *I/O Scheduler*, which is responsible for the deployment and configuration of the ad-hoc storage, the specification of Quality-of-Service metrics and the implementation of I/O and data scheduling policies. To make appropriate decisions, the *I/O Scheduler* requires information about the current state of the HPC system and the expected I/O from applications, information which comes primarily from the *Job Scheduler*, the *Malleability Manager*, and the applications themselves via specialised APIs:

- The *Job Scheduler* (i.e. Slurm) is the front-end component with which end users interact to execute their applications. As such, its main responsibilities in the context of WP4 are to keep track of the current workload of the system (e.g. *job queue state* in the figure), as well as to collect *user hints* about an application's I/O and datasets.
- The *Malleability Manager* (WP3) is the component in the ADMIRE framework responsible for providing elasticity in the allocation of resources for a job. As such, it will be responsible for making decisions that imply dynamically altering the amount of resources assigned to a job.
- *Applications* (WP7) themselves are also able to communicate additional information about an application I/O by making use of specialised APIs provided by the framework. This information can be really valuable for the framework since it will come from application developers themselves rather than from end users. Nevertheless, note that obtaining this information requires application code to be modified, which means that the ADMIRE framework cannot rely on having this information always on hand.

Additionally, both applications and storage tiers are monitored by the *Sensing and Profiling* framework, which is responsible for collecting system-wide performance metrics that are processed and aggregated by the *Monitoring Manager* (WP5) to produce performance models for these resources. All this information is collected by the *Intelligent Controller* (WP6) that uses it to predict potential performance bottlenecks in the system. This means that the main source of information for the *I/O Scheduler* to perform its tasks will be the *Intelligent Controller*.

This document is organised as follows: Section 2 describes the *I/O Scheduler* requirements and its design, focusing on the abstract sub-components required to execute its responsibilities. Section 3 discusses the software packages that actually implement the core of the component, providing instructions to build and install them and usage instructions. Finally, Section 4 describes additional support software that is going to be integrated to provide the control logic of the component.

# 2. The I/O Scheduler

The main responsibility of ADMIRE's *I/O Scheduler* is to orchestrate the movement of *datasets* (i.e. files or objects) between the PFS and the deployed ad-hoc storage services with the goals of (1) accelerating data processing (due to increased data locality); and (2) reducing I/O contention to the PFS. I/O contention in the PFS typically occurs due to HPC applications using it for "normal" I/O operations (e.g. reading configuration files, reading input data, writing results, checking file attributes, checkpointing operations, etc.). The *I/O Scheduler* addresses this by appropriately scheduling data staging into *Ad-hoc Storage Systems*, combined with defining and enforcing per application QoS constraints: a proper orchestration of data movements changes PFS I/O into well-defined streams of sequential read-only or write-only operations, while QoS enforcement makes it so that a single application cannot overwhelm the PFS and its I/O capabilities can be distributed fairly.

Thus, the *I/O Scheduler* implementation offered by ADMIRE has the following requirements:

1. The *I/O Scheduler* will take into account how data is going to be accessed, and give a higher priority to data shared by several compute nodes to reduce contention as much as possible.
2. The *I/O Scheduler* will offer mechanisms to support the definition and enforcement of QoS constraints for an application I/O. This means that it must be possible to set limits on e.g. the number of metadata operations, the number of I/Os per second (IOPS), or the bandwidth consumed w.r.t. the shared PFS.
3. The *I/O Scheduler* will support multiple storage tiers, including at least the backend parallel file system, node-specific flash storage, and Storage Class Memory attached to the CPUs' memory buses. It will also allow scheduling bulk data transfers between the different storage tiers and will manage fine-grained accesses from one storage tier to another, honouring the aforementioned QoS guarantees.
4. The *I/O Scheduler* will include several data scheduling policies with different levels of aggressiveness, given that applications may benefit differently from each policy.

Additionally, the ADMIRE partners also decided that the *I/O Scheduler* should be responsible for executing all tasks related to deploying and terminating Ad-hoc Storage Systems in response to requests by Slurm. This is required, firstly, so that it is possible to stage data into them. Secondly, the Ad-hoc Storage System may be kept running for subsequent jobs in a workflow. And thirdly, it may also be necessary to complete any pending data transfers from the Ad-hoc Storage System to the PFS once all jobs have been completed. Since the *I/O Scheduler* will have first-hand knowledge of when these transfer tasks are complete, it is simpler to let it take care of this.

## 2.1 Requirements

For the *I/O Scheduler* to enable better usage of the HPC cluster's I/O stack, it needs to be able to capture information from end users' I/O usage, as well as to offer a set of particular functionalities

to successfully play its part in the ADMIRE framework. This section describes these requirements in detail.

### 2.1.1 Definition and capture of user information and I/O hints

Clients of the ADMIRE framework must be able to define I/O requirements for applications, either when requesting resource allocations or via specialised ADMIRE APIs. To this end, the *I/O Scheduler* must expose an API that allows the materialisation of I/O hints in the *Job Scheduler*. The following I/O parameters will be captured by *Job Scheduler* and communicated to the *I/O Scheduler*:

- **Datasets to stage-in:** This hint allows users of the framework to identify which datasets will be consumed by a job to produce meaningful results. This is needed by the *I/O Scheduler* to kickstart the transfer process from the PFS into the desired ad-hoc storage system so that data can be on location before the job requires it.
- **Datasets to stage-out:** This hint allows users of the framework to identify the datasets that are expected to be generated from an HPC job, as well as the location in the PFS where they should be written for long-term storage. This will be helpful to the *I/O Scheduler* in several ways: firstly, intermediate datasets produced by a job within an Ad-hoc Storage System that has not been tagged for PFS storage will be considered as temporary by default, being deleted when the job completes; secondly, it will allow the *I/O Scheduler* to determine what to do with the produced datasets, i.e., transfer them to the PFS for persistent storage or keep them in place/move them to other compute nodes for subsequent job executions.
- **Storage tiers:** The previous two arguments must include information that allows users to unequivocally specify the storage tiers where datasets reside. This will allow routing information for data transfers to be precisely defined and enforced.
- **I/O access patterns and modes:** This hint will allow users to define if a dataset will be used in a read-only, write-only, read-write, or read-modify-write manner. Additionally, it will also be possible to define whether a dataset is intended to be privately consumed or shared by several applications.

### 2.1.2 Definition and enforcement of QoS constraints

The goal of Quality of Service in ADMIRE is to throttle I/O performance to prevent the excessive allocation of I/O resources to a single process/group of processes, to avoid diminished performance due to starvation for the remaining active process in the system. As mentioned, the management of Resources in Job Schedulers (both Slurm or PBS) typically takes care of a fair allocation of system resources such as computation and memory but typically excludes the I/O [11]. This situation can be somewhat mitigated by the ability of some storage systems to cap the available capacity (per user or group/project) using quotas, but this feature misses the control of data and metadata traffic during a time frame.

These limitations have been acknowledged by the Lustre community and a more complete QoS implementation is under development. This implementation, often referred to as LIME [15], is based on the notion of a Token Bucket Filter (TBF), which allows controlling the rate of network service from the storage servers based on the client's identity. As Lustre is a network-attached file system, all demands both in terms of data and metadata are expressed by RPCs to storage servers from compute clients. Nevertheless, the lack of orchestration remains one of the main difficulties for delivering QoS for a PFS, since each server only has partial knowledge of the consumption of resources. The *I/O Scheduler* addresses this by offering a central control point to facilitate

this orchestration. With a centralised controller, it is possible to leverage the existing LIME mechanism in Lustre to guarantee a level of resource to specific processes. Thus, to implement this orchestration effectively, it is mandatory for the *I/O Scheduler* to model both QoS classes and constraints to enforce them when transferring datasets back and forth to or from the PFS.

Note that we consider that the target of a QoS class/constraint can be a dataset, a compute node or an application. We currently envision that the following QoS classes will need to be supported:

- **I/O Data Rate:** The *data rate* QoS class allows restricting the amount of data transferred between storage devices during a certain unit of time. This QoS class allows setting upper limits to the I/O bandwidth used by an application, which is intended to prevent an application with high data production from saturating the I/O channel.
- **I/O Operation Rate:** The *operation rate* QoS class allows restricting the number of operations per unit of time being emitted. This, for instance, allows restricting the metadata IOPs emitted by an application, which is often the cause of increased I/O latency for HPC applications.

Note that in D4.1 we also defined a *network data rate* QoS class. Unfortunately, we were unable to find established mechanisms to enforce it that could be easily integrated into the *I/O Scheduler* prototype. For now, development pursuing this QoS class is abandoned but we will revisit it before the project ends.

### 2.1.3 Execution of asynchronous data movement

The *I/O Scheduler* is responsible for executing and tracking data transfer tasks to move datasets between the PFS and the ad-hoc storage services. For efficiency's sake, it makes sense to perform these data transfers asynchronously, so that the initiators (i.e. applications and other ADMIRE components) don't need to wait for the transfers to be completed and may carry on with other tasks.

## 2.2 Architecture

This section describes the architecture of the *I/O Scheduler* component. To do so, we describe each sub-component while focusing on its desired functionalities. This allows us to determine the requirements for each sub-component and the interactions between them.

### 2.2.1 Design principles

As with other components in the framework, the *I/O Scheduler* follows ADMIREs principle of separation of concerns: the control flows and the data flows are strictly separated:

- The *control plane* for the *I/O Scheduler* encompasses all the messages transferred between the different ADMIRE components and the *I/O Scheduler*, as well as all internal messages exchanged between the *I/O Scheduler* sub-components. As mentioned, I/O requirements relevant to the *I/O Scheduler* are captured from applications via a Slurm-specific interface or a specific ADMIRE API (refer to Section 3.1.3 and Section 3.2), and are collected, processed, and analysed by the Intelligent Controller, which also receives relevant Malleability information from the Malleability Manager and incorporates it into the model. From there, this digested information is pushed regularly to the *I/O Scheduler*, so that it can be used to implement decisions via internal messages (i.e. RPCs) to its sub-components. As we will see

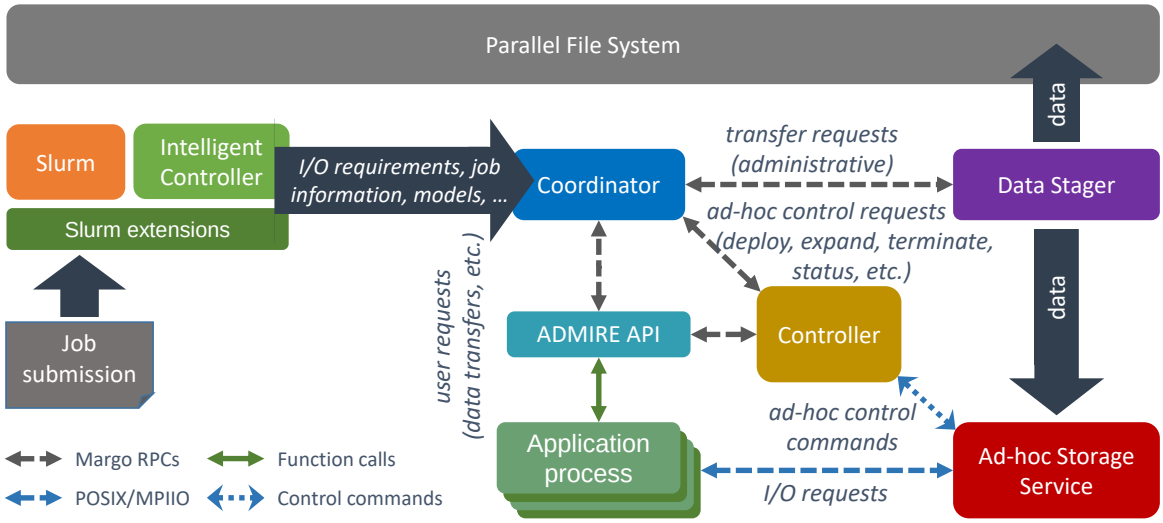



Figure 2.1: Abstract components of the I/O Scheduler

later, all sub-components in the *I/O Scheduler* report and react only to commands coming from the *I/O Scheduler* `Coordinator`, which is the sole contact point for the Intelligent Controller. This simplifies development and testing significantly (allowing us to implement and develop the `Coordinator` in commodity hardware), and also has the benefit of establishing clear boundaries and communication channels between ADMIRE components.

- The *data plane*, on the other hand, refers to all the I/O requirements (and derived models) required to implement its functionalities and is significantly simpler than the control plane. In ADMIRE, I/O requirements can be collected directly from users via the ADMIRE Slurm extensions, and from running applications via the ADMIRE programmatic API. Any derived models originate from data collected by the Monitoring Manager, as well as from application and storage states that are updated regularly. In any case, this information is collected and processed by the Intelligent Controller, which pushes it regularly to the *I/O Scheduler*, so that it is aware of the current I/O state of the system.

### 2.2.2 Components

To make the ADMIRE's *I/O Scheduler* as flexible as possible, its responsibilities are separated into three major sub-components as depicted by Figure 2.1: the `Coordinator`, the `Controller`, and the `Data Stager`.

- The `Coordinator` component is a process that acts as the interface between the *I/O Scheduler* and other ADMIRE components (most notably the Intelligent Controller), which means that it implements most of the interfaces defined by D4.1. The `Coordinator` is the main controller of all *I/O Scheduler* responsibilities, being in charge of exchanging information with the other ADMIRE components, processing it, making decisions, and sending appropriate commands to other *I/O Scheduler* sub-components. Since the `Coordinator` needs to oversee ADMIRE jobs, its lifetime is not linked to any specific application, but rather needs to be started when the ADMIRE framework is first deployed. As we will see later, the `Coordinator`'s actual implementation is that of a daemon capable of running as an infrastructure node by, e.g., being deployed in a login node. This allows it to have a global view of the I/O requirements of the system.

  The `Coordinator` is also in charge of splicing and communicating application-level I/O and data scheduling information to `Controllers` so that they can take it into account when

planning their next actions. We envision that a single `Coordinator` daemon will suffice for the current needs of the ADMIRE framework, but if that proved false in the future, we would explore a distributed solution with multiple `Coordinators`.

- The `Controllers` are management processes that act as the middle-man between applications, ad-hoc storage services, and the `Coordinator` daemon. As such, they handle commands from the `Coordinator` daemon (processing and relaying them) as needed, and also track an application state. The design goals of this component are to reduce the workload of the `Coordinator` by taking full responsibility for an application as its associated ad-hoc storage.

