EuroHPC-01-2019

IO-SEA

IO – Software for Exascale Architectures

Grant Agreement Number: 955811

D1.2

Application use cases and traces

Final

Version:	2.1
Author(s):	M. E. Holicki (FZJ), E. B. Gregory (FZJ), M. Golasowski (IT4I)
Contributor(s):	A. Bischnoi (FZJ), D. Caviedes Voullième (FZJ), D. Chapon (CEA), O. Iffrig (ECMWF), J. O. Mirus (FZJ), Y. Müller, (FZJ), J. Novacek(CEITEC), L. Strafella (CEA), G. Tashakor (FZJ)
Date:	July 20, 2022

Project and Deliverable Information Sheet

IO-SEA Project	**Project ref. No.:**	955811
	Project Title:	IO – Software for Exascale Architectures
	Project Web Site:	https://www.iosea-project.eu/
	Deliverable ID:	D1.2
	Deliverable Nature:	Report
	Deliverable Level: PU *	**Contractual Date of Delivery:** 31 / December / 2021
		Actual Date of Delivery: 21 / December / 2021; resubmitted 29/ July / 2022
	EC Project Officer:	Daniel Opalka

*– The dissemination levels are indicated as follows: **PU** - Public, **PP** - Restricted to other participants (including the Commissions 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).

Document Control Sheet

Document	**Title:** Application use cases and traces	
	ID: D1.2	
	Version: 2.1	**Status:** Final
	Available at: https://www.iosea-project.eu/	
	Software Tool: LaTeX	
	File(s): IO_SEA_D1.2.pdf	
Authorship	**Written by:**	M. E. Holicki (FZJ), E. B. Gregory (FZJ), M. Golasowski (IT4I)
	Contributors:	A. Bischnoi (FZJ), D. Caviedes Voullième (FZJ), D. Chapon (CEA), O. Iffrig (ECMWF), J. O. Mirus (FZJ), Y. Müller, (FZJ), J. Novacek(CEITEC), L. Strafella (CEA), G. Tashakor (FZJ)
	Reviewed by:	V. Kannan (ICHEC) L. Vojáček (IT4I)
	Approved by:	Exec Board/WP7 Core Group

Document Status Sheet

Version	Date	Status	Comments
0.1	30.10.2021	Outline approved	
0.9	30.11.2021	Draft ready for internal review	
1.0	15.12.2021	First internal review complete	
1.1	17.12.2021	First-review edits complete	
1.9	21.12.2021	Second internal review complete	
2.0	21.12.2021	Final draft ready for EU submission	
2.1	01.07.2022	Benchmarking-collaboration update	

Document Keywords

Keywords:	IO-SEA, HPC, Exascale, Software, Benchmarking

Contents

List of Figures

List of Tables

Executive Summary

In this second deliverable of Work Package 1, we describe the effort to build a comprehensive benchmarking suite for the IO-SEA project.

In Part I, we describe the JÜlich Benchmarking Environment (JUBE), which we use to design an automated benchmarking workflow, including organising and archiving results in a remote GitLab repository.

In Part II, we describe a series of synthetic diagnostic benchmarks. We include these in order to maintain a picture of the health and performance of the underlying compute platform. This allows us to more accurately attribute any changes in use-case performance to either system health, code changes, or to new IO-SEA technology developments.

Finally, in Part III, we detail the use case benchmarks. In the previous deliverable D1.1 [1], we identified, how the workflow of each IO-SEA scientific use case will take advantage of technology developed as part of the IO-SEA solution. Here we describe benchmarks specifically targeting use-case steps where we expect these solutions to provide performance enhancements.

Following the recommendations received from the external reviewers during the checkpoint review at M9, we have updated this deliverable in July 2022 with a new section 2.6 that highlights the common benchmarking strategy developed for the DEEP- and IO-SEA projects.

1. Introduction

The IO-SEA project aims to provide novel data management solutions and storage platforms for exascale computing based on hierarchical storage management and on-demand I/O services. The project includes five scientific use cases, both to guide the design of, and test the implementation of these solutions. In order to gauge the effectiveness of the technologies developed in the IO-SEA project, it is important to be able to quantify the performance of the use cases, in particular the workflow steps with heavy I/O demands.

Such benchmarking should capture a number of metrics — not just the elapsed wallclock time — in order to pinpoint changes in performance. Furthermore, the benchmarks should be packaged in such a way that all project partners can run them. Finally, they should be run regularly in an automated fashion to demonstrate improvements over time.

To this end we use the JÜlich Benchmarking Environment (JUBE), which can automate the compilation, batch submission and execution of applications, as well as extract metrics from the output and tabulate the results. In combination with the GitLab CI/CD functionality, the benchmarks can be triggered to run on a regular schedule or when there is a code update.

In the completion of this deliverable and IO-SEA Milestone 2, we have exploited a very natural synergy between IO-SEA and our “sister project”, DEEP-SEA. Both projects have benchmarking needs. DEEP-SEA, in fact, has an extremely similar benchmarking milestone and deliverable. Moreover, there is a large overlap in personnel in the benchmarking tasks of these two projects. While DEEP-SEA benchmarks will focus on somewhat different metrics, and mostly different use cases, the goals are similar enough that a common benchmarking methodology and workflow has been developed. This collaboration is reflected in the complimentary narratives of this report and the companion deliverable, DEEP-SEA D1.2 [2]

Efforts toward this deliverable began at the beginning of the third month of the IO-SEA project. On July 1, 2021, IO-SEA Task 1.2 held a Benchmarking Workshop in cooperation with the EU DEEP-SEA and RED-SEA projects. In the full-day, hands-on workshop participants from all three projects learned to use JUBE. From this starting point the members of IO-SEA Tasks 1.3–1.7 began to set up benchmarks for their use-case applications.

We have prepared two types of benchmarks. First, synthetic benchmarks use standard codes to test a diverse aspects of general system performance. Second, use-case applications are instrumented to record performance metrics chosen to illustrate performance changes in response to developing I/O and data-management technology. In some cases where the use-case workflow contains several distinct steps, multiple benchmarks have been prepared.

At the time of this writing, the IO-SEA project does not yet have a fixed prototype machine for installing IO-SEA solution technology and testing use-case applications. We anticipate this changing in the coming months. Our accomplishment to date is to build and test a comprehensive suite of scripts to run the benchmarking workflows. We provide in each case sample results from a test system to demonstrate functionality. As soon as there exists an IO-SEA prototype platform, we can deploy this suite and begin performing baseline benchmarks, and continue to monitor application performance.

