H2020-JTI-EuroHPC-2019-1

Project no. 956748

ADAPTIVE MULTI-TIER INTELLIGENT DATA MANAGER FOR EXASCALE

D5.5
Extra-P Extended with I/O Modelling Capabilities

Version 1.0

Date: February 29, 2024

Type: Deliverable
WP number: WP5

Editor: Ahmad Tarraf
Institution: TUDA

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	Changes
1	13/11/23	Jesus Carretero	UC3M	Document creation
2	13/11/23	Ahmad Tarraf	TUDA	Copied main layout
3	15/11/23	Ahmad Tarraf	TUDA	Added use cases
4	17/11/23	Ahmad Tarraf	TUDA	Added introduction
5	17/11/23	Jean-Baptiste	PARATOOLS	Added metric proxy
6	20/11/23	Ahmad Tarraf	TUDA	Added TMIO
7	21/11/23	Ahmad Tarraf	TUDA	Added Extra-P description
8	21/11/23	Ahmad Tarraf	TUDA	Added FTIO description
9	22/11/23	Jean-Baptiste Besnard	PARATOOLS	Added link between metric proxy and Extra-P
10	22/11/23	Francieli Zanon Boito	INRIA	Added I/O Scheduling
11	15/12/24	Ahmad Tarraf	TUDA	Structure review and Todos added
12	08/01/24	Ahmad Tarraf	TUDA	Added use case chapter (Chapter 4)
13	10/01/24	Ahmad Tarraf	TUDA	Adjusted introduction and conclusion
14	21/01/24	Ahmad Tarraf	TUDA	Refined Chapter 3
15	30/01/24	Ahmad Tarraf	TUDA	Extended FTIO
16	12/02/24	Ahmad Tarraf	TUDA	Updated FTIO and TMIO
17	13/02/24	Marc-André Vef	JGU	Added just in time staging
18	19/02/24	Jean-Baptiste Besnard	PARATOOLS	Description of the Model-Server
19	19/02/24	Marc-André Vef	JGU	Finalized just in time staging
20	20/02/24	Simone Pernice	Unito	Added system model use case
21	21/02/24	Taylan Özden	TUDA	Added job scheduling use case
22	22/02/24	Ahmad Tarraf	TUDA	First internal revision
23	23/02/24	Guillaume Pallez	Inria	Added elements in I/O Modeling; pass on I/O-set experiments
24	26/02/24	Ahmad Tarraf	TUDA	Second internal revision
25	27/02/24	Jesus Carretero	UC3M	Document review
26	28/02/24	Ahmad Tarraf	TUDA	Finalization

Executive Summary

Executing large scientific applications efficiently on an HPC system with various resource configurations can be a challenging task. Upon scaling up an application, performance bottlenecks are often encountered, preventing the effective usage of such systems. In the context of the ADMIRE project, the performance models generated by Extra-P are not directly used for spotting scalability bugs, but rather to aid the malleability manager and the intelligent controller in their decision processes. In particular, these models can be utilised by different components of the ADMIRE toolchain to adapt various resources and ultimately increase the system performance as will be outlined in this deliverable. Previous to the ADMIRE, the performance models generated by Extra-P were mainly concerned with the scalability of computational or communications parts of the code. To exploit the full potential of the tool and use it to aid the decision regarding the balance of the resources, the functionality of Extra-P was extended to additionally considering I/O. Moreover, several components have been developed and extended in WP5 that further provide the approach in ADMIRE with enhanced monitoring as well as modelling capability, including, for example, the metric proxy for monitoring and FTIO for predicting I/O phases.

This deliverable presents the extended functionalities of Extra-P developed in the ADMIRE project. We described the importance of those extensions and their role in making decisions regarding the resource consumption of an application. We demonstrate the new capabilities of Extra-P based on various applications and describe the role of the performance models in other components of the ADMIRE toolchain. As the final deliverable in this work package, we further provide an overview of the modelling and monitoring components developed in WP5, alongside their interactions and the established use cases with the other WPs in ADMIRE.

Contents

List of Figures

1 Introduction

In HPC, large scientific applications are usually executed on huge clusters with a vast number of resources. As these systems increasingly become more complex and powerful, so do the applications across various domains (e.g., fluid dynamics, molecular dynamics, and environmental simulations) that try to exploit them as much as possible. However, scalability bugs can lead to performance bottlenecks preventing the effective usage of such systems. Consequently, monitoring and modelling applications on an HPC system is considered one of the essential steps for performance optimisation. Identifying scalability bugs at an early stage of the development process is an indispensable prerequisite to ensure early and sustained productivity. In this context, performance modelling plays an essential role, as elaborated later in Chapter 3. Performance modelling has a long research history in HPC [1,9,12,18,22,29,32,35,36,42,45,50]. For example, Extra-P [12] is an automatic performance-modelling tool that generates empirical performance models and aids the users in spotting scalability bugs. Aside from spotting these bugs, performance modelling could also aid scheduling decisions (e.g., for malleable jobs). In particular, the performance of the application at various scales in regard to different resources (e.g., I/O, compute, etc.) is modelled and thus known. Consequently, effective decisions can be made by different components (job scheduled, I/O scheduler, etc.) with the ultimate goal of improving the throughput of HPC systems. The aspect is extensively examined and has been realised in the ADMIRE project.

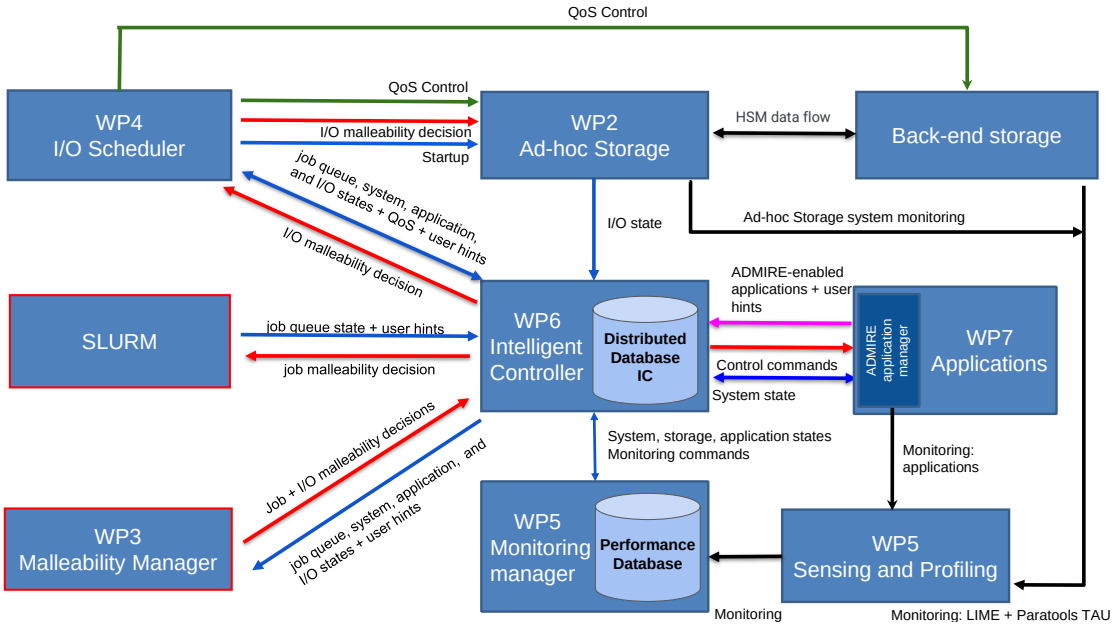


Figure 1.1: ADMIRE global architecture showing the relationship as well as the information flow between the different components of the framework. Using the data gathered in the performance database, application models are generated in WP5 and are forwarded from there to components like the malleability manager from WP3 and the intelligent controller from WP6.

The global architecture of the ADMIRE framework is illustrated in Figure 1.1. As observed, the intelligent controller acts as the main component forwarding and processing data to the various other components. One of the key aspects examined in the ADMIRE project is job malleability, which refers to extending or shrinking

the resources of a job. In this context, the project distinguishes between compute and I/O malleability. The malleability manager decides the best-suited configuration for each application based on the selected scheduling algorithm (see Section 4.1). To make reasonable decisions, the malleability manager needs to consider the current state of the system, the jobs in the queue, user hints, and the *scaling* performance of an application. Consequently, this manager needs performance models that describe how the application's behaviour changes at different scales. Moreover, as the malleability manager tries to balance both compute and I/O resources, the performance models should cover and describe the scaling behaviour of an application in both regards. Once a decision is taken, it's forwarded to the job scheduler SLURM and the I/O scheduler from WP4 through the intelligent controller.

Previous to the ADMIRE project, Extra-P has been used to model the call path of the application, focusing on computational and communication aspects but excluding I/O. However, recent terms such as the *storage wall* [23] try to quantify the I/O performance bottleneck from the application scalability perspective. Indeed, due to storage's shared nature and slow improvement compared to other resources on a cluster, I/O contention, slow I/O, and low I/O performance are often encountered in HPC. One of the main objectives of the ADMIRE project is to maximise performance by establishing control by creating an active I/O stack that dynamically adjusts computation and storage requirements through intelligent global coordination. In particular, the ADMIRE project has created a feedback loop between control (through the intelligent controller and malleability manager) and models/measurements from the developed monitoring framework in WP5. As just described, knowledge of the application I/O performance and the current system I/O load is required to achieve this.

The latter aspect has been the content of Deliverable 5.4 (D5.4). While Deliverable 5.3 (D5.3) describes how the I/O profiles of an application are gathered. In this deliverable, which is in the direct continuation of both D5.3 and D5.4, we demonstrate how the gathered data can be used to generate performance models with Extra-P extending its capabilities to model I/O.

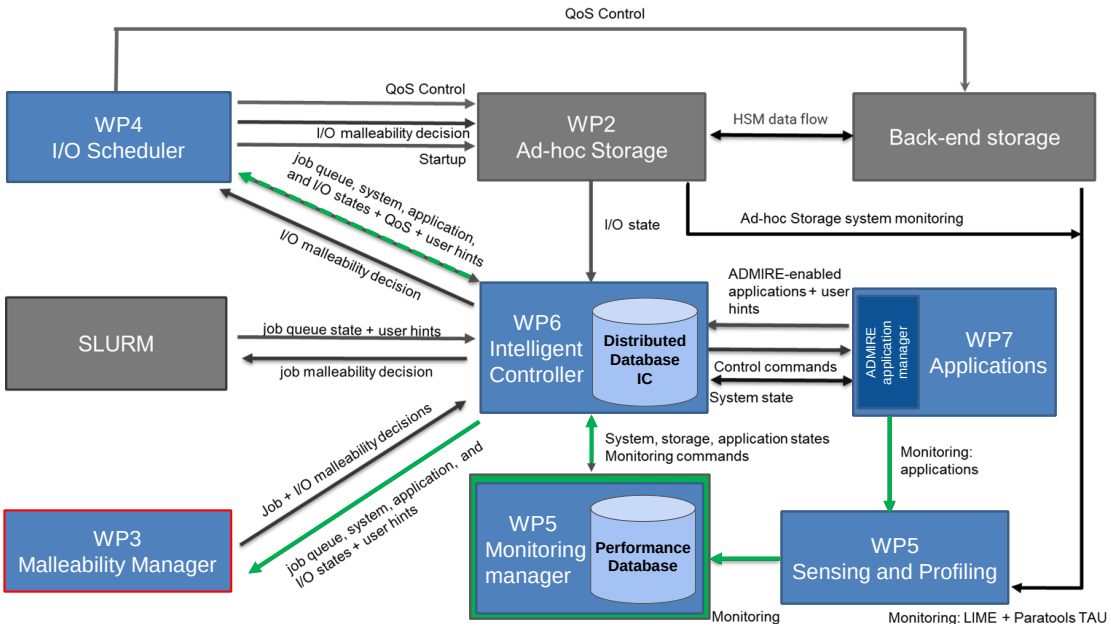


Figure 1.2: ADMIRE global architecture highlighting the components that directly influence or interact with the performance models from Extra-P. These components are coloured, while the remaining components are grey. Moreover, the data flow from and to Extra-P in WP5 are highlighted in green.

In Figure 1.2, only the components that directly influence or interact with the performance models are coloured, while the remaining are grey. Moreover, the data flows that are related to Extra-P are highlighted in green. As part of WP5, the performance models are generated and stored in the performance database. Note that the intelligent controller can also invoke Extra-P at different intervals to generate performance models at demand. As an input, the collected application traces through the sensing and profiling module in WP5 are used and converted to profiles as later described in Chapter 2. Note that the detailed approach has been

described in Deliverable 5.3. Once these models are generated, they are forwarded to the malleability manager in WP3 to aid the scheduling decisions. In particular, the performance models can be either generated on-the-fly or accessed through a Redis database by the intelligent controller. Consequently, the intelligent controller manages the forwarding of the models on one hand. On the other hand, the intelligent controller can also utilise the performance models to refine the system model and predict the system's future behaviour as described in detail in Section 4.2. Other components, such as the I/O scheduler, could also benefit from the I/O models to enhance the system performance even further. However, due to I/O variability and changing I/O behaviour, it often makes sense to analyse phases rather than the entire application behaviour. Consequently, prediction from tools like FTIO developed in WP5 can be used to help I/O schedulers improve the I/O performance as described later in Section 4.3.

From an abstract view, the life-cycle of the performance models from Extra-P in the ADMIRE project can be divided into four stages: monitor, profile, model, and employ. In the first three stages, the performance models are generated, while in the last stage, the models are utilised by different components of the ADMIRE framework. In particular, the following actions are performed in each stage:

Monitor: As a first step, the relevant performance metrics need to be collected from the application. This is done through the developed metric proxy. The collected data are stored in a dedicated trace format, profiles and a Prometheus database as shown in Figure 1.1. This is further elaborated in Section 2.1.

Profile: To generate performance models, Extra-P required profiles, not traces. Profiles can be grouped together through different interfaces (e.g., through a command line interface) into a single profile that contains the application's behaviour at different scales. This file can be read by Extra-P to generate performance models. This is further described in Section 2.2.

Model: In the last stage of the model generation process, Extra-P reads the profile file to generate the desired performance models. That is, for everything that has been traced and consequently turned into profiles, Extra-P can generate a performance model. To refine the models further the tool can be re-invoked whenever new data is available to generate and update performance models continuously. Through the added functionality in the ADMIRE project, the tool is no longer limited to modelling the runtime but can additionally generate, for example, performance models for the bytes transferred during the I/O operations at various levels of the software stack. This is explained in detail in Chapter 3.

Employ: After the performance models are generated, they are *employed* by different components in the ADMIRE framework. This is handled in detail in Chapter 4.

Figure 1.3 shows an abstract representation of these stages. As observed, the refining process is executed whenever new traces are available. Note that the traces are not generated in particular for Extra-P, but for monitoring purposes (e.g., monitor the applications and system state over time). However, through our approach,

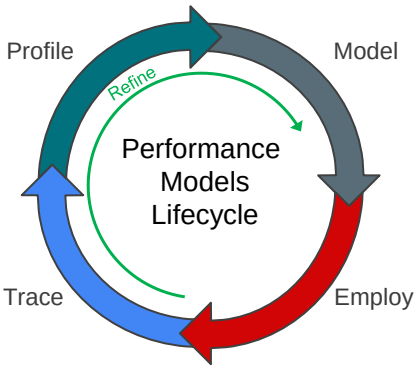


Figure 1.3: Lifycle of performance models in ADMIRE.

different components can utilise the same set of traces. Consequently, we reuse the collected traces by grouping and converting them to profiles. This way, through every execution of an application, further traces are collected, which help refine the models as the profiles are enriched by the newly added measurement points. At the same time, due to the detailed information contained in the traces ranging from low-level system calls such as `pwrite64` up to high-level calls like `MPI_File_write`, performance models can be generated with different granularity. Finally, as compute and communication calls are traced, performance models can be generated that show the *compute to I/O ratios* and describe how resource-demanding different scaling configurations are. This is especially valuable to components as the malleability manager in WP3 as described in the scheduling algorithm in Section 4.1. Furthermore, these models are then again employed by the different components in the ADMIRE project (e.g., the system model in WP6 as described in Section 4.2) to adapt the system and application behaviours. This again yields traces and the cycle is restarted again.

As mentioned, a new demand for models that describe the period of the I/O phases was observed during the development of the components, and, thus, at a later point in the project, efforts have been dedicated to achieving this. In particular, to enhance I/O scheduling in WP4 and burst buffer management, knowledge regarding the period of the I/O phases was required. As this exceeds the scope of Extra-P, a new tool was developed named FTIO: **F**requency **T**echniques for **I/O**. The tool has been previously briefly mentioned in deliverables D4.2, D5.3, and D6.3. In this deliverable, we cover this aspect in detail in Section 3.5. Furthermore, we provide use cases in Chapter 4.

This deliverable is structured as follows: In Chapter 2, we elaborate on how the traces are collected and grouped to generate profiles. We also examine the added functionalities and new interface of the metric proxy. Consequently, this chapter handles the first two stages of the life-cycle of the performance models. Chapter 3 deals with how performance models are generated with Extra-P (stage three of the life cycle) and closely examines the added I/O modelling functionalities. We explain our continuous modelling approach, the model server in the metric proxy, and how the I/O modelling aspect is realised in Extra-P. Starting from Section 3.5, we examine the tool FTIO and highlight its importance for the ADMIRE project. In Chapter 4, we demonstrate the employment of the performance models generated in WP5 in the remaining WPs from ADMIRE based on use cases. Finally, we provide a conclusion in Chapter 5 and handle future perspectives and research directions.

2 Gathering the Monitoring Data

In D5.3, we have described how we explored, simulated, and measured I/O capabilities on a given system. Moreover, we described the I/O profiling interface in detail. Since this deliverable, the monitoring facilities have drastically evolved, leading to improved insights into the running program on the machine. Indeed, to fit in the ADMIRE monitoring loop concept, we had to reconsider our software approach to further extend the monitoring capabilities of the system. The previous monitoring methodology did not provide either job-level data or traces for each job in a convenient manner. However, starting from the design of the previous proxy, we were able to work on a reduction tree derived from the ideas pioneered by LIMITLESS to provide real-time machine-wide job tracking. This capability comes in complement to the previous profile support and now allows: (1) real-time tracking of the current application state and (2) backtracking in time to obtain the state of the given application. In addition, the profiles are now stored in a common store, which is exposed with a proper API. This API allows the Intelligent Controller (IC) to query data in a scalable and portable manner. In this chapter, we are going to describe these aspects in more detail, providing explanations of the changes and their underlying motivation for each of the aforementioned components.

2.1 Enabling Job Tracking in the Metric Proxy

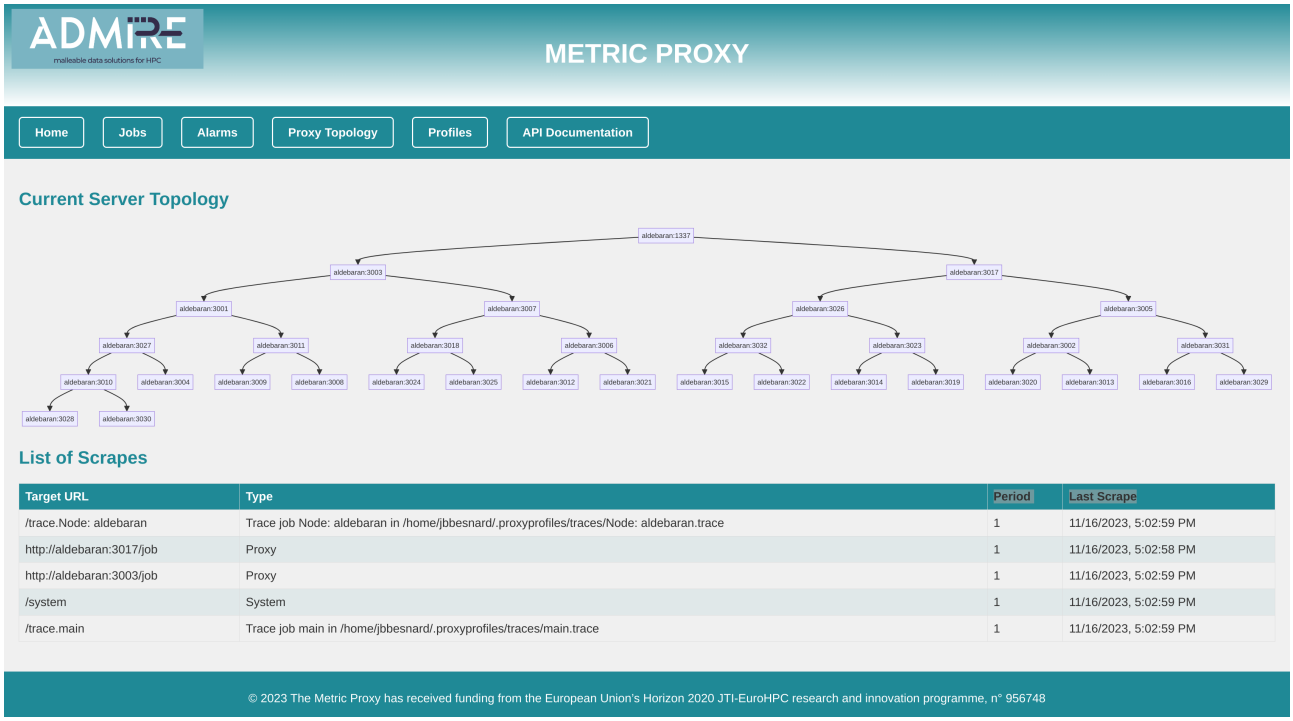


Figure 2.1: Layout of the proxy servers as displayed in real-time in the proxy web interface. In this example, 32 servers send their data up every second.

Up to D5.3, our monitoring infrastructure was not able to track jobs; it was aimed at monitoring all the

nodes in a scalable manner, and the job information was stored separately, requiring a slicing in both space and time of the performance data to obtain the corresponding time series for a given job. This was a working approach, however, and as shown in Figure 2.1, we decided to simplify it drastically by relying on a dynamic Tree-Based Overlay Network (TBON) to perform resource aggregation. In this new layout, the multiple jobs on the system, even distributed, are able to contribute metrics to the local proxy, which in turn streams data to the root proxy that is in charge of persisting the data.

On the Prometheus side, we are still able to consume the performance data, keeping them for the long term, a design choice we made early in the project and that we still find valid. What changes here is the number of required Prometheus servers, indeed, the previous design required one server per node and thus led to overhead on the compute nodes. Conversely, our new model exposes the root visibility of all the jobs with a large number of Prometheus endpoints (one per node, one per job, and one summing all of those). The "main" endpoint is tracked in Prometheus, but nothing prevents tracking at finer-grain with the difference that now only a single scraping endpoint is required, drastically improving the scalability of the model.

The main consequence of this model is the availability of trace data in real-time with a relatively low sampling period. This was a requirement of the FTIO; the frequency modelling approach (see Section 3.5), which needed real-time counter data to provide its predictive insights. This work, therefore, simplified the implementation of real-time I/O modelling as the previous spatial and temporal slicing is now unneeded. All data are now available using JSON HTTP endpoints, which can then be shown in a graphical user interface, also embedded in the proxy itself. This design allows the proxy to be user-friendly and open to various data consumers of the ADMIRE projects – namely the Intelligent Controller.

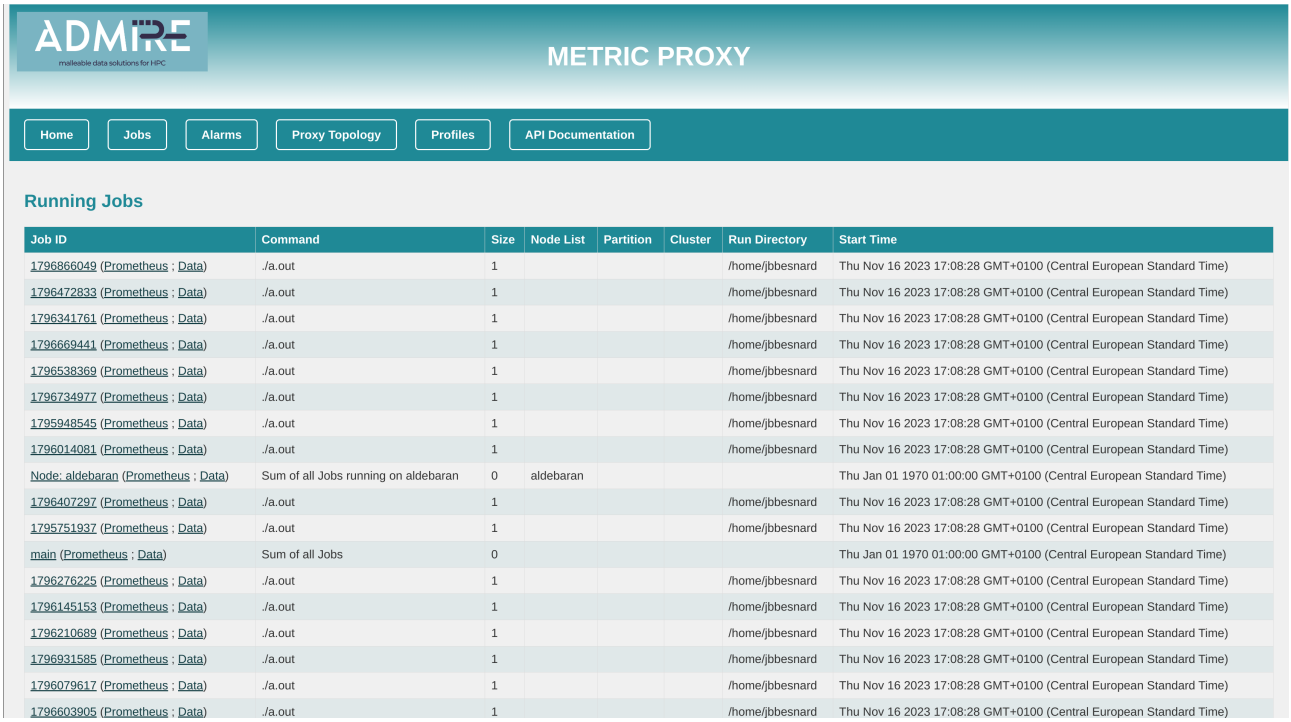


Figure 2.2: Real-time tracking of the global state of jobs on the parallel machine. Each job can run on a subset of the machine.

