H2020-JTI-EuroHPC-2019-1

Project no. 956748

ADAPTIVE MULTI-TIER INTELLIGENT DATA MANAGER FOR EXASCALE

D5.3
Report on the Implementation of Application I/O Profiling

Version 1.0

Date: April 26, 2023

Type: Deliverable
WP number: WP5

Editor: Jean-Thomas Acquaviva
Institution: DDN

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

Change Log

Rev.	Date	Who	Site	What
1	14/09/22	Jean-Baptiste BESNARD	PARA	Document creation.
2	15/03/22	Jean-Baptiste BESNARD	PARA	Description of the measurement infrastructure.
3	20/03/22	Ahmad Tarraf	TUDA	Added frequency techniques description.
4	24/03/22	Francieli Boito	Inria	Extended the discussion on frequency techniques with work done on additional metrics.
5	25/03/22	Jean-Thomas Acquaviva	DDN	internal review and subsequent edits.
6	25/03/22	Ahmad Tarraf	TUDA	Added future work to frequency techniques.
7	28/03/22	Francieli Boito	Inria	Updated plots and discussion in Section 3.2.1.
8	02/04/22	JT. Acquaviva	DDN	Additional references and reviews.
9	17/04/22	Konstantinos Chasapis	DDN	Review.
10	20/04/22	Nafiseh Moti	JGU	Some updates to the Section 2.3.2
11	22/04/22	JT. Acquaviva	DDN	Integration of additional review comments

Executive Summary

This document contains certain parts that are currently undergoing a double-blind review process for the Supercomputing 2023 conference. To ensure fairness and impartiality in the review process, we kindly request that you refrain from publicly distributing this document until after the review period, which is expected to conclude around June 2023. This measure is essential to maintain the anonymity of the review process, which is a fundamental aspect of the conference.

The ADMIRE project is dedicated to implementing a feedback loop between measure and control. This deliverable, D5.3 covers WP5's efforts towards the implementation of an actionable measurement system dedicated to the whole project. In particular, a common challenge when considering performance data is data valorization. Indeed, and as covered more extensively in D5.4, documenting the measurement chain, we can generate a verbose set of machine-wide measurements. However, processing such data in real-time, or at least productively in a meaningful time for malleability can be challenging. As such, this document describes the data models we have unfolded to (1) understand I/O behaviour and (2) project measurements to meaningful metrics for other work packages. To this purpose, this deliverable first present the means we have deployed for I/O modelling, setting our efforts to shape a general understanding of I/O. Then, in the second part, we describe various models provided to the projects, with a particular focus on Extra-P. Eventually, the modelling section concludes with a highly innovative approach for performance modelling, namely: projecting measurements in the frequency domain. Thanks to these coordinated efforts for performance data projection, this deliverable demonstrates the availability of a structured measurement and *modelling* chain for ADMIRE.

Contents

Chapter 1

Introduction

The ADMIRE project aims at defining a new way of addressing resources in supercomputers. First, by defining a holistic manner of targeting parallel resources, including I/O inside the configuration parameters. And second, by defining a centralized component in charge of *automating* the job configuration. These two approaches combined lead to an entirely new way of expressing parallel execution. ADMIRE based its approach on the ability to monitor system-wide and in real-time the evolution of the pressure on multiple shared resources. The ability to collect, with a limited overhead, all the information, to prioritize them and to funnel them to the central orchestration component is one of the core contributions of WP5.

1.1 On the Current Usage of HPC Resources

To further contextualize the challenges associated with this work, one can consider how current HPC-class resources are being used:

- payloads are optimized for *time-to-result* (TTR) and not, for example, for improved efficiency.
- the understanding of the overall scaling behaviour of the application is often diffuse and only driven by the TTR.
- programs are generally bulk-synchronous, leading to large static allocations and therefore (1) potential resource wastes and (2) inefficient use of backfilling.
- the crucial Admhal law (impact of serial phases in parallel applications) can only be captured with fine grain profiling.

Witnessing these challenges, ADMIRE aims at defining a new way of running parallel programs, first by delegating configuration to the Intelligent Controller (IC) whose role is to find the suitable parameters for a given program. And second, by controlling the complete I/O chain to fine-tune it accordingly to the foreseen load. This scheme is what we call *smart-scheduler*, and ADMIRE as a whole is an implementation of such a new component. This document aims at describing our work on describing the measurement chain in the project with a particular focus on I/O modelling. Indeed, and as we will further unfold in the next pages, implementing the features we have just showcased is not only a matter of measure but also of performance prediction.

1.2 Plan Recall

In the rest of this document, we will first cover the feedback loop involved in the project, outlining how it can be parametrized with a particular interest in I/O modelling. As the second step, we will introduce our developments for synthetic I/O modelling and how they allowed us to grasp important I/O parameters. Eventually, we will conduct our scaling study on more practical cases, presenting how we articulate the various models we have devised from the initial studies before concluding on planned actions.

1.3 On the Role of Performance Modeling in ADMIRE

ADMIRE is all about implementing a feedback loop between instrumentation and execution. Regarding Instrumentation, in order to be able to observe the parallel behaviour of an application, ADMIRE deploys a complete and suitable measurement infrastructure. Execution is related to tuning the parameters of the applications, either at launch time (moldability) or during run-time (malleability). This process, illustrated in figure 1.1, requires the design and implementation of heuristics and corresponding classifiers to perform the *smart-scheduler* duties. There is a general assumption in scientific computing that applications can be sorted into a small number of categories. This assumption is reflected by various initiatives such as the Unified European Application Benchmark Suite [1] which tries to capture the HPC workload with a set of 13 applications, or the Berkeley Dwarfs with a list of numerical kernels [2]. Interestingly enough, the Berkeley Dwarfs originally isolated 13 key algorithms. This assumption is only backed by partial and empirical observations, an ambition of ADMIRE is to *measure* code diversity with systematic and automated classification.

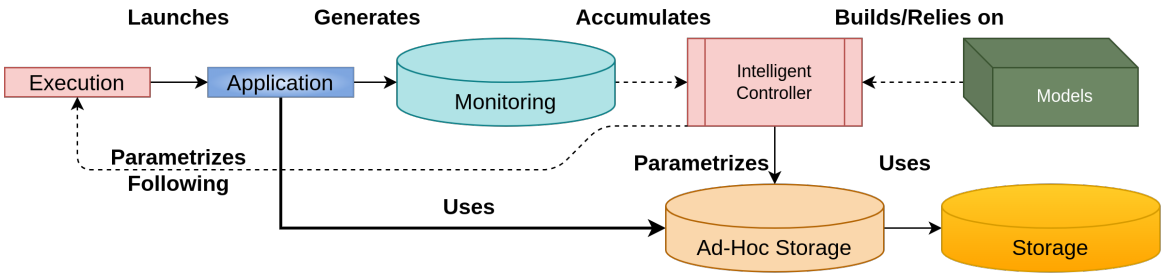


Figure 1.1: Overview of the ADMIRE feedback loop.

In the context of WP5, our role is to (1) capture data and structure it in a scalable manner (as exposed in D5.4) and (2) expose actionable models for I/O and general application scaling. This second aspect, related to models and extrapolation of these models, is the core of this Deliverable. The modeling effort undergone in ADMIRE aims to predict the performance of an application at any scales based on a limited set of measurements. Therefore, ADMIRE value proposition is to build an educated guess using a history of applications' behaviour at multiple scales and diverse configurations. Out of an analysis of this history, we build a *compact* model expressing how programs behave and estimate their resource requirements. This model is then forwarded to the Intelligent Controller (IC) whose role is to implement the *control* part of the feedback loop.

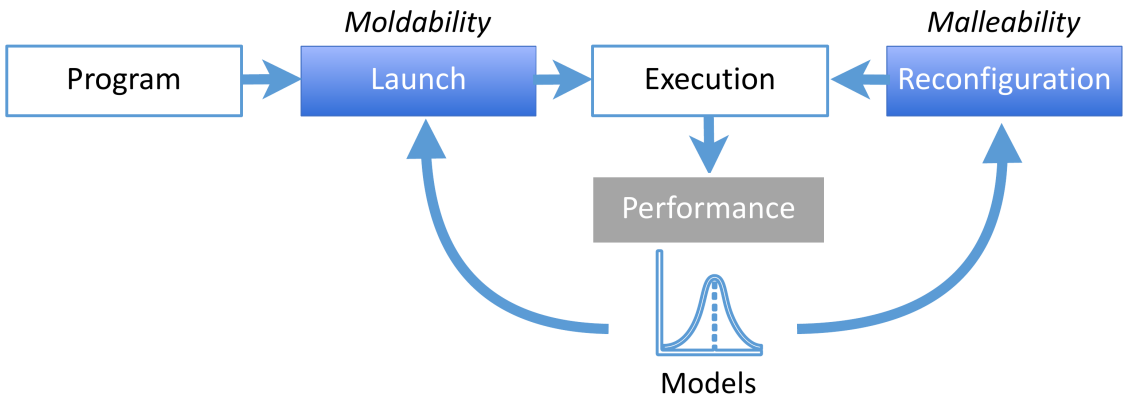


Figure 1.2: ADMIRE's feedback loops for moldability and malleability.

Figure 1.2 presents a simplified vision of the articulation between the components presented in figure 1.1. Within WP5, the goal is to unfold or rehydrate models in order to produce either configuration (moldability) or reconfiguration (malleability) decisions, or both. Considering the potential impact of mis-modelling issues, a correct definition and implementation of these models is important to achieve the planned goals of the project.

[1] https://repository.prace-ri.eu/git/UEABS/ueabs/

To minimize risk, ADMIRE is following an Agile and iterative strategy, starting with an initial analysis of the I/O themselves, and secondly moving on to more practical modelling at the application level.