In this report we describe the design of benchmarks to measure use-case-workflow performance, and synthetic benchmarks to measure general system performance. We detail the implementation of these benchmarks in JUBE, and give preliminary results on various test systems.

Part I.

Benchmarking

2. Benchmarking

In computer science, *benchmarking* refers to running software to collect performance metrics on its execution, in order to establish a performance baseline or compare the performance to an established baseline. In this context, performance metrics are measurable quantities used to quantify the performance of a software, or its components. A common metric is the runtime of a software. In the context of the IO-SEA project metrics of interest, for example, are I/O time and bandwidth.

Scheduled benchmarking is a necessary part of the IO-SEA project, ensuring continued improvement in IO-SEA–relevant metrics as changes in hardware and software affect the performance of the use-case applications.

In the IO-SEA project, a combination of synthetic and use-case application benchmarks will be used. The use-case benchmarks will demonstrate how changes to the underlying software as part of the IO-SEA project affect application performance. The role of the synthetic benchmarks is to benchmark the I/O-related hardware and software optimisations made during the project. This allows us to distinguish between performance changes due to changes in the use-case applications themselves, and those due to changes in the underlying hardware and software I/O stack.

To further improve cross-project collaboration between DEEP- and IO-SEA the same general benchmarking framework will be used for both projects, especially for synthetic benchmarks. This means that both projects will directly benefit from benchmarking developments in the other project, which will free up resources for better benchmarking. As a result the synthetic benchmarking sections for this deliverable for both projects are nearly identical.

2.1. The JÜlich Benchmarking Environment (JUBE)

JUBE [3] is a set of Python scripts that facilitate the organisation of benchmarks in the IO-SEA project by providing a consistent framework and keeping track of benchmark runs. It also provides tools to aggregate benchmark results into tables for analysis. We then upload the results to GitLab for easy access for our project partners.

Project partners were introduced to JUBE in a cross–SEA-project JUBE benchmarking workshop on July 1, 2021, with forty participants across the three SEA projects, DEEP-, IO- and RED-SEA. The workshop started in the morning with an introduction to JUBE by its creator Sebastian Lührs (FZJ) followed by a hands on session where participants worked through curated examples. In the afternoon Max Holicki (FZJ) presented how Benchmarking would work as part of the SEA projects and what the benefits would be to all involved partners. Following that participants were aided in setting up initial JUBE benchmarking scripts for their software. All interested participants left workshop with initial prototypes and the workshop was considered a success.

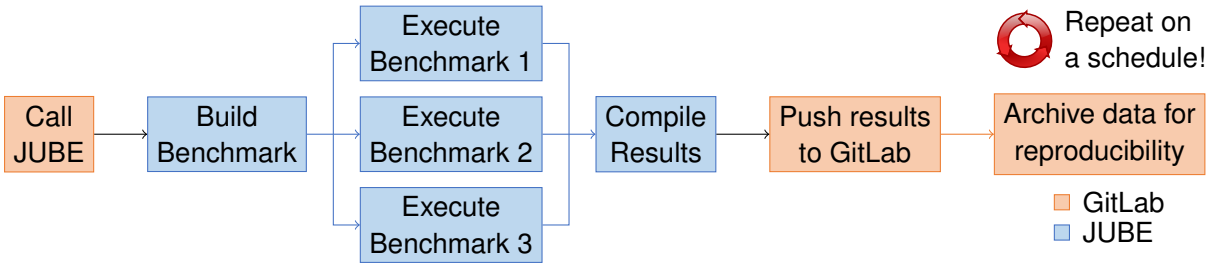


Figure 1.: Workflow illustrating a GitLab runner starting JUBE, which in turn first builds benchmark code and then uses the built binaries to execute three related benchmarks in parallel. Once these are finished JUBE aggregates the results into a table, which the GitLab runner pushes to the relevant repository before archiving the benchmark results.

2.2. The Benchmarking Workflow

Successful scheduled benchmarking as part of the IO-SEA project requires a benchmarking workflow. Our proposed solution is the multi-stage branching workflow illustrated in Figure 1.

The workflow consists of two major components: (1) the GitLab Runner that launches (2) JUBE, which orchestrates the benchmarking run. The GitLab Runner is responsible for executing JUBE, which runs the benchmark run, for pushing the results up to GitLab and for archiving the benchmark run for later reference. JUBE executes the benchmarking run, which generally consists of a build stage and an execution stage. During the build stage, the to-be-benchmarked software is built, alternatively, binaries may be derived from elsewhere. In the second stage, JUBE executes the benchmark, which may consist of multiple steps which run either in serial or parallel. Finally, JUBE can also collect results if desired. To ensure the rapid detection of benchmarking issues in this workflow, emails are sent out to the owners of GitLab Runners if errors occur, for example if a timeout occurs. The owners of the GitLab Runner are then responsible for investigating the problem and restarting the benchmark when possible.

For the use cases, the newest versions will be benchmarked to ensure that performance changes due to code changes are detected as early as possible. For the synthetic benchmarks a fixed version will be used. The version will only be updated if deemed necessary. The reason for this is that updates in the synthetic benchmarks may result in performance improvements that are then incorrectly attributed to changes in the I/O stack.

2.3. I/O Metrics

Performance-defining metrics can broadly be classified into three groups. First are those that are easy to measure with minimal or no changes to a use-case application. Second are those which are more difficult to access, but which specialised synthetic benchmarking codes provide. Finally, there are metrics which are of interest, but are next to impossible to measure without, perhaps, specialised access to hardware components. With this in mind, the I/O metrics of interest as part of the IO-SEA project are:

- Runtime: This is the easiest metric to measure and is simply the elapsed wallclock time of a program's execution. If the software performs I/O operations and these operations become faster during the project, the runtime should decrease. Note that the runtime is not a sufficient metric, though, as it is affected by many other factors. All use cases have implemented this metric.
- I/O time: This metric is simply the time it takes to perform a given I/O operation. This is the metric we wish to improve as part of the IO-SEA project. This metric is straightforward to measure and has been implemented by the use cases.
- I/O latency: This metric is the time spent in an I/O command not actually doing I/O, it includes things like setting up buffers and spinning up drives. This metric should depend only weakly on the amount of I/O to be done by a given command. For very small data volumes this metric dominates the I/O time. This metric is nigh impossible to measure and has not been implemented by any use case nor synthetic benchmark.
- I/O bandwidth: This metric is a measure of how much data can be stored or read by an I/O command per unit time. For large data volumes bandwidth effects dominate the I/O time. Some use cases approximate a lower bound for the I/O bandwidth.
- Network bandwidth: A lot of I/O is not local and occurs over a network. This means that the network bandwidth could be a bottleneck for the I/O-bandwidth, and so should be measured separately to ensure that it is not adversely affecting I/O bandwidth. This will be measured separately using dedicated network benchmarks and, as such, no use cases collect this metric.
- Network latency: This metric is not directly measured and is assumed to be part of the I/O latency. As with the previous item, this is not measured by the use cases.
- PCIe bandwidth and latency: Peripheral Component Interconnect Express (PCIe) is the common communication bus on motherboards for high-bandwidth communication between devices like GPUs or FPGAs, often via the CPU. Therefore, it is hoped that we will find suitable benchmarks for it in this project.

Beyond these I/O Metrics the synthetic benchmarks measure a plethora of other metrics including things like CPU performance to ensure the health of the system and to be able to better correlate observed performance fluctuations with their potential cause.

2.4. Data Management

Benchmarking always generates data of some form. Some of this data is of interest, as it either pertains to establishing a baseline or allows for comparison to a baseline. The additional data generated usually consists of logs and other output that can help diagnose unexpected benchmark behaviour. For example, runtime benchmarking a scientific application may result in three data types: 1) the elapsed runtime of an application, hereafter referred to as runtime, is the ultimate metric in which we are interested, 2) logs describing the system state at the start, during and at the end of the benchmark, including the necessary input data, and 3) the scientific output of the benchmark, which is not of direct interest to the benchmarking effort. The most valuable data in the benchmarking context is the runtime. As such, after each benchmark run the resulting runtime is uploaded to GitLab in a

commit for safekeeping and easy access to project partners. 2) and 3) are not directly useful for the benchmarking effort. However, they are required for approximate reproducibility given a sufficiently similar benchmarking system. They need also to be stored, but losing this data is not as significant a problem as losing the actual benchmarking data, so we opt to locally archive this data where possible. This ensures that the data can be queried at a later stage for forensic analysis or reproducibility of benchmark runs that exhibit anomalous behaviour. Note that input data to a benchmark should also be stored when relevant if it is not readily reproducible. All data aside from the direct benchmark results are private and it is the GitLab Runner owners responsibility to ensure that these are stored safely. The benchmarks results are public and will be shared among project partners, and may be later published. For more information please see IO-SEA Deliverable 7.5. [4].

2.5. Future Plans

Our goal is to regularly run our synthetic and use-case benchmarks throughout the life of the project once suitable hardware prototypes for the project have been established.

The schedule is to be determined in collaboration with the other work packages to establish a suitable benchmarking frequency and to ensure that the impact on other work is as small as possible.

ATOS will in the future provide I/O logging features. It is of great interest to integrate these into the benchmarks to increase and improve the generated data. We plan to examine the functionality once a benchmarking schedule has been established and a hardware test bed with the feature is available.

We also plan to expand the result visualisation and potentially integrate it into the benchmarks to enable an easier-to-digest access to benchmarking results. Some of the visualisation techniques will be demonstrated throughout this report in the included figures and tables, however, a lot more data than those presented here have been generated, and even more will be generated over the project lifetime. As such result visualisation is necessary for some benchmarks to allow our partners to remain on top of the situation.

2.6. Benchmarking Collaboration

Following the recommendations received from the external reviewers during the checkpoint review at M9, we have updated this deliverable with this new section that highlights the common benchmarking strategy developed for the DEEP- and IO-SEA projects. For both projects benchmarking is a key ingredient. As such, the same benchmarking framework and core team is shared across the Tasks 1.2 of DEEP- and IO-SEA, which both perform the same function in their respective projects. Additionally, LinkTest serves as a common benchmark among the three SEA projects: DEEP-SEA, IO-SEA and RED-SEA.

The DEEP system will also be used as the benchmarking testbed for IO-SEA. This strategy simplifies collaboration and benchmarking between the two projects. Furthermore, it allows for the use of the new Jacamar CI [5] based GitLab runners over traditional GitLab runners that formerly posed a potential security risk to the system. The Jacamar GitLab runners, which are provided by Task 3.6 of DEEP-SEA, will further enhance inter-project collaboration. The use of Jacamar CI will also enable

project partners to execute benchmarks on demand, for example when they have optimized their code and wish to check if the benchmarks can corroborate this.

Figure 2 summarizes the benchmarking collaboration. The Figure also highlights how TSMP, see chapter 14, is unique, in that it is part of both DEEP- and IO-SEA. However, the TSMP use case has a slightly different focus in each project. Nevertheless, the TSMP benchmarks for both projects are similar and organized by the same individuals, which promotes cross-project pollination of ideas.

A consequence of this close collaboration is that the main benchmarking chapter as well as the synthetic-benchmark chapters, Chapters 4—10, in this deliverable for DEEP-SEA and the equivalent IO-SEA are virtually identical. The intention to highlight the extensive sharing of resources between the projects. This also explains why the same synthetic benchmarks were chosen for both projects. Furthermore, this will allow us to detect potential issues that would have previously gone undetected because corresponding benchmarks were missing.

LinkTest is actively used in all three projects and is further developed as part of the RED-SEA project for BXI communication. With the recent on-loan addition of BXI hardware to the DEEP system, we hope that benchmarking on both platforms will lead to cross-project seeding of ideas, and to the establishing of the DEEP system as a potential testbed for all three projects.

The most important advantage of using a uniform benchmarking framework, however, does not lie in the fact that it saves time and effort, but in the fact that in the end the scripts are cross-project compatible. This means that once all use cases have been ported to the DEEP system, one project can execute the benchmarks of another, effectively allowing them to leverage the use cases of the other project. We hope that this will encourage collaboration between different work packages and use cases across the SEA projects.