  `Controllers` are processes that run alongside applications and are in charge of receiving commands and requests from the `Coordinator` daemon and reacting to them. As such, there is a `Controller` process per application, that will be started once the node reservation is made available by Slurm. Before an application starts running, its `Controller` is in charge of setting up its environment according to the *user hints* defined when the job was submitted. This may include deploying, configuring and starting external software, such as an ephemeral ad-hoc storage system or the ADIOS2 framework for in-situ processing, for example.

  `Controllers` are also responsible for communicating I/O malleability requests from the `Coordinator` to running ad-hoc storage services (e.g. to expand or shrink an ad-hoc service) as well as servicing any application request coming from ADMIRE's specialised APIs.

- The `Data Stager` is a process responsible for asynchronously moving data back and forth between the ad-hoc storage services and the PFS on behalf of applications. For ADMIRE to be successful in limiting I/O contention, applications must be constrained to work primarily in their dedicated ad-hoc storage services, keeping them from accessing the PFS arbitrarily as much as possible. The `Data Stager` offers a mechanism to applications so that they can *request* transfers to/from the PFS while giving the `Coordinator` (and hence the ADMIRE framework itself) control over when to execute them. From a deployment point of view, two alternatives exist: a system-wide `Data Stager` or multiple per-application `Data Stagers`. As we will discuss in Section 3.1, when considering the implementation details we chose the latter since it simplifies the eventual adoption of the software.

As depicted by Figure 2.1, the *I/O Scheduler* workflow begins with a job submission from a user, which allows the ADMIRE framework to capture I/O requirements for the job via Slurm. The *Intelligent Controller* may now be able to generate extra information such as prediction models for the application and information about the system state gathered by the *Monitoring Manager*. All this information (I/O requirements, extended information, and system state) is forwarded via Margo RPCs to the `Coordinator`, which will process it and determine if the application requires an on-demand ad-hoc storage service. If so, a `Controller` will be started in the job's allocation which will be instructed to deploy the requested ad-hoc storage service. Once the service is running, the `Coordinator` will determine the best moment to start transferring data into it, as well as appropriate QoS constraints depending on the PFS's current condition. Then, transfer requests will be sent via RPCs to the `Data Stager`. When sufficient data is available in the ad-hoc storage service, the parallel application can initiate its work, which may include additional transfer requests via ADMIRE's specialised APIs.

# 3. Implementation

The abstract components described in Section 2.2.2 need to be implemented as distinct software solutions. Figure 3.1 revisits the *I/O Scheduler* architecture, but this time depicting the actual software packages that implement each sub-component of the prototype: `scord`, `scord-ctl`, and `Cargo` which, respectively, implement the `Coordinator`, `Controller`, and `Data Stager` abstract components. We also describe the implementation of the Slurm extensions as a Slurm plugin: `libslurmadmcli.so`. This section, then, moves away from the conceptual design components to discuss the actual software developed in WP4 to enable the ADMIRE vision of an HPC I/O infrastructure based on ad-hoc microservices.

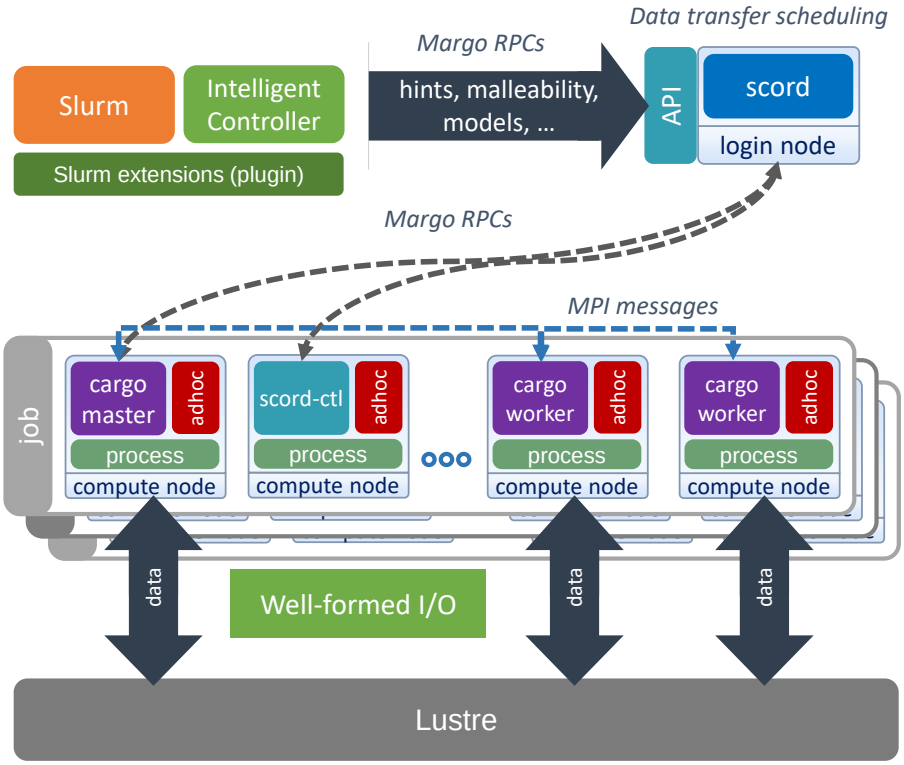



Figure 3.1: Architecture of the *I/O Scheduler* implementation

## 3.1 scord: Storage management and coordination

`scord`[1] is an open-source storage management service that provides facilities for system administrators to easily model their storage hierarchies and expose them to users in a consistent manner. Once the desired storage architecture is configured into `scord`, the service automatically deploys on-demand ad-hoc storage services for those HPC jobs requesting them, and efficiently

[1] https://storage.bsc.es/gitlab/eu/admire/io-scheduler

orchestrates data transfers between those services and the available storage tiers. In this way, the `scord` service simplifies the construction of complex HPC workloads using node-local storage, burst buffers and ad-hoc storage systems and makes careful usage of the I/O resources provided by the infrastructure.

The `scord` service (a portmanteau for *Storage Coordinator*), is the core of ADMIRE's *I/O Scheduler* implementation and, as such, is responsible for collecting job information and application models from other ADMIRE components. It then uses this information to orchestrate both the deployment of ad-hoc storage services and the data transfers required to move datasets between storage tiers. Similarly to other components in ADMIRE, the `scord` service is a daemon built around Margo RPCs[2], a highly scalable RPC library from Argonne National Labs, that offers a common interface for several HPC fabrics such as TCP, InfiniBand, Verbs, or Omni-Path. The `scord` service implements the APIs defined in D4.1, which are exposed as Margo endpoints to other components of the ADMIRE framework.

The service itself is composed of several sub-components:

- The `scord` server is the service's core component and its functionalities directly correspond to the `Coordinator` component described in Section 2.2.2. Thus, the `scord` server holds much of the control and orchestration logic and also serves as a single point of communication for other ADMIRE components. Its responsibilities are to internally model the I/O stack, keep track of jobs and their requirements, execute decision-making algorithms, and send commands to other sub-components for the deployment of on-demand ad-hoc storage services. It is also responsible for initiating, tracking, and steering data transfers between the different storage tiers and updating the *Intelligent Controller*'s REDIS database with up-to-date information about storage tiers. Finally, it serves as a control hub for any QoS enforcement policies such as those described in Section 2.1.2[3]. As an overseeing component, the `scord` server's lifetime is not associated with a single application and should thus be installed and run as an infrastructure service (e.g. a login node). The server itself does not require special privileges for its operation.

- The `scord-ctl` servers are `scord`'s workhorses and serve as a nexus between the `scord` server, ad-hoc storage services and applications. Their functionalities directly correspond to the `Controllers` described in Section 2.2.2. As such, a `scord-ctl` server is initiated alongside each application using a specialised prologue script (refer to Section 3.2 for details) once the resource allocation has been granted. The `scord-ctl` server then establishes a two-way communication channel with the `scord` server which, in the near future, will send an RPC requesting the deployment of an ad-hoc storage service for the initiating job. When this happens, the `scord-ctl` server executes the pre-configured scripts for the requested ad-hoc type (refer to Section 3.1.2) and reports back to `scord` so that the user's application can initiate. Note that the `scord-ctl` server is started as a process of the requesting user, which means that there are no issues with data ownership or permissions of execution.

- The `libscord.so` library and `scord.h[pp]` headers provide an interface for any ADMIRE components that need to communicate with the `scord` server. Rather than forcing other components to manually construct the Margo RPCs themselves, the library encapsulates the RPC communication into a C and C++ API that closely follows the interfaces defined in D4.1, Section 3.3. It offers administrative interfaces to register user jobs, storage tiers, and request and manipulate data transfers. The intended clients for this library are Slurm and the *Intelligent Controller*.

[2] https://mochi.readthedocs.io
[3] This functionality is currently under development

- `libscord-user.so` library and `scord-user.h[pp]` headers provide a C and C++ interface for application-specific functionalities involving the `scord` service. The library offers APIs, for instance, for applications to explicitly request asynchronous data transfers while they are running. Such transfer requests need to arrive at the `scord` server so that they can be taken into account in the overall data scheduling. Other planned interfaces that will be provided by this library are those supporting in-situ/in-transit data operations as described in D4.1, Section 3.5, which will be available once Task 4.3 completes.

### 3.1.1 Building and installing

This section describes how to build and install the `scord` service. Two main options are basically supported out of the box:

1. Automatically building both `scord` and its dependencies using Spack.
2. Building and installing `scord` manually.

For the sake of brevity, we only describe here the first option, since it is also significantly simpler. The instructions to manually build and install `scord` are described in detail in the project's README.

#### Building with Spack

`scord` and all its dependencies can be built using Spack, a flexible, multi-platform, package manager for supercomputers that can build, install, and make available multiple versions and configurations of scientific software.

**Installing Spack**

Spack can be installed by directly cloning its official repository, which will create a directory called `spack` in the user's machine. Once Spack is cloned, we recommend sourcing the appropriate script for the preferred shell, which will add Spack to the `PATH` and enable the use of the `spack` command. The following commands take care of this:

```
$ git clone -c feature.manyFiles=true
↪  https://github.com/spack/spack
$ . spack/share/spack/setup-env.sh       # For bash/zsh/sh
$ source spack/share/spack/setup-env.csh # For tcsh/csh
$ . spack/share/spack/setup-env.fish     # For fish
```

Since `scord` is not yet available in the official Spack repositories, `scord`'s Spack repository needs to be manually downloaded and added to the local Spack namespace in the target machine. To do so, the `/spack` directory found in `scord`'s repository[4] must be downloaded to the target machine (e.g. under `~/projects/scord/spack`), and the following commands need to be executed:

```
$ spack repo add ~/projects/scord/spack/
$ spack info scord
```

[4] https://storage.bsc.es/gitlab/eu/admire/io-scheduler/-/tree/main/spack

If that worked, `scord` is now ready to be installed:

```
$ spack install scord
```

Spack can optionally be asked to run any package tests when installing the software. For `scord`, tests are integrated in CTest, CMake's testing facility, and their execution can be requested with the following command:

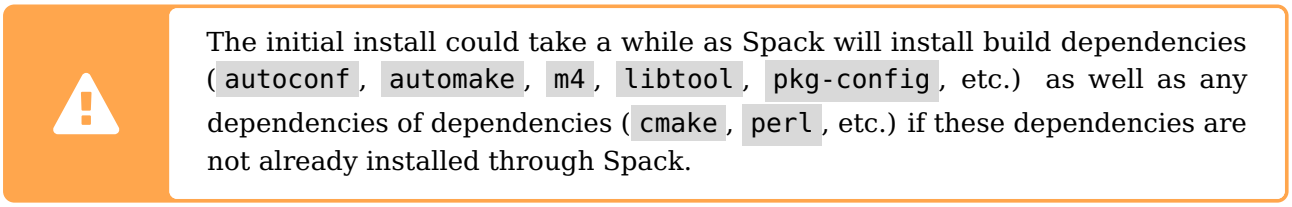
```
$ spack install --tests=root scord
```

Variants can be selected with Spack in case a custom `scord` build is desired, including specific options that are directly propagated to `scord`'s and `scord-ctl`'s configuration files. This can be useful for system administrators who wish to configure the servers from the command line without having to actually edit the configuration files in an interactive manner. The currently available variants are listed below:

| Variant | Command | Default | Description |
|---|---|---|---|
| OFI | `+ofi` | True | Use libfabric as transport library |
| UCX | `+ucx` | False | Use ucx as transport library (experimental) |
| address | `address` | `ofi+tcp://127.0.0.1:52000` | Specify the protocol, host and port where `scord` should listen for RPCs. |

For example, the following command could be used to install and configure a `scord` deployment where `libfabric` and `tcp` is used for communication, and the `scord` server listens for requests on address `192.168.1.128:52000`:

```
$ spack install --test=root scord +ofi address=ofi+tcp://192.168.1.128
```




After the installation completes, `scord` needs to be loaded in order to use it:

```
$ spack load scord
$ scord -h
scord - An orchestrator for HPC I/O activity
Usage: scord [OPTIONS] [SUBCOMMAND]

Options:
  -h,--help                   Print this help message and exit
  -f,--foreground             Run in foreground
  -C,--force-console Excludes: --output
                              Override any logging options defined in the
                              configuration file and send all daemon output
```

```
                              to the console
  -c,--config-file FILENAME   Ignore the system-wide configuration file and use the
                              configuration provided by FILENAME
  -o,--output FILENAME        Write any output to FILENAME console
  -v,--version                Print version string and exit
```

### 3.1.2 Configuration

The configuration for both `scord` and `scord-ctl` is done using configuration files that are installed by default in `$PREFIX/etc/scord.conf` and `$PREFIX/etc/scord-ctl.conf`. The listing below depicts the configuration options currently available for the `scord` server:

```yaml
## global server settings
global_settings:
  logfile: "/var/log/scord/scord.log"  # log file
  rundir: "/var/run/scord"             # working directory (for pidfile, etc.)
  address: "ofi+tcp://192.0.0.1:42000" # address where scord should expect requests
```

Conversely, the listing below depicts the settings currently available for the `scord-ctl` server:

```yaml
config:
  # Specific configurations for supported adhoc storage systems
  adhoc_storage:
    hercules:
      ...
    expand:
      ...
    gekkofs: # The tier ID presented to the user
      # The default working directory for adhoc services of this type
      working_directory: /tmp/gekkofs
      startup:
        # Specific environment variables that should be set for the adhoc
        # service. These will be merged with the environment variables
        # already set by Slurm.
        environment:
          VAR0: value0
          VAR1: value1
        # The command that scord-ctl will use to start an adhoc service of
        # this type. The following variables are supported that will be
        # automatically replaced by scord-ctl if found between curly braces:
        #  * ADHOC_NODES: A comma separated list of valid job hostnames that
        #    can be used to start the adhoc service.
        #  * ADHOC_DIRECTORY: A unique working directory for each specific
        #    adhoc service. This directory will be created by scord-ctl under
        #    `working_directory` and automatically removed after the adhoc
        #    service has been shut down.
        #  * ADHOC_ID: - A unique ID for the adhoc service.
        command: /usr/share/scord/adhoc_services.d/gekkofs.sh
                    start --hosts {ADHOC_NODES} --workdir {ADHOC_DIRECTORY}
                          --datadir {ADHOC_DIRECTORY}/data
                          --mountdir {ADHOC_DIRECTORY}/mnt
      shutdown:
```

```
        environment:
        command: /usr/share/scord/gekkofs.sh stop --workdir {ADHOC_DIRECTORY}

# default storage tiers made available to applications
storage:
  lustre:
    type: "pfs"
    mountpoint: "/mnt/lustre"
```

This configuration file is especially interesting because it demonstrates how the `scord-ctl` server supports the deployment of different ad-hoc storage services: any ad-hoc storage implementation that wishes to be integrated into the ADMIRE framework needs to provide one (or several) scripts implementing several control commands (e.g. `start`, `stop`, or `expand`, among others). By honouring this interface, the `scord-ctl` server can deploy and control widely different ad-hoc storage services without having to actually know any specific details about the services themselves: the actual logic is provided by the ad-hoc developers in the provided scripts.