Chapter 2

I/O Modelling

I/O performance is a critical factor that affects the performance and scalability of high-performance computing (HPC) applications. This is especially acute with modern applications which tend to rely more heavily on data, either to feed as necessary input to produce science or as results. The File System (FS) in HPC is shared among all the compute nodes, and applications use the file system by bursts [1]. To this extent, the file system is one of the very few resources shared among the whole system and subjected to bursts of activities, both factors are prone to create contention, making I/O performance modelling and benchmarking essential to optimize HPC workloads. Proper I/O modelling can help to identify the potential bottlenecks and performance issues in the I/O subsystem and assist in choosing the appropriate storage configuration and FS for a given application. Overall, proper I/O modelling and benchmarking improve the performance and scalability of HPC applications by optimizing the I/O subsystem and identifying potential bottlenecks. In this chapter, we present our different efforts to understand and project I/O behaviours to be able to understand and project them for use in the general ADMIRE infrastructure. The objective of this modelling effort is to quantify systematically the key characteristics of applications. These characteristics are usually informally captured as expertise by end-users, such as rule of thumb associated to the limited amount of I/O generated by CFD codes, or at the opposite by the challenging I/O patterns attached to 3D-Mesh simulation.

We first start with an in-silico model of the I/O subsystem based on a simple flow analogy. In the second part, we present our practical benchmarking infrastructure which is used to characterize the system's response. We then finish this chapter by describing how we leverage the roofline model to determine the correct I/O back-end to be used. Note that we made, on intent, a very practical description of the various tools to ensure the reproducibility of what we present. All codes are available online in open-source.

2.1 `iosim`: I/O Simulator

I/O is commonly understood as a continuous data flow, which is a useful approximation to understanding system behaviour but only partially reflects reality. In practice, reading data is often more complex due to caching effects and the location of data, which could be in a local slab, a further cache, or even on tape. Furthermore, metadata can be a bottleneck in parallel file systems, especially for HPC applications optimized for bulk writes that can lead to contention issues related to directory management due to POSIX locking semantics. While the `iosim` simulator can estimate write bandwidth, it is important to note that it may not reflect the system's behaviour for all I/O operations. The simulator assumes a steady data flow, which may not always be the case for read operations, and caching effects can significantly impact read performance. It is then essential to use the `iosim` simulator only for simulating steady-state writes and to consider its limitations.

2.1.1 Architecture of the Simulator

The `iosim` simulator is a tool that can create a sequence of I/O operations on a given machine configuration. This machine configuration can include different levels of storage and their respective bandwidths. The simu-

[1] LIU, Ning, COPE, Jason, CARNS, Philip, et al. On the role of burst buffers in leadership-class storage systems. In: 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST). IEEE, 2012. p. 1-11.

lator can also model multiple data sources, which represent nodes or storage devices with different capacities. Once the simulator is run, it calculates the bandwidth response of the system and provides an estimation of the achievable write bandwidth for the given configuration. A given execution is then driven by two main files:

- a system description file with an extension of ".io", which explains how the storage back-end is structured,
- an operation description file with an extension of ".ops", which defines the size and timestamp of the write operations.

In the following sections, we will provide a detailed explanation of each of these files, including the capabilities of the simulator's descriptive language. After this, we will present the experimental results obtained with the simulator.

2.1.2 Simulator Installation

The source code of the simulator is open-source and publicly available on gitHub. One may install the simulator we present in this section by doing the following commands:

```
git clone git@github.com:besnardjb/iocache.git
cd iocache
pip install .
```

2.1.3 Storage Description File

The storage description file is a JSON file that defines two ensembles: caches and links. Caches are storage layers that have their own capacity and bandwidth, while links are connections between the storage layers that have their own bandwidth. For simulation purposes, it is assumed that incoming links have balanced Quality-Of-Service (QOS), which means that I/O capabilities are equally shared. Similarly, outgoing links are considered equal, but their individual bandwidths are still taken into account.

Here is a sample description file:

```
{
    "caches": {
        "burst" : {"bw": "1G", "size": "10G"},
        "storage" : {"bw": "500M", "size": "1T"}

    },
    "links": {
        "a_local": {"bw":"1G", "to": "burst"},
        "b_local": {"bw":"1G", "to": "burst"},
        "c_local": {"bw":"1G", "to": "burst"},
        "burst_stor": {"bw":"1G", "from": "burst", "to" : "storage"}
    }
}
```

Here we have a two-level cache with first a burst buffer and second a main storage with respective sizes of 10GB and 1TB. As far as the links are concerned, there are three inputs which are depicted as "named links": `(a,b,c)_local` each with 1BG/s of bandwidth. Eventually, the cache is connected to the back-end storage with a single 1GB link. The `iosim` program has the capability of generating the grapviz output (and then transitively a visual representation) of the corresponding graph as presented in figure 2.1 using the command `iosim -c ./burst.io -d out.dot`.

It is of course possible to devise more convoluted scenarios, for example, to account for in-memory caching, the following description proposes to add 5GB of high bandwidth storage close to inputs, leading to the configuration presented in figure 2.2.

```
{
    "caches": {
        "burst" : {"bw": "1G", "size": "10G"},
        "storage" : {"bw": "500M", "size": "1T"},
        "a_localcache" : {"bw": "10G", "size": "5G"},
```

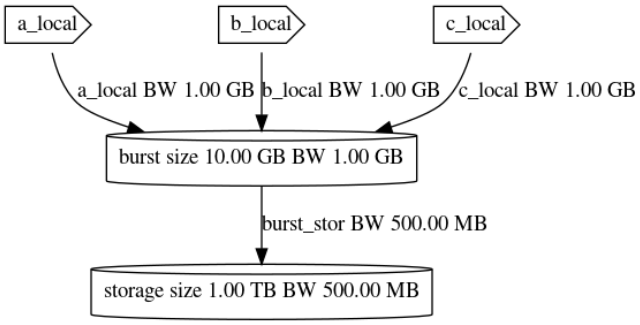


Figure 2.1: Grahviz output for the previous IO configuration description.

```
        "b_localcache" : {"bw": "10G", "size": "5G"},
        "c_localcache" : {"bw": "10G", "size": "5G"}

    },
    "links": {
        "a_local": {"bw":"1G", "to": "a_localcache"},
        "b_local": {"bw":"1G", "to": "b_localcache"},
        "c_local": {"bw":"1G", "to": "c_localcache"},
        "a_localcache_burst": {"bw":"1G", "from":"a_localcache", "to": "burst"},
        "b_localcache_burst": {"bw":"1G", "from":"b_localcache", "to": "burst"},
        "c_localcache_burst": {"bw":"1G", "from":"c_localcache", "to": "burst"},
        "burst_stor": {"bw":"1G", "from": "burst", "to" : "storage"}
    }
}
```

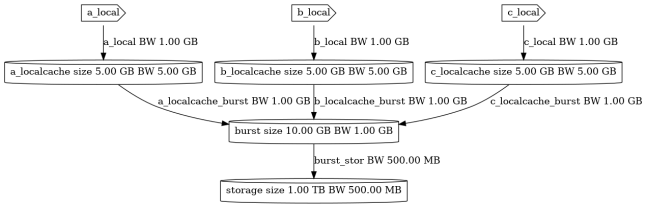


Figure 2.2: Grahviz output for a two-level cache configuration.

The two-level storage configuration is expected to deplete the higher-level caches as I/O operations take place, with any performance issues ultimately attributed to the lower bandwidth of the following layers of the system. Overall, the Storage Description File instantiates a flow graph describing the infrastructure. This flow graph has to be runs against a workload in order to produce a result. Before examining the experimental results, however, it is necessary to first understand how I/O operations are defined.

2.1.4 Operation Description File

We have seen how I/O resources were laid-out, now we need to apply I/O operations to these links. To do so, a second JSON file is needed. This file describes the volume of data to write on each link. Each operation has a size and a timestamp. The operation is considered to be running as long as the sink has not consumed the data to be written on the link (they behave as asynchronous operations). An example of operation description file:

```
{
    "reference":"storage",
    "tick": 0.5,
    "ops": [
        ["a_local", "5G", 0.0],
        ["b_local", "10G", 3.0],
        ["c_local", "30G", 20.0]
    ]
}
```

We first define reference storage. This storage unit defines the stop point of the simulation when all bytes will have "flowed" into this given cache. By construction, it is the lower-level storage layer. Second, we have an array of operations. These operations simply define a link on which to inject data, a total operation size, and eventually a timestamp of where it takes place. With all these inputs we are now ready to look at a sample output of the simulator.