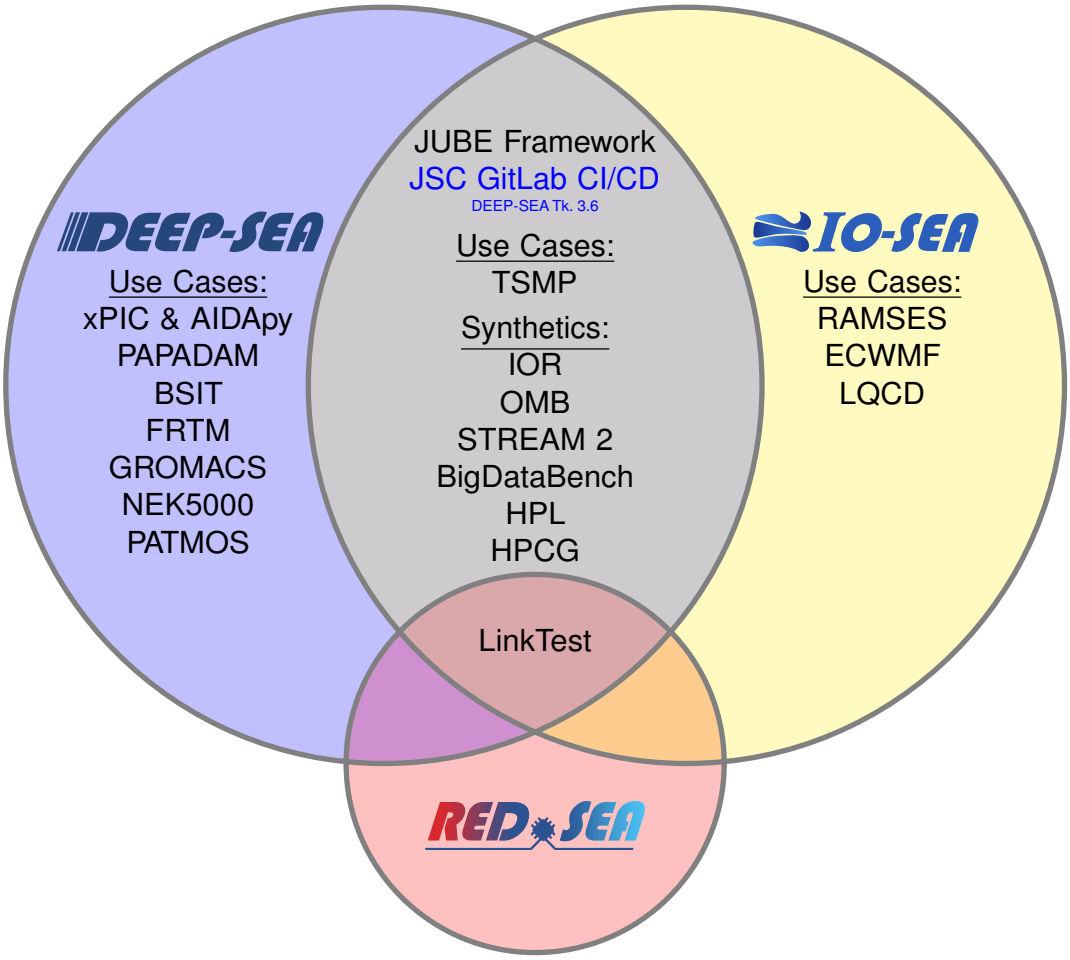


Figure 2.: Benchmarking-collaboration Venn diagram that highlights the benchmarking activities of DEEP- and IO-SEA Task 1.2 and how they are spread out across the three SEA projects.

Part II.

Synthetic Benchmarks

3. Synthetic Benchmarks

The aim of the synthetic benchmarks is to monitor the health of the IO-SEA systems and to help determine if changes in use-case performance are due to changes in the use cases themselves, changes in the underlying IO-SEA I/O software or due to other changes in the hardware or software of the system.

In the following chapters we describe seven synthetic benchmarks integrated into our benchmarking suite. We provide results as examples to demonstrate that the benchmarks execute on generic hardware systems, but also to show the type of data collected and its presentation. In general, the quantitative details are not important at this stage.

As there is not yet a dedicated IO-SEA prototype machine, the synthetic benchmarks were run on several JSC systems (DEEP, JUWELS, JURECA-DC) where we have access.

In particular, our ties to the DEEP-SEA project allowed us access to the DEEP prototype system at JSC. There exists the possibility that a future IO-SEA prototype might be a installed as a module on the DEEP system, though discussions regarding technical feasibility are not complete. For the benchmarks run on the DEEP system (IOR, OMB, STREAM 2, HPL and HPCG) we therefore provide a bit more detail of the results, as it may later be a useful baseline reference.

4. Interleaved Or Random (IOR) I/O Benchmark

The Interleaved or Random (IOR) benchmark [6] allows for the assessment of the I/O performance of parallel storage systems using a diverse range of APIs (e.g., MPI-I/O, GPFS, LUSTRE) and access patterns. Included with IOR is the Mdtest benchmark, which focuses on testing metadata performance, e.g., the creation and removal of directories and files. Taken together, IOR and Mdtest allow for testing the performance of systems with respect to a variety of I/O workloads.

4.1. Description

Parallel storage systems gain their speed from storing chunks of consecutive data on different physical storage devices. This technique is called data striping. Thus, any incoming or outgoing data stream is split respectively into or assembled from smaller chunks, whose size depends on the configuration of the storage system. This data striping fits well with many scientific applications, which in certain time steps write data either for later analysis or for continuing the computations after a failure.

IOR allows simulation of such real-world data loads by enabling the user to configure a data stream by defining (Figure 3):

(1) the block size, which is the amount of data accessed by each process per time step,

(2) the transfer size, i.e. the above-mentioned chunks, which are accessed by a single I/O function call,

(3) the number of segments to access. Here, a segment consists of as many blocks as there are processes.

Apart from the actual data that is read or written, each access also involves metadata, i.e., information about what is accessed, e.g., file name, owner, permissions. Directories, which allow organizing files hierarchically, are also metadata. For the highest performance, modern storage systems manage data and metadata separately on different storage servers. For this purpose, the IOR distribution includes the metadata benchmark Mdtest, which allows measurement of, for example, the creation and deletion of a defined number of files and a hierarchy of directories.

4.2. Results

For our IOR benchmarks (Figure 4) using the parallel API MPI-I/O, we have used the DEEP system cluster module, for more information see the sister-deliverable DEEP-SEA Deliverable 1.2 [2]. The measured average read throughput using four or less nodes of the cluster module demonstrate that the flash-based storage module is capable of saturating the available read bandwidth of up to 100 Gbit s^{-1} per node with a maximum of 600 Gbit s^{-1} for the All-Flash Storage Module (AFSM).