Moreover, the `scord` service further leverages this API to support the deployment of concurrently running ad-hoc storage services even in the same compute nodes: the configuration file defines the template for each control command and the `scord` service simply generates different arguments for each independent service (e.g. separate working directories with `ADHOC_DIRECTORY`, unique ad-hoc identifiers with `ADHOC_ID`, etc.). The example below demonstrates how `scord-ctl` generates specific startup command so that two concurrently running ad-hoc storage services can be deployed in overlapping compute nodes:

```
################################################################################
# The following variables are generated by `scord` and `scord-ctl` for each
# service: ADHOC_ID, ADHOC_NODES, ADHOC_DIRECTORY
################################################################################

### First ad-hoc service: GekkoFS
ADHOC_NODES="tux[1,3-4],snoo[1-2]" # Based on SLURM_JOB_NODELIST
ADHOC_ID="gekkofs-iHRFGlekPYM41RjXUEC9j9gI7jnmDyuw"
ADHOC_DIRECTORY="/tmp/gekkofs/gekkofs-iHRFGlekPYM41RjXUEC9j9gI7jnmDyuw"

# Command eventually executed (variables applied to `scord-ctl.conf` template)
/usr/share/scord/adhoc_services.d/gekkofs.sh
    start --hosts "tux[1,3-4],snoo[1-2]"
          --workdir  "/tmp/gekkofs/gekkofs-iHRFGlekPYM41RjXUEC9j9gI7jnmDyuw"
          --datadir  "/tmp/gekkofs/gekkofs-iHRFGlekPYM41RjXUEC9j9gI7jnmDyuw/data"
          --mountdir "/tmp/gekkofs/gekkofs-iHRFGlekPYM41RjXUEC9j9gI7jnmDyuw/mnt"

### Second ad-hoc service: Hercules
ADHOC_NODES="tux[1,3-10],snoo[4-8]" # Based on SLURM_JOB_NODELIST
ADHOC_ID="hercules-X4fpMR9HHraYIt83jtqAkY6y5hTSUGG"
ADHOC_DIRECTORY="/tmp/hercules/hercules-X4fpMR9HHraYIt83jtqAkY6y5hTSUGG"

# Command eventually executed (variables applied to `scord-ctl.conf` template)
/usr/share/scord/adhoc_services.d/hercules.sh
    start --hosts "tux[1,3-10],snoo[4-8]"
          --workdir "/tmp/hercules/hercules-X4fpMR9HHraYIt83jtqAkY6y5hTSUGG"
```

As shown by the example, both `scord` and `scord-ctl` ensure that each ad-hoc storage

service is assigned a unique identifier and is started on separate directories. Note that the script invocations for each service are not identical: this is intentional to give developers a certain leeway on the arguments they need as long as they are based on the provided information. The design for the interface allowing this functionality was led by WP2 following a co-design effort with other WPs and is described in detail in D2.1.

### Testing the configuration

To test that everything works as it should, the service can be started by running `scord` server in foreground mode and redirecting its logging output to the console with the following command:

```
$ scord -f --force-console
[2023-07-11 16:06:51.320162] [scord] [0743] [info] ================================
[2023-07-11 16:06:51.320171] [scord] [0743] [info] Starting scord daemon (pid 0743)
[2023-07-11 16:06:51.320173] [scord] [0743] [info] ================================
[2023-07-11 16:06:51.320174] [scord] [0743] [info]
[2023-07-11 16:06:51.320174] [scord] [0743] [info] [[ Configuration ]]
[2023-07-11 16:06:51.320182] [scord] [0743] [info]   - running as daemon?: no
[2023-07-11 16:06:51.320196] [scord] [0743] [info]   - address for remote requests:
↪  "ofi+tcp;ofi_rxm://192.18.0.128:52000"
[2023-07-11 16:06:51.320197] [scord] [0743] [info]
[2023-07-11 16:06:51.320198] [scord] [0743] [info] [[ Starting up ]]
[2023-07-11 16:06:51.320199] [scord] [0743] [info]  * Installing signal handlers...
[2023-07-11 16:06:51.320251] [scord] [0743] [info]
[2023-07-11 16:06:51.320253] [scord] [0743] [info] [[ Start up successful, awaiting
↪  requests... ]]
```

Now we can use the `ADM_ping` program distributed with `scord` to send a `ping` RPC to the server `scord`:

```
$ ADM_ping ofi+tcp://192.168.0.128:52000
```

The server logs should update with an entry similar to the following one:

```
[2023-07-11 16:08:12.299] [scord] [0743] [info] rpc => id: 1 name: "ADM_ping" from:
↪  "ofi+tcp;ofi_rxm://192.18.0.128:32825" body: {}
[2023-07-11 16:08:12.299] [scord] [0743] [info] rpc <= id: 1 name: "ADM_ping" to:
↪  "ofi+tcp;ofi_rxm://192.18.0.128:32825" body: {retval: ADM_SUCCESS}
```

Once finished, the server can be stopped either by pressing Ctrl+C or when receiving a `SIGTERM` signal:

```
^C
[2023-07-11 16:09:26.448] [scord] [0743] [warning] A signal (SIGINT) occurred.
[2023-07-11 16:09:26.552] [scord] [0743] [info] ===================================
[2023-07-11 16:09:26.552] [scord] [0743] [info] Stopping scord daemon (pid 0743)
[2023-07-11 16:09:26.552] [scord] [0743] [info] ===================================
[2023-07-11 16:09:26.552] [scord] [0743] [info] * Stopping signal listener...
[2023-07-11 16:09:26.552] [scord] [0743] [info]
[2023-07-11 16:09:26.552] [scord] [0743] [info] [Stopped]
```

### 3.1.3 API examples

This section shows some examples of how the `libscord.so` and `libscord-user.so` APIs can be used to interact with a running `scord` service. Once again, we only demonstrate the C API, but the C++ API functionalities are analogous. Since the examples are necessarily brief, we urge the reader to peruse the `scord.h` and `scord-user.h` headers, as well as to look into the `/examples` directory in `scord`'s repository.

#### Administrative API

The listing below demonstrates how the `libscord.so` library can be used to register an upcoming job requiring an ad-hoc storage system into the `scord` service. Note that this is an administrative-level library that is intended to be used by internal ADMIRE components such as Slurm or the *Intelligent Controller*. The job's owner must have provided some basic information about the desired ad-hoc service, as well as the datasets used as input and the datasets to be generated. This code is very similar to the one we use in the codeslurm_spank_user_init() function in the Slurm SPANK plugin whenever a new job submitted from `srun`, `sbatch`, or `salloc` is detected (see Section 3.2).

```c
/**
 * Register a new job in the system.
 *
 * @param adhoc_name The alias of the ad-hoc storage to be used by the job.
 * @param adhoc_type The type of adhoc storage to be used by the job.
 * @param adhoc_walltime The walltime of the adhoc storage used by the job.
 * @param adhoc_nodes The nodes to be used by the adhoc storage.
 * @param nadhoc_nodes The number of nodes to be used by the ad-hoc storage.
 * @param job_nodes The nodes of the job.
 * @param njob_nodes The number of nodes of the job.
 * @param inputs The input datasets for the job.
 * @param ninputs The number of input datasets for the job.
 * @param outputs The output datasets for the job.
 * @param noutputs The number of output datasets for the job.
 */
void
register_job(const char* adhoc_name,
             ADM_adhoc_storage_type_t adhoc_type, int adhoc_walltime,
             ADM_node_t adhoc_nodes[], size_t nadhoc_nodes,
             ADM_node_t job_nodes[], size_t njob_nodes,
             ADM_dataset_t inputs[], size_t ninputs,
             ADM_dataset_t outputs[], size_t noutputs) {

    const char* scord_address = "ofi+tcp://192.18.0.128:52000";
    const char* scordctl_address = "ofi+tcp://192.18.0.4:42000";
    ADM_return_t ret = ADM_SUCCESS;
    ADM_server_t server = NULL;

    // adhoc information
    ADM_adhoc_resources_t adhoc_resources;
    ADM_adhoc_context_t adhoc_ctx;
    ADM_adhoc_storage_t adhoc_storage;

    // job information
    ADM_job_t job = NULL;
```

```
36      ADM_job_resources_t job_resources = NULL;
37      ADM_job_requirements_t job_reqs = NULL;
38      uint64_t slurm_job_id = 42;
39
40      // Let's prepare all the information required by the API calls.
41      // ADM_register_job() often requires an adhoc storage to have been
42      // registered onto the system, so let's prepare first the data required
43      // to call ADM_register_adhoc_storage():
44      //   1. the adhoc storage resources
45      if((adhoc_resources = ADM_adhoc_resources_create(adhoc_nodes,
46                                                       nadhoc_nodes)) == NULL) {
47          fprintf(stderr, "Fatal error preparing ad-hoc resources\n");
48          abort();
49      }
50
51      //   2. The adhoc storage execution context
52      if((adhoc_ctx = ADM_adhoc_context_create(
53                  scordctl_address, ADM_ADHOC_MODE_SEPARATE_NEW,
54                  ADM_ADHOC_ACCESS_RDWR, adhoc_walltime, false)) == NULL) {
55          fprintf(stderr, "Fatal error preparing adhoc context\n");
56          abort();
57      }
58
59      // All the information required by the ADM_register_adhoc_storage() API is
60      // now ready. Let's actually contact the server:
61      //   1. Find the server endpoint
62      if((server = ADM_server_create(scord_address)) == NULL) {
63          fprintf(stderr, "Fatal error creating server\n");
64          abort();
65      }
66
67      //   2. Register the adhoc storage
68      if((ret = ADM_register_adhoc_storage(server, adhoc_name, adhoc_type,
69                                           adhoc_ctx, adhoc_resources,
70                                           &adhoc_storage)) != ADM_SUCCESS) {
71          fprintf(stderr, "ADM_register_adhoc_storage() failed: %s\n",
72                  ADM_strerror(ret));
73          abort();
74      }
75
76      // Now that we have an existing ad-hoc storage registered into the
77      // system, let's prepare all the information for actually registering the
78      // job with ADM_register_job():
79      //   1. the job's resources
80      if((job_resources = ADM_job_resources_create(job_nodes, njob_nodes)) ==
81         NULL) {
82          fprintf(stderr, "Fatal error preparing job resources\n");
83          abort();
84      }
85
86      //   2. the job's requirements
87      if((job_reqs = ADM_job_requirements_create(
88                  inputs, ninputs, outputs, noutputs, adhoc_storage)) == NULL) {
89          fprintf(stderr, "Fatal error preparing job requirements\n");
90          abort();
```

```
    }

    // All the information required by the ADM_register_job() API is now ready.
    // Let's actually contact the server:
    if((ret = ADM_register_job(server, job_resources, job_reqs, slurm_job_id,
                               &job)) != ADM_SUCCESS) {
        fprintf(stderr, "ADM_register_job() failed: %s\n", ADM_strerror(ret));
        abort();
    }

    char* adhoc_storage_path;
    // We can now request the deployment to the server
    if((ret = ADM_deploy_adhoc_storage(server, adhoc_storage,
                                       &adhoc_storage_path)) != ADM_SUCCESS) {
        fprintf(stderr, "ADM_deploy_adhoc_storage() failed: %s\n",
                ADM_strerror(ret));
        abort();
    }

    // At this point, we have registered into scord a job's initial information
    // and we can let the job execute...
}
```

## Application-level API

The listing below demonstrates how applications can use the `libscord-user.so` library to request data transfers from the cluster's configured `lustre` storage tier, to the `gekkofs` ad-hoc storage service requested by the application upon submission. Please note that this library is currently under development and the final API may vary:

```
/**
  * User transfer example: Data transfers can be requested using the
  * ADM_transfer_datasets() interface. If successful, the call returns
  * an `ADM_transfer_t` handle that clients may use to check the
  * status of an ongoing data transfer using `ADM_transfer_wait()`.
  *
  * Supporting types and macros are provided: `ADM_transfer_status_t`,
  * `ADM_TRANSFER_SUCCEEDED`, `ADM_TRANSFER_FAILED`,
  * `ADM_TRANSFER_PENDING`, `ADM_TRANSFER_IN_PROGRESS`
  */
void
transfer_example() {
    const char* paths[N] = {"input00.dat", "inpu01.dat", "input02.dat",
                            "input03.dat", "input04.dat"};
    enum { N  = 5 }; // the number of files to transfer
    ADM_dataset_t sources[N];
    ADM_dataset_t targets[N];

    for(size_t i = 0; i < N; ++i) {
        sources[i] = ADM_dataset_create("lustre", paths[i]);
        targets[i] = ADM_dataset_create("gekkofs", paths[i]);

        if(!sources[i] || !targets[i]) {
            abort();
```

```
        }
    }

    ADM_return_t ret = ADM_SUCCESS;
    ADM_transfer_t tx;

    // the library will automatically route the request to the `scord`
    // server configured in the cluster
    if((ret = ADM_transfer_datasets(sources, N, targets, N, &tx)) !=
       ADM_SUCCESS) {
        fprintf(stderr, "ADM_transfer_datasets() failed: %s\n",
                ADM_strerror(ret));
        abort();
    }

    ADM_transfer_status_t status;
    do {
        struct timespec timeout = {.tv_sec = 1, .tv_nsec = 0};
        // Wait for the transfer to complete
        ret = ADM_transfer_wait(tx, &status, &timeout);

        if(ret != ADM_SUCCESS && ret != ADM_ETIMEOUT) {
            fprintf(stderr, "ADM_transfer_wait() failed: %s\n",
                    ADM_strerror(ret));
            abort();
        }

        if(ADM_TRANSFER_SUCCEEDED(status)) {
            fprintf(stdout, "Transfer completed successfully\n");
        } else if(ADM_TRANSFER_FAILED(status)) {
            fprintf(stderr, "Transfer failed: %s\n",
                    ADM_strerror(ADM_TRANSFER_ERROR(status)));
        } else if(ADM_TRANSFER_PENDING(status)) {
            fprintf(stdout, "Transfer pending\n");
            continue;
        } else if(ADM_TRANSFER_IN_PROGRESS(status)) {
            fprintf(stdout, "Transfer in progress\n");
            continue;
        } else {
            fprintf(stderr, "Transfer status unknown\n");
            abort();
        }
    } while(!ADM_TRANSFER_SUCCEEDED(status));
}
```