Figure 2.2 presents the list of all running jobs on the supercomputer, this list is built dynamically and updated every second, showing all processes contributing data. In addition to the job granularity, we have kept the node-level granularity with "Jobs" named "Node: XXX" with XXX as the hostname of the given node. This allows us not to give up on the spatial view we previously emphasised in ADMIRE. Second, a "main" job gathers all the metrics in a single view, practically watching the whole system in a single place.

2.2 Per Job Profiles

In our previous deliverable, we described how profiles were stored for each job. This is still the case for the new version of the proxy, all jobs are eventually stored in the filesystem, including the final counter snapshot and a job description. One difference is that thanks to the TBON, it is not needed to perform a filesystem reduction, all is done by the root proxy, which can save the performance data once the job finishes, as shown in Figure 2.3.

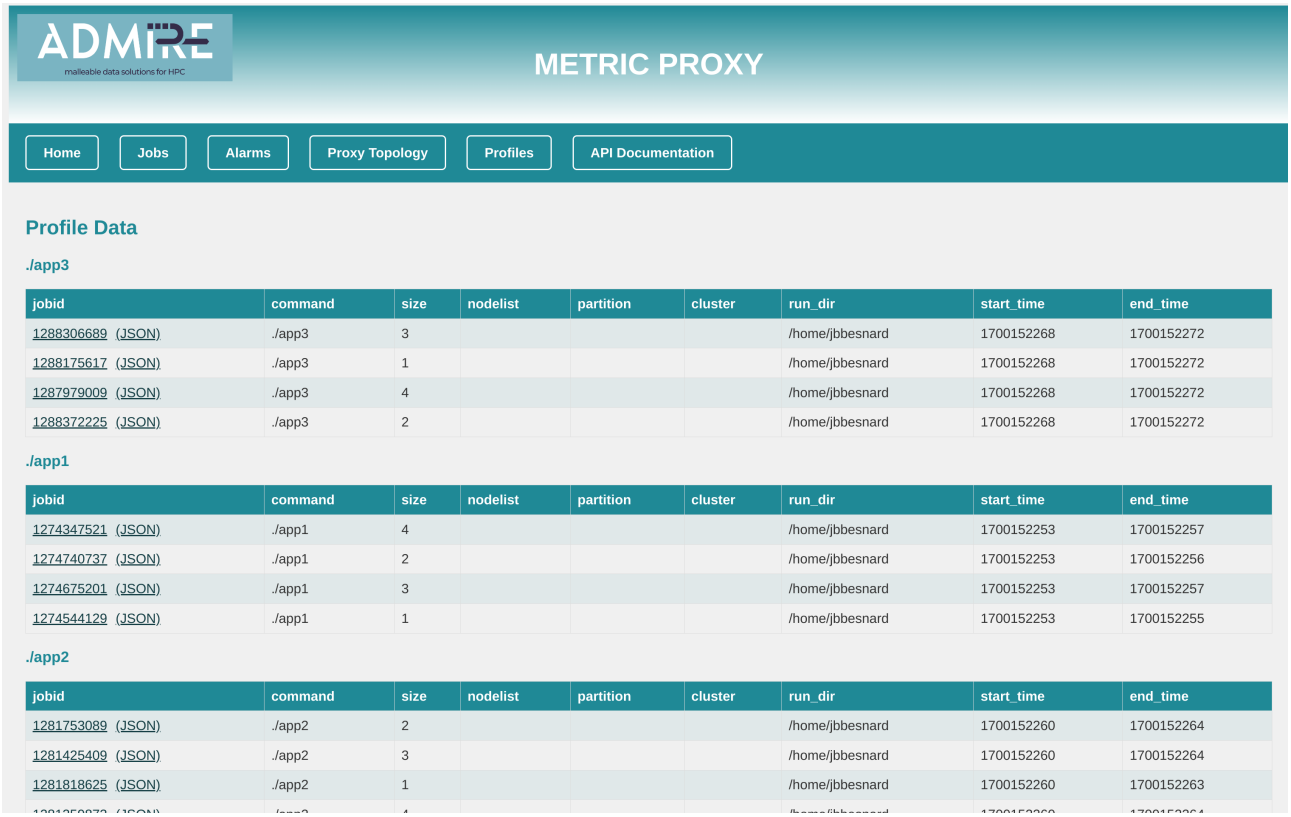


Figure 2.3: List of profiles matching past jobs gathered by command line.

These profiles play an important role in the Extra-P modelling capabilities as they are used to generate the performance models. A performance model is a regression of the various points in the profile to build a fitting function matching the scaling behaviour of the given application. Applications are gathered by command, and then, at any moment, the end user is able to request the generation of a model for the currently running job. The principle is that the command line of the currently running job is hashed and compared to existing models in the model store, matching the same command. If the model is available, it is returned directly for consumption by Extra-P, as discussed in the following section.

2.3 Generating Extra-P Supported Traces

As previously mentioned, profiles are stored for each job when they end by the root proxy. Each time a new profile is stored, all the previous jobs with the same command are scanned, and the corresponding JSON Lines (JSONL) file is generated as described later in Section 3.3 in details. This data is made available on an HTTP endpoint for both currently running jobs (looking for past profiles of the same command) and past jobs:

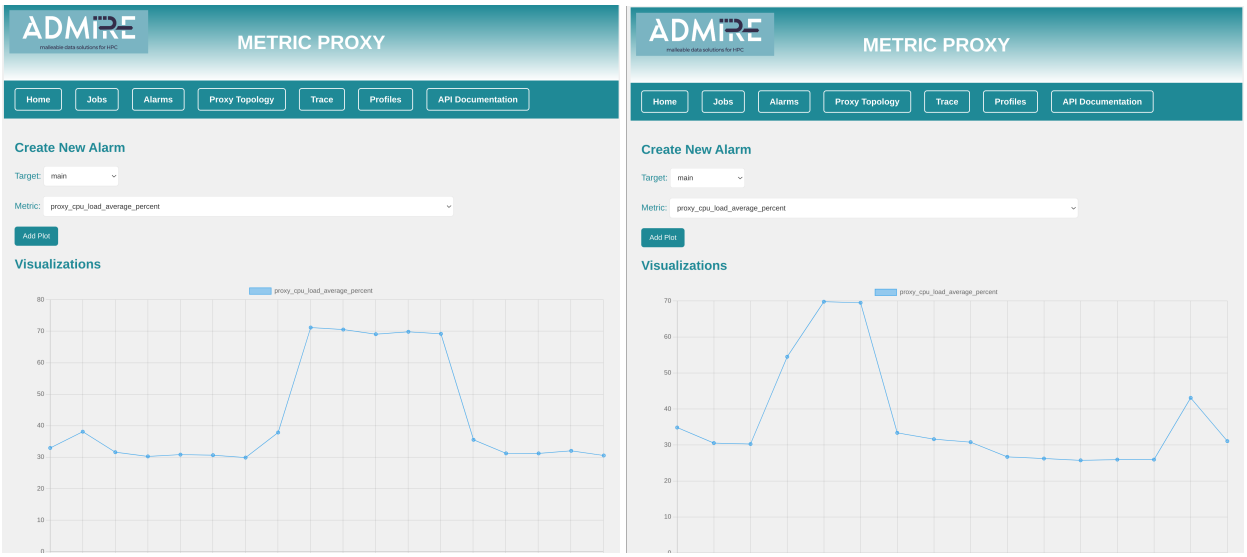
```
$> curl http://127.0.0.1:1337/profiles/extrap?jobid=1288306689
```

```
⋮ ⋮
{"params":{"size":1.0},"metric":"proxy_component_temperature_celcius","callpath":"{component=\"pch_haswell
↪  temp1\"}","value":55.5}
{"params":{"size":2.0},"metric":"proxy_component_temperature_celcius","callpath":"{component=\"pch_haswell
↪  temp1\"}","value":55.5}
{"params":{"size":3.0},"metric":"proxy_component_temperature_celcius","callpath":"{component=\"pch_haswell
↪  temp1\"}","value":56.0}
{"params":{"size":4.0},"metric":"proxy_component_temperature_celcius","callpath":"{component=\"pch_haswell
↪  temp1\"}","value":56.0}
{"params":{"size":1.0},"metric":"time","callpath":"{scall=\"getcwd\"}","value":0.00001500000000000002}
{"params":{"size":2.0},"metric":"time","callpath":"{scall=\"getcwd\"}","value":0.00002400000000000004}
{"params":{"size":3.0},"metric":"time","callpath":"{scall=\"getcwd\"}","value":0.000042}
{"params":{"size":4.0},"metric":"time","callpath":"{scall=\"getcwd\"}","value":0.000054}
{"params":{"size":1.0},"metric":"hits","callpath":"{scall=\"sched_getaffinity\"}","value":3.0}
{"params":{"size":2.0},"metric":"hits","callpath":"{scall=\"sched_getaffinity\"}","value":6.0}
{"params":{"size":3.0},"metric":"hits","callpath":"{scall=\"sched_getaffinity\"}","value":9.0}
{"params":{"size":4.0},"metric":"hits","callpath":"{scall=\"sched_getaffinity\"}","value":12.0}
```


In turn, this JSONL description is passed to Extra-P, which can generate performance models as further described in Chapter 3 of this document.

2.4 Trace Support

Traces are a new aspect of the metric proxy. They were driven by the need to feed FTIO with real-time performance data. It is a complement to the Prometheus approach with the notable difference that traces are not dependent on the Prometheus server to be consumed; this allows easier prototyping of analysis on models in a post-mortem fashion. Initially, we devised that traces were too large to be stored long-term, however, we developed a new way of tracing profiles. Indeed, the Metric Proxy is built around the notion of profiles over time, also called snapshots in the performance field. These snapshots are the current state of the measurement counters, either gauges varying over time or monotonic counters.



(a) 1 second sampling period.

(b) 2 seconds sampling period.

Figure 2.4: Illustration of trace resampling. In this case, we limited the trace size to 100 KB and observed a resampling between the two figures, the consequence of the resampling is a doubling of spacing between points on the x-axis.

However, as shown in Figure 2.4, these snapshots have summative proprieties, meaning that starting with a trace at a given sampling rate, we can halve the sampling rate by averaging the gauges and keeping the largest counter. Doing such at any moment, it is possible to halve the current trace, by multiplying by two the sampling period. It means our traces start at a high frequency (a snapshot per second) up to the point they exhaust their storage space (32 MB in our current configuration). When this is the case, the frequency is divided by two, and the trace is folded in two, freeing half the used space by practically resampling the existing trace. This adaptive approach then attempts to keep the highest sampling frequency while maintaining a reasonable storage budget

for these traces. Another optimisation is storing data in binary format, avoiding a costly JSON overhead. As far as traces are concerned, we propose the following endpoints:

- **/trace/list**: list of available traces with their job description
- **/trace/read?job=[JOBID]**: read all events in a given trace identified by its jobid
- **/trace/plot?job=[JOB]&filter=[METRIC NAME]**: get the values of the given metric over time
- **/trace/metrics?job=[JOB]**: list metrics in a given trace

While profiles and their translation in Extra-P JSONL format were intended for performance modelling of counters in a cross-job fashion, i.e., over the scaling of these jobs, the trace has a different intent. Indeed, the trace was designed to provide FTIO, which is, as explored in further detail in Section 3.5, a frequency analysis toolkit for modelling temporal signals. As such the availability of a high-frequency signal for all of the metrics collected in the ADMIRE project is critical as it allows capture periodicity in the various signals at higher frequencies due to the Nyquist theorem, tying the sampling frequency to the maximum frequencies observed in the underlying signal.

2.5 TMIO

Performance models are essential for the ADMIRE approach as described later in Chapter 4. The monitoring framework previously described in this chapter and developed in the scope of the ADMIRE project is capable of capturing the traces of an application. Those traces are then merged to generate a single profile providing thereby measurements at different scales. The generated file can then be easily imported into Extra-P to generate a performance model for the application. While this full-blown solution is ideal for the approach followed in ADMIRE, we aimed as well at providing a more lightweight solution that can be easily deployed on arbitrary systems. In particular, for example, FTIO only requires very little data (just the bandwidth over time as described later in Section 3.5). Thus, having all the information the metric provides can be too much, depending on the use case. Consequently, we developed a C++ library that can be easily attached to an application using the `LD_PRELOAD` mechanism. The tool is called TMIO: **T**racing **M**PI-**IO**. As the name implies, and contrary to the previous approach, the functionality of this library is limited to intercepting high-level MPI calls using the PMPI interface.

Through intercepting MPI-IO calls, TMIO can gather metrics on a per-rank basis. This includes the start and end times of the I/O operations and the number of bytes transferred. The library supports two modes: online and offline gathering of data. In the offline mode, the library is attached using the `LD_PRELOAD` mechanism. Once the application terminates, i.e., at `MPI_Finalize`, a file (MsgPack, JSON, or JSONL) is generated containing the collected information. In the online mode, two lines must be added to an application: one to include the library and one to indicate when the data is flushed to a file. The newly collected data is appended to the file whenever the latter line is researched. This data can be directly processed by FTIO as explained later in Section 3.5. To obtain application-level metrics, like the total bandwidth or the time consumed by the application, the rank-level metrics are aggregated internally in FTIO. As this *not* occurs on the node where the application code is executed, but where FTIO is running, TMIO has a very low footprint on the application.

TMIO is publicly available on GitHub: `https://github.com/tuda-parallel/TMIO/`. Together with FTIO, TMIO was used for the I/O scheduling use case (Set-10) as described in Section 4.3.

3 Modelling I/O in ADMIRE

As mentioned in the introduction in Chapter 1, performance modelling has a long research history in HPC [1, 9, 12, 18, 22, 29, 32, 34–36, 42, 45, 50]. A performance model is a mathematical formula expressing a performance metric of interest, such as execution time or energy consumption, as a function of one or more execution parameters (e.g., the size of the input problem or the number of processors). No models are perfect: they are a trade-off between the collection of inaccurate information and the simplicity needed to be able to interpret them [34]. These models are often used to examine the scaling behaviour of an application or specific call paths and spot scalability bugs.

While there are various performance modelling tools in the HPC domain, Extra-P is an automatic performance-modelling tool that supports the user in identifying these bugs by generating empirical performance models that predict the scaling behaviour of the different parts of an application. Thus, for each call path of an application, Extra-P generates a performance model showing its scaling behaviour at different resource configurations (usually the number of processors) and optionally other parameters (problem size, etc.). The tool has a long research history, with recent updates adding noise-resilient empirical performance modelling capabilities based on Deep Neural Networks [42] and a strategy to reduce measurement efforts [41]. While this performance modelling tool is usually concerned with the scaling behaviour of the compute/communications functions, I/O functions are often overlooked. However, this depends on the data gathered and subsequently provided to Extra-P, rather than being a limitation of the tool itself.

To balance the resource consumption of an application, as is our intention in ADMIRE, knowing the scaling behaviour of the applications requires knowing the scaling behaviour of the compute/communications functions as well as the I/O functions. Moreover, Extra-P usually shows the scaling behaviour of the execution time as a function of the number of processes. However, this does not paint the full picture, especially when analysing I/O. When considering I/O, knowing the scaling behaviour in terms of the total transferred bytes can be beneficial to optimise the application execution, especially in the presence of novel storage components like burst buffers exploited in the ADMIRE project. For example, knowing how many bytes the applications write at different scales could influence the kind of burst buffer (shared or per node) used and aid the decision on how many times the burst buffers need to be flushed. At the same time, the total times (e.g., total compute time and total I/O time) are also needed to examine how resource-demanding an application is at different scales. Together, these models could be very valuable for I/O contention avoidance strategies, job scheduling, and, ultimately, increased system performance.

To aid the different components of the ADMIRE project in their decision-making, Extra-P was extended to model the I/O behaviour of an application. In this chapter, we describe the extensions and the added functionality in detail. We start by describing the global context of continuous modelling and its importance for the ADMIRE project (Section 3.1). Next, we describe Extra-P, its added functionalities, the model extraction, and its connection to the metric Proxy throughout Section 3.2. In Section 3.3, we describe how Extra-P can be used to generate I/O models from JSONL files as, for example, provided by the metric proxy (see Section 2.3). In Section 3.4, we demonstrate its applicability based on real applications. Furthermore, we demonstrate how performance models can be generated outside the scope of ADMIRE, using Darshan traces as explored in Section 3.4.3 or to model the I/O requirements of applications with asynchronous I/O Section 3.4.4. Finally, we examine FTIO, a tool that allows us to predict the period of the I/O phases in Section 3.5, and describe the gained benefits for the project.

3.1 Continuous Modelling: Refining the Models

One of the goals of the ADMIRE project is to enable the malleability of parallel applications concerning their I/O behaviour. To do so, we deployed a dedicated measurement infrastructure which aims at being integrated into the supercomputer as an always-on service. This metric proxy, which was the object of Chapter 2, has been designed to act as a *model server* (detailed in Section 3.2.4) for the rest of the ADMIRE ecosystem of tools. More practically, it constantly transforms performance data into more compact metrics as processed by the various advanced modelling from the project, namely Extra-P and FTIO for parametric and frequency-domain modelling. This approach, that we called *continuous modelling*, takes advantage of the management of the full instrumentation chain and a close integration between the tools.

As handled in Chapter 2, the required data for model generation can be collected by profiling an application. By continuously parsing the data from the metric proxy whenever new profiles are available, the models are continuously refined. This is especially relevant for the ADMIRE project, as more and more profiles are gathered with every execution of the application on the system. Thus, the collected data can be leveraged in this toolchain. The obtained performance models are then used to aid different components in their decision-making, closing the feedback loop and, consequently, the life-cycle of the performance models as shown in Figure 1.3 from Chapter 1.

3.1.1 From Data to Models

As its name implies, the proxy is mostly a data-forwarder with the capability of aggregating several metrics both spatially (over the machine with the TBON) and temporally with good scalability. These metrics are then persisted in both traces (intended for FTIO) and profiles (intended for Extra-P). As such, these stored data can be transparently forwarded to the respective tools to generate *models*. A model is an understanding of the behaviour of a given application over various parameters, in most cases, we focus on the number of processes (or `size`) and the temporal behaviour – identifying phases which are of uttermost importance for I/O prediction. With a global view of the system, a new profile is added each time a job finishes. Each time a profile is added, the model is automatically refined in the background by invoking the respective tools. Eventually, queries can be made on this model to infer the behaviour of the application with parameters outside of the already explored space – exploiting the projective capability of the models extracted by the measurement chain.

As far as data are concerned, the proxy gathers metrics of several types at various granularity. As such, both node-level metrics (memory, load, disk space) and per-job metrics are collected conjointly. Extra-P models are generated for jobs as they require *profiles*, which are generated on job termination. Whereas, traces can be queried on any job or running node. At the job level, interface calls are tracked, including (I/O syscalls, MPI calls, and ad-hoc performance). The proxy will greedily generate models for all the metrics it collects. Moreover, the accuracy of the models is judged using the experimental data and metrics such as the RSS.

3.1.2 Reliability of I/O Information

There are many parameters that can be used and measured to describe I/O behaviour. Not all parameters can be trusted similarly, and an algorithmic strategy needs to consider this. We provide several examples [5]:

- The number of compute resources used by each application can be obtained from the resource manager. It is easily obtained and reliable.
- Aggregated information such as the total amount of transferred data and compute time of an application can be obtained from previous runs, for example, with profiling tools, such as Darshan [13], or provided by the user. In both cases, the actual observed values could vary, and this data is only semi-reliable.
- For an application, obtaining its bandwidth as a function of the number of I/O resources requires multiple previous runs and is naturally sensitive to variability. The system could accumulate this information over time (so it would only be available to *some* of the running applications), or the user could provide it (less reliable).

3.1.3 Integration With the Intelligent Controller

Eventually, as depicted in Figure 1.2, those insights, acquired and refined from the ADMIRE always-on measurement chain, are consumed by the Intelligent Controller, the core of the ADMIRE project. Practically, a Mercury [44] interface is exposed by the proxy, allowing it to directly make high-performance requests to the model server. These requests are intended to guide the malleability decision outlining performance behaviour over scales and time (phases).

3.2 Extra-P in ADMIRE

Extra-P uses measurements of various performance metrics at different execution configurations as input to generate performance models of code regions (including their calling context) as a function of the execution parameters. This implies that the modelling results depend on the profiles used and that performance models can only be generated for the functions profiled. Consequently, only the scaling behaviour of these functions can be examined. To generate performance models, Extra-P requires repeated performance measurements. It is recommended that at least five measuring points per parameter should be performed.

In a typical workflow with Extra-P, the user would execute an application on a cluster with a profiler (for example, Score-P) to gather the performance metrics. After several runs, the gathered performance metrics can be passed to Extra-P, which generates a performance model for the different call paths of an application. The user would then spot the scalability bugs by examining the performance models in Extra-P's GUI, adapt the poor scaling parts of the code, and repeat the experiments to see if the bug has been fixed. Thus, in this optimisation cycle, the user receives input from Extra-P and adapts the code with the optimisation target of improving the application's performance.

However, in the context of the ADMIRE project, Extra-P is used differently. Different components in ADMIRE receive the performance models and try to make effective decisions with the target to balance the system resources and the application execution simultaneously. As there is typically no human in this loop, the programming interface of Extra-P was used. Moreover, the overall performance regarding different resources, rather than call-path-specific information, is needed. Starting from the user interfaces of Extra-P (Section 3.2.1), we explain how specific performance models can be extracted (Section 3.2.2) and how they are generally made accessible in the ADMIRE framework (Section 3.2.3). As the monitoring proxy is a special component that integrates Extra-P into the model server, we described in Section 3.2.4 how we further improved the accessibility of the models through a model sever, which is linked to the metric proxy.

3.2.1 Extra-P Interfaces

For a typical user, Extra-P provides two user interfaces: the GUI and the command line interface. To start the GUI, the command `extrap-gui` is executed. This opens a window and allows the user to load the file/folder containing the profiles through the toolbar. Additionally, Extra-P allows the user to pass command line arguments when starting the GUI to load the files/folders directly[1]. The GUI of Extra-P is shown and explained in Figure 3.1 for the application KRIPKE [27].

Through the metric selection tab, different metrics can be selected. This example includes the metrics: time, min_time, max_time, various PAPI counters (e.g., PAPI_FP_INS) and the bytes received (during `MPI_Testany`) and sent (during `MPI_Isend`) in the Sweep kernel. Figure 3.1 only shows the performance models for the execution time.

The second user interface is the command line interface[2], which can be accessed using the `extrap` command. Various arguments are available, including specifying:

- the file type (e.g., `--json`, `--text`, etc.),

[1] https://github.com/extra-p/extrap/blob/master/docs/quick-start.md#graphical-user-interface

[2] https://github.com/extra-p/extrap/blob/master/docs/quick-start.md#command-line-interface

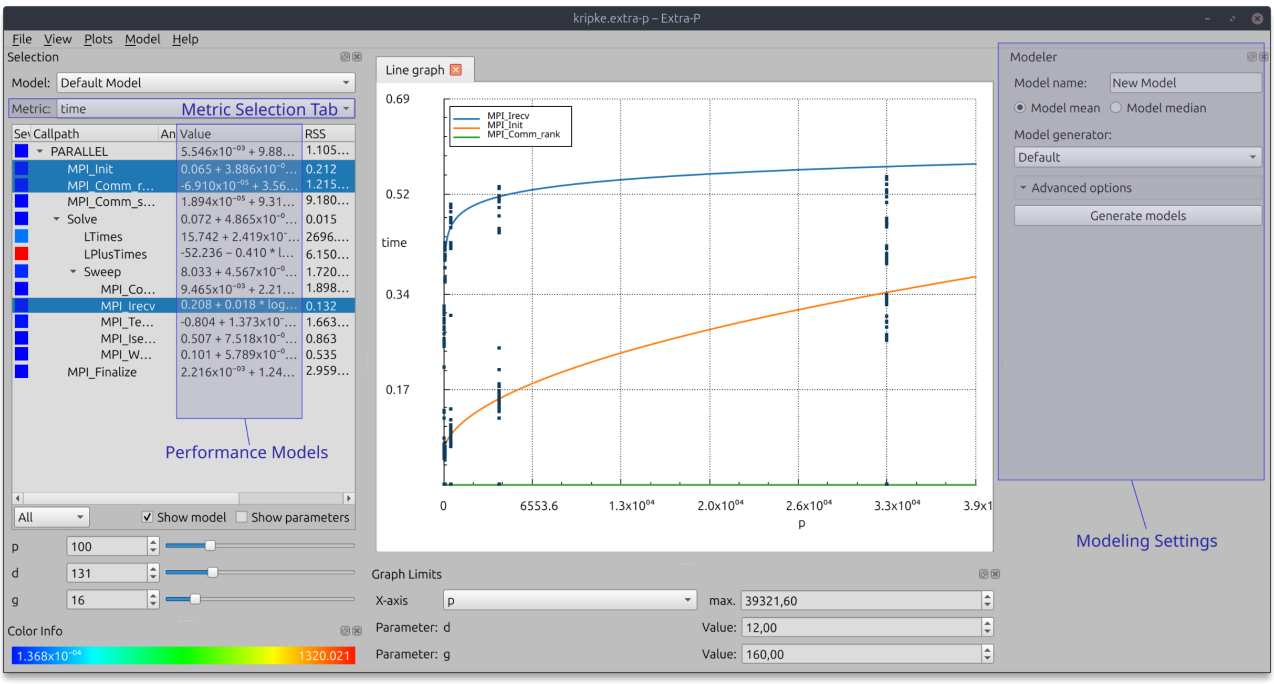


Figure 3.1: Extra-P GUI showing the performance models for the different functions of KRIPKE [27].

- an Extra-P experiment from a previous execution (`--experiment`),
- the aggregation of the values (default is mean), and
- the desired modelling technique (`--modeller <technique>`).