In contrast to the AFSM, the performance of the Scalable Storage Service Module (SSSM) is much lower and peaks at only a single node with a write speed of about 4 GiB s^{-1} and a read speed of 1.3 GiB s^{-1}.

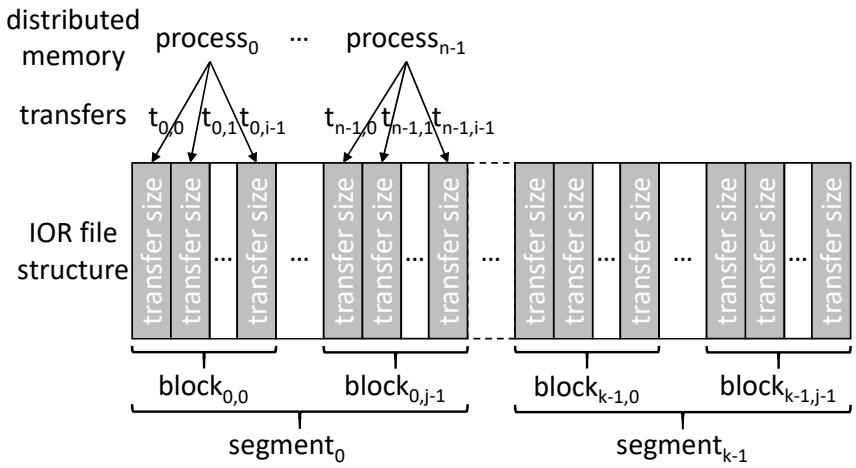


Figure 3.: IOR file structure and access pattern. An IOR data stream consists of one or more segments. Each allocated process in distributed memory accesses exactly one block per segment. Each block consists of either a single or multiple chunks. The size of these chunks is equal to the transfer size and corresponds to the amount of data transferred per I/O function call. The figure was adapted from [7][8].

The results of the metadata benchmarks underline the performance advantage of current flash-based mass storage devices. (Figure 5). Most notably, the AFSM leads clearly over the SSSM in directory creation and removal operations, which are up to 3 and 5 times faster, respectively (Figures 5C and 5D).

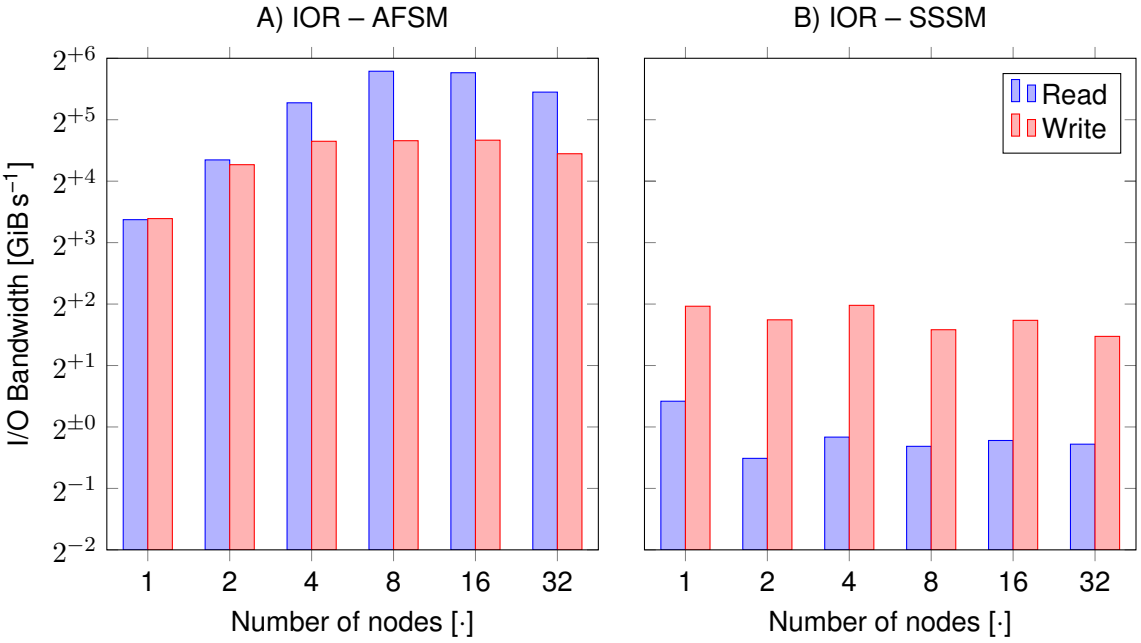


Figure 4.: Strong scaling of write and read performance with IOR on (A) the AFSM and (B) the SSSM of the DEEP system is shown. Throughput was measured with a transfer size of 512 KiB (equal to chunk size of storage system), the block size was set to correspond to the transfer size. Each MPI process wrote to its own file. In order to avoid skewing the results by caching effects three measures have been taken: (1) the aggregate test file size was fixed at 4 608 GiB for AFSM and 768 GiB for SSSM, which corresponds to two times the total RAM installed in the storage servers of the respective storage module; (2) in a multi-node test each node reads back data written by another node; (3) about 90 % of client RAM was occupied, when only a single node was tested. The benchmark jobs were executed in successive order. The Intel compiler version 2021.2.0 20210228 and ParaStation MPI version 5.4.9-1 were used. At the time the benchmarks were carried out, the SSSM was still connected via a 40 Gbit s^{-1} Ethernet network.