2.1.5 Two-Level Cache Simulation

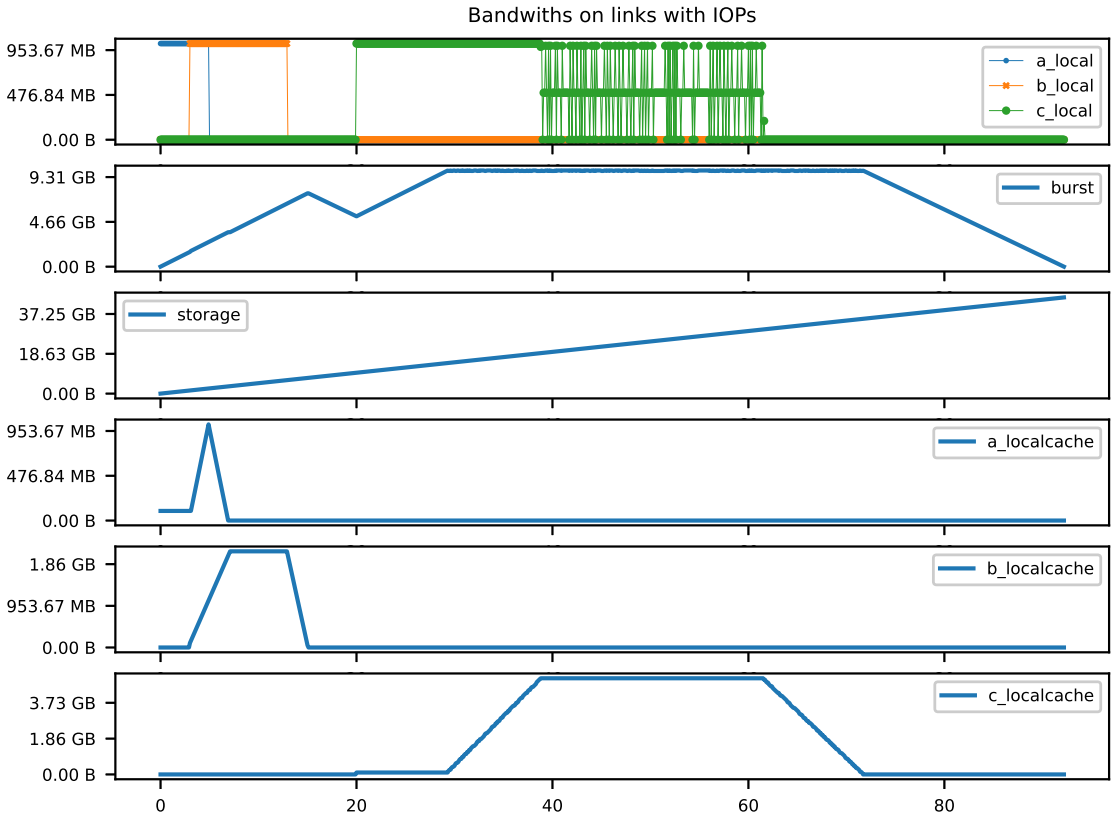


Figure 2.3: Output for simulating architecture of figure 2.2 with I/O operations of previous section.

Figure 2.3 presents the simulation output for the layout of figure 2.2 to which the operations of the previous section were applied. The simulation was run as follows:

```
iosim -c ./21.io -s ./sim.ops -o out.eps
```

This graph shows as blue lines the current occupation levels of the various caches. Whereas, the top-level graph shows the bandwidth on the individual input links. Here we can observe three different I/O behaviours. The first operation on `a_local` is completely amortized by the burst buffer. The second operation on `b_local` however depletes the buffer and requires flushing in the second level buffer. The third operation is the most complex on `c_local`, it does consume the capacity of the two caching levels and therefore leads to degraded bandwidth of 500MB/s on the link, corresponding to the lower bandwidth of the lower storage system. Moreover, note how between T0+20 and T0+30 the second level of burst buffer was able to maintain bandwidth, emptying the L1 cache as it was filling at the same pace, this up to saturation where the L1 storage filled up until blaming the lower-level storage bandwidth to the input.

The goal of this simulator is to capture simple steady write behaviour, possibly in convoluted configurations.

In addition, one important point to note is that an in-memory cache implemented inside an ad-how file system may lead to caching behaviour as depicted in previous sections.

2.2 `iocrawl`: Self-Instrumented I/O Benchmark

This simple benchmark is intended to simulate the probe output from the ADMIRE framework when running on known I/O patterns to enable model calibration. The benchmark was written using the following constraints:

- trivial I/O patterns
- no O_DIRECT to run as an application would (most of the time)
- system configuration extraction and storage in output
- can be easily launched as a one-liner unlike some others

Its main interest is to generate time series of the overall I/O behaviour including on multiple nodes. To do so the benchmark relies on MPI to span numerous nodes. This benchmark does not have other dependencies. For now, only the POSIX interface is tested:

- high-level I/O (FILE*)
- file descriptors (read/write)

2.2.1 Measured I/O Patterns

The following patterns are evaluated, mostly in a steady state. Moreover, all the MPI processes have their independent file. The benchmark will manipulate by default volumes of data equivalent to half of the overall memory on the system (possibly over multiple nodes). This allows the benchmark to calibrate itself while not taking too much time to run. The environment variable `IOC_MEM_RATIO` can be used to alter this behaviour. For example to manipulate three-time the RAM: `export IOC_MEM_RATIO=3`. Conversely to manipulate 10% of the RAM `IOC_MEM_RATIO=0.1`. This will naturally play an important role in terms of local (or remote) cache exhaustion. This size bound to this memory ratio will be called `payload size` in the rest of this section.

2.2.1.1 Massive Write

This benchmark writes the total payload size using 1MB buffers and calls either `fwrite` or `write` in a tight loop until overall data is written. The file size is split over MPI process in an even manner (e.g. If 100 processes need to write 100 GB, each process will write 1GB). It is run both for FILE* and FD interfaces.

Output probes for FILE* are the following:

- **massive_write_FILE::data_written**: bytes written over time
- **massive_write_FILE::close_time**: time spent in close over time
- **massive_write_FILE::open_time**: time spent in open over time

Output probes for FD are the following:

- **massive_write_FD::data_written** : bytes written over time
- **massive_write_FD::close_time** : time spent in close over time
- **massive_write_FD::open_time** : time spent in open over time

2.2.1.2 Periodic Write

Periodic write is also writing the total payload size evenly split over N MPI processes except that this writing is evenly split into 10 buckets separated by a defined number of seconds. The goal of this benchmark is to simulate the effect of periodic I/Os. Therefore if 100 MPI ranks manipulate a total payload size of 100 GB, each rank will write 1 GB total in 10 bursts of 0.1 GB each. These bursts are measured with a variable period of 1,2,4,8 and 16 seconds. Moreover, this benchmark is repeated for both FILE* and FD interfaces.

Output probes for FILE* are the following:

- **nperiodic_write_FILE::close_time** : time spent in close over time
- **periodic_write_FILE::open_time** : time spent in open over time
- **periodic_write_FILE::data_written_1s**: bytes written over time for 1-second bursts
- **periodic_write_FILE::data_written_2s**: bytes written over time for 2-second bursts
- **periodic_write_FILE::data_written_4s**: bytes written over time for 4-second bursts
- **periodic_write_FILE::data_written_8s**: bytes written over time for 8-second bursts
- **periodic_write_FILE::data_written_16s**: bytes written over time for 16-second bursts

Output probes for FDs are the following:

- **nperiodic_write_FD::close_time** : time spent in close over time
- **periodic_write_FD::open_time** : time spent in open over time
- **periodic_write_FD::data_written_1s**: bytes written over time for 1-second bursts
- **periodic_write_FD::data_written_2s**: bytes written over time for 2-second bursts
- **periodic_write_FD::data_written_4s**: bytes written over time for 4-second bursts
- **periodic_write_FD::data_written_8s**: bytes written over time for 8-second bursts
- **periodic_write_FD::data_written_16s**: bytes written over time for 16-second bursts

2.2.1.3 Read Self

Self-read consists in writing the payload size evenly split between MPI processes (as massive_write) and then the data is read back (locally from the same file). This is done with the FILE* interface.

Output probes for FILE* are the following:

- **read_self_FILE::data_read** : bytes read over time

2.2.1.4 Metadata Stress-Tests

Metadata is an important performance factor in parallel systems, and in general, it is one of the weak points in the infrastructure. Our benchmark also targets this component by running the repetitive creation of many files, and their deletion, measuring individual costs. These measurements are done per process in separate directories.

This then leads to the generation of the following probes:

- **metadata_many_files_flat::file_create** number of files created over time
- **metadata_many_files_flat::file_delete** number of files deleted over time

2.2.2 Installation Process

The benchmark consists of a measurement program only depending on an MPI implementation (i.e. mpicc). It can be compiled as follows:

```
wget https://france.paratools.com/ADMIRE/iocrawl.tar.gz
tar xf iocrawl.tar.gz
cd iocrawl
mkdir BUILD
cd BUILD
../configure --prefix=\textdollarHOME/iocrawl CC=mpicc
make -j8 install
# iocrawl should be located at \textdollarHOME/iocrawl/bin/iocrawl
```

In complement, the source tree contains a plotting program depending on python3, it can be installed for the current user as follows (considering you are in the root of the iocrawl source tree):

```
cd ./plot/
python3 -m pip install --user .
# You should have iocplot in your path.
```

2.2.3 Executing the Benchmark

The benchmark is straightforward to run, it only takes a single mandatory argument which is the target directory to measure. This directory should be writable to enable the benchmark to run.

For example:

```
iocrawl /mnt/nfs/
iocrawl \textdollarHOME
iocrawl /mnt/ssd/
\ldots
```

As it is an MPI program, it can be run on several MPI processes and also on multiple nodes:

```
mpirun -np 32 iocrawl /mnt/nfs/
```

For more advanced usage the benchmark can take command-line parameters:

```
iocrawl -rwbm -s [SIZE in percent of MEM] -o [RESULT DIR] [TARGET DIR]
Environment:
  - IOC_MEM_RATIO=0.5 (default) percentage of memory to write/read
  - IOC_OUTPUT_PREFIX=XXX (default '') string to put before output files
Arguments:
  -c  do not wait between benchmarks for cool down
  -C  cool down duration (it is also the burst period)
  -r  do not run read benchmarks
  -w  do not run write benchmarks
  -b  do not run burst benchmarks
  -m  do not run metadata benchmarks
  -s  problem size as a percentage of memory. 1.0=total memory, default=0.5
  -p  how to prefix the results from the run
  -h  show this help
```

2.2.4 Post-Processing Multiple Outputs

A companion program called `iocplot` is provided to process the benchmark results. This program's first use is to gather multiple output files into a single one. For example, if you have run the benchmark on both NFS and NVME and that you would like to plot them together you first need to merge them in a single file as follows:

```
# Note the TAG
iocplot -m NFS:nfs.json NVME:nvme.json  -o merged.json
```

If both `nfs.json` and `nvme.json` contain **massive_write_FILE::data_written** the tags (`NFS` and `NVME`, arbitrary values) will separate them in `merged.json` as **NFS::massive_write_FILE::data_written** and **NVME::massive_write_FILE::data_written** facilitating later processing.