By default, Extra-P prints the performance models on the console. Different print options are available, which can be specified using the `--print options`. Executing `extrap -h` prints all options available. Below, the output on the console is shown:

```
$> extrap -h

usage: extrap [-h] [--version] [--log {debug,info,warning,error,critical}]
              (--cube | --extra-p-3 | --json | --talpas | --text | --experiment)
              [--scaling {weak,strong}] [--median]
              [--modeler {multi-parameter,default,basic,refining}]
              [--options KEY=VALUE [KEY=VALUE ...]]
              [--help-modeler {multi-parameter,default,basic,refining}] [--out OUTPUT_PATH]
              [--print {all,callpaths,metrics,parameters,functions,FORMAT_STRING}]
              [--save-experiment EXPERIMENT_PATH] [--model-set-name NAME]
              FILEPATH

Extra-P, automated performance modeling for HPC applications

Positional arguments:
  FILEPATH              Specify a file path for Extra-P to work with

Optional arguments:
  -h, --help            Show this help message and exit
  --version             Show program's version number and exit
  --log {debug,info,warning,error,critical}
                        Set program's log level (default: warning)

Input options:
  --cube                Load a set of CUBE files and generate a new experiment (default: False)
  --extra-p-3           Load data from Extra-P 3 (legacy) experiment (default: False)
  --json                Load data from JSON or JSON Lines input file (default: False)
  --talpas              Load data from Talpas data format (default: False)
  --text                Load data from text input file (default: False)
  --experiment          Load Extra-P experiment and generate new models (default: False)
```

```
  --scaling {weak,strong}
                        Set weak or strong scaling when loading data
                        from CUBE files (default: weak)

Modeling options:
  --median              Use median values for computation instead of mean values (default: False)
  --modeler {multi-parameter,default,basic,refining}
                        Selects the modeler for generating the performance models (default:
                        ↪  default)
  --options KEY=VALUE [KEY=VALUE ...]
                        Options for the selected modeler (default: {})
  --help-modeler {multi-parameter,default,basic,refining}
                        Show help for modeler options and exit (default: None)

Output options:
  --out OUTPUT_PATH     Specify the output path for Extra-P results (default: None)
  --print {all,callpaths,metrics,parameters,functions,FORMAT_STRING}
                        Set which information should be displayed after modeling. Use one of {all,
                        callpaths, metrics, parameters, functions} or specify a formatting string
                        using placeholders (see
                        https://github.com/extra-p/extrap/tree/v4.1.0-al
                        pha2/docs/output-formatting.md). (default: all)
  --save-experiment EXPERIMENT_PATH
                        Saves the experiment including all models as Extra-P experiment (if no
                        extension is specified, '.extra-p' is appended) (default: None)
  --model-set-name NAME
                        Sets the name of the generated set of models when outputting an experiment
                        (default: 'New model') (default: New model)
```

As described in the help manual of Extra-P, the experiments can be saved with the `--save-experiment` flag. The GUI provides similar options; When clicking on *"File > Save experiment"*, the experiment presented in Figure 3.1 can be stored. With `kripke.extra-p` containing the exported Extra-P experiment from the GUI, the experiment can be read by Extra-P as well through the command line interface by executing:

```
$> extrap --experiment kripke.extra-p
```

This generates performance models for all call paths, specified parameters, and sampled metrics. For this example, in particular, this prints the following output on the console:

```
Callpath: PARALLEL
  Metric: visits
    Measurement point: (8.00E+00,2.00E+00,3.20E+01) Mean: 1.00E+00 Median: 1.00E+00
    Measurement point: (8.00E+00,2.00E+00,6.40E+01) Mean: 1.00E+00 Median: 1.00E+00
    ⋮
    Measurement point: (3.28E+04,1.20E+01,1.60E+02) Mean: 1.00E+00 Median: 1.00E+00
    Model: 1.0000000000000024
    RSS: 8.95E-28
    Adjusted R^2: 1.00E+00
  ⋮
 ⋮
Callpath: PARALLEL->Solve->Sweep->MPI_Irecv
  Metric: visits
    Measurement point: (8.00E+00,2.00E+00,3.20E+01) Mean: 1.20E+04 Median: 1.20E+04
    Measurement point: (8.00E+00,2.00E+00,6.40E+01) Mean: 1.20E+04 Median: 1.20E+04
    ⋮
    Measurement point: (3.28E+04,1.20E+01,1.60E+02) Mean: 2.32E+04 Median: 2.40E+04
    Model: 11250.000000000005 + 900.0000000000003 * log2(p)^(1.0)
    RSS: 3.24E+08
    Adjusted R^2: 0.00E+00
  Metric: time
    Measurement point: (8.00E+00,2.00E+00,3.20E+01) Mean: 2.47E-01 Median: 2.45E-01
    Measurement point: (8.00E+00,2.00E+00,6.40E+01) Mean: 2.50E-01 Median: 2.46E-01
```

```
   ⋮
   Measurement point: (3.28E+04,1.20E+01,1.60E+02) Mean: 5.51E-01 Median: 5.18E-01
   Model: 0.20798724304290037 + 0.017867804631285397 * log2(p)^(1.0) +
   ↪  0.005179387204829805 * log2(d)^(1.0) + 2.9500644858418412e-06 * g^(2.0)
   RSS: 1.32E-01
   Adjusted R^2: 8.77E-01
⋮
```

Similar to the GUI, the different metrics are presented for each call path. Moreover, nested call paths are specified using the arrow operator (->). Note that we only showed a very small portion of the outputted text. For each function, the metric is provided first, followed by the measurement points. The performance model is provided afterwards (cf., `Model`) in addition to statistical quality control metrics: the residual sum of squares (RSS) and the adjusted coefficient of determination ($Adj.\ R^2$). The $Adj.\ R^2$, for example, shows how well the found model replicates the measurements in the conducted performance measurements while also acknowledging the complexity of the found performance model.

While it is possible to extract the desired performance model by parsing the output of Extra-P (e.g., using the shell command `grep`), another way is to rely on the Python interface of Extra-P, as outlined in the next section.

3.2.2 Extracting Specific Performance Models

Since Extra-P is developed in Python, its modules can be easily imported, and custom functions can be deployed for specific purposes. This brings the advantage that the performance models are available in Python and can easily be evaluated at specific measurement points. With the JSONL file `test1.jsonl` from the Extra-P GitHub[3], the code that can (1) extract specific metrics or (2) iterate over all of them looks as follows:

```python
# Imports
from extrap.entities.callpath import Callpath
from extrap.entities.metric import Metric
from extrap.fileio.file_reader.jsonlines_file_reader import(
    read_jsonlines_file)
from extrap.modelers.model_generator import ModelGenerator
import numpy as np

# Load the JSONL data. For different file formats, there are
# different read libraries in extrap.fileio
experiment = read_jsonlines_file("./data.jsonl")

# Initialize model generator
model_generator = ModelGenerator(experiment)

# Create models from data
model_generator.model_all()

# ------ At this point, the models are generated
# Next, let's see how to evaluate them

# 1) Specify the metric and call path, e.g., default and root, respectively
cp0 = Callpath("<root>"), Metric("<default>")
pm = model_generator.models[cp0].hypothesis.function
print(f"Model evaluated at x = 5 and y = 5:\n{pm.evaluate([5,5])}\n")

# 2) Or iterate over all of them:
```

[3] https://github.com/extra-p/extrap/blob/master/tests/data/jsonlines/test1.jsonl

```
28 for model in model_generator.models.values():
29     # Measurment points
30     pts = model.measurements
31     print(f"Measurement points are:\n{pts}\n")
32
33     # Evaulated function @ measurement points
34     pred = model.predictions
35     print(f"Prediction points are:\n{pred}\n")
36
37     # The actual model
38     print(f"Model is:\n{model.hypothesis.function}\n")
39
40     # Evaluate at specific points
41     # Here, data.jsonl is a 2D data set, hence, a 2d array must be provided
42     m = model.hypothesis.function.evaluate(
43         np.array([[1, 24, 58], [1, 29, 30]]))
44     print(f"Model evaluated at x = [1 24 58] and y = [1 29 30] is:\n{m}\n")
45
46     # Or evaluate just at a single point [x,y]
47     pm = model.hypothesis.function.evaluate([2, 3])
48     print(f"Model evaluated at x = 2 and y = 3:\n{pm}\n\n")
```

As the code shows, the models are generated with the single function call to `ModelGenerator` with an experiment as the input argument. Afterwards, the models can be easily extracted by specifying the function name and the metric of interest (lines 23-24) or by iterating over all models (lines 28-48).

3.2.3 Accessing the Performance Models

In the context of the ADMIRE project, we initially provided two ways to access the performance models before the development of the model server (see Section 3.2.4). The first way is that the Intelligent Controller can trigger Extra-P to generate performance models on demand. In particular, the Intelligent Controller gains access through the interface described in the last section. The second way is through a Redis database. As Extra-P is written in Python, and Redis provides a Python interface, storing the performance models is straightforward using, for example, this script:

```
1  # Import
2  from extrap.entities.experiment import ExperimentSchema
3  from extrap.entities.callpath import Callpath
4  from extrap.entities.metric import Metric
5  from extrap.fileio.jsonlines_file_reader import read_jsonlines_file
6  from extrap.modelers.model_generator import ModelGenerator
7  import numpy as np
8  import redis as rd
9
10
11 experiment = read_jsonlines_file('./data.jsonl')
12 model_generator = ModelGenerator(experiment)
13
14 # Create models from data
15 model_generator.model_all()
16
17 redis = rd.Redis(host='localhost', port=6379, decode_responses=True)
18
19 callpaths = ''
20 metrics   = ''
21 models    = ''
```

```
22 for key,model in model_generator.models.items():
23     callpaths += key[0].name + '\n'
24     metrics   += key[1].name + '\n'
25     models    += str(model.hypothesis.function) + ' --- '
26
27 redis.hset('Application_X', mapping={
28 'callpath': callpaths,
29 'metric': metrics,
30 'model': models,
31 })
32
33
34 redis.hgetall('Application_X')
35
```

As shown, the models are merged into a single string and stored at line 27 with the application name as a hash. While accessing the performance models becomes relatively easy, this shows the disadvantage of this approach as well: The hash key (here `Application_X`) can be quite complex and long as it must allow to distinguish simulation performed with different command line options, input files, etc. One way of realising this aspect is by hashing the application executable. Other options are currently still examined in the project.

3.2.4 Implementing the Model Server in the Metric Proxy

To complement these manual integrations of Extra-P, we created a fully automated version of the continuous modelling approach in the metric proxy. Unlike previous examples relying on calling in the Python code of Extra-P, this implementation extracts the models (i.e. the equations from the output of Extra-P and then evaluates them in the metric proxy). This has the advantage of exposing Extra-P models coherently with the rest of the ADMIRE components and also provides faster model evaluation times as the proxy is a compiled binary.

The updated proxy version now possesses the capability to gather data points from various program executions. These collected data points are referred to as profiles and are transmitted to Extra-P in the form of a JSONL file. However, for these data points to be effectively utilised, it was necessary to enable the projection of values onto the models that Extra-P can construct from them. Consequently, by transmitting the JSONL file to Extra-P and establishing a connection between its output and the Metric Proxy, we have successfully established a model server.

The fundamental concept of a model server lies in its automation of the projection. On one hand, each new profile is intended to be incorporated into the reference model for a specific program. This is achieved by appending the JSONL file (as described in the preceding section) whenever a new profile for a particular program is added to the profile store. On the other hand, the proxy maintains an in-memory snapshot of the Extra-P model and updates it whenever there is a change in the JSONL file (by examining the metadata of the JSONL file and comparing it to the most recent model generation). This caching mechanism is essential as the process of Extra-P interpolation across numerous metrics can be time-consuming (typically around 30 seconds). By adopting this approach, we can deliver a real-time stream of projected points, utilising the pre-fetched model, to the Intelligent Controller.

Internally, this materialises as the following HTTP interfaces at the server level:

- `profiles/points?jobid=X`: Retrieves the values corresponding to the execution size for all metrics. This constitutes the known execution points of the application.
- `model/get?jobid=X`: Retrieves the Extra-P models for jobs corresponding to a given JOBID (matched by command-line). Models include the equation and the modelling error, indicating how well the model aligns with the data points.
- `model/evaluate?jobid=X&metric=X&size=X`: Interpolates a point using the Extra-P model at a given size, utilizing the Extra-P model interpolation.

- `model/plot?jobid=X&metric=X&start=X&end=X&step=X`: Evaluates the Extra-P model across a range of values simultaneously. This functionality is employed to plot both the *profile/points* and the model on the same graph, providing a view similar to that of Extra-P.

Furthermore, alongside these JSON-based HTTP endpoints, we have incorporated Mercury endpoints to facilitate high-performance RPC-based communication with the Intelligent Controller. The primary function provided is the ability to assess a specific metric at a given size, akin to the functionality offered by `model/evaluate` in the HTTP interface.

In summary, the proxy now updates the JSONL file targeting Extra-P each time a job finishes by aggregating all existing profile values. Additionally, a caching mechanism is implemented to store results when accessing the model. This caching mechanism expedites future model queries to the server. Ultimately, the proxy consistently maintains and exposes the latest models for a given program, facilitating informed decisions regarding its adaptability to the Intelligent Controller.

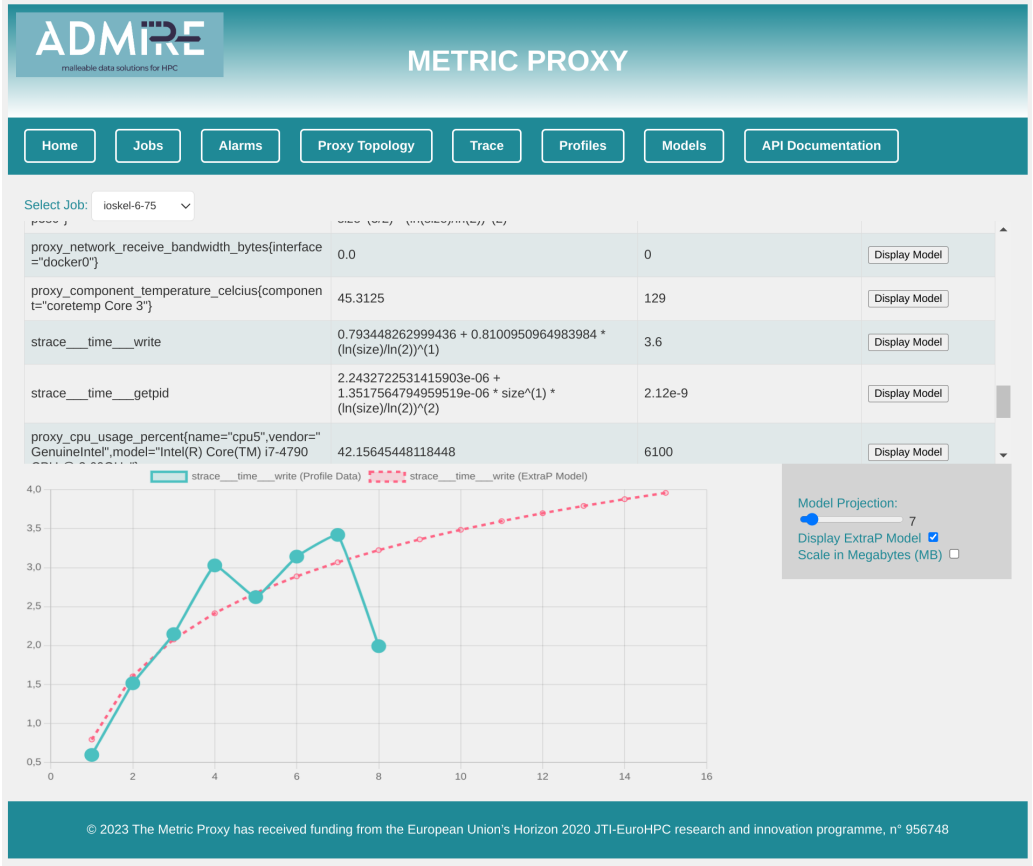


Figure 3.2: Metric proxy HTML interface demonstrating live model generation and evaluation.

The practical use of this model in the ADMIRE infrastructure is foreseen to rely on the Mercury endpoint, which allows tight, low-latency coupling between the components. As illustrated in Figure 3.2, we also implemented an HTML/JS interface to illustrate the data gathered by the metric proxy internally. On the `http://localhost:1337/model.html` page, we present an output similar to the one of Extra-P. After choosing a JOBID (past or currently running), the models are fetched using `model/get`. Then, when the user chooses a model by clicking on it in the list, it will be plotted for the available points extracted from the profiles (using `profiles/points` and `model/plot`). In addition, the user can evaluate the model at larger sizes using the "Model Projection" slider. In this figure, the blue line represents the true measurements for the given program, and the dashed line is the Extra-P model. Here, we see that despite matching the overall shape of the curve, there are some outliers. In order to measure the quality of a model prior to using it, Extra-P provides an RSS value that we leverage to measure how well the model fits the original points.

3.3 Modelling I/O With Extra-P

To generate I/O performance models with Extra-P, profiles containing measurement points at different application scales are required. As described in Chapter 2, the metric proxy is capable of delivering profiles at different levels. In particular, we distinguish between high-level profiles like MPI function calls and low-level profiles like those obtained from syscalls. For each Job, the metric proxy aggregates all functions in a single trace file as described in Section 2.1. Next, a single profile file is generated from these traces as described in Section 2.2. This file is provided to Extra-P to create the performance models. Consequently, as these profiles are rich with I/O metrics, including the number of bytes read and written by the various I/O functions and at different levels, Extra-P can easily model the application behaviour in this regard. In particular, I/O modelling capabilities can be attained by providing the profiles in a particular file structure supported by Extra-P.

By default, Extra-P supports a variety of file formats like plain text, JSON, JSONL, Cube4 format, and many more[4]. Since we intend to provide the data to Extra-P continuously, the JSONL file format was chosen, as new data can be easily appended to the end of the file. In general, a single line in the JSONL format for Extra-P has the following structure:

```
{ "params": { "<parameter1>": 0, "...": "..." }, "value": 0.0 }
```

To demonstrate how a typical JSONL profile generated with the metric proxy looks, consider the following NEK5000 experiments executed on the HPC4AI cluster of the University of Turin [2,33]. NEK5000 represents one of the applications handled in the ADMIRE project, and is thus a part of WP7. For our experiment, we used a 60-node partition with Intel Broadwell processors and an Omnipath interconnect (100 Gb/s) on the Turin cluster. We executed the NEK5000 *turbPipe* test case with checkpointing enabled for process configurations ranging from 32 cores (1 node) up to 1404 cores (39 nodes) by using a step size of 32 cores. Consequently, we obtained 39 different configurations. Next, the traces from the different configurations are gathered and converted to a single JSONL file. As the number of processes is the only parameter for our example, the JSONL file looks as follows:

```
1    {"params": {"Processes": 36}, "callpath": "strace_size..->scall->pread64", "metric": "Size (B)", "value": 1868271840.0}
2    {"params": {"Processes": 72}, "callpath": "strace_size..->scall->pread64", "metric": "Size (B)", "value": 1868340384.0}
.          .
.          .
.          .
38   {"params": {"Processes": 1368}, "callpath": "strace_size..->scall->pread64", "metric": "Size (B)", "value": 1870807968.0}
39   {"params": {"Processes": 1404}, "callpath": "strace_size..->scall->pread64", "metric": "Size (B)", "value": 1870876512.0}
40   {"params": {"Processes": 36}, "callpath": "strace_time..->scall->pread64", "metric": "Time (s)", "value": 1.069}
41   {"params": {"Processes": 72}, "callpath": "strace_time..->scall->pread64", "metric": "Time (s)", "value": 0.918}
.          .
.          .
.          .
77   {"params": {"Processes": 1368}, "callpath": "strace_time..->scall->pread64", "metric": "Time (s)", "value": 1.092}
78   {"params": {"Processes": 1404}, "callpath": "strace_time..->scall->pread64", "metric": "Time (s)", "value": 1.097}
79   {"params": {"Processes": 36}, "callpath": "strace_hits..->scall->pread64", "metric": "Hits", "value": 384.0}
80   {"params": {"Processes": 72}, "callpath": "strace_hits..->scall->pread64", "metric": "Hits", "value": 738.0}
.          .
.          .
.          .
116  {"params": {"Processes": 1368}, "callpath": "strace_hits..->scall->pread64", "metric": "Hits", "value": 12672.0}
117  {"params": {"Processes": 1404}, "callpath": "strace_hits..->scall->pread64", "metric": "Hits", "value": 12960.0}
118  {"params": {"Processes": 36}, "callpath": "strace_size..->scall->pwrite64", "metric": "Size (B)", "value": 303548544.0}
.          .
.          .
.          .
235  {"params": {"Processes": 1404}, "callpath": "strace_hits..->scall->pwrite64", "metric": "Hits", "value": 3000.0}
.          .
.          .
.          .
452  {"params": {"Processes": 144}, "callpath": "tau_size..->function->mpi_send", "metric": "Size (B)", "value": 9922929351.0}
.          .
.          .
.          .
9516 {"params": {"Processes": 1224}, "callpath": "tau_time..->function->mpi_send", "metric": "Time (s)", "value": 132.192}
```

Note that, for better readability, we limited the time metric to 3 digits after the decimal point, though it is up to 19 digits. Moreover, we sorted the lines according to the number of processes and grouped similar function calls. In the JSONL format, newly added data is appended to the end of the file, and thus, this sorting is most likely more random. As observed, for the 39 different process configurations, 39 measurements are obtained for each function call per metric, which are for this example bytes transferred, time, and function visits. Lines 1 to 39 present the measurement points for the `pread64` calls traced for transferred (read) bytes. The following 39

[4] https://github.com/extra-p/extrap/blob/master/docs/file-formats.md

lines (i.e., from 40 to 78) present the profile in terms of time and the next 39 lines (from 79 to 117) in terms of function visits (hits). Thus, 117 lines describe the measurement points over the different process configurations for each function call traced. Line 118 continues with the next function profile obtained with `strace`, namely `pwrite64`. The high-level function calls are listed after all measurement points obtained using `strace` are handled. These profiles start with `tau_*` as indicated in the call path. Note again, that this sorting is only for better readability, in reality, the functions are not sorted.

With this file at hand, the performance models for each of the traced functions can be generated by executing:

```
$> extrap-gui --json  nek5000.jsonl
```

This opens the GUI of Extra-P and directly starts the performance model generating. As mentioned above, three metrics can be selected for the experiment performed: the transferred bytes, the execution time, and function visits. Figure 3.3 presents the performance models for the transferred bytes versus the number of processes for selected syscall functions.

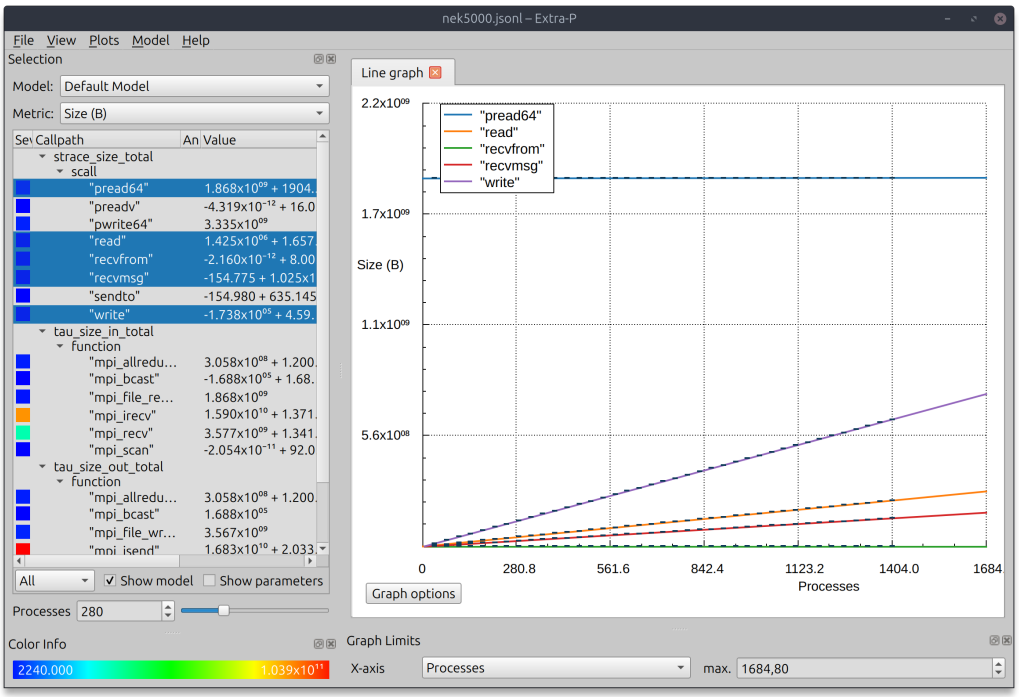


Figure 3.3: Extra-P models for selected syscall functions for NEK5000.

Based on the profiles gathered by the metric proxy, Extra-P models various I/O functions, including low-level syscalls and high-level MPI (i.e., MPI-IO) functions.

While the extended modelling capabilities are great for examining individual I/O functions, a more substantial contribution for the ADMIRE project is to show the overall scaling behaviour of the different parts (i.e., compute and I/O) of an application. Especially in the context of job scheduling, as explained later in Section 4.1, knowing the scaling behaviour in terms of different resources is essential to balance the applications on an HPC system.

As described in Section 3.1, by continuously passing data from the proxy (see Chapter 2), it is possible to improve the models and thus refine them. As the collected monitoring data can be continuously appended to the same file, the performance models are refined whenever more data is available. In particular, each new measurement point usually improves the performance models of Extra-P. As such, the modelling capability is now able to track application behaviour over time, with the possibility of applying a temporal filter over the profiles used to model (for example, the last n profiles). This becomes relevant when the application is changed, though this aspect would also be visible through the hashing key (see Section 3.2.3). Additionally, as

Extra-P offers metrics that provide quality control of the models generated (see Section 3.2.1), decisions about the accuracy of the models can be made as well.

3.4 Examples

In the previous section, we demonstrated the extended modelling capabilities of Extra-P with NEK5000 as an experiment. In this section, we provide further examples to demonstrate the gained performance insight provided by Extra-P.

3.4.1 IMB-IO

To demonstrate the extended I/O modelling capabilities of Extra-P let us examine the scaling behaviour of the IMB-IO benchmark from the Intel MPI benchmarks [8] executed on the Turin cluster [2, 33] with up to 576 processes. The experiment has been recently published in a blog post[5]. We executed the IMB-IO benchmark with the default settings (except for the iteration flag set to 10). The benchmark suite usually repeats the individual benchmarks[6] (e.g., P_write_shared, S_read_indv, etc.) with a different number of processes, such that the highest number is limited to the number of ranks specified. Thus, the unused processes wait at an MPI barrier. For our analysis, we examined the benchmark with up to 576 processes (16 nodes).