## 3.2 Slurm extensions

As discussed in Section 2.1.1, information about applications (and the jobs used to execute them) needs to be made available to the *I/O Scheduler* components. To support this, `scord` includes an extension plugin for Slurm that extends `srun`, `sbatch`, and `salloc` with additional CLI arguments to capture this information. Moreover, it also adds the machinery required to communicate Slurm with `scord` via the API defined in D4.1. In this way, the plugin can capture both job information and ADMIRE I/O hints from the submission process itself, and send it to

`scord` using Margo RPCs. This kickstarts all the process that allows `scord` to (1) register a job and its requirements into the system; (2) prompt `scord-ctl` to deploy its required ad-hoc storage service; and (3) prompt `cargo` to transfer data into it.

This Slurm plugin consists of a shared object (`libslurmadmcli.so`) that is loaded by Slurm's plugin system and serves to extend its functionalities without having to actually modify Slurm's code itself. It also includes some prologue and epilogue scripts[5] for Slurm that are used to prepare a job's environment for utilisation in the ADMIRE ecosystem. The `libslurmadmcli.so` shared library itself is a Slurm SPANK[6] plugin that extends Slurm's command line arguments to enable users to provide information for `scord`, thus serving as a simple entry point into ADMIRE's automatic functionalities.

### Installation and configuration

The core of the plugin is written in C and requires the `libslurm.so` and the `libscord.so` C libraries to be compiled. It also requires access to the Slurm `spank.h` header. The plugin is compiled as a shared object that is automatically loaded by the Slurm daemons and submission programs, which gives us a fair amount of control over the different stages in a job execution.

ADMIRE's Slurm plugin is included in `scord`'s distribution (under the `plugins/slurm/` directory) and will be compiled by default when building the service either manually or from a Spack recipe. If users wish to only build the plugin, they can do so with the following commands:

```
$ git clone https://storage.bsc.es/gitlab/eu/admire/io-scheduler.git scord
$ cd scord
$ mkdir build && cd build
$ cmake ..
$ make -j <nprocs> slurm-plugin install
```

Once the shared object is compiled, Slurm needs to be configured in order to find it and use it, which is done by adding an appropriate entry in the Slurm plugin configuration file (usually `/etc/slurm/plugstack.conf`):

```
include /etc/slurm/plugstack.conf.d/*.conf
optional  /path/to/libslurmadmcli.so  scord_addr=<ADDRESS>
↪  scordctl_bin=/path/to/scord-ctl
```

The key-value pairs following the plugin are optional configuration variables:

1. `scord_addr`: The address to contact the Scord service in Mercury format. For instance, if `scord` has been configured to listen on port `52000` on a machine with the IP address `192.168.1.128` and using `tcp` as the transport protocol, the resulting address would be `ofi+tcp://192.168.1.128:52000`.

2. `scordctl_bin`: The absolute path to the `scord-ctl` binary to run on every node of an allocation, can be the path to an executable (default to scord-ctl).

For example, if `libslurmadmcli.so` is installed in `/usr/local/lib/`, `scord-ctl` is installed in `/usr/local/bin/`, and `scord` is configured to run on `ofi+tcp://192.168.1.128:52000`, the following entry should be added to `plugstack.conf`:

[5] https://slurm.schedmd.com/prolog_epilog.html

[6] Please refer to manual page `man 7 spank` and `<slurm/spank.h>` for details

```
include /etc/slurm/plugstack.conf.d/*.conf
optional  /usr/local/lib/libslurmadmcli.so  scord_addr=ofi+tcp://192.168.1.128:52000
↪  scordctl_bin=/usr/local/bin/scord-ctl
```

Besides the shared library, the plugin also installs `prologue` and `epilogue` scripts for job control under `$PREFIX/share/scord/slurm/`. In order to enable them for Slurm use, the following lines should be added to Slurm's configuration file (where `$PREFIX` should be replaced with the path where `scord` is installed):

```
Prolog=$PREFIX/share/scord/slurm/scord_prolog.sh
Epilog=$PREFIX/share/scord/slurm/scord_epilog.sh
```

### Usage

The plugin extends Slurm's command line arguments to allow users to request the deployment of ad-hoc storage services for their jobs. The following arguments for this purpose are added to `srun`, `sbatch` and `salloc`:

- `--adm-adhoc`: The job requires an adhoc storage service. The following types are supported:
  - `gekkofs`: The job requires the deployment of the GekkoFS ad-hoc file system.
  - `expand`: The job requires the deployment of the Expand ad-hoc file system.
  - `hercules`: The job requires the deployment of the Hercules ad-hoc file system.
  - `dataclay`: The job requires the deployment of the dataClay ad-hoc object-store.

  The ad-hoc storage service will be deployed on the same nodes as the compute nodes and its nodes will be shared with the application. The number of nodes assigned for the ad-hoc storage service can be controlled with the `--adm-adhoc-nodes` option. If not specified, the deployed ad-hoc storage service will share all the nodes assigned to the job.

- `--adm-adhoc-exclusive`: The ad-hoc storage service will be deployed on the same nodes as the compute nodes, but the ad-hoc nodes will not be shared with the application. The number of nodes assigned for the ad-hoc storage service MUST be specified with the `--adm-adhoc-nodes` option and cannot be greater than the number of nodes assigned to the job. Note, however, that the value of `--adm-adhoc-nodes` must be smaller than the value of `--nodes` (or `--ntasks`). Otherwise, the application would have no resources to run on.

- `--adm-adhoc-dedicated`: The ad-hoc storage service will be deployed in an independent job allocation and all the nodes for the allocation will be available for it. An `adhoc_id` will be generated for it and will be returned to the user so that other jobs can use the deployed ad-hoc storage service. In this mode, the resources assigned to the ad-hoc storage service can be controlled with the usual Slurm options (e.g. `--nodes`, `--ntasks`, `--time`, etc.).

- `--adm-adhoc-remote <adhoc_id>`: The job will use a remote and dedicated ad-hoc storage service that must have been previously requested in a different submission with the `--adm-adhoc-dedicated` option. An identifier for that ad-hoc storage service must be provided as an argument.

Users can also fine-tune to a certain extent how their requested resources will be distributed between the deployed ad-hoc storage service and the application by using the following CLI options:

- `--adm-adhoc-nodes`: The number of nodes to use for the ad-hoc storage service. The nodes will be allocated from the same partition as the compute nodes. This option is only valid when used with `--adm-adhoc` or `--adm-adhoc-exclusive`.

In addition to the options for ad-hoc management described above, the plugin also extends Slurm commands with the following options for capturing I/O hints from the user:

- `--adm-input "<origin> <target>"`: Define datasets that should be transferred between the PFS and the deployed ad-hoc storage system and where they should be stored.
- `--adm-output "<origin> <target>"`: Define datasets that should be automatically transferred between the ad-hoc storage system and the PFS. The ad-hoc storage will guarantee that the dataset is not transferred while there are processes accessing the file. The datasets will be transferred before the job allocation finishes if at all possible, but no hard guarantees are made.
- `--adm-expect-output "<origin> <target>"`: Define datasets that are expected to be generated by the application. When using this option, the application itself must use the ADMIRE programmatic APIs to explicitly request the transfer of the datasets (see Section 3.1.3 for more details).
- `--adm-inout "<origin> <target>"`: Define datasets that should be transferred from the PFS to the ad-hoc storage and back when finished.

Both `origin` and `target` must be valid dataset definitions of the form `storage-id:path`, where `storage-id` represents the identifier of the involved storage tier (e.g. `lustre`, `gekkofs`, `tmp`, etc.) and `path` represents a valid dataset identifier for that particular storage tier (e.g. an object ID for a KV store or a path for a POSIX-compatible file system)[7]. Storage identifiers can be separated into *static* and *dynamic* ones: static ones represent stable infrastructure-level storage tiers that rarely change, such as a PFS (e.g. `lustre`, `gpfs`, ...) or a shared burst buffer. They are defined by system administrators as part of `scord`'s configuration process so that users can refer to them in any ADMIRE-enabled job submission. Dynamic storage identifiers, on the other hand, are generic aliases that allow referencing the specific mount points created on demand by `scord-ctl` for each deployed ad-hoc storage system (refer to Section 3.1.2 for more details). Thus, a `gekkofs`/`hercules` storage identifier in a job submission will always refer to the GekkoFS/Hercules instance requested (and deployed) for this particular job, regardless of where the services were actually mounted. To simplify batch scripting, the provided prologue script also exports the `ADM_ADHOC_MNT` environment variable so that users may refer to a dynamically generated ad-hoc storage mount point in a generic way.

### Examples

The listing below shows some examples of how the new CLI options can be combined to request the automatic deployment of ad-hoc storage services and the movement of input and output datasets:

[7] Due to limitations in Slurm argument processing, the dataset routing must be quoted.

```
$ cat script.sh
#!/bin/bash
#SBATCH --adhoc-input="lustre:/$USER/input000.dat gekkofs:/input000.dat"
#SBATCH --adhoc-inout="lustre:/$USER/in_out_132.info gekkofs:/in_out132.info"
#SBATCH --adhoc-output="gekkofs:/results000.dat lustre:/$USER/%j/results000.dat"
#SBATCH --ntasks-per-node=32
#SBATCH --time=00:120:00
#SBATCH --output=script.%j.out
#SBATCH --error=script.%j.err

# [Step 1] Data preparation:
#   `genmap` is a temporary file generated directly into the ad-hoc service.
#    It never makes it to the PFS
srun -n 10 prepare ${ADM_ADHOC_MNT}/genmap

# [Step 2] Execute the actual simulation:
#   `input000.dat` is an input file consumed by the job
#   `results000.dat` is an output file generated by the job
#   `in_out132.info` is an input/output file updated by the job
srun -n 256 parallel_job \
     --input ${ADM_ADHOC_MNT}/input000.dat \
     --output ${ADM_ADHOC_MNT}/results000.dat \
     > ${ADM_ADHOC_MNT}/in_out132.info

# Run `script.sh` with a GekkoFS ad-hoc storage service in the same allocation
# and share allocation resources with it
$ sbatch --adm-adhoc gekkofs script.sh
Submitted batch job 30

# Run `script.sh` with a GekkoFS ad-hoc storage service in the same allocation.
# Resources are kept separate:
#   ad-hoc services uses 8 nodes,
#   application uses remaining nodes (8)
$ sbatch --adm-adhoc gekkofs --adm-adhoc-exclusive --adm-adhoc-nodes 8 script.sh
Submitted batch job 34

# Request the deployment of a dedicated GekkoFS adhoc storage service in its own
# allocation (since this job is only used for the ad-hoc service and nothing else,
# no script it's actually needed, but `sbatch` doesn't allow us to use an empty
# script. Thus, we trick it by wrapping bash's noop command `:`)
$ sbatch --adm-adhoc gekkofs --adm-adhoc-dedicated --adm-adhoc-nodes 10 --time
↪  00:200:00 --wrap :
Submitted batch job 41
Will deploy adhoc storage 123456

# Run `script.sh` with a dedicated GekkoFS storage service in its own allocation.
# To ensure that the GekkoFS instance is running, we set an explicit `after`
# dependency for its jobid.
$ sbatch --adm-adhoc gekkofs --adm-adhoc-remote 123456 --dependency=after:42
↪  script.sh
Submitted batch job 42
```

**Differences from D4.1**

There have been several changes from the API designed in D4.1. Most of these changes were made for the sake of clarifying user experience or due to limitations in the implementation. We list them here for the sake of completeness.

1. The extensions to the dataset naming mechanism that allowed defining dataset slices (see D4.1, Section 3.1) have been dropped. The reason is that none of the project's use cases actually had a need for them.

2. The ad-hoc options controlling *Lifetime and resource allocation* (see D4.1, Section 3.2.2) with `--adm-adhoc-context` have been refactored and renamed. The rationale behind this decision has been to simplify the user experience as well as to adapt existing Slurm terminology rather than inventing our own:

   - The `--adm-adhoc-context` and `--adm-adhoc-context-id` options have been removed in favour of simpler options.
   - Option `in_job:shared` has become the default configuration mode when requesting an ad-hoc service via `--adm-adhoc <type>`.
   - Option `in_job:dedicated` has become `--adm-adhoc <type> --adm-adhoc-exclusive`.
   - Option `in_job:new` has become `--adm-adhoc <type> --adm-adhoc-dedicated`.
   - Option `separate:existing` has become `--adm-adhoc <type> --adm-adhoc-remote <id>`.

3. The `--adm-adhoc-walltime` has been removed since Slurm's stock `--time` option can be used for its intended purposes.

4. Extra options controlling an ad-hoc behaviour after deployment such as `--adm-adhoc-access` or `--adm-adhoc-distribution` will be integrated in the near future into a more general `--adm-adhoc-options` option accepting a list of key-value pairs. The reason is that this approach is more flexible than hard-coded options.

## 3.3 Cargo: Parallel data staging

`Cargo`[8] is a data staging service for HPC that runs alongside applications, transferring their data between compute nodes and the PFS in a parallel manner. `Cargo` implements the final `Data Stager` sub-component of the *I/O Scheduler* described in Section 2.2.2 and, as such, its main responsibilities are to receive requests for transferring data from its clients, and to make effective use of the available bandwidth as much as possible.