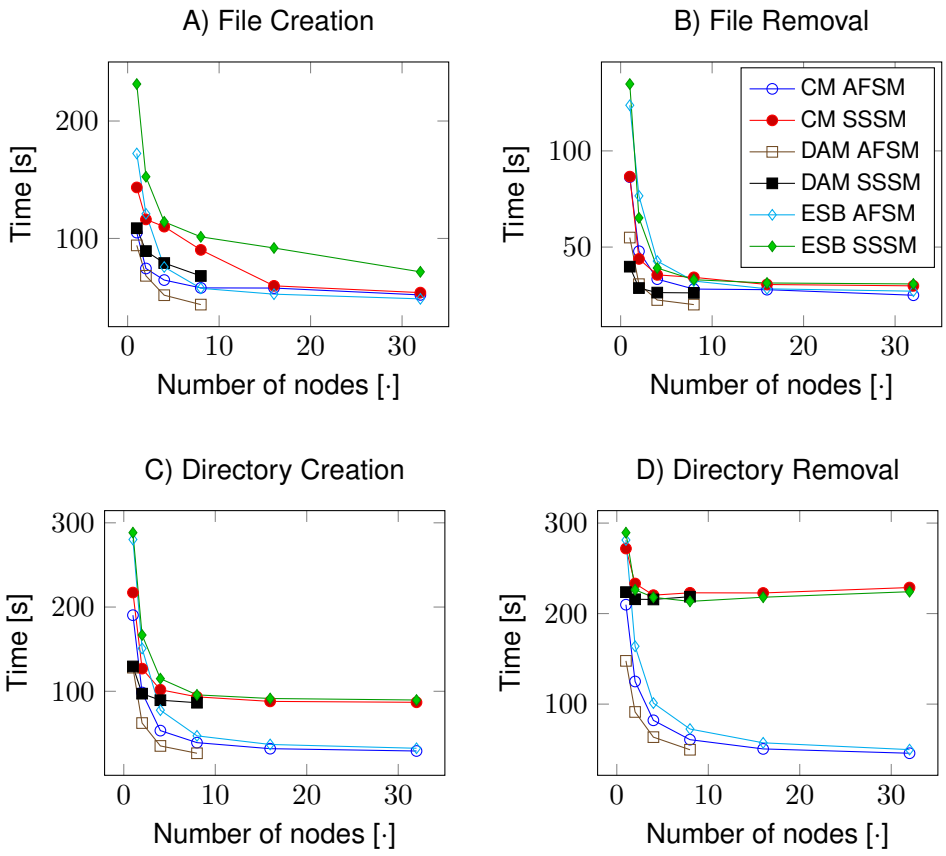


Figure 5.: Strong scaling of file and directory creation and removal operations on the AFSM and the SSSM of the DEEP system is shown. Separate tests using up to 32 nodes were carried out one after the other to avoid mutual interference. Each test was repeated five times by Mdtest and the average values were plotted. The depth of the directory tree was set to two and the branching factor to eight. The number of files per directory was calculated as $10^6/(8^2 * N)$, where N is the number of MPI processes working in parallel. This fixes the total amount of files at 10^6. The leaf level of the tree was used for file tests. The Intel compiler version 2021.2.0 20210228 and ParaStation MPI version 5.4.9-1 were used. At the time the benchmarks were carried out, the SSSM was still connected via a 40 Gbit s^{-1} Ethernet network.

5. The Ohio State University (OSU) MicroBenchmarks (OMB)

The Ohio State University MicroBenchmarks (OMB) [9] is a suite of Message-Passing–Interface (MPI) benchmarks aimed at testing all available MPI functions. It consists of many small executables, each designed to test one aspect of MPI. The goal of this synthetic benchmark is to ascertain the communication performance of the IO-SEA test infrastructure.

5.1. Description

The OMB test suite consists of many smaller MPI benchmarks with dedicated tasks and specializations. Nearly every MPI call has an associated benchmark, consisting of open-access code that can be compiled into an executable. There are benchmarks to test bandwidth, latency, remote memory access and collective operation. As part of this project we have focused on testing bandwidth latency and collectives, as these are the most commonly used MPI functions.

Execution options for these generally consist of the number of messages to path as well as the message size. The number of messages defines the number of communications that results are averaged over, while the message size influences exactly what is measured, as MPI behaviour strongly depends on the message size passed to it. Furthermore, large message size helps to avoid caching effects, which may yield more realistic results for actual performance at the expense of not resulting in realistic numbers as most real world applications pass small messages.

5.2. Results

As the OMB can be used to generate a plethora of graphs displaying the MPI performance of a system, we have here chosen to restrict ourselves to the most readily understandable results, MPI latency and bandwidth. MPI communication latency describes the inherent unavoidable time delays associated with communication. In general MPI communication times t consist of two components, a message-size–independent component t_l, the latency, and a message-size–dependent component t_b, which tends to increase as message size increases. The time interval t_b depends on the bandwidth of the network over which the message is communicated. Mathematically the total MPI communication time can be written as:

$$t = t_l + t_b(m), \tag{5.1}$$

where m is the message size. If the message size is zero $t_b(m) = 0$, the communication latency can be directly inferred from the total communication time t. In the limit as m tends to positive infinity the contribution of t_l to t becomes vanishingly small when compared to the contribution by $t_b(m)$. This causes the measurable total communication time to become a good upper bound for the communication time actually related to communication. By dividing the message size by the total communication time a good lower bound on the bandwidth can be determined.

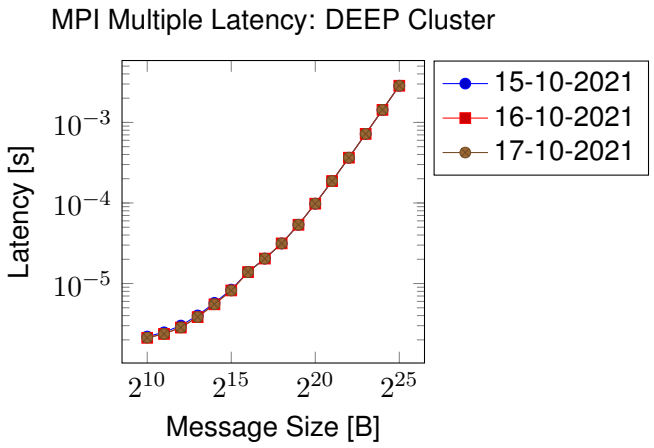


Figure 6.: A comparison of multi-latency tests OMB run on the DEEP Cluster module.