To monitor the benchmark during its execution, we used the monitoring proxy. For the execution with 576 process, the Grafana dashboard, which shows the metrics stored in Prometheus, is shown in Figure 3.4.

Figure 3.4: Grafana dashboard showing in real time the execution of the IMB-IO benchmark on the Turin cluster with 576 ranks.

After the traces are collected, a single file containing the profiles from all runs is generated. For this example, the number of processes is the only parameter, and the metrics collected are transfer size, hits, and time. Consequently, these metrics can be selected in the Extra-P GUI. Thus, instead of displaying absolute call paths in Extra-P as traditionally done, we show the call path relative to the tools used to capture the information (i.e, MPI-related information with tau, while strace is used to capture the syscalls). Figures 3.5 and 3.6 show the performance models generated in Extra-P. While Figure 3.5 shows the scaling behaviour of the function calls `pwrite64` and `MPI_File_write` in terms of size (in bytes) versus the number of processes, Figure 3.6

[5] https://admire-eurohpc.eu/extrap-model/

[6] https://www.intel.com/content/www/us/en/docs/mpi-library/user-guide-benchmarks/2021-2/imb-io-blocking-benchmarks.html

shows the scaling behaviour of the execution time (in seconds) versus the number of ranks for low-level syscalls as well as for high-level MPI calls.

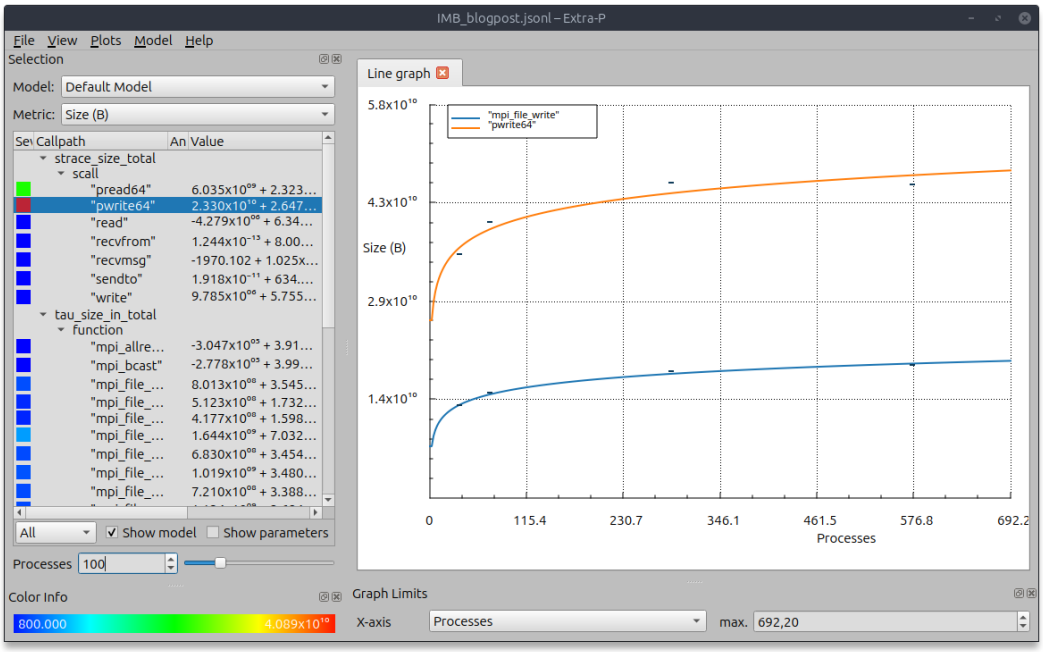


Figure 3.5: Scaling behaviour of the function calls `pwrite64` and `MPI_File_write` in terms of size (in bytes) versus the number of processes.

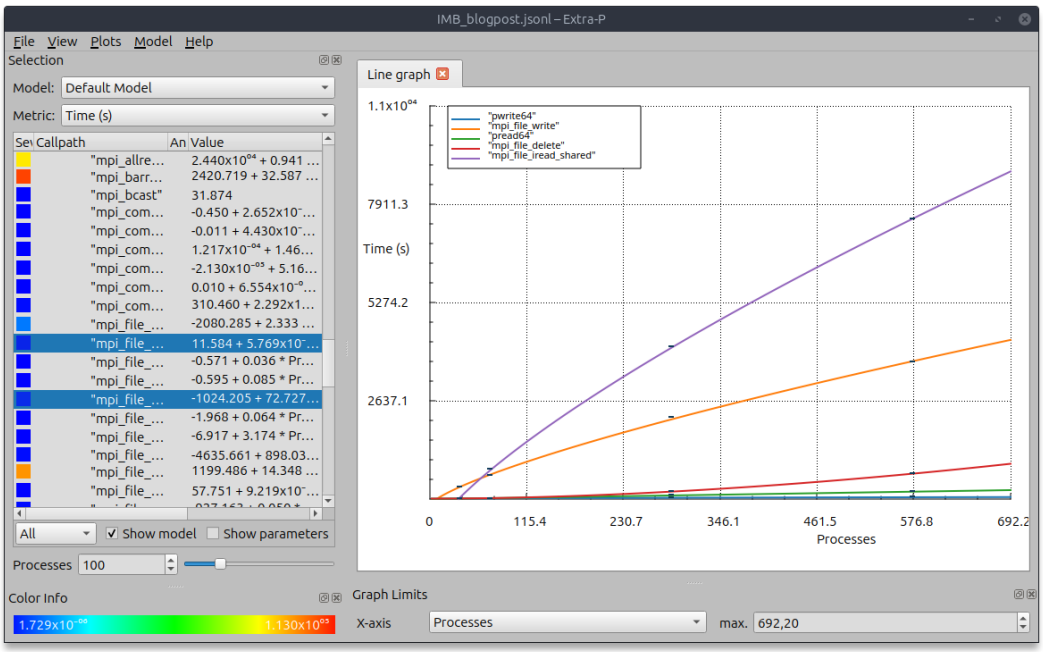


Figure 3.6: Scaling behaviour of the execution time (in seconds) versus the number of ranks for low-level syscalls and high-level MPI calls from the same setup.

Figure 3.7 presents the output of the metric proxy when leveraging the models from Extra-P for the same example. It is shown that the proxy is capable of interpolating the values using the function provided. In addition, the proxy invokes the modelling process each time a new point is added (when a job with the same command finishes). As a consequence, the model is in constant evolution, being refined for each job. In this

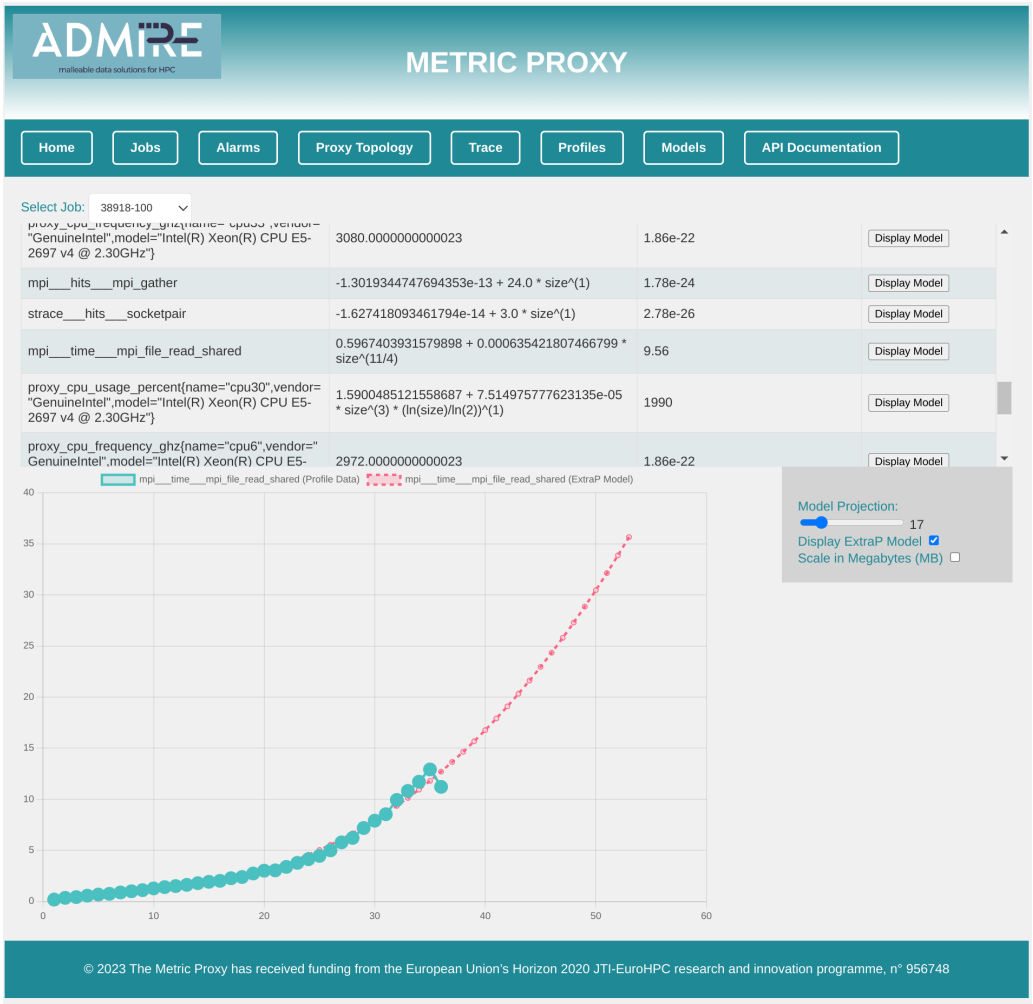


Figure 3.7: model interpolation in the metric proxy of `Pread_shared` from the IMB benchmark.

case, we show the total time spent in `MPI_File_read_shared`, and it can be seen that the model is close to the true measurement point (except for an outlier). Note that runs with the same size have their times averaged.

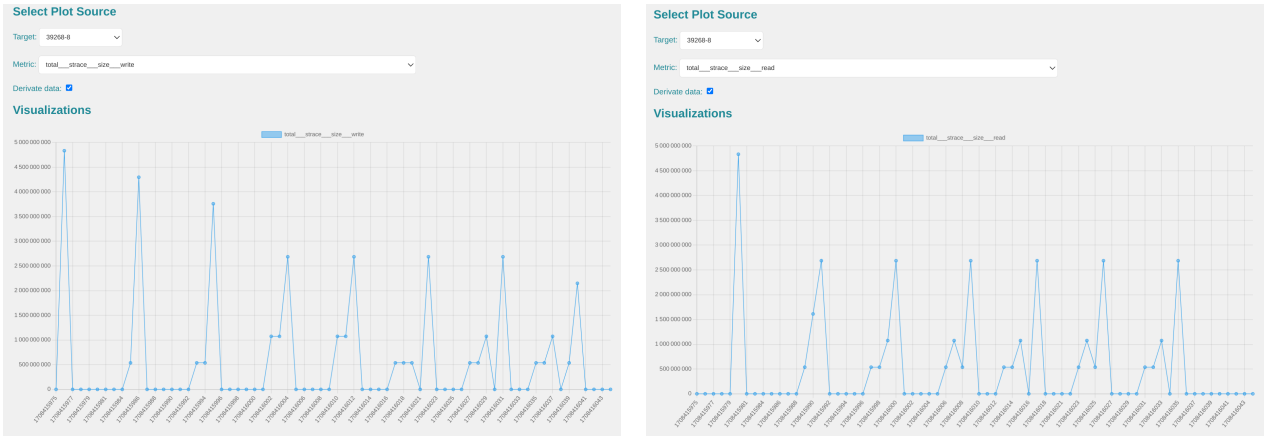
3.4.2 IO Skeleton Application

As a vehicle to measure the I/O performance at scale, we have developed a simple I/O benchmark exhibiting various behaviours as functions of provided parameters. The goal of this benchmark is to synthesise payloads mimicking the actual applications. In fact, in HPC, most applications are bulk-synchronous and often do I/O at a regular pace, an observation which motivated, for example, the work on FTIO (see Section 3.5). The parameters we have retained to build this simple proxy application are the following:

- **IO burst size**: size written at each burst;
- **IO period**: time between each burst
- **IO itterations**: number of iterations to be done
- **R/W probability**: the probability of transitioning from read to write (and conversely)

These configurations were gathered in a benchmark application called *ioskel*[7]. This application has the peculiarity of configuring itself from the binary name. As such, as the proxy gathers commands by name

[7] https://github.com/besnardjb/ioskel



(a) Write bandwidth.

(b) Read bandwidth

Figure 3.8: Measurements traces shown by the proxy for `ioskel.4.536870912.15.100` writing then reading 512 MB every 4 seconds for 15 iterations.

to generate models, the various behaviours are correctly aggregated for a given configuration. Two naming patterns are supported:

- **ioskel.PERIOD.SIZE.ITER**: a write-only benchmark, writing SIZE every PERIOD time for ITER iterations;
- **ioskel.PERIOD.SIZE.ITER.PROBA**: a read/write benchmark with a PROBA probability of transition, manipulating SIZE every PERIOD time for ITER iterations.

Thanks to this simple naming convention, it is possible to generate several configurations by simply creating symbolic links to the same binary with various names accounting for a wide range of configurations. This has the effect of creating a parametric analysis for the various I/O patterns. We used this benchmark to generate models representing a wide range of behaviours, and this guided the modelling and malleability decision in the ADMIRE project. In addition, each of these configurations is run at various scales to study the variation as a function of the number of processes – the main parameter guiding malleability.

Figure 3.8 presents the traces of `ioskel` alternating between read and write. It can be seen that every 8 seconds, we have a spike of write bandwidth (see Figure 3.8a) in opposition to a phase with the read operations (see 3.8b). In practice, we then have, per process, a 512MB burst in write followed by a burst in read for 15 iterations. This leads to a total write size of eight peaks of 512 MB, i.e., 4096 MB in total. Conversely, for the read operations, we have 7 peaks of 512 MB and a total read size of 3584 MB.

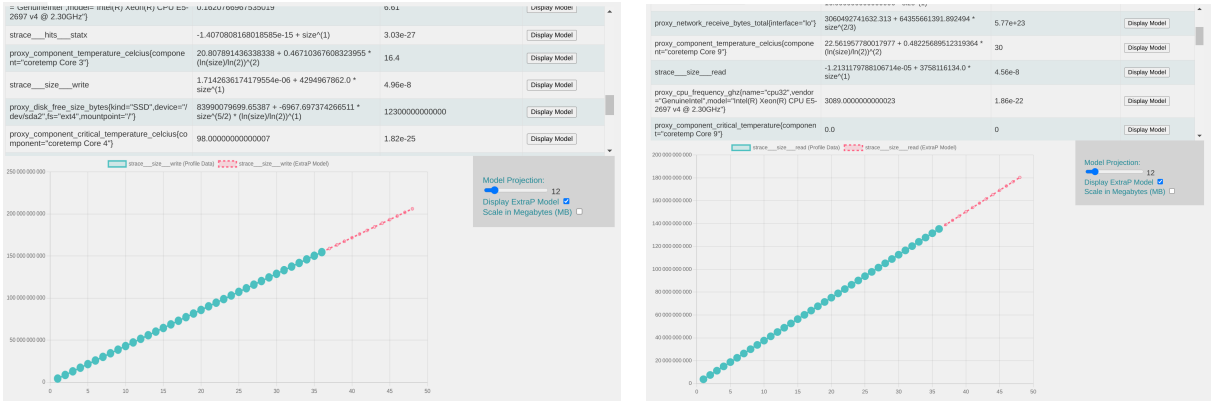
As observed in Figure 3.9, we can see in the top part of the screenshots the following equations for total written (Figure 3.9a) and read bytes (Figure 3.9b):

$$write = 1.714 * 10^{-6} + 4294967862 * size$$
$$read = -1.213 * 10^{-5} + 3758116134 * size$$

Both equations are affine functions, it shows that Extra-P did extract correctly the linear increase of the write/read sizes as the number of processes increases. If we ignore the very small constant, Extra-P parameterised the operations as follows:

$$write = \frac{4294967862.0}{1024 * 1024} = 4096.000539\,\text{MB per process}$$

$$read = \frac{3758116134.0}{1024 * 1024} = 3584.018835\,\text{MB per process}$$



(a) Extra-P model for write operations.

(b) Extra-P model for read operations.

Figure 3.9: Extra-P models displayed by the metric proxy for the execution of the application `ioskel.4.536870912.15.100`.

As outlined by the very small *RSS* (third column in the top table of Figure 3.9) for read and write bytes, these values are spot on with the behaviour we expected through the empirical evaluation done for the traces in the previous paragraphs. This shows that Extra-P is capable of detecting such patterns easily. These linear equations are for Extra-P one of the simplest case. However, in terms of total size applications are often "boring" as they tend to be symmetric in their use of I/Os, thus leading to such linear response in terms of total sizes. One example of such a pattern is the program check-pointing its results every n^{th} iteration, writing a fixed amount of data over time.

3.4.3 Darshan Support

To provide even further support for profiles collected outside the project, an extension was developed that parses several Darshan [14, 43] profiles and traces (DXT) into a single file that can be read by Extra-P. Consequently, Extra-P can generate performance models that show how the I/O performance of an application scales aside from the remaining phases (compute/communication). The essential advantage of this approach is that it provides universal I/O model generation capabilities to Extra-P from existing profiles. As later discussed in Section 3.5 and demonstrated in Section 3.5.7, FTIO also supports these traces, as this parsing represents the back-end of FTIO.

In the scope of the IO-SEA[8], HPC I/O traces were gathered from the *I/O Trace Initiative* website[9] [30] which has been an on-going collaboration between several EuroHPC projects led by JGU in the ADMIRE and IO-SEA projects. We download all NEK5000 Darshan profiles from this website. Next, from the FTIO repository[10], we parsed the Darshan files into a single Extra-P supported profile:

```
$> ioparse *.darshan
```

Executing this command generates the file *scale.jsonl* in the current working directory, which has the following structure:

```
{"params":{"Processes":1024},"callpath":"read_sync->total_bytes","metric":"Bytes","value":1.177825e+08 }
{"params":{"Processes":2048},"callpath":"read_sync->total_bytes","metric":"Bytes","value":1.177988e+08 }
{"params":{"Processes":4096},"callpath":"read_sync->total_bytes","metric":"Bytes","value":1.178320e+08 }
{"params":{"Processes":8192},"callpath":"read_sync->total_bytes","metric":"Bytes","value":1.178996e+08 }
{"params":{"Processes":1024},"callpath":"read_sync->max_bytes_per_rank","metric":"Bytes","value":1.152250e+05 }
{"params":{"Processes":2048},"callpath":"read_sync->max_bytes_per_rank","metric":"Bytes","value":5.775300e+04 }
{"params":{"Processes":4096},"callpath":"read_sync->max_bytes_per_rank","metric":"Bytes","value":2.901800e+04 }
{"params":{"Processes":8192},"callpath":"read_sync->max_bytes_per_rank","metric":"Bytes","value":1.461800e+04 }
{"params":{"Processes":1024},"callpath":"read_sync->max_transfersize_over_ranks","metric":"Bytes","value":1.152250e+05}
⋮                                                                                                                    ⋮
```

[8] https://iosea-project.eu/

[9] https://hpcioanalysis.zdv.uni-mainz.de/

[10] https://github.com/tuda-parallel/FTIO

```
{"params":{"Processes":2048},"callpath":"io_time->async_read_t","metric":"Time","value":0.000000e+00 }
{"params":{"Processes":4096},"callpath":"io_time->async_read_t","metric":"Time","value":0.000000e+00 }
{"params":{"Processes":8192},"callpath":"io_time->async_read_t","metric":"Time","value":0.000000e+00 }
```


This file can be easily imported into Extra-P as it supports the structure mentioned in Section 3.3. To import it directly into the GUI, we simply execute:

```
$> extrap-gui --json  scale.jsonl
```

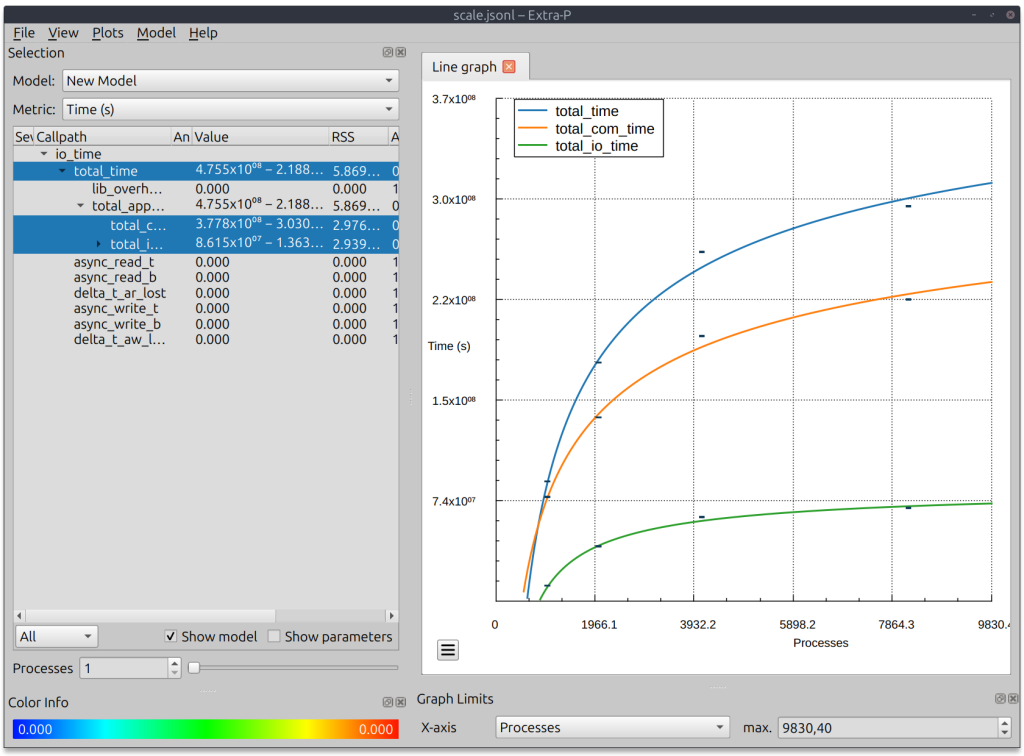


Figure 3.10: Extra-P models for the total time, the I/O time and the compute time.

As Darshan is widely used in the HPC community, by parsing the profiles we added the ability to generate performance models easily through Extra-P. Aside from the number of hits and the total bytes transferred, we can model the total time which can be divided in to the total I/O time and the total compute time (including communication) as shown in Figure 3.10.

3.4.4 TMIO and Extra-P: Asynchronous I/O Requirements

As mentioned in Section 2.5, TMIO provides an offline (using the `LD_PRELOAD` mechanism) and an online mode. To generate performance models with Extra-P, TMIO can be used in both modes, as the models are generated post-run. In particular, the generated traces with TMIO for the different configurations can be merged into a single profile similar to Section 3.4.3. Thus, it makes more sense to use the offline mode with Extra-P, as there are currently no gained benefits in using the online mode. Thus, TMIO provides an easy-to-use tool without any code modifications and minimal overhead.

One of the important metrics to model is the I/O bandwidth. While the approach handled so far can easily achieve this (i.e., the bandwidth is among the gathered metrics, and consequently can be modelled), this becomes especially interesting for asynchronous I/O. TMIO was developed with a focus on asynchronous I/O. As asynchronous I/O is usually done during (i.e., behind) the computational phases, TMIO finds the bandwidth required to perform the I/O *completely* in the background of the compute phases. Consequently, if the system provides this bandwidth to the application, the application would perform its asynchronous I/O entirely in the background of the compute phases without wasting time at the matching blocking I/O calls like `MPI_Test` and `MPI_Wait` or any of its derivatives. Note that the MPI standard demands matching pairs for asynchronous

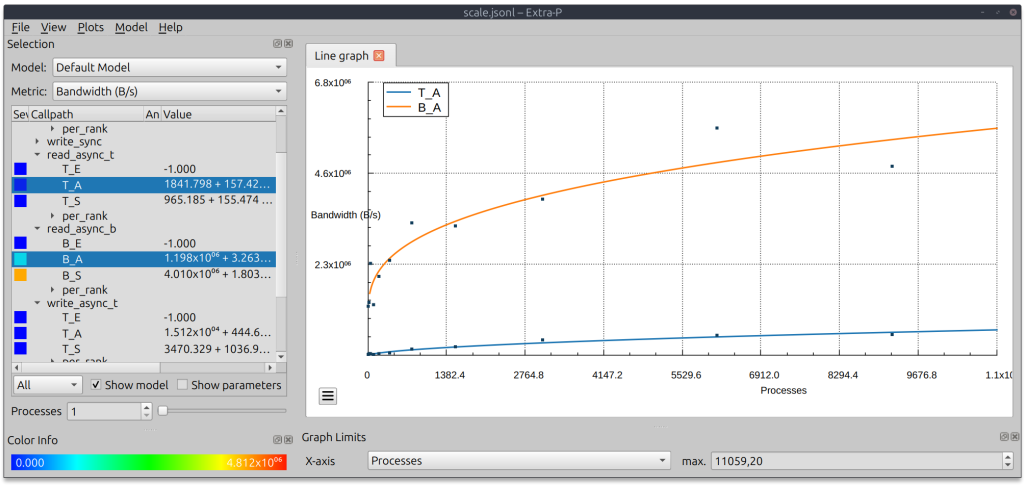


Figure 3.11: Scaling behaviour of the required reading bandwidth B_A versus the actual reading bandwidth T_A.

I/O; a separate request complete call (`MPI_Wait`, `MPI_Test`, or any of their variants) is needed to complete the I/O request [31, Ch. 14.2] and thus confirm that the data has been read or written.