2.2.5 Output Format

Each run will generate a JSON file for each node and one which sums up all the contributions from the nodes suffixed with "merged". These files do contain the information relative to the machine where they were run and to the target system, enabling them to be self-descriptive. As far as measurements are concerned, they are simply arrays of (x,y) data points.

```
{
    "SERIE": {
        "config" : {
            #Information on the system
            ...
        },
        "data" :
        [
            # Data-points
            [ TIMESTAMP, VALUE],
            [ TIMESTAMP, VALUE],
            ...
        ]

    }

}
```

And here is a true example:

```
{
  "periodic_write_FD::data_written_2s": {
    "config": {
      "node_count": 3,
      "mpi_count": 12,
      "fstype": "nfs4",
      "mountpoint": "/media/raid",
      "free_space": 9568960643072,
      "numa_count": 3,
      "socket_count": 3,
      "core_count": 24,
      "total_mem": 50224680960
    },
    "data": [
      [
        0,
        209270000
      ],
# ...
      [
        6.4227,
        691772000
      ]
    ]
  }
}
```

2.2.6 Plotting Results with `iocplot`

The most interesting part of the iocrawl benchmark is of course the ability to directly plot performance results, this while realizing some direct data transformations such as derivation and density maps. As our work unfolded, this tool became a relatively capable I/O analysis component:

```
*usage: iocplot [-h] [-p] [-g] [-L] [-j] [-t] [-l] [-i [INPUT ...]] [-s [SERIES ...]] [-a] [-f
   (cont.)FILTER]
                [-S] [-A AVERAGE] [-D] [-m [MERGE ...]] [-o OUTPUT]
```

```
IOCrawl data-processor.

optional arguments:
  -h, --help            show this help message and exit
  -p, --plot            Plot given series
  -g, --gather-plots    Plot on the same plot
  -L, --ls              Plot in logarithmic scale
  -j, --json            Output the given series in JSON
  -t, --text            Output the given series in text format
  -l, --list            List available series in the file
  -i [INPUT ...], --input [INPUT ...]
                        Input iocrawl file
  -s [SERIES ...], --series [SERIES ...]
                        Series to be processed (add D: before to derivate)
  -a, --all             Load all series in the file
  -f FILTER, --filter FILTER
                        Select series using regular expression
  -S, --statistics      Generate statistics for given series
  -A AVERAGE, --average AVERAGE
                        Apply sliding average of N to the series
  -D, --density         Generate a rate density map of a series
  -m [MERGE ...], --merge [MERGE ...]
                        Merge multiple file together (optionnal tag with TAG:PATH)
  -o OUTPUT, --output OUTPUT
                        File to store the result to
```

We are now going to describe the main use-cases for this program using simple examples.

2.2.6.1 List Series in a File (or several):

Each iocrawl file contains time series identified by a unique name. The `-l` flag allows their listing, it also outputs the corresponding configuration.

```
ioplot -i NFS.json -l
(...)
- nfs_12procs:periodic_write_FD::data_written_16s
   * node_count = 3
   * mpi_count = 12
   * fstype = nfs4
   * mountpoint = /media/raid
   * free_space = 9568960643072
   * numa_count = 3
   * socket_count = 3
   * core_count = 24
   * total_mem = 50224680960
(...)
```

Note that it is possible to process multiple output files at once as long as their series name do not collide. For example:

```
ioplot -i NFS.json NVME.json -l
```

Each entry in the list displays the configuration of the machine used to run the experiment:

- **node_count** : number of nodes
- **mpi_count** : number of mpi processes
- **fstype** : type of underlying file-system
- **mountpoint** : mount point of target partition
- **free_space** : free space on target mount point
- **numa_count** : number of NUMAs in the overall run
- **socket_count** : number of sockets in the overall run

- **core_count** : number of cores in the overall run
- **total_mem**: total memory for the run (sum of the nodes)

2.2.6.2 Plot a Given Metric

Once metrics(s) of interest were identified, it is possible to plot them using the '-p' flag. To select metrics, multiple flags are available, first '-s' takes a list of the metrics' names, it is the simplest selector. Second, '-f' enables series selection through regular expression. Eventually, '-a' selects all series from the file(s).

Here are some examples, first plot one metric:

```
# Select one serie from the file and plot it
iocplot -i ./NVME.json -s nvme_16procs:read_self_FILE::data_read -p
```

Optionally, one can specify an output file to directly generate the graph inside the target:

```
iocplot -i ./NVME.json -s nvme_16procs:read_self_FILE::data_read -p -o nvme.png
```

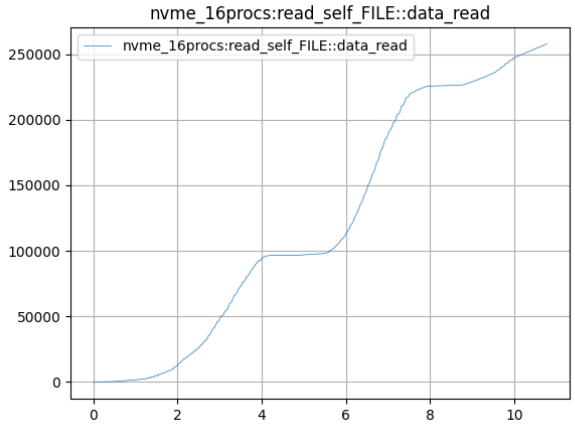


Figure 2.4: Sample output for 16 MPI processes reading from an NVME (cumulative read size).

One can also specify multiple series after the '-s' flag:

```
iocplot -i ./NVME.json -s nvme_16procs:read_self_FILE::data_read  nvme_96procs:read_self_FILE::
  (cont.)data_read -p -o nvme2.png
```

Note that by default, series are on different plots, to gather them use the `-g` flag:

```
iocplot -i ./NVME.json -s nvme_16procs:read_self_FILE::data_read  nvme_96procs:read_self_FILE::
  (cont.)data_read -pg -o nvme2g.png
```

Eventually, it is also possible to plot distributions for a given metric:

```
iocplot -i ./Dropbox/unito.json -D -s 576:massive_write_FD::data_written  -A 100 -o bw.eps
```

2.3 Towards I/O Modelling

In light of the previous tools and measurements, our goal in this concluding section is to define a simple model for I/O in the HPC context. This model should be compact to be easily manipulated by the Intelligent Controller. Meanwhile, it should still be relevant by capturing the most important components of the I/O behaviour at scale on a parallel system. To do so, we propose to start from measurements done with `iocrawl` to then unfold

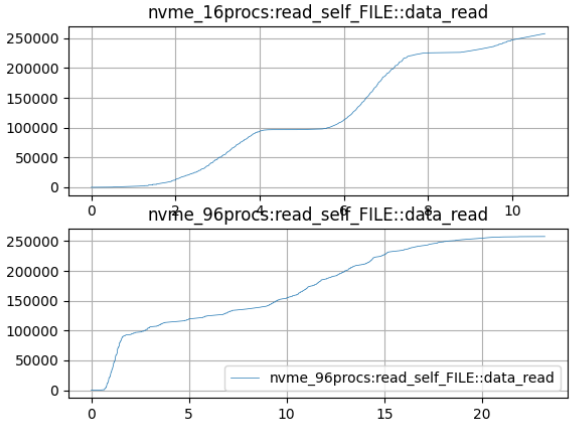


Figure 2.5: Data from Figure 2.4 alongside the same cummulated read measurement for 96 processes.

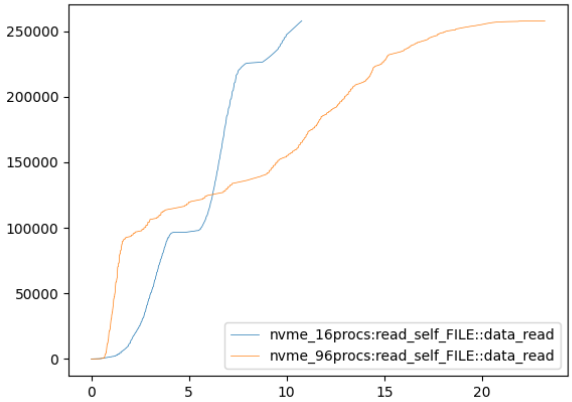


Figure 2.6: Gathering plots from Figure 2.5 on the same graph.