As with `scord`, `Cargo` is built around Margo RPCs, and has been designed to be able to work either as a central infrastructure service or as a companion process for a parallel job. In ADMIRE we chose to deploy a `Cargo` instance for each job running in the system and coordinate all of them using `scord`. The reason is that while it would be simpler to have a single `Cargo` service managing

[8] https://storage.bsc.es/gitlab/hpc/cargo.git

the transfers, this service would need to write data belonging to different users and, as such, would require special privileges or capabilities which will hinder widespread adoption until the project is well-trusted. To prevent this, `Cargo` is deployed in the context of a running job, belonging to the job's owner, which allows it to access their data without any special considerations.

To achieve these goals, `Cargo` is organised as follows:

- A `master` process listening for requests (Margo RPCs) from `Cargo` clients. This process is responsible for processing any transfer requests arriving via RPC and dispatching work as MPI messages.

- Several `workers` that execute the transfer tasks in parallel. The workers are MPI processes that coordinate using the MPI/IO parallel I/O interface [5, 13] to transfer data.

- A `libcargo.so` C++ library and `cargo.hpp` header file that clients can use to send data transfer requests to a running `Cargo` service. For ADMIRE, the intended client for this API is only `scord`, since any transfers requested in this way will be handled directly by the targetted `Cargo` instance. Running ad-hoc storage systems and applications may also initiate data transfers, but they should be requested and handled either via the `libscord.so` or `libscord-user.so` APIs described in Section 3.1, respectively. The reason is that these transfers should be taken into consideration by `scord` for data scheduling purposes and, thus, `scord` should be notified of the client's *desire* to execute them but it should `scord`'s sole responsibility to actually *initiate* them.

**Differences from DoA/D4.1**

The initial plan stated in the DoA (and also D4.1) was to use the `NORNS` service[a] to support ADMIRE's data staging requirements. `NORNS` is an infrastructure service developed by BSC that allows moving data between different storage backends in a one-to-one fashion, including file-to-file data transfers via POSIX I/O, as well as file-to-memory and memory-to-memory transfers via RDMA [10].

As Task 4.2 progressed, however, we realised that ADMIRE's requirements were more oriented towards parallel data transfers than point-to-point communication. Moreover, `NORNS` requires special capabilities in order to read and write data from different Linux users, which made it less convenient. For this reason, we decided to write the `Cargo` service as a new development, as a more flexible alternative focused towards ADMIRE needs. Note, however, that even if `NORNS` was not finally adopted into the framework, most of `Cargo`'s code was repurposed from it.

[a] https://storage.bsc.es/gitlab/hpc/norns

### 3.3.1 Building Cargo

Similarly to `scord`, `cargo` and its dependencies can be built using Spack. Once again, since `cargo` is not yet available in the official Spack repositories, the `cargo` Spack repository needs to be added to the Spack namespace in the target machine. To do that, the `/spack` directory in `cargo`'s repository must be downloaded (e.g. to `~/projects/cargo/spack`, and the following commands executed:

```
$ spack repo add ~/projects/cargo/spack/
$ spack install cargo
```

Include or remove variants with Spack when a custom ‘Cargo‘ build is desired. The available variants are listed below:

| Variant | Command | Default | Description |
|---|---|---|---|
| OFI | `+ofi` | True | Use libfabric as transport library |
| UCX | `+ucx` | False | Use ucx as transport library (experimental) |
| address | `address` | `ofi+tcp://127.0.0.1:62000` | Specify the protocol, host and port where `cargo` should listen for RPCs. |

After the installation completes, `cargo` needs to be loaded in order to use it:

```
$ spack load cargo
$ cargo -h
Usage: cargo [options]

Options:
  -C [ --force-console ]  override any logging options defined in configuration
                          files and send all daemon output to the console
  -v [ --version ]        print version string
  -h [ --help ]           produce help message
```

### 3.3.2 Testing Cargo

Since `cargo` is an MPI application it cannot (yet) be tested automatically when installing. Even so, the tests can be manually executed provided that Spack is instructed to keep the build tree instead of deleting it automatically. This can be achieved with the following commands:

```
$ spack install --keep-stage cargo +tests
$ spack cd --build-dir cargo

# start the Cargo master + MPI workers
spack-build-z7f7xav $ mpirun -np 4 src/cargo -C &

# in another TTY execute the tests
spack-build-z7f7xav $ ./tests -S ofi+tcp://127.0.0.1:52000

# cleanup
$ spack clean --stage cargo
```

### 3.3.3 Using the Cargo API

As discussed in the previous sections, `Cargo` offers a C++11 API for clients that wish to request data transfers from a running server. The listing below demonstrates how this API can be used to

read data in parallel from a PFS into a POSIX file system while executing some compute-intensive code in the meantime:

```
/**
 * Read files from a PFS into a non-parallel FS and do some compute
 * intensive code in the meantime.
 *
 * @param[in] sources: A list of existing paths to read data from.
 * @param[in] targets: A list of output paths were data should be written.
 * @param[in] hv: A hypervector with data to process
 * @return: A hypervector resulting from processing `hv` and `sources`
 */
somelib::hypervector
read_files_and_compute(const std::vector<std::string>& sources,
                       const std::vector<std::string>& targets
                       somelib::hypervector& hv) {

    // transform the list of pathnames into something Cargo understands
    std::vector<cargo::dataset> cargo_sources;
    std::transform(sources.begin(), sources.end(),
                   std::back_inserter(cargo_sources),
                   [](auto path) {
                       return cargo::dataset{path, cargo::dataset::type::parallel};
                   });

    std::vector<cargo::dataset> cargo_targets;
    std::transform(targets.begin(), targets.end(),
                   std::back_inserter(cargo_targets),
                   [](auto path) {
                       return cargo::dataset{path, cargo::dataset::type::posix};
                   });

    // The address to the Cargo instance that should execute the transfers
    cargo::server server{"ofi+tcp://192.168.1.5:62000"};

    // Send the transfer request:
    //   `cargo::transfer_datasets()` does not block, but returns a
    //   `std::future<cargo::transfer_status>` that clients can use to
    //   wait for completion if needed
    std::future<cargo::transfer_status> f =
        cargo::transfer_datasets(server, cargo_sources, cargo_targets);

    // Do some interesting preprocessing for some time with the input
    // hypervector while data is begin moved...
    auto metadata = compute_intensive_code_phase1(hv);

    try {
        // Try to get the `cargo::transfer_status` from the future:
        //   This will block if the transfer has not been completed yet, or
        //   it will throw if the transfer finished in error. If everything
        //   worked well, however, a valid `cargo::transfer_status` will be
        //   returned with information about the transfer.
        cargo::transfer_status = f.get();

        // do something even more interesting with the data processed
```

```
53          // in the first stage and the files transferred asynchronously
54          return compute_intensive_code_phase2(hv, metadata, targets);
55      }
56      catch(const std::exception& e) {
57          std::cerr << "Transfer failed: " << e.what() << "\n";
58          throw; // rethrow exception to the caller
59      }
60  }
```

### 3.3.4 Experiments

While `Cargo`'s parallel data staging should intuitively be faster than sequential copying under most circumstances, it is worth it to actually evaluate its implementation to see where we stand. This section presents an initial evaluation of `Cargo`'s data staging performance without the usage of ad-hoc storage systems. To evaluate the service we ran a number of experiments targeting process and node scalability on the MOGON II supercomputer at JGU. The MOGON II supercomputer consists of 1,876 nodes in total, with 822 nodes using Intel 2630v4 Intel Broadwell processors (two sockets each) and 1,046 nodes using Xeon Gold 6130 Intel Skylake processors (four sockets each). The Skylake nodes were used in all experiments. MOGON II uses a 100 Gbit/s Intel Omni-Path interconnect to establish a fat tree between all compute nodes, providing a 7.5 PiB storage backend managed by several Lustre file systems. Further, since `Cargo` only relies on the POSIX and MPI-IO interfaces to read and write data, no special conditions were required for the experiments. The ROMIO configuration was also not modified for these experiments.

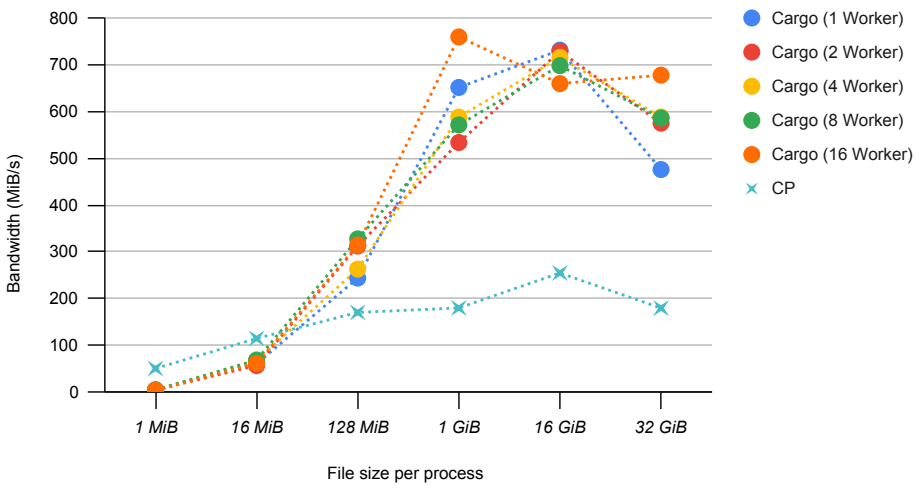



Figure 3.2: Cargo's single node performance when copying files of varying sizes from Lustre to the node's memory in several worker configurations compared with the Linux `cp` command.

In the first experiment, `Cargo` transferred files from Lustre to a single node in multiple worker configurations to evaluate its worker scalability. We established a baseline using the standard Linux `cp` utility to copy files, and we compared `Cargo`'s performance against it. Both tools copied files of varying sizes, ranging from 1 MiB to 32 GiB, from a specified directory in Lustre to the compute node's memory. In this experiment, the number of `Cargo` workers corresponding to MPI processes ranged from 1 to 16. Figure 3.2 depicts `Cargo`'s performance against the `cp` baseline in terms of bandwidth, with different numbers of workers.

As described in Section 3.3, the service creates a master process and some worker processes to handle I/O operations and establish connections for file transfers. These operations impose an

overhead which will be called transfer overhead in this section. The transfer duration consists of the latency of reading from the source directory (in Lustre), the latency of writing to the target directory (compute node memory), and the transfer overhead (network communication). The transfer overhead remains constant for every file transfer due to the same operation for each transfer, while the latency of read/write operations increases with the file size. For small files, e.g., 1 MiB and 16 MiB files, we expected `cp` to be faster than `Cargo`, which was confirmed by the experiment. However, the small difference in performance between `Cargo` and `cp` suggests that the transfer overhead is negligible. As the file sizes exceed 128 MiB, `Cargo` exhibits a significant performance improvement compared to `cp` regarding bandwidth. For instance, `Cargo` can transfer a 32 GiB file approximately 3.7 times faster than `cp`.

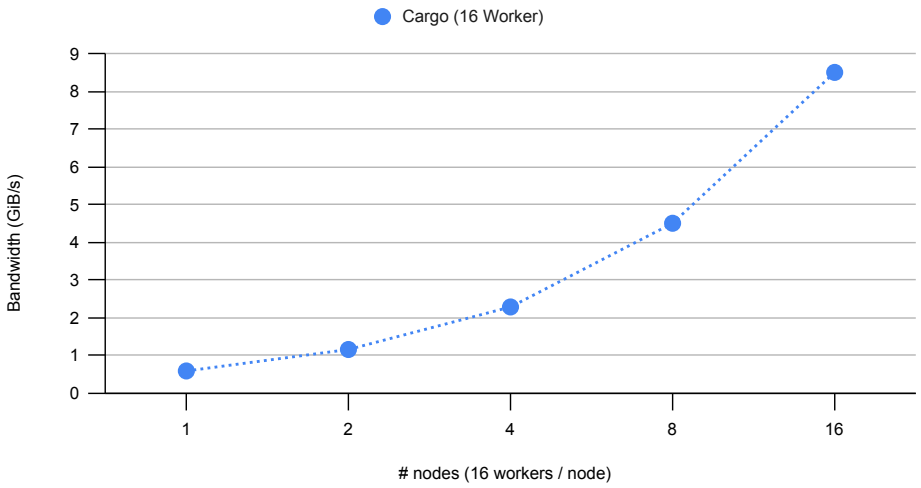



Figure 3.3: Cargo's node scalability when copying 32 GiB files from Lustre to the nodes' memory.

The second experiment assesses `Cargo`'s scalability. To achieve this, a theoretical scenario is assumed where each node runs an independent job that only performs read operations on the same dataset. Since none of the nodes attempts to modify the files, conflicts in access are avoided. Based on these assumptions, it is possible to copy files to the local space of each node, enabling them to perform their respective jobs using these files. To ensure acceptable performance and scalability, a copy tool is required to improve its performance as the number of nodes increases. Note, however, that this experiment essentially replicates the dataset on each node involved: when an ad-hoc storage system is used, the entire dataset only needs to be copied once into the ad-hoc storage system's single namespace, which is shared across all nodes that are part of the compute job. In this case, investigating `Cargo`'s scalability is relevant because input files can be copied from Lustre to the ad-hoc storage system's namespace from multiple nodes in parallel to increase stage-in performance.

Figure 3.3 illustrates this process and shows `Cargo`'s bandwidth when transferring a 32 GiB file from Lustre to the node's memory on up to 16 nodes. The `Cargo` workers were set to 16 as it performed best in the first experiment. The experiment demonstrates that Cargo scaled linearly with the number of nodes, achieving up to 8.5 GiB/s data throughput.

# 4. Supporting software and algorithms

The software described in Section 3 provides the necessary scaffold for the *I/O Scheduler* to support the ADMIRE framework, but does not describe the control logic behind it. This section describes additional supporting software developed in the project that will be integrated in the near future to actually provide the control mechanisms for data transfers, PFS I/O, and QoS. The general goals that we want to fulfil in the remainder of the project are the following:

1. *Contention-aware scheduling of I/O requests at the PFS level.* This includes the automatic classification of applications into groups according to their I/O requirements as well as the automatic prioritisation of such application groups to determine request ordering.