To demonstrate this aspect, we modified the HACC-IO [28] benchmark to perform asynchronous I/O. HACC-IO mimics an I/O phase of HACC (Hybrid/Hardware Accelerated Cosmology Code) [21]. From an abstract view, HACC-IO fills arrays of different types with the current index of a for loop, which iterates over the number of particles. Next, a header (containing metadata information) and the arrays are written to a file. Finally, the file's contents are read again and compared against the values of the variables which are still in memory. We classified the application into four blocks in the same order as described above: compute, write, read, and verify. For our purposes, we added a for loop around these blocks to execute them several times. Moreover, we used the MPI-IO version of HACC-IO to write using an individual file pointer to distinct files. In the standard version, HACC-IO uses non-collective blocking I/O routines with explicit offset (`MPI_File_write_at` and `MPI_File_read_at`), which we replaced with matching non-collective non-blocking I/O routines (`MPI_File_iwrite_at` and `MPI_File_iread_at`). Moreover, we adjusted the code such that the read/write occurs asynchronously with the compute/verify blocks. To avoid data races between the read and write blocks, we used wait blocks (`MPI_Wait`) at the end of the compute and verify blocks. Additionally, to make the data from one compute block available to the verify block of the same phase, we create a copy of the data using memcpy just before the wait block. Finally, we added global broadcast operations during the compute and verify phases, to add more variability to these phases.

With TMIO preloaded, we executed HACC-IO with 10 loops and 10^6 particles per rank on the Lichtenberg cluster with a varying rank configuration. Figure 3.11 shows the scaling behaviour of the actual bandwidth (denoted as T) versus the required (bandwidth denoted as B). The index "A" stands for the average. Thus, during the execution of the application, TMIO gathered both the required and the actual bandwidth at the rank level. Next, the rank-level metrics were aggregated to obtain the application-level metrics. TMIO can either perform this during `MPI_Finalize`, or this is handled by the parsing script that merges the different traces into a single profile for Extra-P. For this example, the latter option was the case to decrease the overhead of TMIO on the application. Consequently, the metrics shown in Figure 3.11 represent application-level values. As observed, as the required reading bandwidth is higher than the actual one, the application is blocked during the asynchronous reading.

Figure 3.12 shows the scaling behaviour of the application versus the number of processes. The top part of Figure 3.12 shows the total compute and I/O time. As observed, the I/O time increases much slower compared to the compute time with an increasing number of ranks. The bottom part shows that with an increasing number of ranks, the application increasingly spends more time waiting for the asynchronous read to finish. This is because the verify plus the memcpy blocks consume more time than the compute block. Thus, the writing phase has a longer duration enabling the application to hide it completely behind the verify and memcpy blocks.

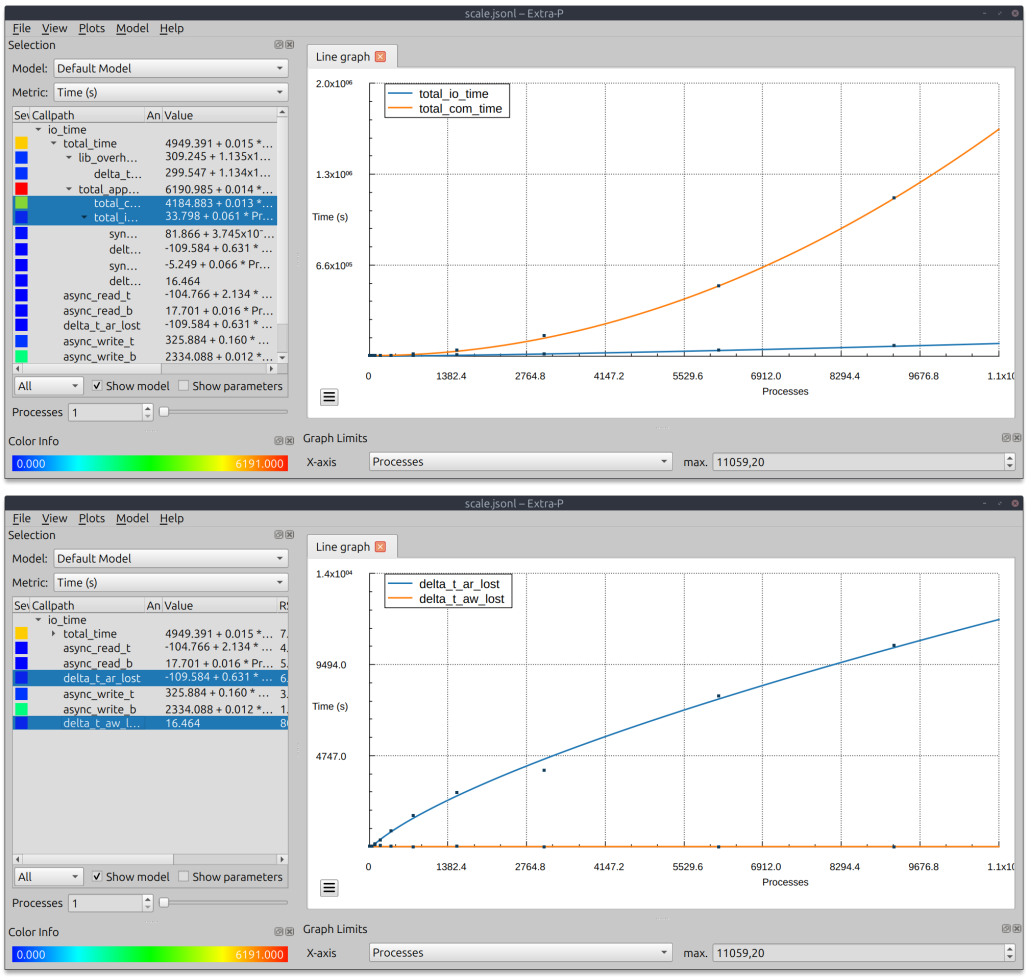


Figure 3.12: Scaling behaviour of the application time versus the number of MPI ranks. The top part shows the total compute and I/O time, while the bottom part shows the time spent by the application waiting (i.e., *lost*) for the asynchronous read (t_{ar}) and write (t_{aw}) to finish.

In contrast, the compute block is very short, forcing the read phase to be visible by blocking the application till its completion.

3.5 FTIO

As the behaviour of an application can vary during execution, it can be beneficial to generate models that describe the current behaviour rather than to have post-execution performance models or complicated model descriptions. In particular, while performance models describe the whole scaling behaviour of an application, knowing when the different phases of an application occur can be very beneficial, for example, for contention avoidance algorithms like I/O scheduling [10, 16, 19, 52] and burst buffer management [3, 4]. This is where FTIO comes in, as described in this section. An overview of FTIO is provided in Figure 3.13.

We start by explaining the difficulty of identifying I/O phases in Section 3.5.1. In Section 3.5.2, we describe the importance of the period of the I/O phases. FTIO treats the I/O bandwidth over time as a *signal*. Afterwards, the signal is discretised and then analysed using the discrete Fourier transform as explained in Section 3.5.3. To find the period of the I/O phase, we use outlier detection methods as described in Section 3.5.4. We provide metrics that gauge the confidence in the results in Section 3.5.5. Finally, we explain how to predict the period of the I/O phases of an application at runtime using FTIO in Section 3.5.6.

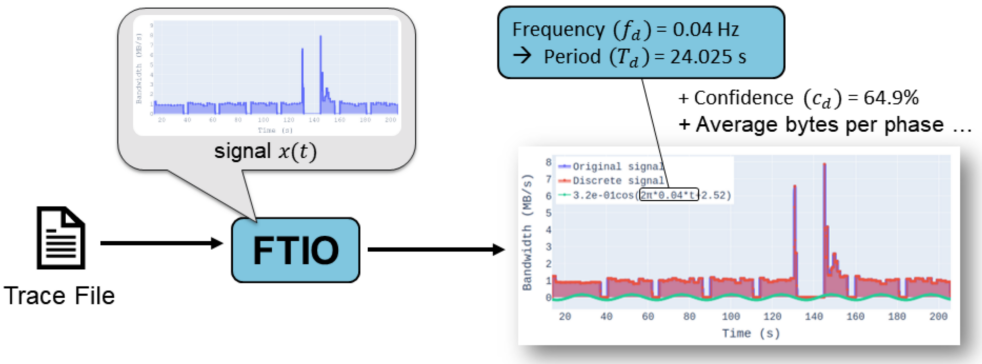


Figure 3.13: Overview of FTIO. FTIO requires a trace file containing the bandwidth as an input and provides the frequency and its confidence as an output. Further metrics that build on these results are also provided.

3.5.1 The Challenge of Finding I/O Phases

In HPC, large scientific applications often alternate between I/O and compute phases [15]. Periodic behaviours often describe the I/O phases due to repeated actions like checkpointing [20]. On the other hand, due to the nature of I/O being a shared resource on HPC systems, long file system access, I/O congestion, as well as I/O bursts are often encountered. While several research works have been dedicated to reducing these problems by providing, for example, burst buffers to flatten the I/O [48] or I/O scheduling [11, 17], they still require information about the I/O behaviour, which is often not easy to acquire. This is also the case in the ADMIRE project, where designing an active I/O stack requires quick responses, which by predicting the I/O behaviour, can be made.

Several tools have been devoted to aggregating I/O metrics (.e.g., Darshan). However, aggregated metrics do not properly represent the *temporal* behaviour of applications [51]. Since I/O tends to be bursty and periodic, knowing how many bytes are accessed does not paint the whole picture. We need to know *when* (or rather *how often*) these accesses happen: two applications that write each 1 TB over 2 hours, one with a single I/O phase at the end of the execution and the other one with multiple I/O phases every 2 minutes, impose very different loads on the system. This has motivated us, as mentioned in Chapter 2, to extend our monitoring tool (metric proxy) and develop another one that solely focuses on this aspect (TMIO) to be as lightweight and fast as possible.

Finding the I/O phase and, in particular, the period of these phases can be a challenging task in the time domain. The HPC I/O stack only sees a stream of issued requests and does not provide I/O behaviour characterisation. Thus, the notion of an *I/O phase* is often purely logical, as it may consist of a set of independent I/O requests issued by one or more processes and threads during a particular time window, and popular APIs do not require that applications explicitly group them. Consequently, a major challenge is to draw the borders of an *I/O phase*.

If we consider, for example, an application with 10 processes that write 10 GB by generating a sequence of two 512 MB write requests per process, then performs computation and communication for a certain amount of time, after which it writes again 10 GB. How do we assert that the first 20 requests correspond to the first I/O phase and the last 20 to a second one? An intuitive approach is to compare the time between consecutive requests with a given threshold to determine whether they belong to the same phase. Naturally, the suitable threshold should depend on the system. The reading or writing method can make this an even more complex challenge, as accesses can occur, e.g., during computational phases in the absence of barriers. Hence, the threshold would not only be *system dependent* but also *application dependent*, making this intuitive approach more complicated than initially expected. Consequently, there is a need for a tool that first correctly identifies the I/O phase without using such thresholds and then predicts their future occurrences.

3.5.2 Temporal I/O Behaviour

Often one might desire a detailed description of I/O activity over time. However, finding an extremely precise profile comes at the cost of a higher overhead, both in terms of measurement and data accumulation. Moreover, a detailed time model can be hard to explore a contention-avoidance algorithm that is lightweight enough to be used in practice, especially as models with high predictive accuracy are often *black box* and cannot be interpreted directly for explaining I/O performance [24]. Hence, depending on the use case, models at higher abstraction levels might be tolerated, which can be easier to interpret and often to generate, especially online.

Recent work on I/O scheduling [10, 16, 25] has shown that knowledge of periodic I/O patterns, even when not *perfectly precise*, leads to good contention avoidance. Consequently, one approach could be to predict the *period of the I/O phases* during runtime and provide this information to such approaches rather than finding detailed time models. This is where our newly developed tool FTIO [46] comes in, which is publicly available on GitHub: `https://github.com/tuda-parallel/FTIO`.

With FTIO, we focus on predicting the *I/O phases* rather than the I/O requests. As the name suggests, FTIO characterises the behaviour of an HPC application using well-known frequency techniques from signal processing. In particular, rather than examining the behaviour of an application in the time domain, FTIO examines its behaviour in the frequency domain. Thus, we move away from detailed modelling and focus on a simple metric: The *period* of the I/O phases. Consequently, the provided metric presents a trade-off between aggregated information and a detailed time model.

All that FTIO requires is the application bandwidth over time, which we denote as $x(t)$. If rank-level metrics are provided to FTIO, the tool internally calculates the application-level bandwidth by overlapping the rank-level metrics. Different tools exist that can provide this metric. TMIO and the metric proxy are examples of these. Darshan and recorder traces are also supported. However, for the online version of FTIO, only TMIO and the metric proxy are currently suitable, as later explained in Section 3.5.6.

Since we focus on I/O phases, by applying DFT on the application-level signal, we overcome the challenges mentioned in Section 3.5.2. Compared to time analysis, frequency techniques, such as DFT, decompose a signal into its frequency components, giving us control over the *relevant* I/O. That is, through the parameters of the DFT, we can control the granularity of the captured behaviour, as will be described in the next section.

3.5.3 Discrete Fourier Transformation

Fourier analysis has a wide range of applications, including signal and image processing, analog signal design, physics (optics, astronomy, etc.), and many more. In essence, it decomposes a signal into its frequency components such that their sum allows for reconstructing the signal. While the term ”Fourier transform” usually refers to the continuous one, which deals with continuous signals, the discrete Fourier transform DFT works with discrete samples of the signal in the time domain. Here, we use the latter one and discuss how we apply it and profit from its unique properties.

FTIO treats the bandwidth over time $x(t)$ as a continuous signal. As a first step, FTIO performs the discrete Fourier transformation on $x(t)$. Thus FTIO first needs to discretise the continuous signal by sampling it with a s constant sampling rate T_s in a time window Δt, to obtain $N = \Delta t \cdot f_s$ samples:

$$\{x_n = x(n/f_s) \quad | \quad n \in [0, N)\} \tag{3.1}$$

The evenly spaced sequence x_n is then transformed from the time domain into a sequence X_k of the same size in the frequency domain:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi kn}{N} i} \tag{3.2}$$

Where the bins $k \in [0, N)$ correspond to the frequencies: $f_k = \frac{k}{N} f_s = \frac{k}{\Delta t}$. Following the terminology of signal analysis, we refer to $X_{k=0}$ as the *DC* offset (or DC value), $f_{k=1}$ the fundamental frequency of DFT, and $f_{k>1}$ its harmonics.

Since the I/O signals is always *real*, x_n consists of purely real values, and thus for $k \in [1, N)$:

$$X_k = X^*_{(N-k)},$$

such that X_k^* denotes the complex conjugated of X_k. Thus, the signal is fully contained with the first $1 + \frac{N}{2}$ values of X_k ($k \in \{0, \cdots, N/2\}$). Moreover, the highest captured frequency for the analysis is $\frac{f_s}{2}$, which directly shows the well-known Nyquist theorem.

Consequently, when plotting the amplitude X_k against the frequencies f_k (i.e., in the frequency domain), only half of the spectrum (*single sided spectrum*) needs to be inspected. In this case, the amplitude of symmetric signals (around $\frac{N}{2}$) needs to be multiplied by two. Thus, only half of the frequencies are needed to reconstruct the original signal with the inverse DFT (IDFT):

$$x_n = \frac{1}{N}\left(X_0 + \sum_{k=1}^{\frac{N}{2}} 2|X_k| \cos\left(\frac{2\pi kn}{N} + arg(X_k)\right)\right) \tag{3.3}$$

with the amplitude $|X_k|$ and the phase $arg(X_k)$.

$$|X_k| = \sqrt{\mathrm{Re}(X_k)^2 + \mathrm{Im}(X_k)^2}$$
$$arg(X_k) = atan2(\mathrm{Im}(X_k), \mathrm{Re}(X_k))$$

This reduces the calculation needed for the reconstruction of the signal and limits the constituting signals to cosine waves only, simplifying the interpretation of results. Moreover, the DC offset X_0 is expected to be among the highest components as the I/O data transferred is always a positive number of bytes, and the cosine waves obtained with DFT must be shifted upwards.

As I/O exhibits high variability and is often affected by noise, a more robust approach is to replace the frequency (i.e., amplitude) spectrum with the *power* spectrum:

$$p_k = \frac{1}{N} X_k^2$$

Thus, by taking the square of the amplitude, frequency components with higher contributions can be easily spotted. To find the dominant frequency f_d that describes the frequency of the I/O phases, a simple approach would be to find the component with the highest contribution while excluding the DC offset from the analysis. The corresponding frequency of this component is then simply f_d. However, this is not enough, as there might be frequency candidates with similar high contributions, which is often the case for non-periodic signals. Thus, f_d must not only have the highest contribution, but it must also be an *outlier*. This is explored next.

3.5.4 Outlier Detection

FTIO offers different outlier detection methods, including the Z-score [26], DBSCAN, isolation forest, local outlier factor, and the find_peaks algorithm from SciPy. By default, FTIO favours easier calculation and thus uses the Z-score, which reveals how many standard deviations σ a power p_k is from the mean $\bar{p}$ of all powers:

$$z_k = \frac{|p_k| - |\bar{p}|}{\sigma} \tag{3.4}$$

A Z-score above three is usually a direct indication of an outlier. However, for our purpose, just finding an outlier is not representative enough to find the dominant frequency; it must be at the same, dominating the behaviour of the signal. That is, it must be distinct from the remaining outliers. For that, we introduce a tolerance value of 80% (that can be modified) multiplied by the maximum power. Consequently, dominant frequency candidates satisfy the following equation:

$$\mathcal{D}_f = \{f_k | z_k \geq 3 \text{ and } z_k / z_{\max} \geq 0.8\} \tag{3.5}$$

Depending on the number of dominant frequency candidates $\mathcal{D}_f$, we distinguish between three cases: If only a single frequency f_k satisfies Eq. (3.5), the corresponding frequency is the dominant frequency f_d, and we have a high confidence c_k in the obtained results. If two frequencies satisfy Eq. (3.5), i.e., $\mathcal{D}_f = \{f_{k_1}, f_{k_2}\}$ (two candidate frequencies), the one with the higher power is the dominant frequency. Moreover, the signal

has some variation in its behaviour but is still periodic, and we thus have a moderate confidence c_k in the results. Finally, if more than two frequencies satisfy the equation, the signal is most likely not periodic. There is an exception when the candidates are multiples of two of each other. In this case, the higher frequencies are ignored. The presence of this kind of harmonics with decreasing high contributions is an indication that there are periodic I/O bursts in the signal as will be demonstrated in Section 3.5.7. Note that we only described the approach briefly, as several more steps are involved in the calculation of c_k and extraction of f_d. The detailed approach has been described in [46].

3.5.5 Confidence Metrics

The gauge the results from FTIO, we provide confidence metrics. That is, as long as Eq. (3.5) returns only two dominant frequency candidates, we provide the c_k in f_k. Let us denote $\mathcal{I}_1 = \{i \mid z_i \geq 3\}$ as the set of frequencies that are outliers, and $\mathcal{I}_2 = \{i \mid z_i/z_{\max} \geq 0.8\}$ as the set of frequencies whose Z-score is within 80% of the maximum Z-score, then the confidence c_k of the frequency f_k is:

$$c_k = \frac{1}{2}\left(\frac{z_k}{\sum_{i\in\mathcal{I}_1} z_i} + \frac{z_k}{\sum_{i\in\mathcal{I}_2} z_i}\right)$$

Thus, c_d is the confidence of the dominant frequency f_d. To further refine the confidence metric, we optionally provide a second method that does not rely on the result from DFT, namely autocorrelation. The results from both analyses are merged in the final step of FTIO. However, we skip this explanation, as it has been described in detail in [46]. To sum up, an overview of FTIO's algorithm (Sections 3.5.3 to 3.5.5) is provided in Figure 3.14.

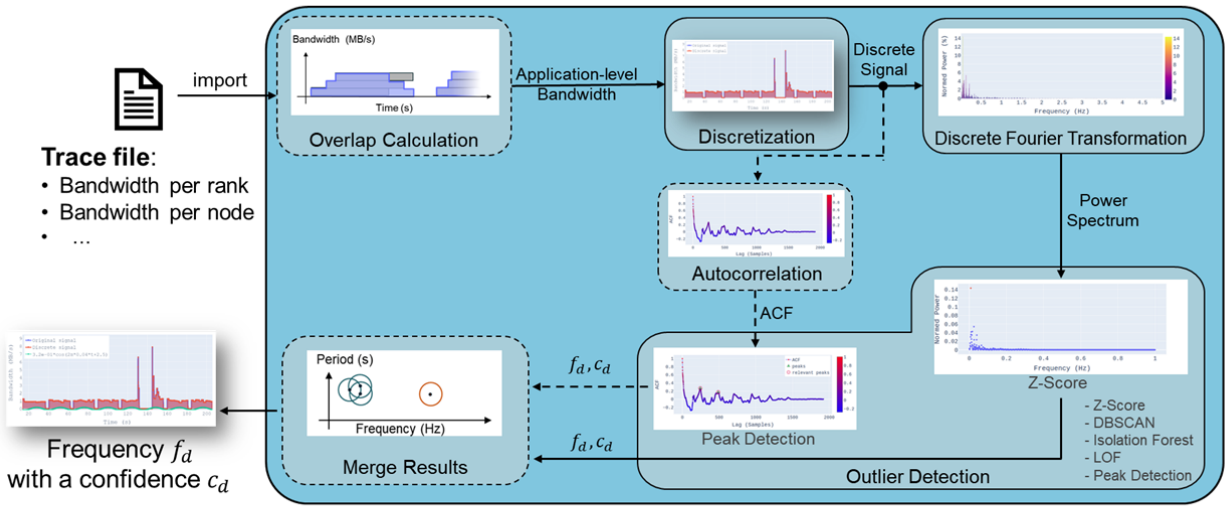


Figure 3.14: Internal FTIO architecture. The dashed lines indicate optional steps. FTIO internally calculates application-level bandwidth. Afterwards, the dominant frequency is found using DFT and outlier detection. If autocorrelation is additionally used, the results are merged with the results from DFT to refine the confidence.

3.5.6 Online Approach: Predicting the Frequency of the I/O Phases at Runtime

FTIO can be executed offline (detection) and online (prediction) period analysis. In the offline mode, FTIO reads post-simulation profiles or traces to detect the period of the I/O phases. Various existing tools, including Recorder [49], Darshan [14,43], as well as newly developed tools like TMIO (Section 2.5) and the metric proxy (Section 2.1) are supported. In particular, the online mode of TMIO was developed for this purpose. Since D5.3, the metric proxy has been extended to yield profiles with a high sampling rate for FTIO as described in Section 2.1. Consequently, two tools can provide $x(t)$ to FTIO to obtain predictions regarding the period of the I/O phases during an application's runtime.

For the online mode, FTIO monitors a tracing file as shown in Figure 3.15. Whenever this tracing file is modified (e.g., TMIO appends new traces), FTIO launches a new child process that predicts the period of the I/O phases. Once the processes perform the prediction, the results are merged into a shared memory space

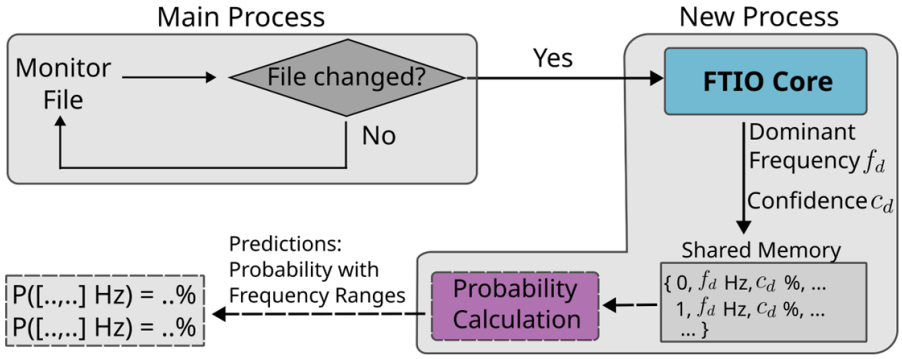


Figure 3.15: Online approach of FTIO. A file is monitored for changes with FTIO. Whenever new traces are appended to the file (e.g., through TMIO in the online mode), a new prediction is executed in a new process.

between them. Consequently, based on the predictions collected so far, the results are evaluated, and the periods are optionally provided in probability intervals. This represents the first step we took to tackle the problem of changing I/O behaviour and variability. Another optional enhancement is to disregard old results at some point and consider a shorter time window for the analysis. Using online time window adaptation, FTIO can, for example, after finding k times a dominant frequency, reduce the time window for evaluation to k times the last found period. Alternatively, one could specify a fixed length or a fixed k. Another enhancement is to provide frequency ranges for the prediction alongside their probability.

We demonstrate the applicability of the online approach of FTIO with an I/O scheduling use case in Section 4.3. This experiment uses the IO-Sets [11] implementation from WP4. In particular, further information about this experiment can be found in [6] and about FTIO in [46]. Furthermore, the data set [47] is also online for the reproducibility of the results.

3.5.7 Examples

In the following, we demonstrate FTIO based on two examples: IOR and NEK5000. The first example demonstrates the applicability of the tool to a large application. Moreover, we show that FTIO allows us to detect I/O bursts in the signal. The second example shows the compatibility of FTIO with profiling and monitoring tools outside the scope of ADMIRE to reach a broader audience.

Example 1: IOR FTIO cannot only find the dominant frequency (if any) in a signal, but it can also detect I/O bursts. To demonstrate this, we executed the IOR benchmark [7] on the Lichtenberg cluster with 7680 ranks. We set up IOR with 8 iterations, 2 segments, a transfer size of 2 MB, and a block size of 10 MB with the MPI-IO API in the parallel mode and TMIO preloaded. After execution, FTIO on the result for the entire time window $\Delta t = 563.11$ s (i.e., from 65.17 s to 628.27 s) with a sampling frequency $f_s = 10$ Hz (default). Thus, we executed the following call:

```
$> ftio 7680.jsonl -re -v -e mat
```

FTIO then performs DFT followed by the Z-score. For this example, FTIO found a dominant frequency at $f_d = 1.24 * 10^{-02}$ Hz, corresponding to a period of $T_d = 80.442$ s. Moreover, FTIO provides a confidence of $c_k = 62.05\%$ in the results. The normalised power spectrum is shown in Figure 3.16.