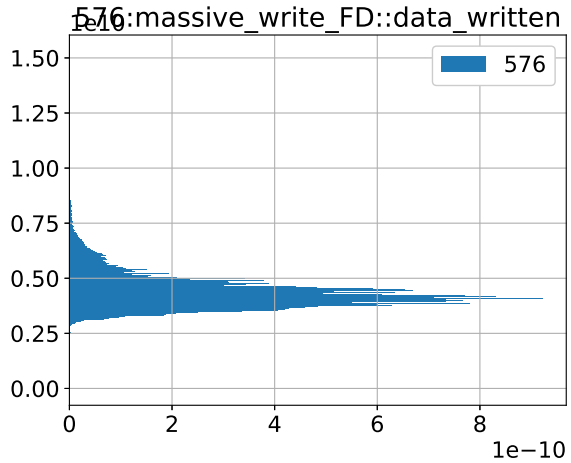
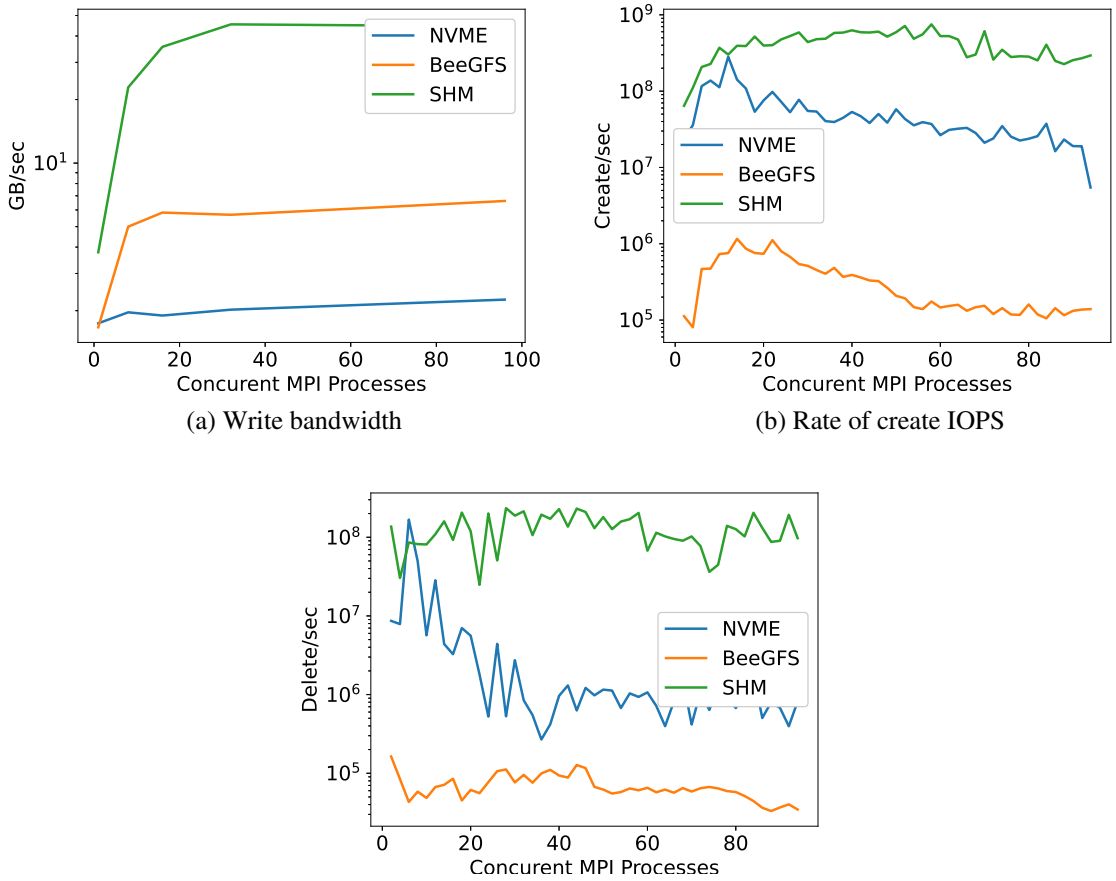


Figure 2.7: Sample output for bandwidth distribution as measured on the UNITO integration cluster.

them to a multi-variadic roofline model that we will use as canvas for resource mitigation in the ADMIRE framework.



(a) Write bandwidth

(b) Rate of create IOPS

(c) Rate of delete IOPS

Figure 2.8: Experimental FS peak performance in function of MPI processes.

2.3.1 Experimental Performance Measurement

In HPC, file systems (FSs) are shared resources that are often subject to contention, which can impact their performance. To assess the potential impact of I/O on performance, we relied on *iocrawl* to measure peak performance in terms of bandwidth and I/O operations per second (IOPS). As presented in figure2.8, the peak performance measurements of multiple FSs, including SHM, BeeGFs, and local NVME, were conducted on a dual socket 64-core AMD Milan, and their behaviours were observed to match the roofline models. The bandwidth and IOPS parameters are important for understanding the performance behaviour of I/O subsystems. Bandwidth increases with the number of nodes up to a point where the storage capabilities are saturated, while IOPS experience an increase, a plateau, and often contention due to locking on meta-data operations. These results and observations are derived from a measurement campaign conducted on various file systems, (you may download the data samples by clicking on the corresponding link):

- Lustre on DDN cluster (5.1MB)
- Performance of Western Digital Ultrastar DC SN630 NVMe SSD. (45MB)
- Performance of an NFSv4 server (3.4MB). The network is gigabit ethernet, each node has 16GB of ram. The storage node runs a fours disk storage in RAID 6 and has network bonding enabled on five 1GB interfaces and a local ram of 16 GB.
- Performance of a GPFS storage (52MB). IBM GS4S high-performance file server utilizing the Spectrum Scale storage engine. The ESS server cluster has 96 4TB SSDs providing a total of 250TB of storage. The ESS cluster is linked to the private storage network with 200Gbps of Ethernet bandwidth.

- Measurement on the UNITO integration cluster (288MB). The backend storage is provided by a BeeGFS installation deployed as a single I/O server with 32 SSD organised as a RAID, with 8 SSD per controller channel. It uses 64 I/O workers. BeeGFS runs under buffered mode as the default cache type.

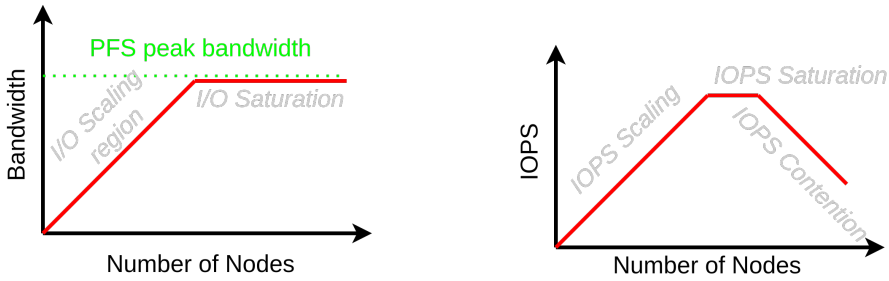
2.3.2 Cross Project I/O Profiling Effort

At the initiative of WP2 and with close cooperation with WP5, a collaboration between EuroHPC projects was established to create an open-access database of I/O performance data for applications running on different parallel I/O libraries and in different layers of the I/O stack. The goal is to help the scientific community identify I/O bottlenecks and design more efficient system software. The study uses profiling and tracing tools selected to require minimal modification to the running setup and be readily applicable in different environmental settings with minimal overhead. The selected tool for profiling the workloads is Darshan, which records statistics about the I/O operations performed by an application. The initiative includes a mailing list and a website to collect the traces and metadata information of the submitted applications. JGU has presented the initiative and received contributions and feedback from IO-SEA. The next steps include extending the study and support to cover the major workloads running on European clusters and leveraging the knowledge of applications into the ad-hoc storage systems' malleability mechanisms.

This collaboration for I/O tracing and analysis among EuroHPC projects is a significant input for the ADMIRE project, which is also focused on I/O scheduling. By creating an open-access database of I/O performance data for applications running on different parallel I/O libraries and in different layers of the I/O stack, the initiative provides a unique opportunity to study the I/O behaviour of representative workloads running on European clusters. This comprehensive analysis of I/O characteristics and requirements of various HPC applications is crucial for designing malleable storage solutions, which is the main goal of the ADMIRE project. By leveraging the results of this initiative, ADMIRE can identify I/O bottlenecks and design more efficient system software that is tailored to the specific I/O requirements of different applications.

More information on this topic can be found in Deliverable 2.3.

2.3.3 Roofline Model



(a) Roofline BW. (b) Roofline IOPS.

Figure 2.9: The two simplified I/O roofline models used for modeling.

Figure 2.9 provides a simplified roofline model that outlines the performance expectations based on previous performance measurements. The model provides a useful framework for understanding how different factors can affect I/O performance in HPC systems. One of the key advantages of this simplified model is its ease of parametrization, which makes it a useful tool for budgeting I/O resources.

One important aspect to note is that IOPS are the most sensitive part of the I/O subsystem. This is because HPC applications tend to be very cautious about the files they manipulate, often relying on a limited number of large files. In contrast, machine learning payloads tend to use many small files. This is often due to the use of higher-level languages, such as Python, which rely on many small files.

To expand on this point, we can consider the differences between HPC applications and machine learning workloads in more detail. HPC applications often require high bandwidth and low latency I/O operations to handle large data sets efficiently. In many cases, these applications operate on a limited number of large files

that are carefully managed to ensure data coherency and minimize contention for shared resources. On the other hand, machine learning workloads may require a large number of small files, which can create challenges for I/O performance.

One factor that contributes to the use of many small files in machine learning workloads is the use of high-level programming languages. These languages often require the use of many small files to store configuration data, code, and other metadata. Additionally, machine learning workloads often rely on distributed file systems that are optimized for handling many small files, such as the Hadoop Distributed File System (HDFS) or the Google File System (GFS).

Overall, understanding the performance characteristics of I/O subsystems and the unique requirements of different workloads is essential for designing and configuring HPC systems. Getting back to the ADMIRE framework translates into the use of tools like the roofline model while carefully managing file systems and I/O operations. By choosing between the standard parallel file system or dedicated ad-hoc storage, our goal is to optimize the performance of HPC systems for a wider range of applications and workloads.