2. *Dynamic computation of QoS limits.* Being able to efficiently allocate the bandwidth and IOps from the PFS between applications means that the I/O flows at the PFS level can be restricted and distributed fairly according to system conditions. These limits can be used to steer both Lustre's LIME subsystem (see Section 2.1.2) as well as data staging operations.

3. *Modelling and prediction of an application data needs.* Predicting dataset usage from an application theoretically allows moving data into an ad-hoc storage service just in time the application actually needs it. This has benefits in that it provides fine-grained control of an ad-hoc storage capacity and it also allows separating a job's data transfers in time, which simplifies management.

To achieve these goals, control strategies need to be deployed at different levels in the *I/O Scheduler* hierarchy (e.g. the `Coordinator`, the `Data Stager`, and even the PFS itself), and need to be fed with metrics derived from the actual behaviour of applications. In this section, we describe the control strategies developed in WP4 and the modelling tools they require to properly function. Then, we discuss how we plan to integrate these solutions in the ADMIRE architecture.

## 4.1 IO-Sets

I/O scheduling strategies try to decide algorithmically which application(s) are prioritised (e.g. first-come-first-served or semi-round-robin) when accessing the shared PFS. Previous work [7] thoroughly demonstrated that existing approaches based on either *exclusivity* or *fair-sharing* heuristics showed inconsistent results, with exclusivity sometimes outperforming fair-sharing for particular cases, and vice versa. Based on these observations, we decided to research an approach capable of combining both by grouping applications according to their I/O frequency. As a result, we designed `IO-Sets`, a novel method for I/O management in HPC systems.

In `IO-Sets`, applications are categorised into *sets* based on their characteristic time, representing the mean time between I/O phases. Applications within the same set perform I/O exclusively, one at a time. However, applications from different sets can simultaneously access I/O and share the available bandwidth. Each set is assigned a priority determining the portion of the I/O bandwidth applications receive when performing I/O concurrently. Therefore, proposing a

heuristic in the `IO-Sets` method involves addressing two important questions: (i) how to allocate an application to a set, and (ii) how to define the priority of a set.

We answer these questions by developing a scheduling heuristic called `Set-10`, which is a simple method that follows the `IO-Sets` approach and requires minimal information about the applications. The `IO-Sets` method, the `Set-10` heuristic, and an extensive evaluation can be found in a corresponding paper [3].

### 4.1.1 Set-10

To schedule the I/O requests of applications using `Set-10`, the *characteristic time* of the applications is required. The characteristic time, denoted as $w_{\text{iter}}$, is defined as:

$$w_{\text{iter}} \stackrel{\text{def}}{=} \frac{\sum_{i \leq n} t_{\text{cpu}}^{(i)} + t_{\text{io}}^{(i)}}{n}.$$

Where $n$ represents the number of iterations of the application, and $t_{\text{cpu}}$ and $t_{\text{io}}$ represent the duration of the compute and I/O phases, respectively.

We define `Set-10` as the strategy in the `IO-Sets` method, which consists of:

- The $\pi$ function that maps jobs to sets using Equation (4.1).
- The list of sets with their priorities $\{\ldots, (S_{-1}, 10), (S_0, 1), (S_1, 0.1), (S_2, 0.01), \ldots\}$, where the priorities are computed using Equation (4.2);

**Set mapping:** Given an application $A_{\text{id}}$ with a characteristic time $w_{\text{iter}}^{\text{id}}$, the allocation mapping is denoted as $\pi$:

$$\pi : A_{\text{id}} \mapsto S_{\lfloor log_{10} w_{\text{iter}}^{\text{id}} \rceil} \tag{4.1}$$

Where $\lfloor x \rceil$ represents the nearest integer value of $x$.

In other words, an application is assigned to a set that corresponds to the order of magnitude of its $w_{\text{iter}}$. For instance, there will be a set for applications with $w_{\text{iter}}$ values such that $\log_{10} w_{\text{iter}}$ falls between $0.5$ and $1.5$ (corresponding to $w_{\text{iter}}$ values between 4 and 31, assigned to $S_1$), another set for $\log_{10} w_{\text{iter}}$ between $1.5$ and $2.5$ (corresponding to $w_{\text{iter}}$ values between 32 and 316, assigned to $S_2$), and so on.

**Set priority:** To define priorities for the sets, we make the following observation: if two applications start performing I/O at the same time and share the bandwidth equally, the one with the smallest I/O volume finishes first. If we increase the bandwidth for the application with the smallest I/O volume, it will finish earlier, while the larger one will not be delayed.

Based on this observation, we aim to provide higher bandwidth to the sets with smaller I/O accesses. Intuitively, if the characteristic times ($w_{\text{iter}}$) are of different orders of magnitude, we assume that the I/O accesses also differ significantly. Using $w_{\text{iter}}$ instead of the I/O volume has the advantage of being easier to obtain. Therefore, we define the priority $p_i$ for set $S_i$ (which corresponds to jobs such that $i = \lfloor \log_{10} w_{\text{iter}}^{\text{id}} \rceil$) as follows:

$$p_i = 10^{-i} \tag{4.2}$$

This means that applications with the smallest $w_{\text{iter}}$ receive the highest priority and, consequently, most of the bandwidth. The priorities decrease exponentially: $S_1$ receives a priority of $1/10$, $S_2$ has a priority of $1/100$, and so on.

### 4.1.2 SimgIO

To develop and evaluate `IO-Sets` and `Set-10`, we created `SimgIO`, a simulation tool that implements the `IO-Sets` method on top of the SimGrid framework [8]. Additionally, we developed a Python package called `PySimgIO`, which is used for writing and executing simulation tests. The source code and instructions to reproduce all the results presented in the aforementioned manuscript can be found at https://gitlab.inria.fr/hpc_io/io-sets.

As mentioned, `SimgIO` was developed on top of `SimGrid` v3.30[1] and simulates a platform where a set of jobs run and perform I/O concurrently. The tool implements the `IO-Sets` methodology and can be used to evaluate different I/O scheduling approaches, such as Fair-Share (applications can do I/O concurrently and receive the same amount of bandwidth) or All-Exclusive (only one application at a time can perform I/O).

`PySimgIO` is a Python package initially developed to facilitate the execution of our tests but has now become an integral part of `SimgIO`. It includes functions that allow us to generate a wide range of tests. With `PySimgIO`, we can write tests using the *AppGenerator* and *Evaluation* classes. The *Evaluation* class is responsible for generating the workloads and executing a sequence of tests considering different scheduling strategies. At the end of the execution, the class writes a CSV file with the simulation output for each generated workload. The *AppGenerator* class is responsible for generating jobs following a predefined generation protocol (described in Section V-B of our manuscript [3]).

#### Installation

To execute `SimgIO` and `PySimgIO`, it is necessary to install the following dependencies (version numbers serve as an indication of the versions that were tested):

- C++ compiler (g++ v.9.3.0)
- CMake (v3.16.3)
- Python 3 (v3.9)
- libboost (v1.48)

In a Debian-like system, recent versions of the dependencies can be installed as follows:

```
$ apt install python3 python3-pip
$ apt install g++
$ apt install cmake
$ apt install libboost-dev libboost-context-dev
$ apt install git
```

**Installing and Executing SimgIO** `SimgIO` is executed on top of the SimGrid (v3.30) framework. Therefore, it is necessary to install SimGrid by following these steps:

```
$ git clone https://framagit.org/simgrid/simgrid
$ cd simgrid/
$ cmake -DCMAKE_INSTALL_PREFIX=/opt/simgrid .
$ make
$ make install
```

[1] https://framagit.org/simgrid/simgrid

The above commands will compile the SimGrid library and install it on the system. After the installation is complete, we can verify it by compiling and running SimGrid tests:

```
$ make tests -j8 # Build the tests
$ ctest -j8      # Launch all tests
```

If everything is set up correctly, we should see the output of the tests. Next, we can then compile `SimgIO`. To do this, **return to the root folder of the `IO-Sets`** repository and execute the following commands (By default, the executable file is placed in `bin/`):

```
$ mkdir build
$ cd build
$ cmake ../
$ make
```

`SimgIO` receives two XML files as input: the platform file and the deployment file. The platform file describes the computational environment, while the deployment file describes the workload. An example of these files, along with a complete description of each XML field, is provided below. You can find these examples in the `example/deployment/simple_d.xml` and `example/platform/host_with_disk.xml` files.

```
<!-- Platform File -->
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
<platform version="4.1">
    <zone id="AS0" routing="Full">
    <!--  A host with 2000 cores and speed of 1Gflops (per core)-->
        <host core="2000" id="bob" speed="1Gf">
            <prop id="ram" value="100B"/>
            <!--   Disk with 10000GiB of space and speed of 1GBps of write/read -->
            <disk id="Disk1" read_bw="1GBps" write_bw="1GBps">
                <prop id="size" value="1000GiB"/>
                <prop id="mount" value="/scratch"/>
                <prop id="content" value="storage_content.txt"/>
            </disk>
        </host>
    </zone>
</platform>
```

```
<!-- Deployment File -->
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "https://simgrid.org/simgrid.dtd">
<platform version="4.1">
    <actor host="bob" function="host">
        <argument value="app_1"/>       <!-- App Name -->
        <argument value="5" />          <!--periodicity  -->
        <argument value="5e9" />        <!-- Compute size (flop) -->
        <argument value="10000000000"/> <!-- File size (bytes) -->
        <argument value="1"/> <!-- priority  -->
        <argument value="1"/> <!-- Group ID -->
        <argument value="0"/> <!-- release time (s)-->
    </actor>
```

```
    <actor host="bob" function="host">
        <argument value="app_2"/>       <!-- App Name -->
        <argument value="6" />          <!--periodicity  -->
        <argument value="5e9" />        <!-- Compute size (flop) -->
        <argument value="2000000000"/> <!-- File size (bytes) -->
        <argument value="1"/> <!-- priority  -->
        <argument value="1"/> <!-- Group ID -->
        <argument value="0"/> <!-- release time (s)-->
    </actor>
</platform>
```

A simulation with these files can be executed as follows:

```
$ cd bin/
$ ./simgio -p ../example/platform/host_with_disk.xml -d
↪  ../example/deployment/simple_d.xml -ts 1  -te 9  --log=simgio.thres:verbose
```

**Installing PySimgIO** After installing SimgIO, we can proceed to install PySimgIO. To do this, execute the following commands:

```
$ pip install python-dateutil --upgrade
$ pip install setuptools
$ pip install -e .
$ pip install -r PYsimgio/requirement.txt
```

To execute the simulations using the PySimgIO package, it is necessary to configure the variable `SIMGIO_HOME` in `PYsimgio/evaluation.py`. The variable should be set to the actual directory of SimgIO.

```
class Evaluation:
    # List of special character
    COMMENT_CHAR = '#'
    HOST = 'bob'
    FUNCTION = 'host'

    SIMGIO_HOME = '$HOME/iosets'  # <-- MUST BE FULL PATH TO PROJECT'S SOURCEDIR
    CSV_FOLDER = 'csv'
    XML_FOLDER = 'xml'
    GRAPH_FOLDER = 'graphs'
```

To illustrate the usage of PySimgIO, let's consider the example below. In this example, the class `evaluation` is utilized to execute simulations with 20 distinct workloads, each composed of 60 jobs. During each execution, a workload with a different job distribution is created by calling the `tasks_per_mu` function. Then, the `set_mapping` function is used to define the set of jobs. Finally, the functions `f_fair_share` and `f_set_10` are employed to set the priorities. This example can be found in `example/pysimgio/run_tests.py`.

```
from PYsimgio.app_generator import APPGenerator
from PYsimgio.evaluation import Evaluation

```

```
"""
Set mapping
"""
def set10_mapping(generator: APPGenerator):
    for app in generator.apps:
        group_id = round(math.log10(app.w_iter))
        app.set_set_id(group_id)
def fairShare_mapping(generator: APPGenerator):
    for app in generator.apps:
        app.set_set_id(1 + app.app_id * 10 ** (-6))

"""
Set priority
"""
# Each app receives a different priority, so everyone executes together
def f_fair_share(generator: APPGenerator):
    for app in generator.apps:
        app.set_priority(app.set_id)

def f_set_10(generator: APPGenerator):
    for app in generator.apps:
        app.set_priority(1 / (10 ** app.set_id))

def tasks_per_mu(n_apps):
    x = random.uniform(0, 2 / 3)
    # Using the randomly defined value 'x', we define the number of tasks
    # generated per mu value
    small = int(round(x * n_apps))  # small groups
    medium = int(round((1 / 3) * n_apps))  # medium group
    large = n_apps - medium - small  # large group
    return x, [small, medium, large]

priority_functions = {'FairShare': (fairShare_mapping, f_fair_share),
                      'Set-10': (set10_mapping, f_set_10)}

test = Evaluation(evaluation_csv='evaluation.csv',
                  n_executions=20,
                  n_apps=60,
                  mu_list=[10, 100, 1000],
                  sd_percent=0.1,
                  io_limit=0.8,
                  const=20000,
                  tasks_per_mu=tasks_per_mu,
                  timeframe_start=6000, timeframe_end=14000,
                  with_desyn=True,
                  with_noise=True,
                  platform_file=f"{Evaluation.SIMGIO_HOME}
                  /simulation_results/host_with_disk.xml")

Evaluation.clean_all()
test.set_priority_functions(priority_functions)
test.run_test()
```

## Validating the simulation approach

To validate the simulation strategy, we compare it to an experiment conducted on a real system. For this proof-of-concept implementation, we wrote a centralized scheduler to implement `Set-10` and modified the IOR benchmark [6] to communicate with it before and after each I/O phase.

In this scenario, a total of $N = 16$ concurrent IOR instances are created, and distributed on two sets: $n_H$ high-frequency applications of $w_{\text{iter}} = 64s$ and $N - n_H$ low-frequency ones of $w_{\text{iter}} = 640s$.

The $w_{\text{iter}}$ values were chosen so that the shortest I/O phases would still write a total of 32 GB, which was experimentally observed to be enough to reach peak performance in the used platform [2]. The access pattern was also chosen to allow for reaching peak performance: contiguous 1 MB write requests are issued to per-process files. The experiments were executed five times.

The experiments were conducted on the Bora cluster from the PlaFRIM experimental platform [2]. Each of its nodes has two 18-core Intel Xeon processes, 192 GB of RAM and runs CentOS7.6.1810 (kernel v3.10.0-957.el7.x86 64). The parallel file system is `BeeGFS` v.7.2.3 deployed over two servers, each with four storage targets and one MDT. The servers and compute nodes are connected through a 100 Gbps Omnipath network.