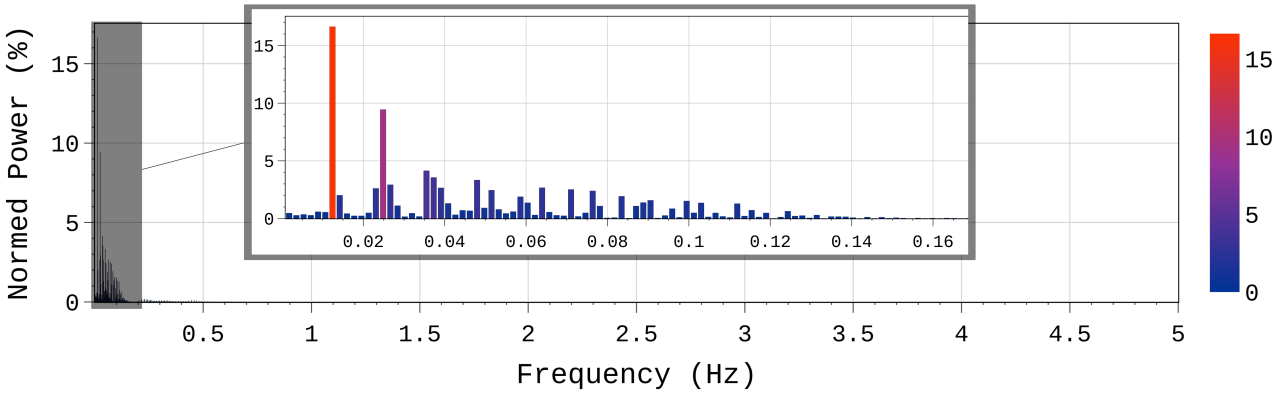


Figure 3.16: normalised single-sided power spectrum obtained from FTIO on IOR with 7680 ranks. The red bar at $1.2 * 10^{-02}$ Hz has the highest contribution and represents the dominant frequency in the signal. The next highest contribution is from the frequency at $2.49 * 10^{-2}$ Hz coloured in purple.

As observed in Figure 3.16, the frequency at $1.2 * 10^{-02}$ Hz has the highest contribution to the power. Consequently, with Z-score as the default method, this frequency was detected as an outlier, as shown in Figure 3.17.

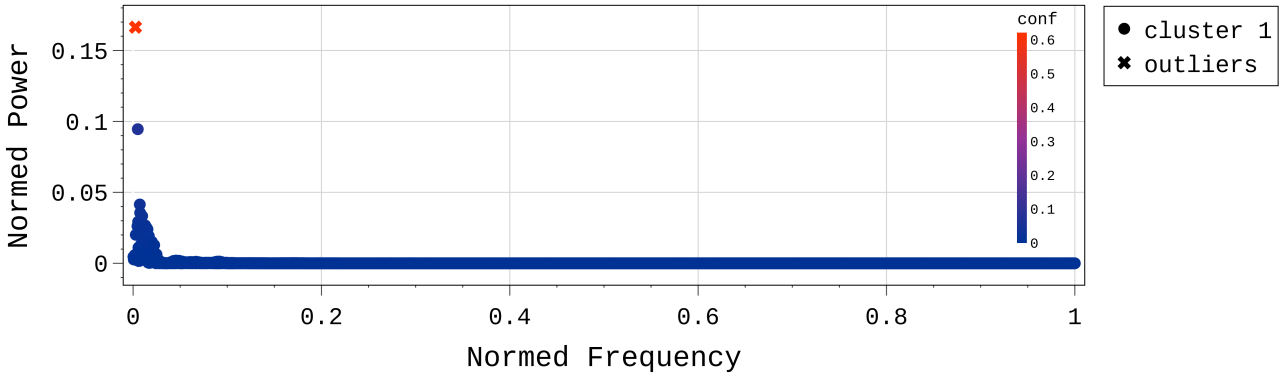


Figure 3.17: Result of the Z-score on IOR with 7680 ranks. As indicated by the red "x" marker, FTIO just detected a single frequency is an outlier with a high confidence. This outlier has, at the same time, the highest contribution (see Figure 3.16). The red colour in this figure indicates that the outlier has also a moderate confidence.

Rather than displaying the plots with Plotly, FTIO allows selecting also Matplotlib as a plot engine. This brings the advantage that the plots are fast generated in case a lot of sample points need to be displayed. For this, we execute FTIO with the `-e mat` flag. The original signal $x(t)$, the discretized signal x_n, and the cosine wave of the dominant frequency are observed in Figure 3.18. Note that, as observed in Eq. (3.3), the dominant

frequency (i.e., the cosine wave) just represents one of the frequencies needed to reconstruct the signal using the IDFT. Consequently, the more frequencies are used, the more accurate the reconstructed signal becomes as we later see in this example.

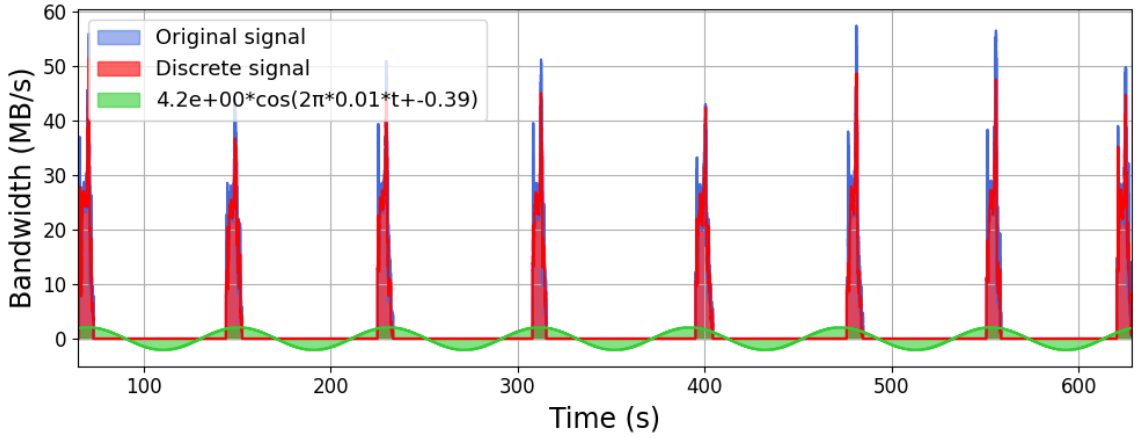


Figure 3.18: Temporal behaviour of IOR with 7680 ranks execute don the Lichtenberg cluster. The cosine wave containing the dominant frequency is shown in green. As observed, the I/O phases align with the drawn cosine wave.

As mentioned in Section 3.5.3, FTIO excludes the DC offset from the analysis. Consequently, as shown in Figures 3.16 and 3.17, the second highest frequency (while ignoring the DC offset) is at $2.49 * 10^{-2}$ Hz. However, this frequency is a harmonic (multiple of two) of the dominant frequency $f_d = 1.243 * 10^{-2}$. According to Section 3.5.4, such frequencies indicate that the signal contains I/O bursts. To demonstrate this aspect, FTIO offers a flag that allows plotting up to the top 10 frequencies presented in the signal $x(t)$. For this, we pass the `-re` flag:

```
$> ftio 7680.jsonl -re -e mat
```

This plots the reconstructed signal using the IDFT up to the number of specified frequencies. In Figure 3.19, we plotted the reconstructed signal with up to 10 frequencies. The top part of the figure shows the reconstructed signal from the frequency $f_d = 1.24 * 10^{-2}$ Hz plus the DC offset. The second figure from the top, which is denoted with "Recon. top 3", shows the reconstructed signal the same as previously plus the contribution of the frequency at $2.49 * 10^{-2}$ Hz and so on. As observed, the more frequencies are included, the closer the reconstructed signal becomes to the original one. Moreover, the highest contributing frequencies are almost all harmonics of $f_d = 1.24 * 10^{-2}$ Hz, indicating that this constitution allows detecting I/O bursts in the signal. This aspect will be further explored in the future.

Example 2: NEK5000 In the second example, we want to highlight the compatibility of FTIO with other tools that provide I/O traces. As mentioned, and like TMIO, FTIO offers online (prediction) and offline (detection) modes. These methods depend on whether the application is traced online or offline traces are used. Moreover, similar to Extra-P, to reach a broader audience, FTIO offline mode supports Darshan traces as mentioned in Section 3.4.3. From the *I/O Trace Initiative* website [30][11], we download the Darshan file[12] (renamed to 1024.darshan) of NEK5000 which was executed on the MOGON II cluster with 1024 ranks. Next, we pass this file to FTIO via:

```
$> ftio 1024.darshan
```

[11] https://hpcioanalysis.zdv.uni-mainz.de/
[12] https://hpcioanalysis.zdv.uni-mainz.de/trace/64ec79b34d5c25a42acabc94

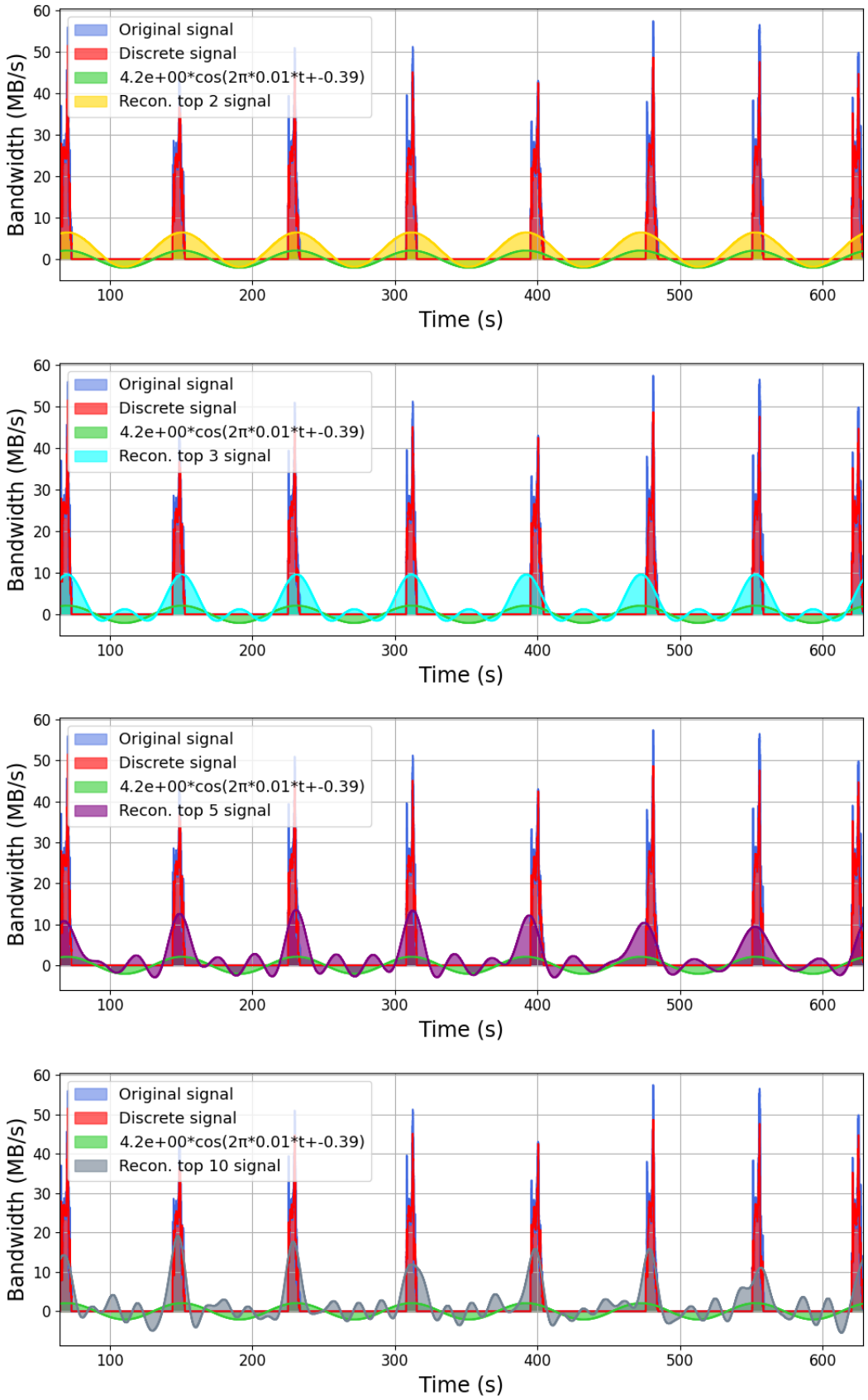


Figure 3.19: Temporal behaviour of IOR with 7680 ranks drawn alongside the reconstructed signal with a varying number of frequencies.

This executes FTIO with the default outlier detection method (Z-score). FTIO discretized the signal of length 86016.0 s with a sampling frequency f_s of $6.1 * 10^{-3}$ Hz. As the trace file provides bins, FTIO automatically adjusted the sampling frequency f_s to the needed value. Consequently, $N = 525$ samples were collected in 0.006 s. Next, DFT is executed. The single-sided power spectrum is shown in Figure 3.20.

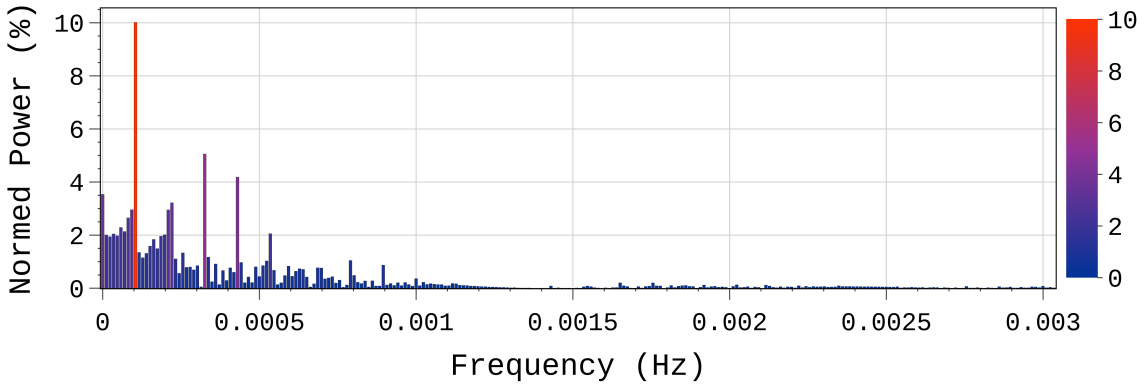


Figure 3.20: Single-sided normalised power spectrum obtained using FTIO on NEK5000 with 1024 ranks. In this figure, we additionally show the DC offset at 0 Hz.

As observed in Figure 3.20, the frequency at nearly 0.0001 Hz has the highest contribution compared to the remaining ones. In fact, FTIO spotted 4 frequencies with a Z-score higher than 3. From these frequencies, only a single one satisfies Eq. (3.5) and is consequently the dominant frequency. Moreover, FTIO provides a confidence of 72.96% in the found value. In Figure 3.21, the time behaviour of the signal as well as the result from FTIO are illustrated. FTIO found the dominant frequency at $f_d = 1.05 * 10^{-4}$, which is plotted in green in this figure.

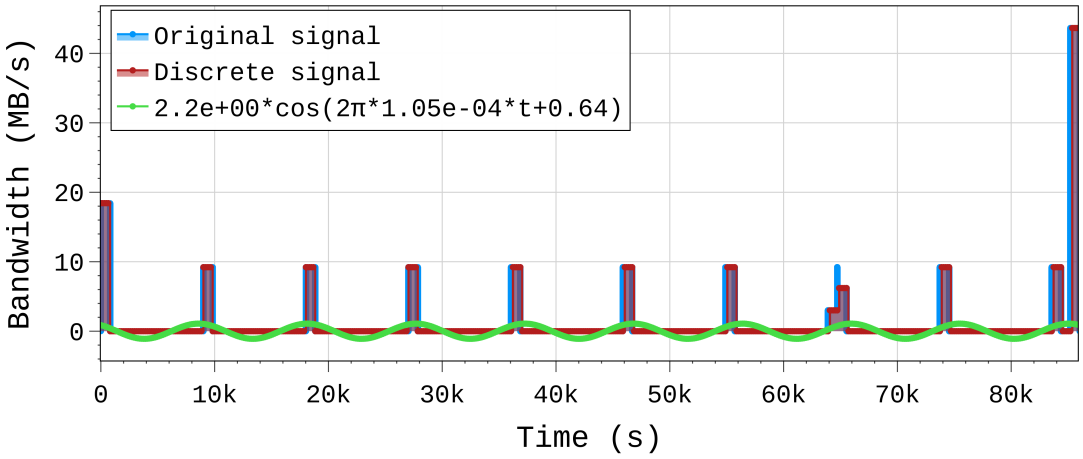


Figure 3.21: Result of FTIO on the Darshan profile of NEK5000 with 1024 ranks. The figure shows the dominant frequency $f_d = 1.05 * 10^{-4}$ (green cosine wave) drawn along the original and discrete signal.

Figure 3.22 shows the top three frequencies in the signal. As illustrated, the time behaviour of the signal changes, especially near the end. In particular, much more data is written at the end, which disturbs the periodic behaviour. Consequently, the obtained low confidence value is justified.

To adapt to changing behaviour, FTIO offers modifying the inspected time window Δt of the analysis. In the online mode, this is performed automatically. For offline evaluation, the user can control the time window by specifying the start time `-ts start_time` and/or the end time `-te end_time` of the analysis. We plan on developing more automated methods for the offline approach in the future.

Rather than executing the above command, with the `-o` flag, we can specify that we would use DBSCAN

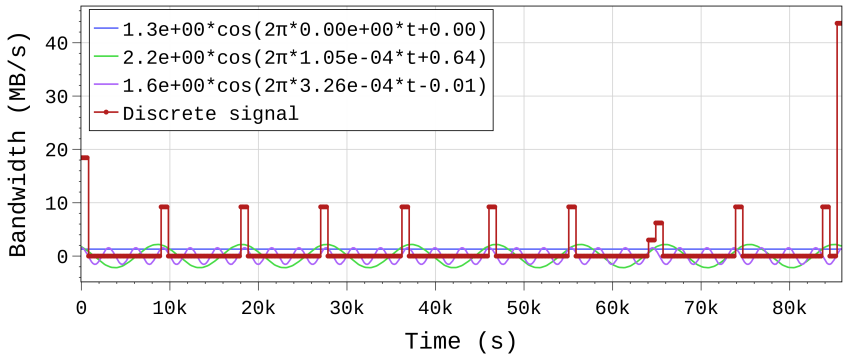


Figure 3.22: Result of FTIO on the Darshan profile of NEK5000 with 1024 ranks. The figure shows the top three frequencies (including the DC offset) in the signal.

to detect the outliers instead of using the Z-score. Moreover, we can adjust the tolerance value, which is by default 0.8 in Eq. (3.5), using the `-t` flag:

```
$> ftio 1024.darshan -o dbscan  -t 0.6 -v
```

As shown in Figure 3.23, similar results are obtained. DBSCAN results in a single cluster (cluster 1). The dominant frequency at $f_d = 1.05 * 10^{-4}$ coloured in red in the figure is the only outlier detected. Thus, the results are similar to the case when the Z-score was used.

To refine the confidence, FTIO offers to use additional autocorrelation. However, we skip this step here, as this is later demonstrated in Section 4.4. Moreover, the signal can be further characterised based on the frequency found. This includes providing a periodicity score and calculating the average bytes transferred. More details on these aspects are provided in [46].

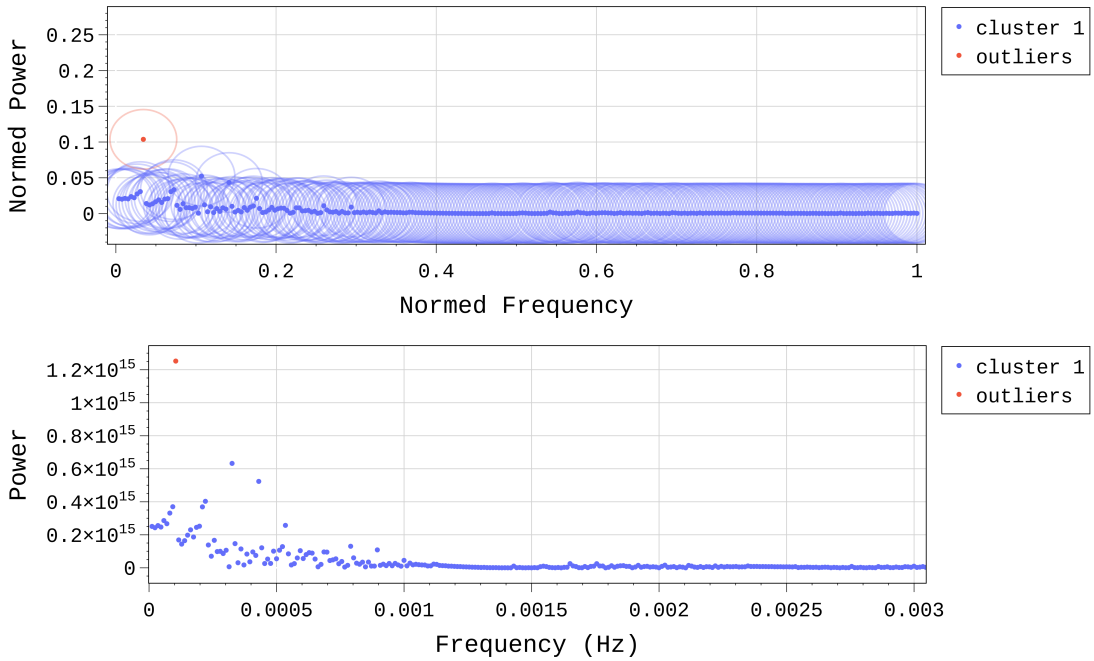


Figure 3.23: Result of FTIO on the same trace as in Figures 3.20 and 3.21, however, using DBSCAN rather than the Z-score for outlier detection. As observed, similar results are obtained.

3.5.8 FTIO Meets Extra-P

FTIO creates prediction for the I/O phases of an application. More generally speaking, FTIO can generate predictions for any traces provided. While we limited this aspect to I/O phases, other options include predicting, for example, the scheduling points, compute phases, and other metrics according to captured information in the traces.

With malleability, predicting the I/O phases can be very beneficial. This becomes even more relevant when the scaling behaviour of the application is examined. Consequently, we recently investigated combining FTIO and Extra-P. While FTIO generates predictions for the phases, Extra-P could model their scaling behaviour.

To demonstrate this aspect, we download from the *I/O Trace Initiative* website [30][13] all currently presented NEK5000 darshan files (from 1024 till 8192 ranks). Next, we executed FTIO on each of them. We merged the found predictions into a JSONL file as described in Section 3.3, which is supported by Extra-P. Finally, we provided the generated file to Extra-P and obtained results presented in Figure 3.24.

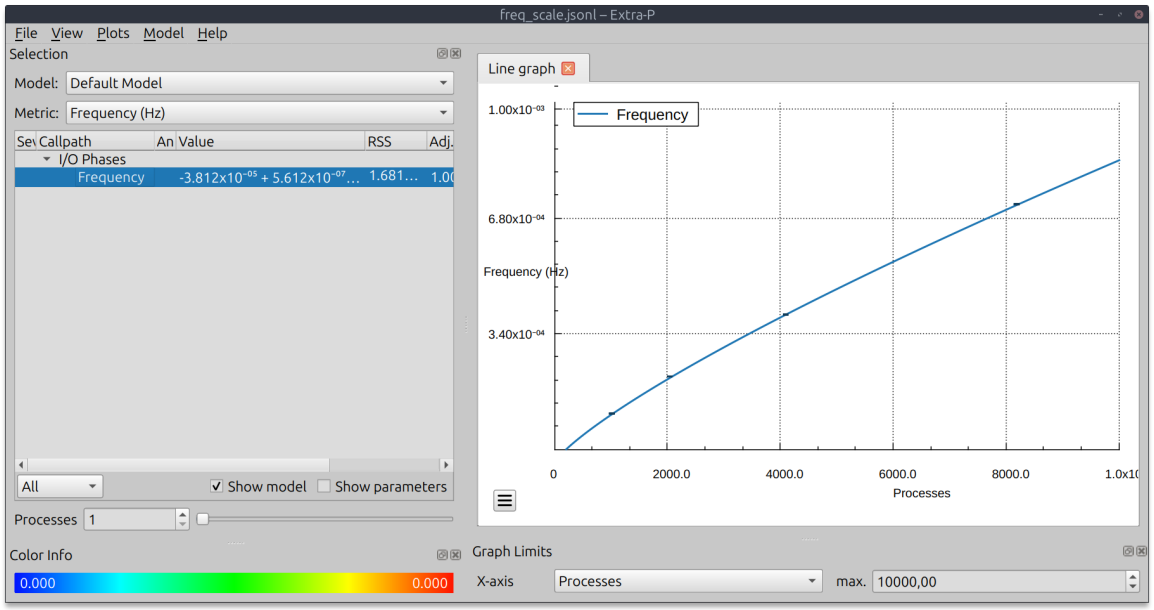


Figure 3.24: Scaling behaviour of the prediction (frequency of the I/O phases) from FTIO in Extra-P for NEK5000.

For this example, each trace file just delivered a single prediction as the offline version of FTIO was used. Future work will focus on integrating the online approach of FTIO with Extra-P to provide Extra-P with more predictions.

[13] https://hpcioanalysis.zdv.uni-mainz.de/

4 Use Cases: Exploiting the Models

The models developed in WP5 can aid different components in the ADMIRE framework in their decision-making. In this chapter, we demonstrate the use cases of these models highlighting in each case the advantages gained through our approach. In particular, we focus on four components: The malleability manager from WP3, the inelegant controller from WP6, the I/O scheduler from WP4, and the ad-hoc file systems from WP2. In the same order, the synergy between these models and the components is handled in Sections 4.1 to 4.4.

4.1 Job Scheduling

In WP3, we developed a scheduling algorithm to balance the computational and I/O load of the workload. As the algorithm's baseline, we define the I/O intensity of a job, the system (i.e., running jobs), and the workload (i.e., running and queued jobs). To assess the I/O intensity of a job, we combine the job metrics I/O time, total time, and the average bandwidth achieved to the PFS. We then derive the system and workload I/O intensity by combining the intensities of the running and queued jobs, respectively, and with every scheduling decision, the scheduler approximates the workload I/O intensity with the system I/O intensity.

Extra-P models provide the algorithm with all the necessary metrics in the decision-making process. The scheduler retrieves the scalability for jobs entering the queue and alters the workload I/O intensity accordingly. As we consider rigid, moldable, and malleable jobs, the scheduler has to account for single or multiple configurations, depending on the job type. While rigid jobs alter the workload I/O intensity with a single configuration, the scheduler considers the preferred configuration for moldable and malleable jobs upon submission. When a job enters the system (i.e., starts executing), the system I/O intensity alters accordingly. In the case of malleable jobs, the scheduler exploits scheduling points to approximate the workload I/O intensity by changing the configuration of a job, which, in return, modifies the system I/O intensity. Figure 4.1 visualises our scheduling concept.