2.3.4 From Measurement to I/O Tuning

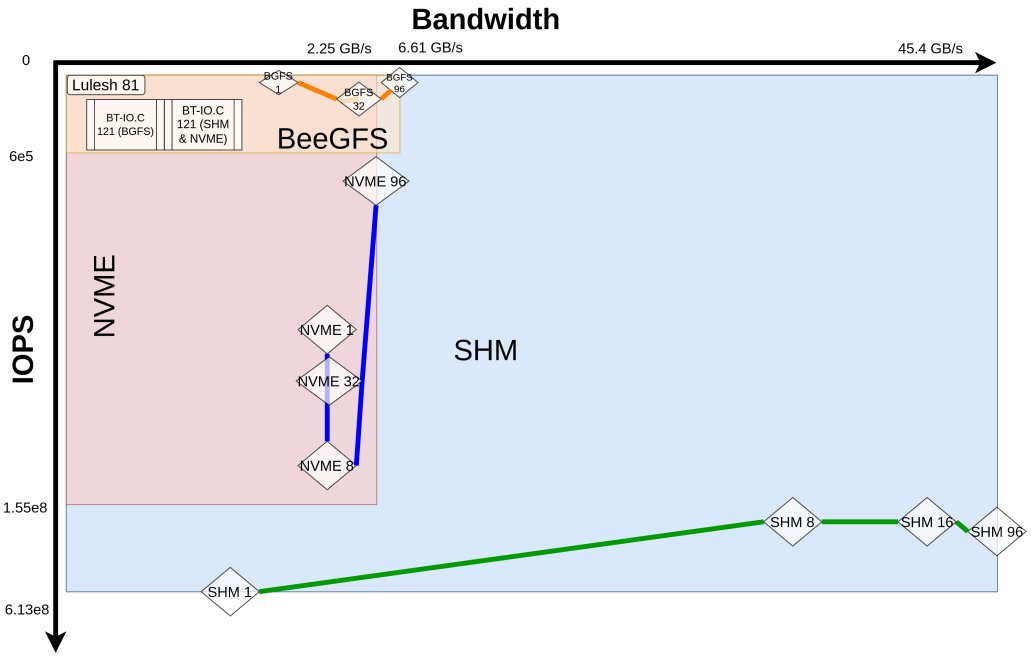


Figure 2.10: Resource saturation diagram over bandwidth and IOPS. All scales are logarithmic. Applications are mapped as per average bandwidth and IOPS.

We have illustrated our I/O parametrization approach in Fig.2.10, which is based on the peak performance measurements shown in Fig.2.8. As we can see from the diagram, metadata operations in BeeGFS, and in most HPC-oriented file systems, have lower efficiency compared to bandwidth. In contrast, the single local NVME has better metadata performance but cannot match the performance of a whole storage array. In comparison, SHM is significantly faster. By providing a practical way to quantify the performance differences between file systems, this diagram enables us to identify the limitations of I/O performance for a given program, whether it is IOPS or bandwidth.

Moreover, we have overlaid the execution coordinates of multiple applications using average bandwidth and average IOPS, creating a combined resource saturation diagram that can be used to measure the sensitivity of a program to I/O. This diagram also reveals that the I/O benchmarks showed variable performance, whereas LULESH (with visualization activated) and BT-IO (class C) mainly remained fixed in this diagram.

To anticipate saturation for a given file system, we are currently using models backed up by Extra-P to project the total dataset size and execution times to compute this mapping, which can guide moldability. We anticipate that machine learning payloads may lead to higher IOPS, leading to patterns diverging from HPC

applications. Hence, our projection models will be particularly valuable in guiding I/O optimization and moldability for machine learning payloads.

Chapter 3

Performance Projection

In the previous sections, we have described how we explored, simulated and measured I/O capabilities on a given system. This was the first step towards building a more elaborate model of I/O this time targeting application. Indeed running a given parallel program leads to an embedding problem where the requirements of the programs are to be satisfied by the back-end storage in its current state. This last point is the complexity of I/O as the current state varies over time depending on the concurrent requirements of other jobs in the machine. ADMIRE is about understanding these requirements while correctly scheduling applications to take them into account. In this section, we will present three means of projecting I/O performance application-wise. First, we describe how Extra-P is able to predict application behaviour thanks to regression analysis. Second, we focus on asynchronous I/O, introducing a dedicated model. Eventually, we introduce an original manner of looking at performance data in the frequency domain.

3.1 Performance Projection with Extra-P

The ADMIRE measurement infrastructure has been designed to store a profile of each running job inside the cluster. The detailed implementation of this infrastructure is part of D5.4. As far as how the performance data are processed, we present here the python consumer interface which was implemented to perform performance modelling on the machine-wide profiles.

3.1.1 `tau_profile_inspect`: probing the performance database

Capabilities for probing the performance database were implemented in the `tau_profile_inspect` command which takes the following arguments:

```
usage: tau_profile_inspect [-h] [-p PROFILES] [-l] [-v] [-g] [-G SELECTGROUP] [-s [SELECT ...]] [-E
   (cont.)EXTRAP]

optional arguments:
  -h, --help            show this help message and exit
  -p PROFILES, --profiles PROFILES
                        PROFILE Path to the profile storage directory
  -l, --list            PROFILE List profile descriptions
  -v, --values          PROFILE List values from the selected profile
  -g, --group           PROFILE Group profiles by commands
  -G SELECTGROUP, --selectgroup SELECTGROUP
                        PROFILE Select a group of commands
  -s [SELECT ...], --select [SELECT ...]
                        PROFILE Select one profile from its jobid to print it
  -E EXTRAP, --extrap EXTRAP
                        PROFILE Path where to store the Extra-P data
```

In practice, the performance database is a simple directory storing multiple profiles which are aggregated from the individual nodes. Querying the database is then simply listing the files in the storage directory. These files are stored in binary to maximize spatial efficiency. It is first possible to list the profiles as follows:

```
tau_profile_inspect -l
```

```
==================
METRICS    : 886
START_TS   : 1679060832
END_TS     : 1679060857
JOBID      : 872480769
SIZE       : 16
COMMAND    : tau_metric_proxy_run -m -s -- ./bin/bt.B.x.mpi_io_full
NODES      :
PARTITION  :
CLUSTER    :
RUNDIR     : /beegfs/home/jbbesnard/NPB3.4.2/NPB3.4-MPI
==================
(...)
```

This will give you the run command, the allocated nodes and the number of stored metrics for each profile, in this case, 886. It is then possible to dump the profile content in JSON by passing in the job-id (note that it is possible to pass multiple jobs):

```
tau_profile_inspect -s 895811585 -v

{
    "2106654721": {
        "tau_hits_total{function=\"mpi_comm_split\"}": {
            "doc": "Number of function calls for MPI_Comm_split",
            "type": "TAU_METRIC_COUNTER",
            "value": 32.0
        },
        "strace_hits_total{scall=\"stat\"}": {
            "doc": "Number of calls for syscall",
            "type": "TAU_METRIC_COUNTER",
            "value": 7167.0
        },
(...)
    }
}
```

What is more interesting in our case is the ability to gather profiles by command line, this enables us to study cross-job scalability and in this particular case, we leverage Extra-P which is designed to do so. First, we list the available execution groups:

```
tau_profile_inspect -g
{
    "5310112b06b1197a638a1be5934e165b": [
        {
            "metric_count": 859,
            "start_time": 1679054666,
            "end_time": 1679054671,
            "jobid": "2114977793",
            "size": 1,
            "command": "tau_metric_proxy_run -m -s -- ./bin/ft.A.x ",
            "nodelist": "",
            "partition": "",
            "cluster": "",
            "rundir": "/beegfs/home/jbbesnard/NPB3.4.2/NPB3.4-MPI"
        },
        (...)
    ],
    (...)
}
```

In addition, once an execution group of interest is identified, it is possible to extract the data towards Extra-P using the JSONL format. This allows the measurement infrastructure to leverage the state-of-the-art capabilities of Extra-P for performance regression. Extraction is done as follows:

```
tau_profile_inspect -G 5310112b06b1197a638a1be5934e165b -E ~/out.jsonl
```

Once done, we can now leverage Extra-P to perform our scaling analysis.

3.1.2 Extra-P

Extra-P is a powerful automatic performance-modelling tool that creates empirical performance models describing an application's scalability behaviour. These models are mathematical formulas expressing performance metrics such as execution time or energy consumption as a function of one or more execution parameters like the size of the input problem or the number of processors. The tool has a long research history and has recently been updated to include noise-resilient empirical performance modelling capabilities for use cases such as Deep Neural Networks and statistical meaningfulness.

To generate these performance models, Extra-P requires repeated performance measurements. It is suggested that at least five measuring points per parameter should be performed, though fewer have been shown to work well in some cases. By profiling an application or continuously passing data from a database like Prometheus, the necessary data for model generation can be collected.

Extra-P supports many input formats, including cube files, text files, and JSON files. JSON lines format is especially useful since it allows for the continuous appending of data, enabling the refinement of performance models whenever new data becomes available. While Extra-P has traditionally focused on computational and communication aspects, there is a growing need to analyze the scalability behaviour concerning I/O as well.

To address this, Extra-P has been extended to generate performance models for various I/O metrics. By generating several performance models, we can judge an application's computational I/O intensity with respect to the number of processors. In the context of a smart scheduler, Extra-P is used to generate offline or online models for all executions on a system, which are then leveraged to guide optimization decisions.

3.2 Performance in the Frequency-Domain: Predicting I/O phases

Characterizing the I/O behaviour of an HPC application is a challenging task. The periodicity of I/O behaviour is one of the most commonly discussed I/O patterns. Informing the system about such patterns can be useful for optimization techniques such as I/O scheduling, burst buffers management, and many more, especially if provided online. I/O analysis is typically performed in the time domain. In this section, we describe a methodology that uses frequency techniques to detect the characterize the I/O behaviour of an application using a single metric, namely the frequency. In this sense, using this metric, scheduling approaches or optimization algorithms can use this metric and respond to expected I/O behaviour.