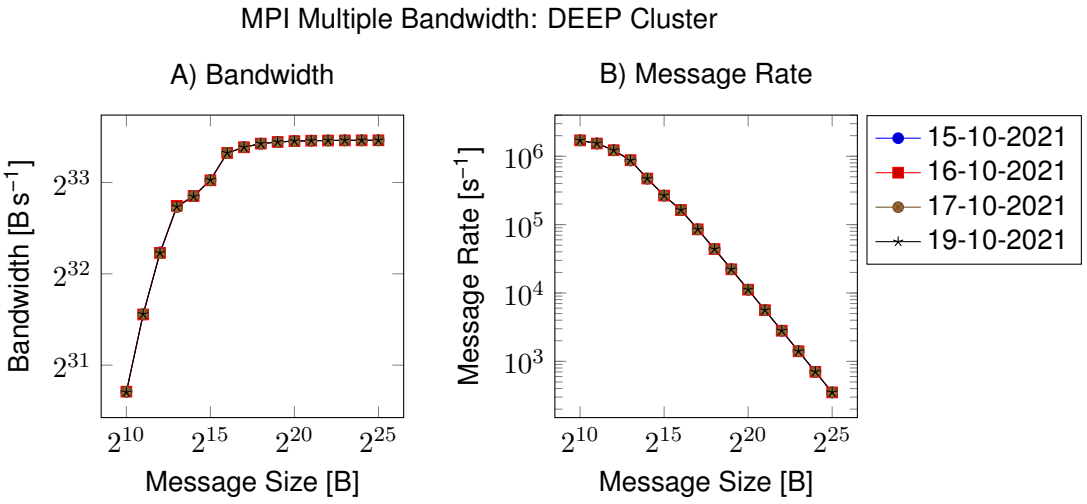


Figure 7.: A comparison of multi-bandwidth OMB tests run on the DEEP Cluster module.

Figure 6 shows a comparison of multiple multi-latency tests run on the DEEP Cluster module. These results demonstrate good run-to-run consistency, which is important for benchmarking. Large run-to-run variances in metrics make it difficult to identify changes in metric performance that are due to hardware or software level changes.

Figure 7 shows a comparison of multiple multi-bandwidth tests run on the DEEP Cluster module. Again we see excellent run-to-run variance. There is a kink in the bandwidth, which is hardly visible in the message rate due to scaling, for message sizes around 8 to 64 KiB presumed to be due to CPU-related cache effects. The peak bandwidth caps out at around 10 GiB s^{-1}. As the bandwidth approaches the cap, the message rate decays linearly.

6. Linktest

Linktest [10] is a communication-API benchmarking tool that tests all possible point-to-point connections between an even-numbered set of hosts and is designed to work on very large numbers of hosts. It supports the benchmarking of MPI, TCP, UCP, IB verbs, PSM2, and NVLink bridges via CUDA. Linktest is used as part of IO-SEA to benchmark communications, which are an important component of I/O.

6.1. Description

Linktest is a JSC-developed tool for benchmarking communication APIs. The APIs and associated communication hardware are benchmarked by sending messages between tasks hosted either on the same or different CPUs/GPUs. Messages can either be sent between two tasks in parallel with one task sending its message to the other as the other is sending its message back. Alternatively the messages can be sent one after the other. Furthermore, the location where these message are stored can be controlled. They can reside either in CPU or GPU RAM.

The communication APIs that can be benchmarked are MPI, TCP, UCP, IB verbs, PSM2 and CUDA for benchmarking of NVLink bridges between NVIDIA GPUs.

Linktest is able to consistently and coherently test the API by constructing a virtual cluster environment that unifies the various communication protocols into simple set of instructions used for the benchmarks. This ensures a quasi-level playing field between the APIs. Nevertheless, some specific benchmarks have custom code to improve performance.

The output of the program is a full timing communication matrix of the message-transmission times for all pairs of tasks written to a SION file. SION files are files optimised for parallel output produced by the SIONlib package, which is actively being worked on in WP 3. Linktest also provides a standard output log that summarises the results is also provided. Additional tools are provided to read the generated SION file into Python and to make one-page reports out of the data.

As part of the IO-SEA project Linktest is used to benchmark communication, an important component of non-local I/O. The MPI benchmark is used here as MPI is the most common HPC communication API. It also allows us to compare the Linktest benchmark results to those of OMB (see Chapter 5). This helps us to corroborate results. Furthermore, using multiple different types of benchmarks that measure the same thing makes it more likely to pick up on errors for which one of the benchmarks is more susceptible. Further note, that although Linktest and the OMB are similar, what they exactly test is different. Only Linktest provides the exhaustive testing of possible communication pairs in an even-numbered pool of tasks.

6.2. Results

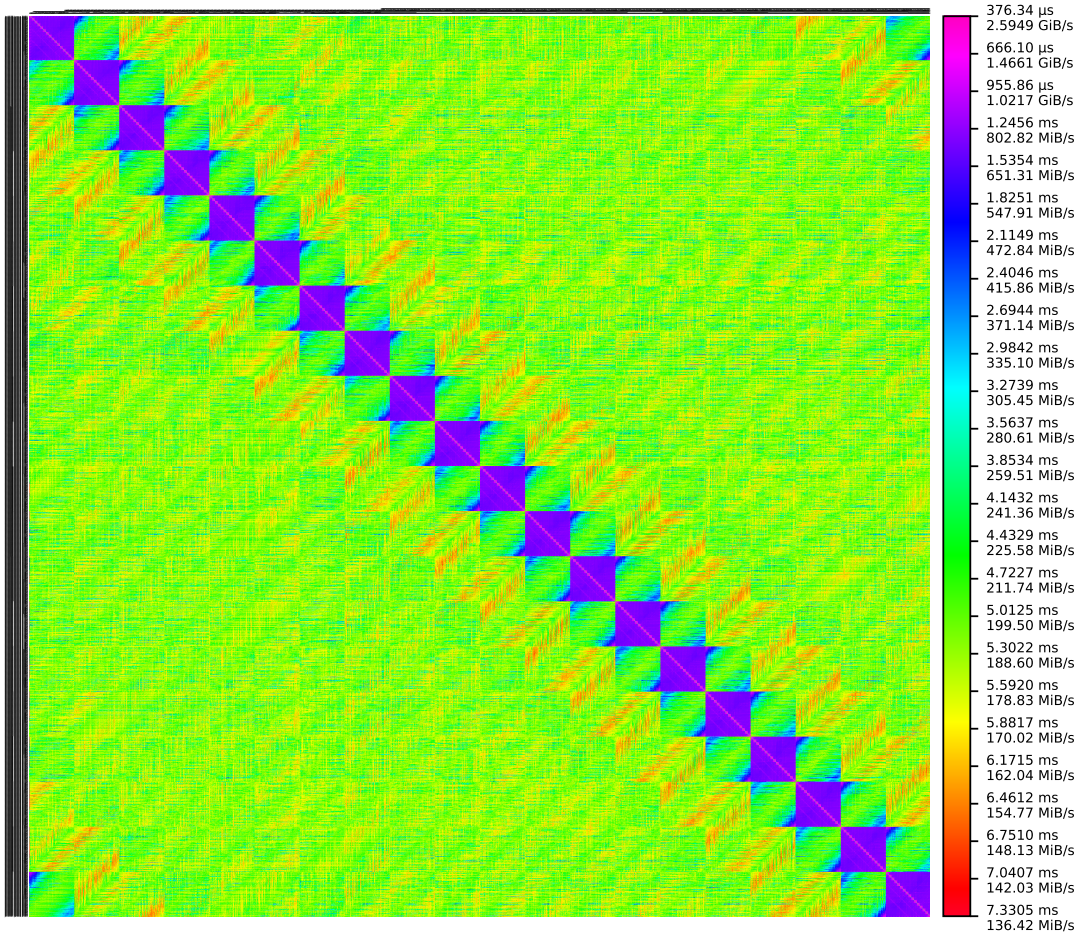
Page 29 shows a Linktest report from the JURECA-DC system at JSC. The top indexed-colour plot shows the average measured communication time to send thirty 1 MiB messages back and forth