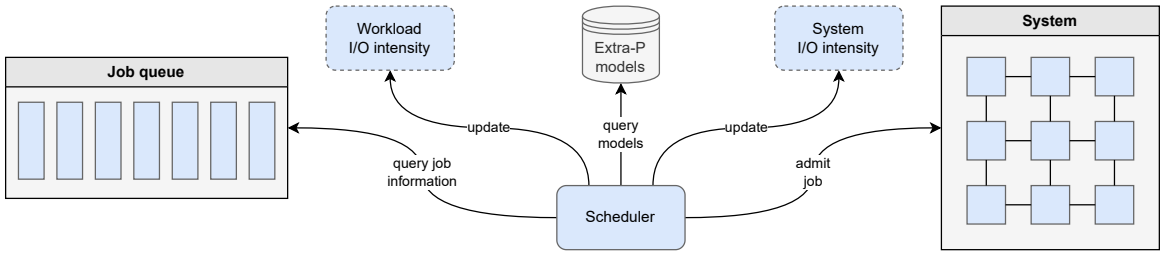


Figure 4.1: Our scheduling concept balancing computation vs. I/O exploiting Extra-P models.

4.1.1 A malleability scheduling algorithm

In ADMIRE, we have developed a new scheduling algorithm, named the *Malleable Easy Backfilling* (MEBF) algorithm, which extends Easy backfilling with malleability features. When a job arrives to the system, the jobs are allocated to resources normally or adapted to the availability of the system's resources.

Following performance models, such as those provided by Extra-P, malleability decisions are applied to the jobs based on their priority and their remaining execution time. MEBF scheduling algorithm first tries to serve

Algorithm 1: Malleable job expansion with Backfilling, MEBF

```
Input: List of jobs: J; set of nodes: N;
// workload lists updated each schedule
jobs ← {j ∈ J : state_j = RUNNING ∧ type_j = MALLEABLE};
free_nodes ← {n ∈ N : state_n = FREE};
for each job, job ∈sorted_jobs do
    if job.state == PENDING then
        for i ∈ pref_nodes, ..., min_nodes do
            if i ≤ free_nodes then
                job.assign(i)
                break
            end
        end
        if free_nodes == 0 & low_r ≠ 0 then
            SHRINK(low_r)
        else
            if free_nodes ≠ 0 & rmjobs ≠ 0 then
                backfilled_job =BACKFILL(pending_jobs)
                if backfilled_job == "None" then
                    EXPAND(high_r)
                else
                    backfilled_job.assign(pref_nodes)
                end
            else
                if (no pending_jobs) then
                    EXPAND(high_r)
                end
            end
        end
    end
end
```

each submitted job according to its priority, giving the higher-priority jobs a chance to be executed before the lower-priority jobs. If an arriving job can't be allocated because of insufficient available resources, the system state is checked to go either with backfilling, expansion, or shrinking. If the system state indicates that there are free resources but not enough for the queued job, BACKFILLING is invoked to find another malleable job that can be allocated with its preferred number of nodes. If BACKFILLING does not find a suitable job to fill the resources, EXPANSION of the longest-running job can be checked. On another hand, if the state of the system indicates that there are no free resources at all, a list of candidates jobs based on their priority is created to make space for the submitted job through SHRINK. The scheduling algorithm is detailed in Listing 1.

Shrinking a job is implemented only after it is confirmed that job performance is not harmed. For each candidate job, the feasibility routine is invoked. It uses a termination time prediction based on the recorded execution time to decide the possibility of shrinking. The feasibility routine executes a comparison between the job execution time with malleability and the job performance without implementing any changes. Only if the job will not be terminated soon or if the ratio between the predicted execution time with shrinking and the static execution time does not exceed a certain threshold (γ), the shrinking can be implemented. As defined in the equations 4.1 and 4.2. Once the job shrinking is considered feasible, a *shrinking_factor* specifies the amount of granted nodes. It is important to re-mention that node stealing is performed only from low-priority jobs.

The job expansion as well is carried out after checking the feasibility. The predicted remaining execution time should be greater than a quarter of the recorded execution time of the job. Otherwise, the job is considered almost complete, and the expansion will not provide the desired benefits. In both malleable events; shrinking and expansion, the de-allocation cost or allocation cost (α) of the nodes was taken into account. After removing or adding new nodes from or to a job, it is appended to the reconfigured jobs list to avoid applying changes on the same jobs several times.

$$granted\_nodes = job\_assigned\_nodes \times sharing\_factor \tag{4.1}$$

Algorithm 2: Malleability Hand-off policy for expansion

Input: List of jobs: J; set of nodes: N;
$rmjobs \leftarrow \{j \in J : state_j = \text{RUNNING} \wedge type_j = \text{MALLEABLE}\}$;
$free\_nodes \leftarrow \{n \in N : state_n = \text{FREE}\}$;
$sorted\_jobs \leftarrow \text{SORT}(rmjobs, \text{"SCAL\_FACTOR"})$;
for *j in sorted_jobs* **do**
  if FEASIBLITY_EXPAND(j) **then**
    for $(n \leq |free\_nodes| - |J_{assigned\_nodes}|, -1)$ **do**
      $nodes\_new \leftarrow n + |J_{assigned\_nodes}|$;
      if $(n > assigned\_nodes) and (nodes\_new \leq |free\_nodes| \wedge nodes\_new \leq |j.max\_nodes|)$ **then**
        j.ASSIGN(n);
        FREE_NODES.REMOVE(n);
        BREAK;
      end
    end
  end
end

where *job_assigned_nodes* is the number of nodes currently used by a job during execution. and *sharing_-factor* defines the shrinkage step (i.e., shrink amount)

$$shrink\_increase = \frac{granted\_nodes}{job\_assigned\_nodes} \times rem\_exe\_time + \alpha \tag{4.2}$$

where *rem_exe_time* is the estimated remaining time of a job until it is completed. We used the recorded execution times of the jobs on a dedicated system to calculate the estimated time. While *alpha* represents the cost of deallocation.

$$new\_est\_exe = rem\_exe\_time + shrink\_increase \tag{4.3}$$

$$\frac{new\_est\_exe}{rem\_exe\_time} <= \gamma \tag{4.4}$$

where the new_time with malleability does not exceed the double of static rem_time.

Malleability policies: In what follows in this section, we briefly describe three proposed policies for the malleability of the resources: hand-off, aggressive, and keep-spare.

1) Hand-off: In this strategy, expansion/shrinking occurs only when the amount of expansion is significant. A set of constraints, as shown in Algorithm 2, limits the expansion/shrinking events to the most useful ones. The job *J* is modified by step *n* if $n \leq free\_nodes - J_{allocated\_nodes}$. This makes the malleability less greedy, as the job takes a portion of the free nodes and leaves an amount equal to its currently allocated node. This strategy does not attempt to expand the job up to its *max*, but instead expands significantly and leaves some of the free nodes for other jobs. Same for shrinking, as we never try to go to the minimum number of nodes in the first place.

2) Aggressive: In the aggressive policy, the jobs are expanded/shrunken to their maximum/minimum nodes, even if expansion costs all *free_nodes*. Algorithm 3 shows how this policy allows jobs to expand. Aggressive expansion is implemented in two ways. The first is greedy enough to allow the job to expand while taking all available free nodes if they are in the range of its maximum nodes, and to exclude it from shrinking after this expansion while allowing the job to expand multiple times. The second allows shrinkage of that job after expansion.

3) Keep-spare: This policy tries to use only a portion of free_nodes for expansion or a portion of allocated nodes for reduction. The expansion algorithm is shown in Algorithm 4. In case of expansion: $new\_nodes = int(i - num\_assigned\_nodes * 0.3)$.

As the algorithms are similar for shrinking decisions, we skip them.

Algorithm 3: Malleability Expand Aggressive policy for expansion

```
Input: List of jobs: J; set of nodes: N;
rmjobs ← {j ∈ J : state_j = RUNNING ∧ type_j = MALLEABLE};
free_nodes ← {n ∈ N : state_n = FREE};
sorted_jobs ← SORT(rmjobs, "SCAL_FACTOR");
for j in sorted_jobs do
    if FEASIBLITY_EXPAND(j) then
        for (n ≤ |j.max_nodes| − |j.assigned_nodes|, −1) do
            if n ≤ |free_nodes| then
                j.ASSIGN(new_nodes)
                FREE_NODES.REMOVE(n);
                BREAK;
            end
        end
    end
end
```

Algorithm 4: Expand Keep-Spare policy

```
Input: List of jobs: J; set of nodes: N;
rmjobs ← {j ∈ J : state_j = RUNNING ∧ type_j = MALLEABLE};
free_nodes ← {n ∈ N : state_n = FREE};
sorted_jobs ← SORT(rmjobs, "SCAL_FACTOR");
for j in sorted_jobs do
    if FEASIBLITY_EXPAND(j) then
        for (n ≤ |j_max_nodes| − |j_assigned_nodes|,-1) do
            if (n > assigned_nodes)and(n ≤ |free_nodes|) then
                nodes_new ← n − |j_assigned_nodes| × limit%};
                if nodes_new ≤ |free_nodes| ∧ nodes_new ≤ |j_max_nodes| then
                    j.ASSIGN(new_nodes)
                    free_nodes.REMOVE(n);
                    BREAK;
                end
            end
        end
    end
end
```

4.2 System Model

WP6 (task 6.2) exploited a workflow to model applications behaviour in HPC system, namely GreatNector, which leverages the power of Extended Stochastic Symmetric Nets (ESSN), a sophisticated high-level formalism that enables the temporal modelling of system dynamics. In particular, it focuses on modelling the dynamics of I/O queues at the system level based on IOPS and network bandwidth measurements, offering the flexibility to define complex rate functions parametrically. The workflow is characterised by three main steps, starting from the elaboration of traces of the application generated from WP5, their clusterisation into different groups which will represent the application templates characterising all the applications with similar behaviour, and the system calibration and simulation to fit the average traces of the application templates. Specifically, the clustering was mainly carried out by exploiting the CONNECTOR package, while the model design, calibra-

tion and simulation by the GreatMod framework. GreatMod was exploited to design the model and to simulate the Continuous Time Markov process underlying the ESSN model using the Stochastic Simulation Algorithm.

Let us focus on the input applications traces, which are generated by the Always-On-Monitoring developed in WP5 based on The TAU performance toolset. These traces are processed for filtering the unnecessary information, and aggregating the time spent in a state, and the number of calls of that state in a specific time interval for each application state that we want to consider (for instance I/O, MPI, etc). Successively, these traces are clustered to highlight differences and similarities among the traces, and also to highlight similar patterns in the application. Moreover, the homogeneous groups characterising a template application are used to estimate the model's unknown parameters to fit a specific log trace through the calibration functionality provided by the framework.
Indeed, the application behaviour is intricately linked to the number of processors utilised. To streamline the analysis and ensure a comprehensive understanding without segregating individual processors or considering different applications the traces obtained using the same applications but with different numbers of processors, Extra-P will be exploited. This normalisation process enables the consideration of the application's dynamics in a collective manner, abstracting away the individual processor distinctions. By applying Extra-P normalisation, the focus shifts to a holistic perspective, allowing for a more generalised modelling approach that encapsulates the influence of varying processor counts on the system dynamics. Specifically, by including Extra-P into the GreatNector framework (as shown in Fig.4.2), it will be possible to consider the application traces in terms of the model obtained by Extra-P to cluster the applications and to calibrate the ESSN model independently by the number of processors. The application's dependency on the number of processors and the utilisation of Extra-P

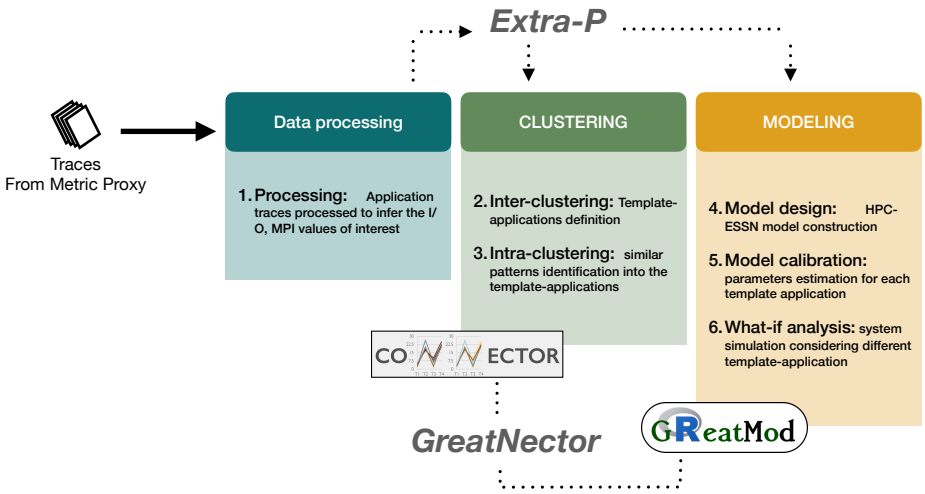


Figure 4.2: GreatNector framework including Extra-P.

normalisation represents an initial conceptualisation. Further refinement and exploration are anticipated as this approach serves as a starting idea, with the potential for deeper investigation and optimisation in subsequent stages of the research.

4.3 I/O Scheduling

In WP4, the IO-Sets [11] method for I/O scheduling was proposed, together with the Set-10 heuristic. In IO-Sets, applications are classified into sets, and applications belonging to the same set perform I/O exclusively (one at a time), while applications from distinct sets can perform I/O at the same time. Moreover, each set has a priority that defines how much I/O bandwidth its applications receive when sharing the PFS with others. Differently from other I/O scheduling techniques, IO-Sets uses only one piece of information about the application: the mean time between the start of consecutive I/O phases (also called "characteristic time"). Implementing

IO-Sets in practice requires, therefore, obtaining the characteristic time for running applications. This can be provided by FTIO, since the characteristic time is the period of I/O phases (reciprocal of the frequency). In this section, we describe an experiment that evaluates this combination of IO-Sets with FTIO.

In case FTIO is used together with Set-10 (later denoted as "Set-10 + FTIO "), the priorities for the groups (i.e., the sets) are calculated based on the period T_d provided by FTIO.

We used the IO-Sets implementation on BeeGFS, developed in WP4 of ADMIRE [6]. Experiments were conducted in the Grid'5000 (`www.grid5000.fr`) experimental platform, using the Gros cluster of the Nancy site. A single node was deployed as the metadata server and management node, and other nodes are either OSS or client nodes. Each OSS has a single OST, which uses the node-local hard disk for storage. See [6] for more details about this experiment.

The IOR benchmarking tool was modified to use the TMIO library. Each IOR instance (representing a different application) has a dedicated prediction mode FTIO, running on the same node and watching for its I/O trace, which is updated after every I/O phase (due to a TMIO call). A custom the script parses the FTIO output for the predicted period and confidence, calculates the priority according to the Set-10 heuristic, and writes it to the */sys/kernel/config/iosets/[job id]/priority* file, from where it is obtained by the BeeGFS client module whenever the application issues new I/O requests.

We aimed at recreating one of the experiments shown in [11, Figure 4]: 16 concurrent applications execute, clearly belonging to two separate sets, and inside each set, they all present the same period. IOR was used to create one high-frequency application and 15 low-frequency ones: all of them use a single node (which is a shared client node for all 16 applications), 8 processes, and write to file-per-process files using the *fsync* option (-e). The high-frequency application has 200 compute phases of 18 seconds, each followed by I/O phases where each process writes 16 MiB. Because the write performance with fsync was measured to be approximately 110 MiB/s, that means this application has a period of approximately 19.2 s. The low-frequency applications, on the other hand, have compute phases of 360 seconds, followed by I/O phases where each process writes 320 MiB (for a period of $\approx$ 384 s).

Figure 4.3 shows the results, comparing four situations:

- "Set-10 + clairv." is a clairvoyant application of the scheduling heuristic, meaning that the *ideal* (in isolation) periods (19.2 or 384 s) are provided manually in advance.
- "Set-10 + FTIO" combines the heuristic with FTIO, which determines the *actual* periods at runtime. In this case, Set-10 uses the most recent prediction from FTIO.
- "Set-10 + error" uses predictions *worse* than FTIO: the predictions given by FTIO are randomly increased or decreased by a factor of 50% before provided to Set-10.
- "Original" corresponds to BeeGFS without any modifications and serves as the baseline.

The metrics were computed using the time frame between 400 and 3200 seconds after the start of the first application, and are shown in Figure 4.3. The *stretch* quantifies the overall slowdown factor for an application caused by inter-job file-system interference; the *I/O slowdown* represents the factor by which its I/O time was increased. Thus, the lowest value of both metrics is 1. Both are calculated by taking the geometric mean of all applications from each execution. The *Utilization* ($\in [0, 1]$) is a system metric that specifies how much of the node time was spent on computation instead of I/O. More details about these metrics are given in [11, Section V-D], and more on this experiment in [6].

The results achieved with FTIO are close to the clairvoyant version—only 2.2% worse in stretch, 19% in I/O slowdown, and 2.3% in utilisation. In contrast, the version where we inject errors to FTIO results made stretch worse by 5%, utilization by 4%, and I/O slowdown 27% higher, compared to the "Set-10 + FTIO" version, in addition to presenting higher variability. Compared to not using Set-10, the FTIO-powered version decreased the mean stretch and I/O slowdown by 20% and 56%, respectively, and increased utilisation by 26%. These results show how well FTIO fills the knowledge gap, making the improvements that Set-10 allows possible in practice, where the period is not known in advance.

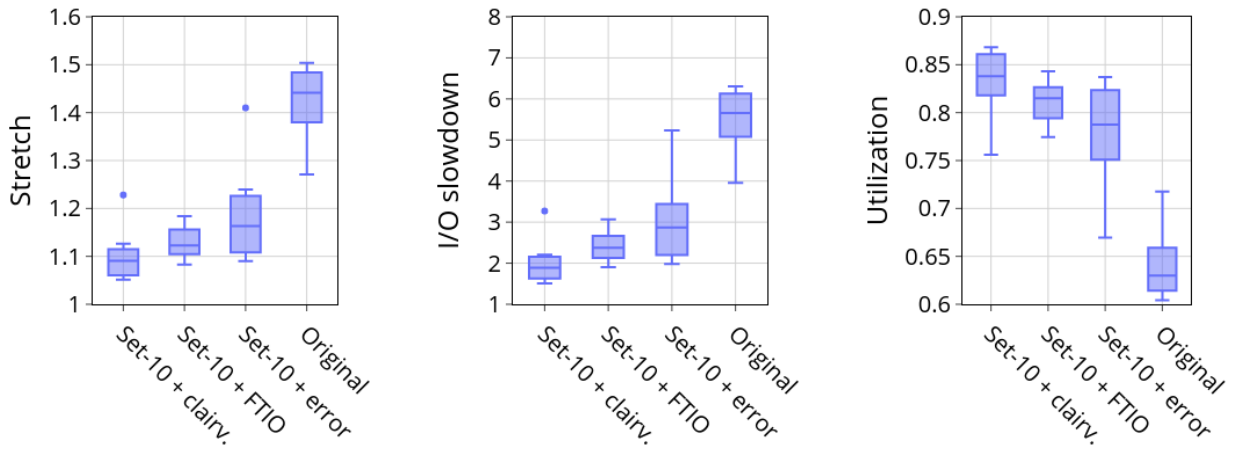


Figure 4.3: Comparison of clairvoyant Set-10, Set-10 with FTIO, Set-10 with 50% error injected to the FTIO-provided periods, and the original configuration without Set-10. The figures show the stretch (how much slower jobs were compared to running in isolation: lower is better), the I/O slowdown (how much slower I/O was compared to isolation: lower is better), and the utilization (how much of the time was NOT spent on I/O: higher is better). The boxplots (with 1.5*IQR whiskers) group ten executions. The y-axes do not start at zero and are all different.

4.4 Just-in-Time Staging

One of the use cases of FTIO is *data staging* between the back-end shared parallel file system (PFS) in a compute cluster and the ad-hoc file system. In general, data staging is necessary because ad-hoc file systems provide a new (initially empty) namespace that applications can use as a temporary high-performing storage layer that scales with the number of participating compute nodes. Therefore, input data must be *staged-in*to the ad-hoc file system from the PFS and output data *staged-out* from the ad-hoc file system to the PFS. However, because data staging usually begins before an application is started and after it ends, additional job wall clock time is required for the staging process. Consequently, an ad-hoc file system may not benefit certain applications with only little I/O intensity, where the time required for data staging results in a net increase of overall runtime.

Another approach leverages a characteristic of many HPC applications that are *bulk-synchronous* and alternate I/O and computational phases. Moreover, bulk-synchronous applications, such as NEK5000, write out data at specific step boundaries that are not read within the same compute job. In principle, it is, therefore, possible to stage-out result data immediately after it was written before the application ends and to stage-in data right before an application needs it; or *just in time*. To minimise interference, staging data should further overlap the computational phase to avoid reducing I/O performance during an I/O phase.

As discussed earlier, FTIO can predict future I/O phases, which information can be used to schedule data staging while the application is running. However, FTIO requires per-process I/O information to make such predictions, and therefore, analytics tools that provide post-mortem analyses, e.g., via tracing, are insufficient. Because the per-process I/O statistics can be collected directly within the ad-hoc file system, we have implemented a proof-of-concept prototype into the GekkoFS client that collects the per-process I/O statistics. This information can then be ingested into FTIO to make the corresponding future I/O predictions. This work is currently work-in-progress and is planned to be completed within the scope of this project.

In the following, we present the instructions to configure the GekkoFS client to collect I/O statistics. At the time of writing, the I/O statistics are written at the end of the application run. In the next step, we will include a periodic dump of the latest I/O activity that is directly ingested into FTIO.

First, GekkoFS needs to be built with client metric support. Please refer to the readme and documentation of GekkoFS for further build instructions[1]:

[1] https://storage.bsc.es/gitlab/hpc/gekkofs

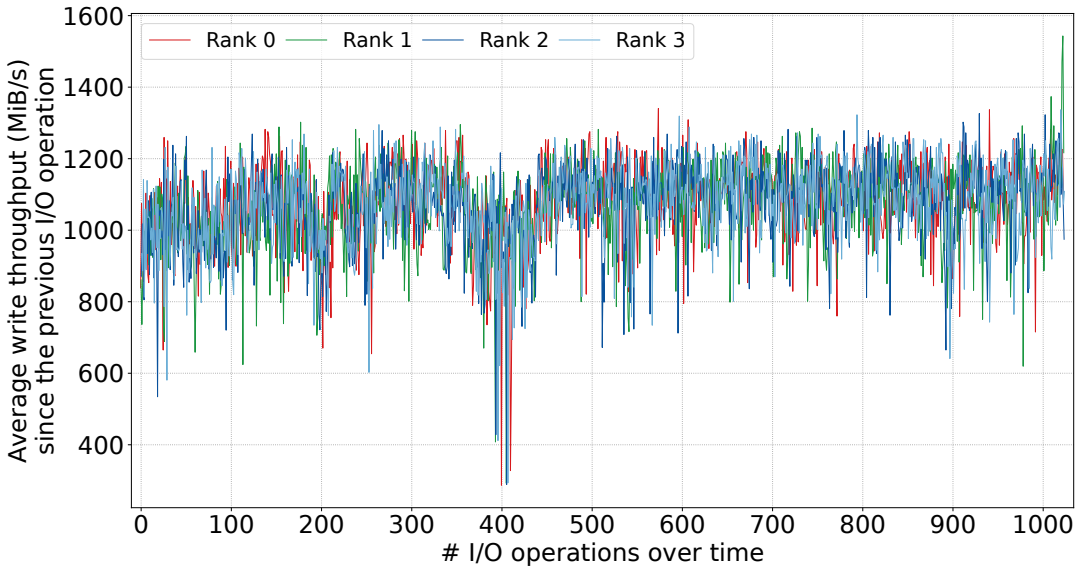


Figure 4.4: The average write throughput for GekkoFS when running IOR with four MPI ranks over the number of I/O operations. Each process wrote 512 KiB for each of the 1024 I/O operations (or 512 MiB in total).

```
cmake [...] -DGKFS_ENABLE_CLIENT_METRICS=ON ..
```

After the GekkoFS server instances are running, each client is configured separately to collect the client-side I/O statistics. In this example, we use the widely-used IOR benchmark[2] to generate an I/O workload sequentially writing and reading 512 MiB per process with 4 participating MPI ranks. The environment variables `LIBGKFS_ENABLE_METRICS` and `LIBGKFS_METRICS_PATH` define whether client-side metrics should be collected and where they are stored:

```
mpiexec -np 4 -x LIBGKFS_ENABLE_METRICS=on \
-x LIBGKFS_METRICS_PATH=/tmp/gkfs_client_metrics \
-x LD_PRELOAD=/usr/local/lib64/libgkfs_intercept.so \
/opt/bin/ior -a POSIX -i 1 -o /tmp/gkfs_mountdir/iortest -t 512k -b 512m -F
```

In this example, the resulting statistics are placed into the `/tmp/gkfs_client_metrics` directory in the *MessagePack* format that FTIO supports. When converted into a JSON format via the included `gkfs_-clientmetrics2json` tool, the individual I/O statistics can be made human-readable. Figure 4.4 presents the corresponding average throughput over 4 processes for 1024 consecutive write operations. The global accumulated write throughput was 4,179 MiB per second over a runtime of 0.49 seconds.