Our approach uses the discrete Fourier transformation (DFT) to find the frequencies constituting a signal (i.e., temporal I/O behaviour). Before applying the DFT, the continuous signal needs to be discretized. For that, the signal is sampled with a sampling frequency f_s. For the continuous function $x(t)$ sampled with sampling rate T_s $(1/f_s)$ in a time window Δt, N samples are obtained such that:

$$N = \frac{\Delta t}{T_s} \tag{3.1}$$

The N sampled values of for x_n are hence:

$$\{x_0, x_1, \ldots, x_{N-1} \quad | \quad n \in [0, N)\} \tag{3.2}$$

The DFT transforms the evenly spaced sequence (see Eq. 3.2) from the time domain into a sequence of the same length in the frequency domain:

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-2\pi k n}{N} i} \tag{3.3}$$

for the frequency bins $k \in [0, N)$ corresponding to the frequencies:

$$f_k = k \frac{f_s}{N} \tag{3.4}$$

According to the Nyquist criterium, the sampling frequency f_s limits the highest frequency captured by the approach to $f_s/2$. Once the discrete signal is at hand, the DFT can be applied to extract the frequencies. The obtained frequencies can be illustrated in the frequency spectrum. For that the amplitudes:

$$|X_k| = \sqrt{\mathrm{Re}(X_k)^2 + \mathrm{Im}(X_k)^2} \tag{3.5}$$

are plotted usually against the frequency bins k or directly against the frequencies f_k.

To demonstrate this, we executed the IOR [1] benchmark in the collective mode on the Lichtenberg cluster at the university of Darmstadt. We executed the benchmark with two segments, a block size of 10 MB, a transfer size of 2 bytes, and eight iterations with the MPI-IO API. Figure 3.1 shows the obtained frequency spectrum (half spectrum). Note that since the DFT results in a symmetric spectrum for purely real signals, it is enough to show half the frequency spectrum. The amplitudes on the y-axis are coloured according to their contribution to the single. A colour closer to dark blue indicates a poor contribution. while a colour close to red indicates a high contribution. In Figure 3.1, we can observe that the frequency at 0 Hz has the highest contribution to

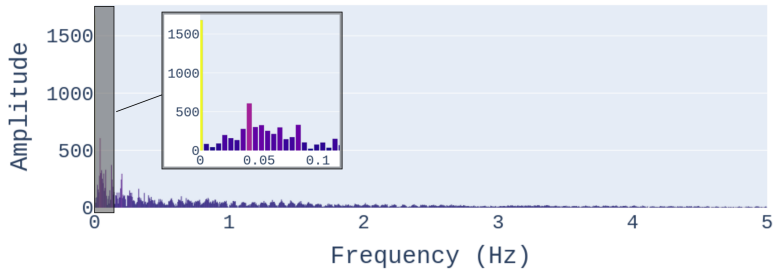


Figure 3.1: Half frequency spectrum from the DFT on IOR with 1536 ranks and f_s = 10 Hz.

the signal. This observation makes sense, as the component at 0 Hz is often referred to as the d.c. offset, and illustrated the average (i.e., average bandwidth in our case). Apart from the component at 0 Hz, to find the periodicity of the I/O signal, the task boils down to finding the dominant frequencies (i.e., the components with significantly higher amplitudes than others). As the task is similar to outlier detection, we use the Z-score to identify dominant frequencies. The Z-score reveals how many standard deviations s an amplitude $|X_k|$ is from the mean $|\bar{X}|$ of the amplitudes:

$$z_k = \frac{|X_k| - |\bar{X}|}{s} \tag{3.6}$$

Thus, for each frequency f_k a Z-score z_k can be found, which specifies how far the corresponding amplitude $|X_k|$ is from the mean of all amplitudes. Usually, a Z-score beyond 3 indicates an outlier. In this sense, if we have just a single outlier (excluding f_0), we have a dominant frequency that specifies the periodicity of the I/O signal. This is just the approach. More sophisticated methods, like filtering the signal to remove short I/O bursts or enhancing outlier detection methods, could be deployed here, which are currently under research. Moreover, as the approach can be performed online during the execution of an application, it can be utilized to predict the I/O periodicity and hence allow us to respond accordingly by diverse I/O strategies (e.g., bandwidth limitation or stag-in/out burst buffers). In this context, adapting the time window Δt allows us to respond to changes in the I/O behaviour, for example, decreasing it to consider only the most recent temporal behaviour inside a given time frame (e.g., 5 minutes back from the current time).

Continuing with the IOR example, the result of our approach is illustrated in Fig 3.2. As shown in blue, the original signal has 8 phases that occur periodically. After the signal is discretized (red), our approach can easily identify the periodicity of a signal which is 0.04 Hz, corresponding to a period of 25 s. As observed in Figure 3.2, our approach delivers a single metric which is f=0.04 Hz, that successfully characterized the I/O behaviour even in the presence of bursts. Note that the sampling is related to the DFT approach and should not be confused with the sampling approaches used to obtain the original signal. A tracing approach usually provides the original signal. Current work involves linking this approach with the monitoring infrastructure and in particular, with the database to deliver a complete solution avoiding unnecessary intermediate formats.

3.2.1 Additional characterization of temporal I/O behavior

For a periodic signal, the DFT finds a dominant frequency that represents it well. However, non-periodic signals may lead to undetermined or simply wrong predictions. Therefore, it is important that our approach provides, in addition to the period, metrics that provide a measure of how periodic the signal is and of the level of confidence in the obtained results.

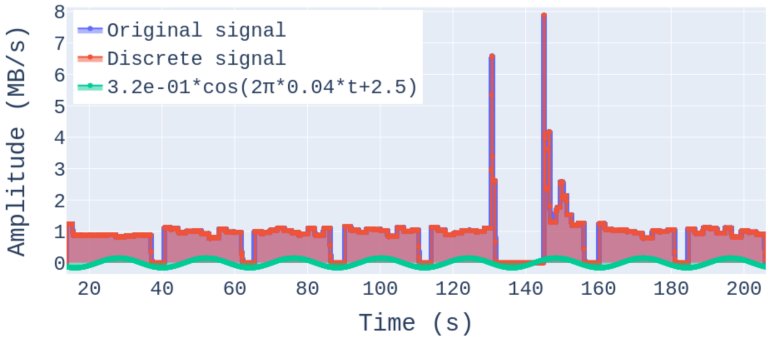


Figure 3.2: IOR on 1536 ranks. After discretization and DFT, the cosine wave that depicts the frequency of the I/O phases is found to be 0.04 Hz (i.e., a period of 25 s).

For the I/O trace $\mathcal{T}$, let $V(\mathcal{T})$ be the amount of data accessed on it (the "volume of I/O"). Given f the frequency identified with DFT, we divide the trace into $L \cdot f$ sub-traces $\mathcal{T}_1, \cdots, \mathcal{T}_{Lf}$ of length $1/f$ and amount of data V($\mathcal{T}_i$) for $i \leq Lf$.

We compute σ_{vol} as the standard deviation of $V(\mathcal{T}_i)$ divided by their arithmetic mean. The lower this value, the more similar the amounts of data accessed per period are, and thus the more consistent the periodicity of the signal is expected to be. This gives an inverse measure of periodicity in *both time and volume* (the lower, the more periodic).

It is important to notice an application could be periodic in time but not in volume: if its I/O phases happen periodically but do not always access the same amount of data. In that case, we could still be interested in analyzing its periodicity *in time*. For that, first we need to determine the periods of time spent on *interesting* I/O. We focus on *substantial* I/O because an application could have low-bandwidth I/O throughout its execution, for example by constantly writing a small log file, but periodic higher-bandwidth I/O phases. In that case, we would like to be able to consider the low-bandwidth activity as *noise* and only count time on the I/O phases. On the other hand, for a signal composed only of the same low-bandwidth "noise" mentioned above, we could want to consider that not as noise but as the I/O behavior of that application.

Let $\mathcal{S}$ be the subset of the time-units of the trace $\mathcal{T}$ where the volume of I/O per time-unit is greater than V/L. To characterize the time spent doing interesting I/O, we define two metrics:

1. an order of magnitude of the bandwidth characterizing the *substantial* I/O time: $B_{IO} = V(\mathcal{S})/L(\mathcal{S})$;
2. the proportion of time spent doing I/O using this order of magnitude: $R_{IO} = L(\mathcal{S})/L(\mathcal{T}), 0 \leq R_{IO} \leq 1$.

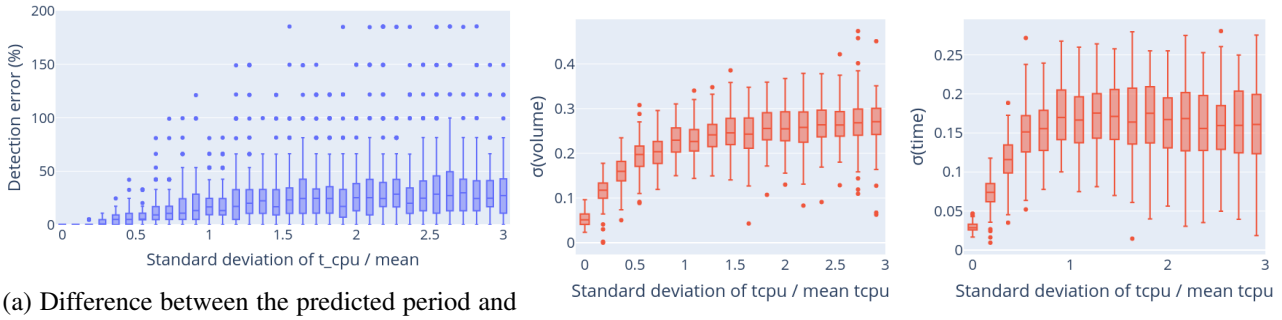
Similarly to $\mathcal{S}$, we let $\mathcal{S}_i$ be the subset of $\mathcal{T}_i$ where the volume of I/O per time-unit is greater than $V(\mathcal{T})/L(\mathcal{T})$. Then

$$\sigma_{time} = \sqrt{\frac{1}{L \cdot f} \sum_{i=1}^{L \cdot f} \left(\frac{L(\mathcal{S}_i)}{L(\mathcal{T}_i)} - R_{IO} \right)^2}$$

In other words, σ_{time} is the standard deviation of the proportion of time spent on I/O inside each period. The intuition is that if signal is periodic *in time*, and the application spends for example 60% of its time on I/O ($R_{IO} = 0.6$), then each of its I/O phases corresponds to approximately 60% of a period. Therefore, the lower the σ_{time}, the more periodic in time the signal is expected to be. Values close to zero for both σ_{time} and σ_{vol} indicate a signal that is periodic and therefore a high confidence in the period obtained with our method. On the other hand, a high σ_{vol} with a low σ_{time} indicates the application is probably periodic in time but not in volume. Finally, high values for both indicate that the signal is probably not periodic and therefore a low confidence on the DFT results.