We used eight nodes to run applications and a separate node for the scheduler. Data sizes, numbers of nodes and processes and their placement, and stripe count (all eight targets) were selected so each application would reach peak performance by itself, as recommended by Boito et al. [2] in their study conducted in the same platform. Moreover, priority-based bandwidth sharing was implemented by using ten times more processes for each application from one set than to each from the other set (160 vs. 16). We experimentally confirmed that this resulted in the ones with more processes receiving roughly ten times more of the available bandwidth than the others when sharing it. More details about these experiments can be found in [3].

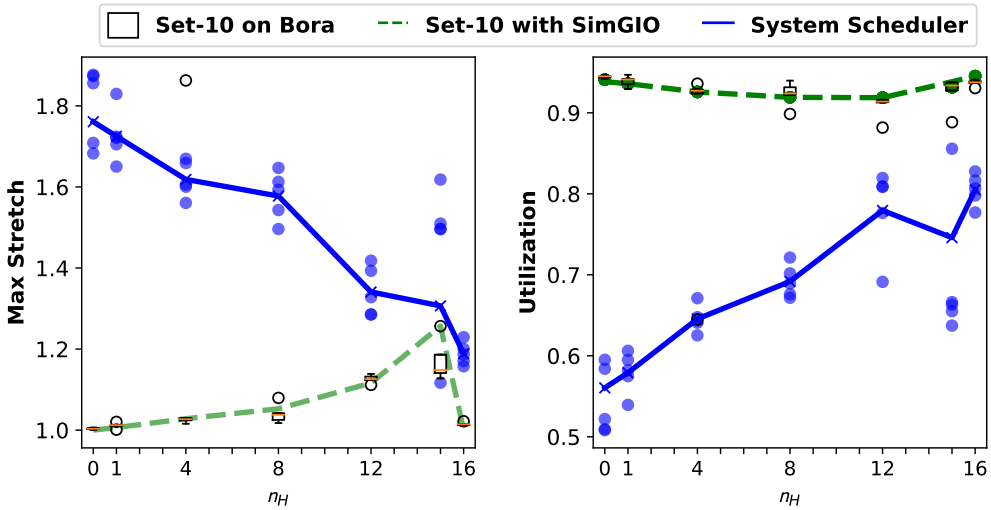



Figure 4.1: Comparison of practical (box and whisker) and simulated (dashed lines) executions with `Set-10`. The blue line and points show baseline executions without `Set-10`. Simulated results are deterministic. `Max Stretch` is shown on the left (lower is better) and `Utilization` on the right (higher is better). The y-axes do not start at 0.

As a baseline, we also evaluated a scenario without our scheduler, which we call the "system scheduler" (which would be close to Fair-Share with additional optimization). For this, all applications used the same number of processes (16).

[2] https://www.plafrim.fr/

The results presented in Figure 4.1 show the excellent precision of the simulator. 80% of the experimental results are within 3.5% (resp. 1.5%) of the simulated `Utilization` (resp. `Max Stretch`). There are two outlying scenarios:

- For $n_H = 4$, one repetition had results severely different from the others. We believe this was due to interference from other jobs on the shared platform.
- When $n_H = 15$, most practical results are slightly better than the simulation. In this case, most I/O phases are small, hence we attribute this to system optimizations such as caching. Note that this is not observable at $n_H = 16$ because the `Max Stretch` is equal to 1, leaving no room for optimizations.

### 4.1.3 Integrating IO-Sets in ADMIRE

As demonstrated by the results presented in the previous section, the `IO-Sets` method, and more specifically the `Set-10` heuristic, is very good at grouping applications with similar I/O patterns. This, in turn, also allows dynamically computing an appropriate bandwidth distribution for the applications currently being managed by the algorithm. Thus, it makes more sense to use it to directly manage the I/O from a system-wide storage tier such as a PFS, rather than using it for scheduling data transfers from `scord` directly. Due to this, we explored extending Lustre to include `IO-Sets` scheduling but the scope of this implementation was deemed too large for the effort allocated in the project. Nevertheless, implementation of smaller scope was possible for `BeeGFS`[3], a commercial parallel file system from ThinkParQ GmbH, which we describe below.

#### Adding IO-Sets to `BeeGFS`[4]

As explained in Section 4.1, there are two components required to implement the `IO-Sets` framework: a way to ensure exclusive access for applications in the same set, and a way to partition the I/O bandwidth "unfairly", according to the priority of the set. For example, in `Set-10`, each set should get 10 times as much bandwidth as the next set. As discussed by the manuscript [3], the designers of `IO-Sets` considered several practical implementations. We explore one of them, integrating `IO-Sets` directly into a shared parallel file system. This has several advantages: first, it is transparent to the applications; second, it is relatively straightforward to implement because parallel file systems already have all the logic required to handle requests from multiple clients, all that is needed is to tweak them so that these requests are treated according to the `IO-Sets` priority.

The `BeegFS` file system was chosen because it already provides two policies to handle requests, either the standard first-come, first-served or "per-user", the latter intended to improve fairness. This is not precisely the policy needed for `IO-Sets`, but because the software already provides a choice, these features are cleanly isolated in the code base and it was easy to add a third one.

The implementation itself is conceptually simple. It focuses on storage servers and ignores metadata servers. When an application wants to write to the file system, the `BeegFS` client splits the data into chunks, called stripes. Each stripe is associated with a request destined for the file system servers, so for example, a 10MiB write on a system with 1MiB stripes will generate 10 requests to the storage servers (simplifying a bit and ignoring some bookkeeping requests). These requests will then reach the storage servers and be processed in queues. This is fairly standard and not particularly specific to `BeegFS`. I/O scheduling is then a simple matter of ordering the requests in the storage servers' queues.

[3] https://www.beegfs.io

[4] Due to limitations in `BeeGFS`'s license agreement, the code of `IO-Sets` integrated into it cannot be freely disclosed in a public deliverable. We are currently discussing with the `BeeGFS` developers how to release this code publicly.

Let's illustrate with small examples. Say we have two members of the same set, applications `APP1` and `APP2` making three requests each (see Figure 4.2). Because the applications are in the same set, they have the same priority, and the `IO-Sets` framework dictates that they should write in an exclusive manner. Here it means that all requests of `APP1` are processed before `APP2`.

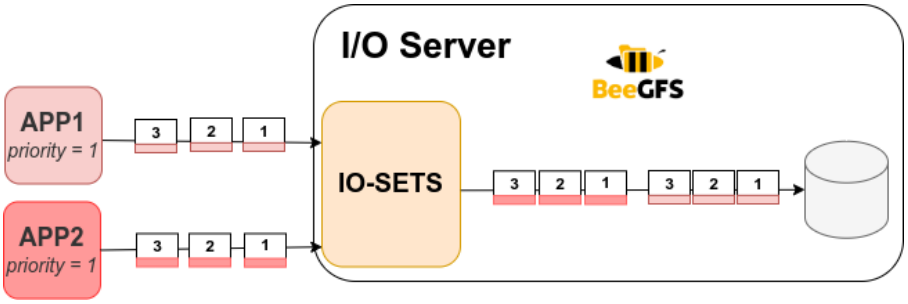



Figure 4.2: Ensuring exclusive access. All requests for APP1 are processed before APP2.

Now, let's assume `APP1` and `APP2` are in different sets, `APP2` having twice the priority (figure 4.3). It means that two requests of `APP2` should be processed for every request of `APP1`.

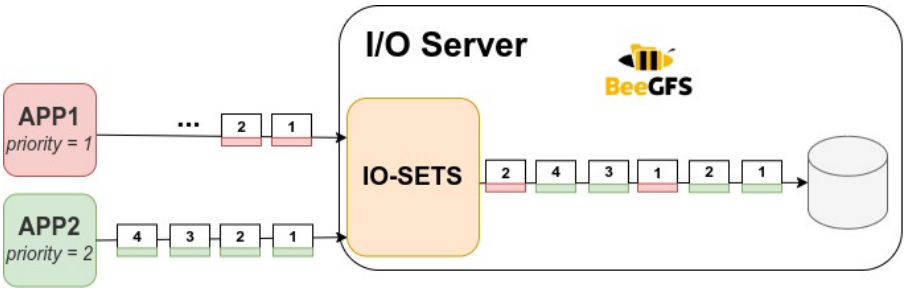



Figure 4.3: Ensuring unfair bandwidth sharing. Two requests of `APP2` are processed for every one request of `APP1`.

**Implementation:** The `IO-Sets` framework requires requests to be tied to a specific application. The `BeeGFS` request header contains a user ID, but no application ID, it was thus extended to include it. Unfortunately, application ID is not a concept as well defined as user ID, there's no standard system call to get an application ID from the operating system. In the context of this work, application IDs are set by the job scheduler of a cluster, and made available to applications via the environment, Slurm writes it to the `SLURM_JOB_ID` environment variable for example. The `BeeGFS` client was modified to read it directly from the application environment. The Lustre file system, which makes use of the application ID, reads it in the same manner. On the `BeeGFS` servers, the queue processing was adapted for `IO-Sets`. Once requests were augmented with application ID, all that is left is to pick requests from the queue according to the priority of the application, and, to ensure exclusive access, process applications with the same priority one after the other. This is straightforward enough that it probably does not warrant more details.

An option `--jobstats` was also added to the `beegfs-ctl` tool, to help visualise the instantaneous bandwidth usage per application. With this, it was easy to check that two applications with the same priority get exclusive access to the bandwidth.

**Results:** We ran a few validation tests. Note that these are not representative in terms of `BeeGFS` performance, being run on general-purpose machines without any optimized configuration (the observed bandwidth with an unmodified `BeeGFS` installation was around 600MiB/s). We merely

| Bandwidth A (MiB/s) | Bandwidth B (MiB/s) |
|---|---|
| 569.13 | 58.52 |
| 586.68 | 62.19 |
| 606.64 | 62.08 |
| 845.43 | 101.36 |
| 525.77 | 55.36 |

Table 4.1: Bandwidth sharing between application A (priority 10) and B (priority 1).

| Bandwidth A (MiB/s) | Bandwidth B (MiB/s) | Bandwidth C (MiB/s) |
|---|---|---|
| 621.96 | 81.07 | 8.11 |
| 555.83 | 68.76 | 6.87 |
| 542.02 | 72.78 | 7.43 |
| 605.36 | 81.35 | 8.32 |
| 556.95 | 67.88 | 7.35 |

Table 4.2: Bandwidth sharing between applications A, B, and C (priorities 100, 10, and 1, resp.).

wanted to validate the approach and make sure the overhead was acceptable. Presented in Table 4.1 is an example of `Set-10` bandwidth partitioning for two applications, using the IOR benchmark with 16 processes each writing to their own files. Application A has priority 10 and application B has priority 1. The result is the mean bandwidth, for five runs.

Table 4.2 presents the results with three applications A, B and C, with the priority still separated by a factor of 10.

### Using IO-Sets for QoS control

Part of the `Set-10` heuristic is the distribution of bandwidth to applications according to their periodicity (see Equation 4.2). This aspect of the proposal is very suitable for dynamically computing (and updating) QoS limits. This is very useful for `scord` since it would allow the service to determine with a high probability the best bandwidth distribution for the current set of data transfers. This calculation can then be used to (1) control Lustre's LIME subsystem described in Section 2.1.2; and (2) to dynamically control the number of worker processes for each `Cargo` service in order to restrict the data to the established QoS constraints (refer to Section 4.2). Both approaches are currently being worked on as part of the final integration effort and will be reported in WP4's final deliverable.

## 4.2 Enforcing QoS through I/O malleability

As discussed in Section 2.1.2, data transfers between the ad-hoc storage systems and the PFS need to honour the established Quality of Service guarantees. An advantage of having a dedicated data staging service that runs independently from applications is that we can control the number of MPI processes dedicated to I/O, which can directly affect the data rate obtained from the PFS [2]. By considering these processes as a malleable entity, we can dynamically adjust the data rate

from each `Cargo` service, thus enforcing QoS even if PFS mechanisms such as Lustre's LIME are not available.

In this section, we propose an algorithm[5] to orchestrate the running `Cargo` instances to control such processes and reduce the number of QoS breaches. The algorithm was tested inside ElastiSim[6], a batch-system simulator for malleable workloads written for ADMIRE by TUDA, that simulates (1) jobs and applications, (2) the scheduling algorithm, and (3) the platform itself. Using ElastiSim, we modelled each `Cargo` instance as a malleable job doing I/O on the PFS and constructed a control algorithm around this.

The algorithm requires the following information:

- Number of available nodes
- Cargo instances
- Increase/Decrease bandwidth of Cargo instances (using FlexMPI or splitting transfers)
- Bandwidth obtained by each Cargo instance (Bytes transferred / Running time)
- QoS required by each Cargo instance

The algorithm itself aims to execute `Cargo` instances while balancing those with high requirements and those with low requirements. Balance can be configured, but a 1:1 ratio is implemented in ElastiSim for each round. Each `Cargo` instance runs with a fixed number of initial processes.

Once all the jobs are executed, we process all the running jobs that have obtained bandwidth information. Each `Cargo` instance can be classified as either `Shrink` (the established QoS limit is being exceeded) or `Expand` (obtained bandwidth is below the established QoS constraint) using the following algorithm:

$$\texttt{Cargo}_{\text{action}} = \begin{cases} \text{Expand}, & \text{if } \texttt{Cargo}_{\text{bw}} < \texttt{Cargo}_{\text{QoS}} + \text{threshold} \\ \text{Shrink}, & \text{if } \texttt{Cargo}_{\text{bw}} > \texttt{Cargo}_{\text{QoS}} + \text{threshold} \end{cases}$$

To test the algorithm, we simulate 35 concurrent transfer jobs (1,000 transfer steps each) executing with a maximum number of 7 nodes. This exhibits 514 QoS breaches with the full algorithm (i.e. both `Expand` and `Shrink` actions are applied), 159,694 QoS breaches when only the `Shrink` action is applied, 6,405 QoS breaches without any control algorithm, and 2,078 violations using submission heuristics. We consider a QoS violation when `Expand` actions are required in a particular `Cargo` instance during the transfer phase (the same `Cargo` job can exhibit multiple violations during its lifetime, and the algorithm considers each violation separately). The benefits of working with the full algorithm are clear.