FTIO offers a customer data format to reach a broader audience. For the example here, we utilised this option to highlight this aspect and thus to pass the data to FTIO. For that, the `-cf` flag must be passed along with a file that defines the parsing of the provided traces. To gain further confidence in the results, we executed FTIO with the autocorrelation flag `-c` and lowered the tolerance value `-t` to detect if there is any dominant frequency in the signal. Thus, we executed the following command:

```
$> ftio file.txt -o dbscan  -t 0.6 -v -cf cutome_pattern.py
```

For this experiment, FTIO returned that the signal is not periodic. The result is illustrated in Figures 4.5 to 4.7. Figure 4.5 shows the single-sided power spectrum, including the frequency at 0 Hz (DC offset). As observed, this frequency has by far the highest contribution, indicating the behaviour of the signal is nearly constant. While FTIO usually excludes this frequency from the analysis, future versions will utilise it to make such conclusions. In Figure 4.6, the temporal behaviour of the application is shown. As illustrated, the temporal

[2] https://github.com/hpc/ior

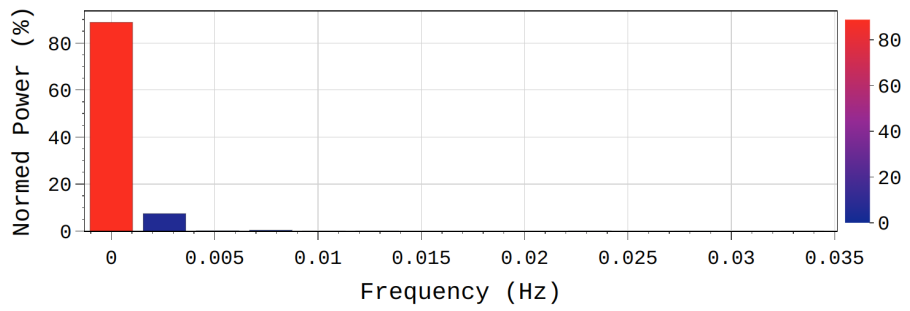


Figure 4.5: Normalized single-sided power spectrum from FTIO on the IOR example with GekkoFS.

behaviour of the signal is centred around the DC offset. The second highest frequency detected by FTIO (coloured green), shows the variability of the signal around this value.

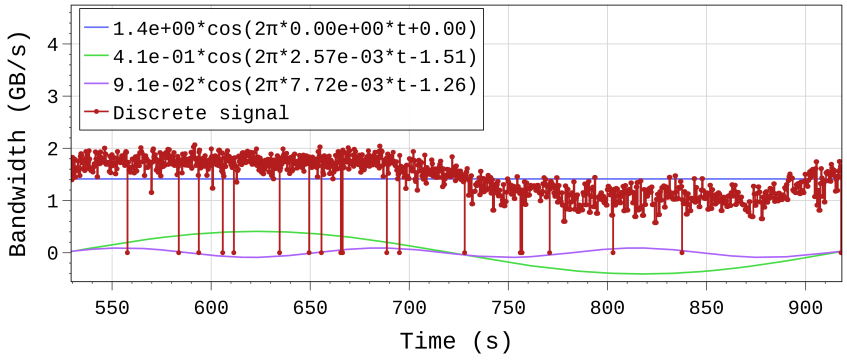


Figure 4.6: Temporal behaviour of the signal alongside the top three frequencies presented in it obtained by executing FTIO on part of the trace.

Figure 4.7 shows the result from autocorrelation on the signal with FTIO. With autocorrelation, FTIO is capable of increasing the confidence in the results. For this example, during the merging of the results from DFT and autocorrelation, FTIO detected that the confidence is too low and that there is no dominant behaviour in the signal.

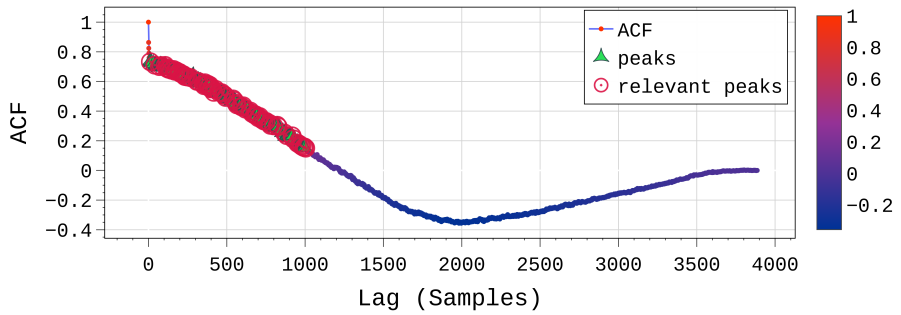


Figure 4.7: Result from autocorrelation on the signal.

To further demonstrate the just-in-time staging aspect, we handle a second example, which has a periodic behaviour. For that, we executed NEK5000 with GekkoFS with in total 200 steps on 8 nodes with in total 32 processes. Every 20 steps, I/O operations are performed. Note that while only Four ranks perform I/O, each one of them is located on a different node. The average write throughput for the four ranks is shown in Figure 4.8.

Each of these ranks saved its trace, shown in Figure 4.8, to a dedicated file. Next, we merge all traces into a single one named file.txt and provide again the same parsing file to FTIO. Furthermore, we use additionally autocorrelation to merge the result with DFT and set the sampling frequency to 0.01 Hz. Thus, we executed:

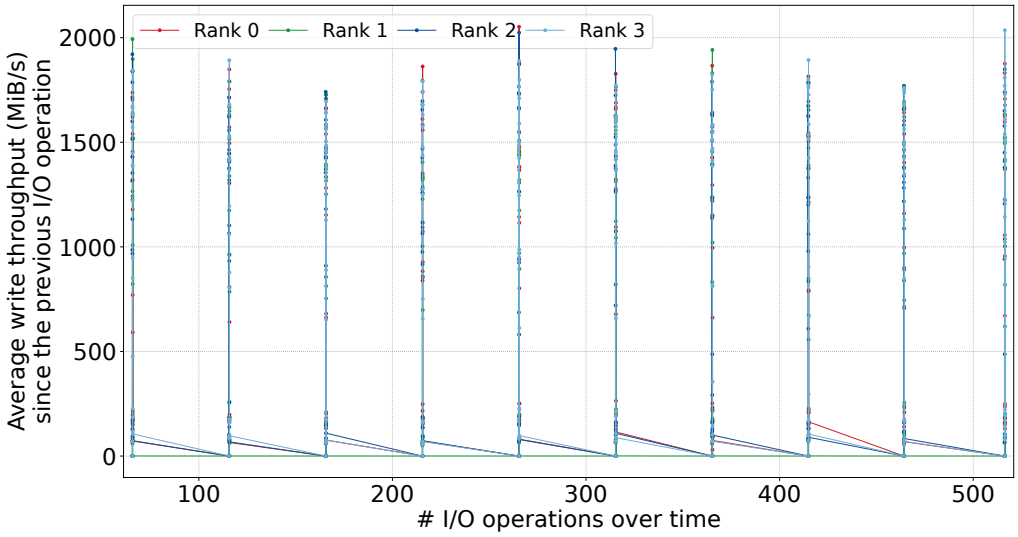


Figure 4.8: The average write throughput for GekkoFS when running NEK5000 with 32 MPI ranks (8 nodes) over the number of I/O operations. Four ranks perform I/O every 20 steps out of the 200 steps.

```
$> ftio file.txt -cf cutome_pattern.py -c -f 0.01
```

The results of this call are shown in Figure 4.9. As illustrated, FTIO detected a dominant frequency at 2e-05 Hz (i.e., 50088.8 s). Furthermore, FTIO overlapped the rank-level metric to obtain an application-level one. During the analysis, 6 harmonics were ignored as they were multiples of two of the dominant frequency. This was expected, as I/O bursts are presented in the signal (see Section 3.5.4). From DFT, FTIO return that the confidence in the results is 51.64%. Since the `-c` flag was enabled, autocorrelation was additionally executed, which in turn returned a confidence of 100%. Consequently, the final confidence FTIO provided for this example was 83.83%, which is obtained by not only merging these values but also examining the agreed predictions as mentioned in [46]. As shown in Figure 4.9, the high confidence is reasonable for this example.

With such as result, and since this example shows the writing behaviour of the application, the burst buffers could be flushed *just in time*, such that they are available before the next phase occurs. This would allow us to overcome the storage space restrictions such components impose. In the future, we will incorporate such analysis in the staging strategy for the burst buffers alongside strategies to reduce congestion during the flushing process. This would allow us to find the right instance to flush the buffers or load the data needed just in time.

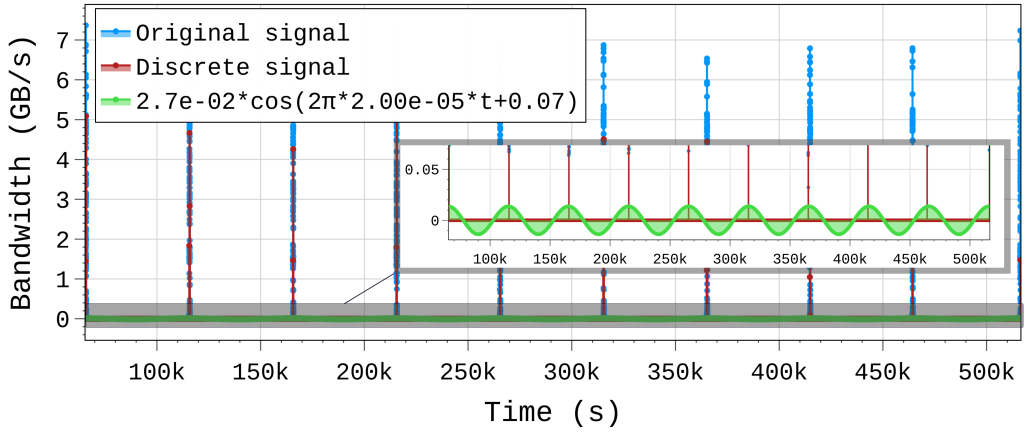


Figure 4.9: Temporal behaviour of the application-level signal from NEK5000 with GekkoFS. FTIO internally overlapped the rank-level metrics from Figure 4.8 to obtain the application-level bandwidth. FTIO detected a dominant frequency as the green cosine wave depicts.

5 Conclusion and Future Perspectives

In this deliverable, we have shown how ADMIRE developed specific methodologies to model and project performance data. This undertaking materialised as an approach that we called *continuous modelling*. Continuous modelling is about accumulating valorised metrics over time to acquire and enhance the understanding of the applications running in a supercomputer. ADMIRE has peculiar needs as it spans the whole software stack (from the I/O back-end to the application) and is proposed to implement holistic malleability at these various levels. As such, we first devised a robust and scalable monitoring capability with the design choice of coalescing metrics from multiple data sources, acknowledging the plurality of the heuristics guiding malleability.

Furthermore, we worked on extending Extra-P to leverage its advanced modelling capabilities to guide modelling for the whole project. To do so, the metric proxy was modified to be able to invoke Extra-P automatically, exposing a *model server* to the IC. Conjointly, a dimension under-exploited despite its paramount importance for I/O has been explored: Time. FTIO is a new way of looking at performance data and one of the big breakthroughs of the ADMIRE project. The ability to identify at scale the period of I/O (instead of simply looking at volume transferred) allowed us to greatly enhance the projection capabilities not only between jobs (moldability) but also inside a given job over time – the most challenging approach. As such, ADMIRE has developed a wide range of models capable of characterising applications in the various steps of their life cycle. As we exemplified, these automatically generated models provide ways of determining the best scale for a given program, predicting total I/O yield, and guiding buffer flushes in the case of burst buffers. They implement a complete toolbox that provides WP6 to guide the decisions of the IC, WP3 to perform job scheduling and balance the resource through the malleability manager, and finally WP4 and WP2 to take effective I/O scheduling decisions that boost system utilisation and to optimise the usage of novel storage components such as burst buffers.

All the tools implemented in this work package are open-source and freely available online to reproduce and experiment with the newly developed features. We are now actively working on integrating and demonstrating these modelling capabilities throughout the different components in ADMIRE and, in particular, to WP6 to close the feedback loop of the project successfully by the end of the ADMIRE project. To this purpose, we expose the modelling interfaces using mercury and provide representative test cases (using traces) to illustrate malleability heuristics. Eventually, we are now working on running applications leveraging the end-to-end ADMIRE infrastructure while tuning the piece-wise integration.

List of Acronyms and Abbreviations

$Adj.\ R^2$	Adjusted coefficient of determination
RSS	Residual sum of squares
ADMIRE	Adaptive multi-tier intelligent data manager for Exascale
DFT	Discrete Fourier transformation
ESSN	Extended Stochastic Symmetric Nets
GUI	Graphical User Interface
HPC	High Performance Computing
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IC	Inteligent Controller
IDFT	Inverse discrete Fourier transformation
IOPS	I/O operations per second
JS	JavaScript
JSON	JavaScript Object Notation
JSONL	JSON Lines file format, also known as newline-delimited JSON
LIMITLESS	light-weight monitoring tool for large scale systems
MEBF	Malleable Easy Backfilling algorithm
MPI	Message-Passing Interface
TBON	Tree Based Overlay Network
WP	Work Package

Glossary

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.

BSC Barcelona Supercomputing Center, Barcelona.

D4.2 Deliverable 4.2 from WP4 of the ADMIRE project: Software to support I/O scheduling policies [37].

D5.3 Deliverable 5.3 from WP5 of the ADMIRE project: Report on the implementation of application I/O profiling [38].

D5.4 Deliverable 5.4 from WP5 of the ADMIRE project: Monitoring solution at Exascale [39].

D6.3 Deliverable 6.3 from WP6 of the ADMIRE project: Runtime tools to tune I/O system behaviour [40].

Extra-P Tool for automated performance modelling of HPC applications developed by TUDA. See `https://github.com/extra-p/extrap` for more info.

FTIO **F**requency **T**echniques for **I/O**. It is a tool developed in the ADMIRE project to predict the period of the I/O phases using frequency techniques. The approach is described in detail in Section 3.5 and in [46]. The tool is publicly available on GitHub: `https://github.com/tuda-parallel/FTIO`.

GekkoFS A Temporary Distributed File System for HPC Applications. The tool is developed by JGU and BSC. See `https://storage.bsc.es/gitlab/hpc/gekkofs` for more info.

IOR Is a parallel IO benchmark that can be used to test the performance of parallel storage systems. For more info, see `https://ior.readthedocs.io/en/latest/`.

JGU Johannes Gutenberg-Universität Mainz

NEK5000 Is a highly scalable spectral element computational fluid dynamics code for solving the incompressible Navier-Stokes equations on 2D quadrilateral and 3D hexahedral meshes. For more info, see `https://nek5000.github.io/NekDoc/`.

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 ship 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 make them persistent.

PFS Parallel File System, type of distributed file system supporting a global namespace and spread across multiple storage servers.

RPC Remote procedure call.

scalability bug A scalability bug is a part of the program whose scaling behaviour is unintentionally poor, that is, much worse than expected.

SLURM Job submission system widely used.

TAU Tuning and Analysis Utilities. See https://www.cs.uoregon.edu/research/tau/home.php for more details.

TMIO **T**racing **M**PI-**IO**. It is a C++ library that intercepts MPI calls using the PMPI interface and is attached to an application using the `LD_PRELOAD` mechanism. The tool was developed in the ADMIRE project and is described in detail in Section 2.5. It is publicly available on GitHub: https://github.com/tuda-parallel/TMIO

TUDA Technical University of Darmstadt, Germany.

WP2 Work package 2 of the ADMIRE project: Ad-hoc storage systems. The main objective of this WP is the development of fast and scalable ad-hoc storage systems.

WP3 Work package 3 of the ADMIRE project: Malleability management. The main objective of this WP is to develop mechanisms to manage the combined malleability of computation and I/O.

WP4 Work package 4 of the ADMIRE project: I/O scheduler. The main objective of this WP is to achieve a inter-job storage scheduling and I/O coordination that is not available today in production systems.

WP5 Work package 5 of the ADMIRE project: Sensing and profiling. The main objective of this WP is to setup systems monitoring, I/O profiling, and performance modelling tools amenable for exascale.

WP6 Work package 6 of the ADMIRE project: Intelligent controller. The main objective of this WP is to integrate cross-layer data to provide a holistic view of the system to enable intelligent decisions for dynamically adapting the system to the current and future workloads.

WP7 Work package 7 of the ADMIRE project: Application co-design. The main objective of this WP is to analyze application codes to provide requirements as co-design input to technical WPs (i.e., WP2-WP6) and to define suitable use cases and benchmarks to prepare applications to optimally run with ADMIRE technologies. It will thus evaluate the usability of the ADMIRE technologies to demonstrate advances for end users.

Bibliography

[1] V. Aggarwal, C. Yoon, A. George, H. Lam, and G. Stitt. Performance modeling for multilevel communication in shmem+. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, New York, NY, USA, 2010. Association for Computing Machinery.

[2] Marco Aldinucci, Sergio Rabellino, Marco Pironti, Filippo Spiga, Paolo Viviani, Maurizio Drocco, Marco Guerzoni, Guido Boella, Marco Mellia, Paolo Margara, Idillio Drago, Roberto Marturano, Guido Marchetto, Elio Piccolo, Stefano Bagnasco, Stefano Lusso, Sara Vallero, Giuseppe Attardi, Alex Barchiesi, Alberto Colla, and Fulvio Galeazzi. HPC4AI, an AI-on-demand federated platform endeavour. In *ACM Computing Frontiers*, Ischia, Italy, May 2018.

[3] Guillaume Aupy, Olivier Beaumont, and Lionel Eyraud-Dubois. What size should your buffers to disks be? In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 660–669. IEEE, 2018.

[4] Guillaume Aupy, Olivier Beaumont, and Lionel Eyraud-Dubois. Sizing and partitioning strategies for burst-buffers to reduce IO contention. In *IPDPS'19*, pages 631–640. IEEE, 2019.

[5] Alexis Bandet, Francieli Boito, and Guillaume Pallez. Scheduling distributed I/O resources in HPC systems. 2024.

[6] Clément Barthélemy, Francieli Boito, Emmanuel Jeannot, Guillaum Pallez, and Luan Teylo. Implementation of an unbalanced I/O bandwidth management system in a parallel file system, 2024. Available at https://inria.hal.science/hal-04417412.

[7] IOR Benchmark. Version 3.3.0. https://github.com/hpc/ior, 2020.

[8] Intel MPI Benchmarks. https://www.intel.com/content/www/us/en/docs/mpi-library/user-guide-benchmarks/2021-2/overview.html, 2023.

[9] A. Bhattacharyya and Torsten Hoefler. PEMOGEN: Automatic Adaptive Performance Modeling During Program Runtime. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*, pages 393–404. ACM, Aug. 2014.

[10] Francieli Boito, Guillaume Pallez, Luan Teylo, and Nicolas Vidal. IO-sets: Simple and efficient approaches for I/O bandwidth management. *IEEE Transactions on Parallel and Distributed Systems*, 34(10):2783–2796, 2023.

[11] Francieli Boito, Guillaume Pallez, Luan Teylo, and Nicolas Vidal. Io-sets: Simple and efficient approaches for I/O bandwidth management. *IEEE Transactions on Parallel and Distributed Systems*, 34(10):2783–2796, 2023.

[12] Alexandru Calotoiu, Torsten Hoefler, Marius Poke, and Felix Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 45, 2013.

[13] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and Improving Computational Science Storage Access through Continuous Characterization. *ACM Transactions on Storage*, 7(3):8:1–8:26, October 2011.

[14] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale I/O workloads. In *Cluster'09 Workshops*, pages 1–10. IEEE, 2009.

[15] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale I/O workloads. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, 2009.

[16] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination. In *IPDPS'14*, pages 155–164. IEEE, 2014.

[17] Matthieu Dorier, Gabriel Antoniu, Rob Ross, Dries Kimpe, and Shadi Ibrahim. Calciom: Mitigating I/O interference in HPC systems through cross-application coordination. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 155–164, 2014.

[18] Paul R. Eller, Torsten Hoefler, and William Gropp. Using performance models to understand scalable krylov solver performance at scale for structured grid problems. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, page 138–149, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. Scheduling the I/O of HPC applications under congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1013–1022. IEEE, 2015.

[20] Ana Gainaru, Guillaume Aupy, Anne Benoit, Franck Cappello, Yves Robert, and Marc Snir. Scheduling the I/O of HPC applications under congestion. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1013–1022, 2015.

[21] Salman Habib, Vitali Morozov, Hal Finkel, Adrian Pope, Katrin Heitmann, Kalyan Kumaran, Tom Peterka, Joe Insley, David Daniel, Patricia Fasel, Nicholas Frontiere, and Zarija Lukic. The Universe at extreme scale: Multi-petaflop sky simulation on the BG/Q. In *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Salt Lake City, UT, November 2012. IEEE.

[22] Torsten Hoefler, William Gropp, William Kramer, and Marc Snir. Performance modeling for systematic performance tuning. In *State of the Practice Reports*, SC '11, pages 1–12, New York, NY, USA, November 2011. Association for Computing Machinery.

[23] Wei Hu, Guang-ming Liu, Qiong Li, Yan-huang Jiang, and Gui-lin Cai. Storage wall for exascale supercomputing. *Frontiers of Information Technology & Electronic Engineering*, 17(11):1154–1175, November 2016.

[24] Mihailo Isakov, Eliakin del Rosario, Sandeep Madireddy, Prasanna Balaprakash, Philip Carns, Robert B. Ross, and Michel A. Kinsy. HPC I/O Throughput Bottleneck Analysis with Explainable Local Models. In *SC'20*, pages 1–13, 2020.

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

[26] K Senthamarai Kannan, K Manoj, and S Arumugam. Labeling methods for identifying outliers. *International Journal of Statistics and Systems*, 10(2):231–238, 2015.

[27] LLNL. Kripke. https://github.com/LLNL/Kripke.

[28] LLNL. CORAL Benchmark Codes - *HACC IO*. https://asc.llnl.gov/coral-benchmarks#hacc, 2020.

[29] Aniruddha Marathe, Rushil Anirudh, Nikhil Jain, Abhinav Bhatele, Jayaraman Thiagarajan, Bhavya Kailkhura, Jae-Seung Yeom, Barry Rountree, and Todd Gamblin. Performance modeling under resource constraints using deep transfer learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[30] Nafiseh Moti, André Brinkmann, Marc-André Vef, Philippe Deniel, Jesús Carretero, Philip H. Carns, Jean-Thomas Acquaviva, and Reza Salkhordeh. The I/O trace initiative: Building a collaborative I/O archive to advance HPC. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, SC-W 2023, Denver, CO, USA, November 12-17, 2023*, pages 1216–1222. ACM, 2023.

[31] MPI Forum. MPI: A Message-Passing Interface Standard, June 2021.

[32] Mohammad Abu Obaida, Jason Liu, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. Parallel application performance prediction using analysis based models and HPC simulations. In *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '18, page 49–59, New York, NY, USA, 2018. Association for Computing Machinery.

[33] University of Turin. Cluster documentation. https://hpc4ai.unito.it/documentation/, 2023.

[34] Guillaume Pallez. *Model Design and Accuracy for Resource Management in HPC*. PhD thesis, Université de Bordeaux, 2023.

[35] F. Petrini, D.J. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, pages 55–55, November 2003.

[36] James Price and Simon McIntosh-Smith. Improving auto-tuning convergence times with dynamically generated predictive performance models. In *Proceedings of the 2015 IEEE 9th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip*, MCSOC '15, page 211–218, USA, 2015. IEEE Computer Society.

[37] ADMIRE project. Deliverable 4.2: Software to support I/O scheduling policies. https://admire-eurohpc.eu/wp-content/uploads/2024/02/D4_2_admire.pdf, 2023.

[38] ADMIRE project. Deliverable 5.3: Report on the implementation of application I/O profiling. https://admire-eurohpc.eu/wp-content/uploads/2023/07/D5_3_Report_on_the_implementation_of_application_IO_Profiling.pdf, 2023.

[39] ADMIRE project. Deliverable 5.4: Monitoring solution at Exascale. https://www.dropbox.com/s/mx831mokb2w2tsy/D5_4_Monitoring_solution_at_exascale.pdf?dl=0, 2023.

[40] ADMIRE project. Deliverable 6.3: Runtime tools to tune I/O system behaviour. https://www.dropbox.com/s/agfddcumv1vfs9s/D6_3_admire.pdf?dl=0, 2023.

[41] Marcus Ritter, Alexandru Calotoiu, Sebastian Rinke, Thorsten Reimann, Torsten Hoefler, and Felix Wolf. Learning cost-effective sampling strategies for empirical performance modeling. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 884–895, 2020.

[42] Marcus Ritter, Alexander Geiß, Johannes Wehrstein, Alexandru Calotoiu, Thorsten Reimann, Torsten Hoefler, and Felix Wolf. Noise-resilient empirical performance modeling with deep neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 23–34, May 2021.

[43] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K Lockwood, and Nicholas J Wright. Modular HPC I/O characterization with darshan. In *2016 5th workshop on extreme-scale programming tools (ESPT)*, pages 9–17. IEEE, 2016.

[44] Jerome Soumagne, Dries Kimpe, Judicael Zounmevo, Mohamad Chaarawi, Quincey Koziol, Ahmad Afsahi, and Robert Ross. Mercury: Enabling remote procedure call for high-performance computing. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8. IEEE, 2013.

[45] Jingwei Sun, Guangzhong Sun, Shiyan Zhan, Jiepeng Zhang, and Yong Chen. Automated performance modeling of HPC applications using machine learning. *IEEE Transactions on Computers*, 69(5):749–763, 2020.

[46] Ahmad Tarraf, Alexis Bandet, Francieli Boito, Guillaume Pallez, and Felix Wolf. Capturing periodic I/O using frequency techniques. In *Proc. of the 38th IEEE International Parallel and Distributed Processing Symposium (IPDPS), San Francisco, CA, USA*, pages 1–14. IEEE, May 2024. (accepted).

[47] Ahmad Tarraf, Alexis Bandet, Francieli Boito, Guillaume Pallez, and Felix Wolf. Capturing Periodic I/O Using Frequency Techniques [Data Set], February 2024.

[48] Marc-André Vef, Nafiseh Moti, Tim Süß, Tommaso Tocci, Ramon Nou, Alberto Miranda, Toni Cortes, and André Brinkmann. Gekkofs - a temporary distributed file system for HPC applications. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 319–324, 2018.

[49] Chen Wang, Jinghan Sun, Marc Snir, Kathryn Mohror, and Elsa Gonsiorowski. Recorder 2.0: Efficient Parallel I/O Tracing and Analysis. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1–8, May 2020.

[50] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, Apr 2009.

[51] Wenxiang Yang, Xiangke Liao, Dezun Dong, and Jie Yu. A quantitative study of the spatiotemporal I/O burstiness of HPC application. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1349–1359, 2022.

[52] Zhou Zhou, Xu Yang, Dongfang Zhao, Paul Rich, Wei Tang, Jia Wang, and Zhiling Lan. I/O-aware batch scheduling for petascale computing systems. In *Cluster'15*, pages 254–263. IEEE, 2015.