Figure 3.3 shows results obtained with synthetic traces, with the x axis representing the variability in the time between I/O phases, i.e. the higher the less periodic the signal is. We can see that, despite the variability, our approach is still able to identify a period that is not too far from the actual one for most cases. We can see that both metrics increase as the signals become less periodic, as intended. Finally, in these experiments (100

random traces per x-axis value), the difference between the actual time spent on I/O and our estimated R_{IO} was never above 10%.



(a) Difference between the predicted period and the actual one (%)

(b) σ_{vol}

(c) σ_{time}

Figure 3.3: Results obtained for synthetic traces where I/O phases of 11 seconds are separated by a compute phase of duration $\mathcal{N}(\mu, \sigma)$. The x axis represents $\frac{\sigma}{\mu}$, i.e. the higher, the less periodic the signal is.

3.2.2 Limitations and Perspectives

In the context of ADMIRE, we plan to make this more dynamic and use this information for various components, including the I/O scheduler in WP4 or the Modeling approaches developed in T6.2. Moreover, we plan to examine the scaling behaviour of the prediction with Extra-P, to obtain performance models for the predictions.

Of course, there are several limitations associated with the approach, like sudden I/O changes, which we try to encounter with adaptive regulations of the time window Δt. Other limitations are also associated with the sampling frequency f_s. If this value is specified too low, the approach can miss fast I/O. Moreover, there are also limitations arising from the use of the DFT. While the DFT provides a perfect *resolution* in the frequency domain, it has no information about the time domain. That is, though we know which frequencies are in the signal, we have no clue when exactly they occurred. Though we tried to overcome this limitation by adapting the time window Δt, there are still several limitations associated with this method.

3.2.3 Future work

To solve these problems, an approach could be to use the wavelet transformation. The wavelets transformation [3] overcomes the limitations of the DFT, however, at a price for accuracy in the frequency domain. However, the result of the wavelet transformation can not be as easily analyzed as the DFT. To illustrate this, we run IOR again with the same settings and on the same system as previously. However, this time we set the number of ranks to 9216. The temporal I/O behavior is shown in 3.4.

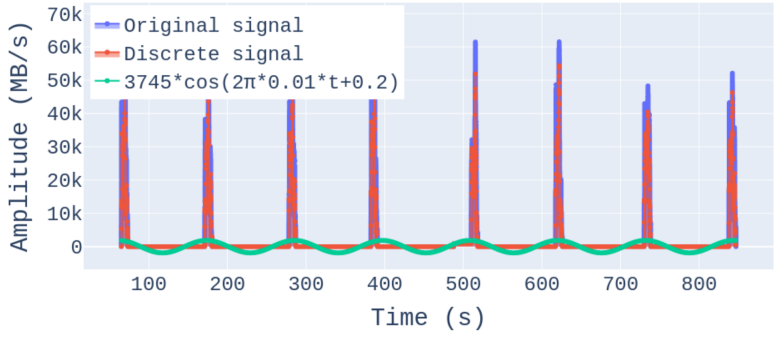


Figure 3.4: IOR on 9216 ranks. After discretization and DFT, the cosine wave that depicts the frequency of the I/O phases is 0.01 Hz.

Figure 3.4 shows, aside from the temporal I/O behavior, the signal that depicts the period in green (period of 0.009 Hz corresponding to a time of 111.57 s). For the same I/O behavior, we applied the wavelet transformation in Figures 3.5 and 3.6. While Figure 3.5 shows the result of the continuous wavelet transformation, the result of the discrete one is shown in Figure 3.6. In both figures, the colors indicate the amplitude of the frequencies at a given time. A tendency to red is an indication that a particular frequency range is more present in a signal, while blue color indicates its absence. As observed in Figure 3.5, not only are we able to see the

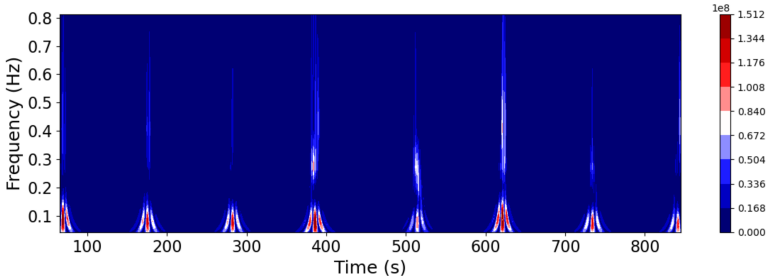


Figure 3.5: Result of the continuous wavelet transformation on the temporal I/O behavior shown in Figure 3.4

frequencies indicated as dominant by the DFT, but we are also able to see when they occurred. However, we only see ranges and not specific values. Moreover, as this is in the continuous domain, the challenge of finding a single period that characterizes the signal is high. Note that there is a trade-off between frequency resolution and time resolution of the wavelet transformation. The higher the time resolution, the lower the frequency resolution. Several parameters, including the decomposition level (here set to 10) can control this trade-off. Interestingly, at the highest frequency resolution, the results of the wavelet are identical to those of the DFT.

An easier approach would be to consider the discrete wavelet transformation. Not only can this method be used to improve the results of the DFT (e.g., filtering high frequencies out), but also as a standalone to compete with the results obtained from the DFT. The result for a decomposition level of three is shown in Figure 3.6. As the sampling frequency was 10 Hz, a decomposition level of 3 states that the wavelet classifies the signal into four chunks: From 5 to 2.5 Hz, from 2.5 to 1.25 Hz, from 1.25 Hz to 0.62, and from 0.62 to 0 Hz. Note that we set the decomposition to three here, as it makes the results easier to visualize. The colour scale on the right of Figure 3.6 shows how to present a particular frequency at a given time is in the signal (i.e., the amplitude of the wavelet coefficients). As seen, compared to the continuous wavelet, the results can be much easier extracted. Moreover, again we are able to the dominant frequency (at 0.01 Hz) as depicted by the DFT is the results from the discrete wavelet clearly.

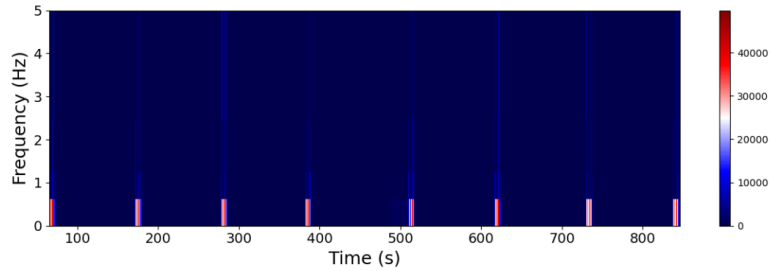


Figure 3.6: Result of the discrete wavelet transformation on the temporal I/O behavior shown in Figure 3.4

We are currently investigating how to use this methodology to extract the information easily and use it to enhance the characterization of the temporal I/O behaviour of an application.

Chapter 4

Conclusion

This report details the modelling effort for WP5, which is closely related to task 6.2 in WP6, also responsible for performance modelling. The boundary between the two efforts was established through extensive discussions resulting in a clear decision. WP5's research focused on understanding I/O system behaviour and defining a potential classification, with compact projective descriptors added, as well as scalability models for both I/O and programs. Additionally, WP5 explored temporal descriptors using the frequency domain, which is innovative in the field of High-Performance Computing and performance modelling. WP6 is collecting these models and descriptors to generate an applicative model, a compact and persistent description of what to expect from a given application that covers all aspects of execution. Given ADMIRE's focus, I/O is of paramount importance. WP6 is responsible for structuring the data and projections provided in this report, which covers both the program and machine capabilities. The intelligent controller's role is to embed one into the other using advanced scheduling capabilities from WP3 and WP4 combined with the aforementioned models and information. As of today, we consider that the monitoring infrastructure is developed and ready to feed the Intelligent Controller, we are in the process of integrating this work on a single machine located at the University of Torino, acting as our integration platform. We expect that these integration efforts will materialise into WP6 which is the central component of the ADMIRE project, with all other WPs acting as providers to achieving the *smart scheduler* pattern. Meanwhile, we will pursue the enhancement of our models while providing active support to enable the correct validation with applications.

The modelling effort will allow to clusterize applications in classes and define the most suitable infrastructure depending on the application portfolio for a given HPC site. We expect such data-driven approaches to have deep impact on the way HPC systems are designed.

Bibliography

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

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research: A view from berkeley. 2006.

[3] A. Graps. An introduction to wavelets. *IEEE Computational Science and Engineering*, 2(2):50–61, 1995.