Conversely, if we submit jobs for execution with at least 3 nodes (but with the capability of expanding to 7 nodes or shrinking to 1 node), we find the next results: 465 QoS breaches with the full algorithm, 149,401 QoS breaches when only the `Shrink` action is applied, 14,228 when only the `Expand` action is applied, 23,156 QoS breaches with no control algorithm, and 23,030 using submission heuristics. Again, running with the full algorithm shows a great improvement over the other alternatives. It is worth noting that the action of shrinking or expanding should be used in tandem, as the environment is dynamic and some tasks may need to apply both. Additionally, note that in this second experiment interference between tasks increases due to an increased number of jobs. This happens because reducing nodes from a transfer task releases resources that are immediately assigned to pending jobs. Note, however, that this is a simulation-specific problem since in a real scenario, `Cargo` instances are linked to a specific job allocation and reducing

[5] https://storage.bsc.es/gitlab/eu/admire/cargo-simple-scheduler
[6] https://elastisim.github.io

processes for an existing `Cargo` instance does not automatically release resources for other jobs. Nevertheless, one way to reduce such in-simulation interference is to apply submission heuristics, such as not allowing a job to be executed if the number of free nodes falls below a threshold.

## 4.3 FTIO: Detecting I/O Periodicity Using Frequency Techniques

Characterising the I/O behaviour of an HPC application can be a challenging task. Various aspects, for example, I/O contention, can cause performance variability making modelling I/O a well-known problem. On the other hand, informing a system about the periodicity of I/O can be extremely valuable for different optimisation techniques, such as I/O scheduling and burst buffer management. For example, by joining an application's I/O periodicity with information about its dataset usage, we could determine the best moment to stage in/out the application's data into an ad-hoc storage service. To achieve this "just in time" data staging, WP4 plans on integrating the `FTIO` tool into `scord`'s decisions.

`FTIO` is a tool from WP5 introduced in D5.3, that can detect an application's I/O periodicity in an online manner. It uses the discrete Fourier transformation (DFT) to find the frequencies constituting a signal (i.e., temporal I/O behaviour) combined with an outlier detection method (Z-score) to find the dominant frequency. With the dominant frequency at hand, the periodicity of I/O can be computed in a simple manner (i.e. it's just its reciprocal). On the other hand, the absence of a dominant frequency is a direct indicator that the I/O behaviour of the application is not periodic. One of the advantages of `FTIO` is that it has a very low overhead, making it well-suited for online prediction. Moreover, as the approach provides a metric of confidence in the obtained results, it can be used in practice in many scenarios, such as I/O and data scheduling decisions.

In particular, decisions can be made with a certain level of confidence, leveraging probabilistic aspects in the developed approaches. Additionally, the approach also allows for identifying I/O bursts. While several of its parameters can be changed (e.g. the considered time window $\Delta t$ and the sampling frequency $f_s$), their selection directly allows for identifying the time regions of interest, as well as I/O of interest (e.g., a low value for $f_s$ will result in ignoring any high-frequency behaviour, according to the Nyquist theorem). For example, in Figure 4.4 below, we executed LAMMPS [14] with 3,072 ranks. We choose the 2-d LJ flow simulation with 300 runs and dump all atoms every 20 runs. Using `FTIO` with a sampling frequency of 10 Hz, the periodicity of I/O was found to be 0.039 Hz (25.73 s) with moderate confidence. For comparison, the real mean period for this execution (which can only be known because the application was manually instrumented) was 27.38 s. More details about the approach and an extensive evaluation can be found in our paper, currently available as a pre-print [12].

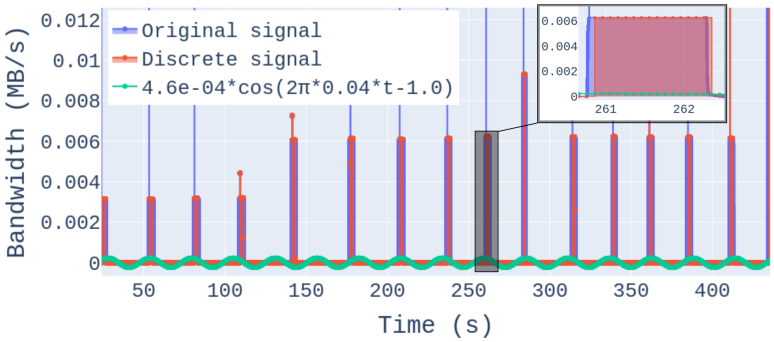



Figure 4.4: FTIO on LAMMPS with 3,072 ranks.

As discussed in D5.3, the approach is much more sophisticated and various extensions are currently being examined, like using wavelet transformation instead of the DFT or finding the

dominant frequency with clustering approaches. Moreover, at the simplest level, in order to detect the periodicity of I/O, the approach requires bandwidth over time. Combined with an ad-hoc storage service such as GekkoFS, this information can be easily obtained, as in ADMIRE these services already capture the application's I/O requests.

While the approach can be used for various aspects not limited to I/O (e.g., prediction of malleable time windows to perform malleable decisions), one of the main motivations behind its development is its use with `IO-Sets` and for ad-hoc storage management. For both cases, we are currently working on the integration of the developed components.

- For `IO-Sets`, the $w_{\text{iter}}$ metric used for assigning applications to sets and defining their priority directly corresponds to the period of the I/O phases (the inverse of their frequency), which is reported by default by `FTIO`. `FTIO` can be used either with information obtained from an application's ad-hoc storage service (bandwidth over time) or with its own tracing library for applications that access the PFS directly. `FTIO` can then communicate the period ($w_{\text{iter}}$) to `scord`, which can apply the `Set-10` heuristic for bandwidth sharing to compute the QoS limits (see Section 4.1.1). `FTIO` can also provide this metric to the PFS clients, so they can communicate it to the server if it implements the `Set-10` heuristic for set-exclusive access (like our prototype with `BeeGFS`).

- For ad-hoc storage management, the idea is to use the predictions provided by `FTIO` for deciding when to stage data in or out, with the goals of using the ad-hoc storage capacity efficiently and controlling the requested bandwidth to minimise I/O interference to concurrent applications. With the information of bandwidth over time provided by ad-hoc storage services during the execution of an application, `FTIO` can offer predictions to `scord` regarding the frequency of I/O phases, their duration, amount of data, etc. Then, that will be combined with what is known about the files accessed by the application and can be used to decide if data must be staged out (to free space) or in before the next I/O phase, and how long it will be before that is needed. That could be used, for example, by the `scord` service to prioritise more urgent data movements[7], or to establish a QoS limit to this movement.

  In this context, there are two possibilities. In case the prediction is wrong and, consequently, a bad decision is taken, the application may have to wait for files to be moved between the PFS and the ad-hoc storage system. On the contrary, if the prediction is correct, an effective decision can be made that allows maximising the usage of the buffers and the usage of the shared I/O resources. Since `FTIO` provides a metric that delivers confidence in the results, decisions with probabilistic occurrences can be made rather than mere guesses. These aspects are currently being examined and will be further developed in the upcoming months.

[7] The extension of `IO-Sets` to a situation where some applications are using an ad-hoc file system is currently being studied — the question to be answered is to what set movements between the PFS and ad-hoc file system belong and what is its priority

# 5. Conclusion

In this prototype deliverable, we presented the software developed to support I/O and data scheduling decisions in the ADMIRE project. We discussed the design considerations that led from the API definitions in D4.1 to the actual software implementations, providing details on how to deploy and use the software packages. While the initial sections of this document describe the scaffolding needed to materialise the ADMIRE ecosystem, the final section describes the additional software that will provide actual logic and decision-making driving the *I/O Scheduler*.

We also presented some preliminary experimental results for the software developed as well as the control algorithms researched. Specifically, `Cargo` demonstrated that data transfers can be made to scale with the number of processes and, as expected, outperform typical `cp`-based approaches. For `IO-Sets`, we showed the precision of its simulator and how the `Set-10` heuristic was found to be quite robust, offering a precise classification even when provided with inaccurate information. Finally, we demonstrate how separate `Cargo` instances can be orchestrated with a simple control algorithm focused on maintaining QoS constraints for PFS bandwidth. Despite its simplicity, the algorithm proved highly effective at preventing QoS breaches.

The next steps (some of which are already in progress) are to integrate these missing pieces into the current prototypes and finalise the integration with other ADMIRE components. More specifically, a version of `IO-Sets` is planned to be integrated into `scord` for the purposes of bandwidth redistribution. The QoS control algorithm presented in Section 4.2 is currently being added to `scord` as a fallback QoS enforcement mechanism for those systems not relying on Lustre. Conversely, a control algorithm for Lustre's LIME subsystem is currently being designed for those systems relying on Lustre. Finally, approaches to monitor and control ad-hoc storage capacity and to provide "just in time" data staging are currently being discussed. The outcomes of these initiatives, along with the progress made in Task 4.3 w.r.t. in-situ and in-transit data operations, will be reported in D4.3, WP4's final deliverable.

# Appendix A

# Terminology

- Ad hoc Storage System, ephemeral storage system that only exists in a determined period, i.e. during a job's execution.
- CLI, command line interface.
- DRAM, dynamic random-access memory.
- EBNF, Extended BackusNaur Form is a family of metasyntax notations, any of which can be used to express context-free grammar. EBNF is used to make a formal description of a formal language such as a computer programming language. They are extensions of the basic BackusNaur form (BNF) metasyntax notation.
- In situ data, processing the data where it originated.
- In transit data, processing the data when it is moved.
- NORNS, data transfer service for HPC developed at BSC.
- NVM, non-volatile memory.
- PFS, parallel file system.
- POSIX, Portable Operating System Interface, family of standardized functions.
- QoS, Quality of Service.
- RDMA, remote direct memory access.
- RPC, remote procedure call.
- Slurm, job submission system widely used.
- SSD, solid state drive.
- Object store, persistent storage system where data are stored not as files but as objects. In its canonical implementation Object are immutable and the API is limited to PUT, GET and DELETE. More sophistical object stores have been developed on the ground of these concepts such as ADMIRE Data Clay.
- Disaggregated Storage, storage systems where all the storage capabilities are centralized in dedicated network attached storage servers. This approach allows connected compute nodes to access a storage capacity without constraints related to the capacity of a single storage device.

- PFS, Parallel File System, type of distributed file system supporting a global namespace and spread across multiple storage servers.
- Node Local Storage, ability for a compute server to store persistent data on physically local storage devices.
- Ephemeral Storage, file systems which are making persistent (surviving across system reboot) but which are designed to be deployed and destroyed over a limited period of time, from a few hours up to a few months.
- API, Application Programming Interface, a mechanism that enables an application or service to access a resource within another application or service. The application or service doing the accessing is called the client, and the application or service containing the resource is called the server.
- Rest API, such APIs can be developed without constraint and the programming language and support a variety of data formats. The only requirement is that they align to the following six REST design principles - Uniform interface, Client-server decoupling, Statelessness, Cacheability, and Code on demand (optional).
- OSS, an Object Store Server in the Lustre terminology is a computing server in charge of managing the ingest of data, including generation of the data protection, and shipping these data to the correct Object Store Target.
- OST, Object Store Target in the Lustre terminology is a storage server accommodating potentially a large number of hard drives and/or NMVes. The OST write the data received from the OSS and makes them persistent.
- MDS, MetaData Server.
- MDT, MetaData Target.
- Stripe, an elementary chunk of data according to the Lustre terminology. A large file is split in multiple stripes and each stripe is sent to an individual OST. The higher the number of stripes, the higher the parallelism.
- Monitoring Manager,
- Intelligent Controller,
- Monitoring Daemon,
- TBON, Tree-Based Overlay Network,
- PromQL, the query language supported by the Prometheus database. Syntax, documentation and examples are available here: https://prometheus.io/docs/prometheus/latest/querying.

# Bibliography

[1] BDVA. BDVA Strategic Research and Innovation Agenda V4. Technical report, BDVA, 2017.

[2] Francieli Boito, Guillaume Pallez, and Luan Teylo. The role of storage target allocation in applications' I/O performance with BeeGFS. In *CLUSTER 2022 - IEEE International Conference on Cluster Computing*, Heidelberg, Germany, September 2022.

[3] Francieli Boito, Guillaume Pallez, Luan Teylo, and Nicolas Vidal. IO-SETS: Simple and efficient approaches for I/O bandwidth management. Pre-print available at https://inria.hal.science/hal-03648225/, May 2022.

[4] André Brinkmann, Kathryn Mohror, Weikuan Yu, Philip Carns, Toni Cortes, Scott A Klasky, Alberto Miranda, Franz-Josef Pfreundt, Robert B Ross, and Marc-André Vef. Ad hoc file systems for high-performance computing. *Journal of Computer Science and Technology*, 35:4–26, 2020.

[5] Peter Corbett, Dror Feitelson, Sam Fineberg, Yarsun Hsu, Bill Nitzberg, Jean-Pierre Prost, Marc Snirt, Bernard Traversat, and Parkson Wong. Overview of the mpi-io parallel i/o interface. *Input/Output in Parallel and Distributed Computer Systems*, pages 127–146, 1996.

[6] IOR Benchmark, version 3.3.0. https://github.com/hpc/ior.

[7] Emmanuel Jeannot, Guillaume Pallez, and Nicolas Vidal. Scheduling periodic I/O access with bi-colored chains: models and algorithms. *Journal of Scheduling*, 24(5):469–481, 2021.

[8] Adrien Lebre, Arnaud Legrand, Frédéric Suter, and Pierre Veyre. Adding storage simulation capacities to the simgrid toolkit: Concepts, models, and api. In *CCGrid'15*, pages 251–260. IEEE, 2015.

[9] N. Mathur and R. Purohit. Issues and challenges in convergence of big data, cloud and data science. *IJCA*, 160:7–12, 02 2017.

[10] Alberto Miranda, Adrian Jackson, Tommaso Tocci, Iakovos Panourgias, and Ramon Nou. NORNS: Extending Slurm to support data-driven workflows through asynchronous data staging. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, 2019.

[11] Yingjin Qian, Xi Li, Shuichi Ihara, Lingfang Zeng, Jürgen Kaiser, Tim Süß, and André Brinkmann. A configurable rule based classful token bucket filter network request scheduler for the lustre file system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2017.

[12] Ahmad Tarraf, Alexis Bandet, Francieli Boito, Guillaume Pallez, and Felix Wolf. FTIO: Detecting I/O Periodicity Using Frequency Techniques. Pre-print available at https://arxiv.org/abs/2306.08601, 2023.

[13] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32, 1999.

[14] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton. LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Comp. Phys. Comm.*, 271:108171, 2022.

[15] Li Xi and Zeng Lingfang. Lime: A framework for Lustre global QoS management. In *Lustre Administrator and Developer Workshop*, 2018.