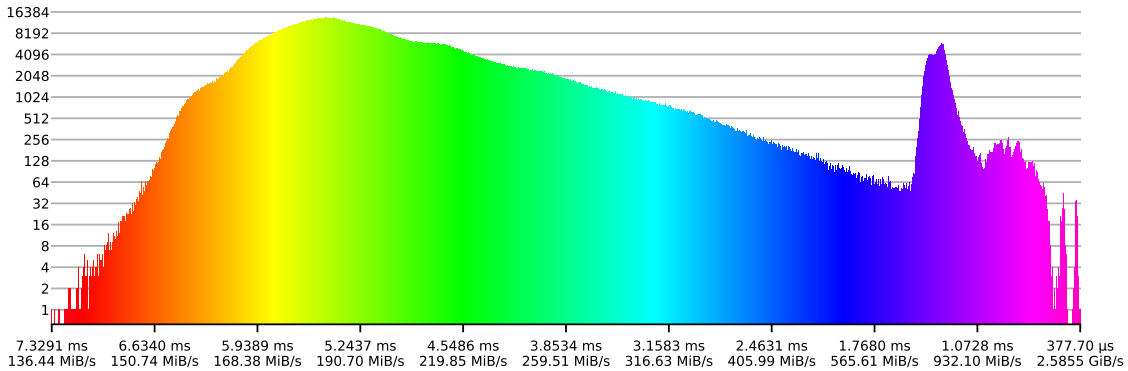
using MPI. Twenty nodes, each with 128 physical cores were used for this test, resulting in 2 560 tasks total. The nodes are connected to each other using an InfiniBand network.

In the histogram the purple spikes, which correspond to fast communication, are related to intra-CPU communication. In the indexed-colour plot these are the blocks down the diagonal. The neighbouring blue-green off-diagonal blocks correspond to one CPU communicating to another in the two-CPU nodes. The neighbouring further-away red-green blocks are topologically the furthest away and hence the connections are slowest, all other connections make up the bulk of the histogram and correspond to connections of varied closeness. Please note that in this test as many connections as possible were tested in parallel so the measured bandwidths do not correspond to peak theoretical performance.

linktest_jureca_mpi_20nx128c_1B.sion

15:09:44 05/08/2021 UTC+0





Communication API:	mpi	Number of Tasks:	2560	Number of Hosts:	20
Number of Messages:	30	Message Size:	1,048,576 B (1.0000 MiB)		
# Warm-Up Messages:	3	# Serial Retests:	0	Domain:	
Execution Order:	Parallel	All-to-All:	No	Mixed Ranks:	No
Bidirectional:	No	Bisection:	No	Use GPUs:	No
Min. Value:	376.34 µs 2.5949 GiB/s	Average Value:	4.9153 ms 203.45 MiB/s	Max. Value:	7.3305 ms 136.42 MiB/s

Linktest Version: 2.1.11

Report generated by the FZJ Linktest Python Report Tool 2.1.11, Forschungszentrum Jülich GmbH

7. STREAM 2

The STREAM and STREAM 2 benchmarks [11] measure sustainable memory bandwidth and corresponding computation rate for simple vector kernels. As a result, it is used for benchmarking the memory hierarchy for modern compute systems.

7.1. Description

The STREAM 2 benchmark is an evolution of the original STREAM benchmark with the intent to measure sustained bandwidths at all tiers of the memory hierarchy and to more clearly expose the differences between read and write. This is important as many common compute kernels, like those used in finite-difference techniques or linear solvers, spend the majority of their time waiting for memory as the gap between CPU performance growth and that of the memory subsystem widens. Benchmarking performance changes between memory tiers therefore gives us direct insight in to how certain operations may perform.

STREAM 2 offers four different vector kernels for benchmarking: 1) FILL populates an array and is a pure write kernel, 2) SUM sums a filled array and is a pure read kernel, 3) COPY copies one array to another and is a read-write kernel, and 4) DAXPY adds two vectors (arrays), where one vector is scaled by an additional constant, and stores the result in the first array. This is a read-read-write kernel. Each kernel stresses CPUs and their vector instruction sets in slightly different ways.

FILL tends to be the smallest memory bandwidth for small array sizes as it only writes and writing to memory physically takes longer than reading from it for modern memory architectures. For larger array sizes the COPY kernel is even slower as both arrays cannot be kept in fast cache simultaneously. For small array sizes, however, it boasts the largest bandwidths as it corresponds to a simple copy from one cache location to another for which the average read-write bandwidth is largest. SUM and DAXPY have roughly equal memory bandwidths, with SUM showing a larger memory bandwidth for very small array sizes. For larger array sizes the aggregated memory bandwidth of DAXPY is greater due to the presence of specialised instructions for the kernel in many modern CPUs.

7.2. Results

This kernel behaviour can easily be corroborated by running the benchmark. Figure 8 shows results from the DEEP cluster module. The aforementioned performance relationships are exactly replicated in the figure. Furthermore the figure clearly illustrates the performance drop-offs associated with changes in memory caches. The change in copy performance is especially apparent as soon as the array size exceeds half the L1-cache size as the memory footprint of COPY is twice the array size.

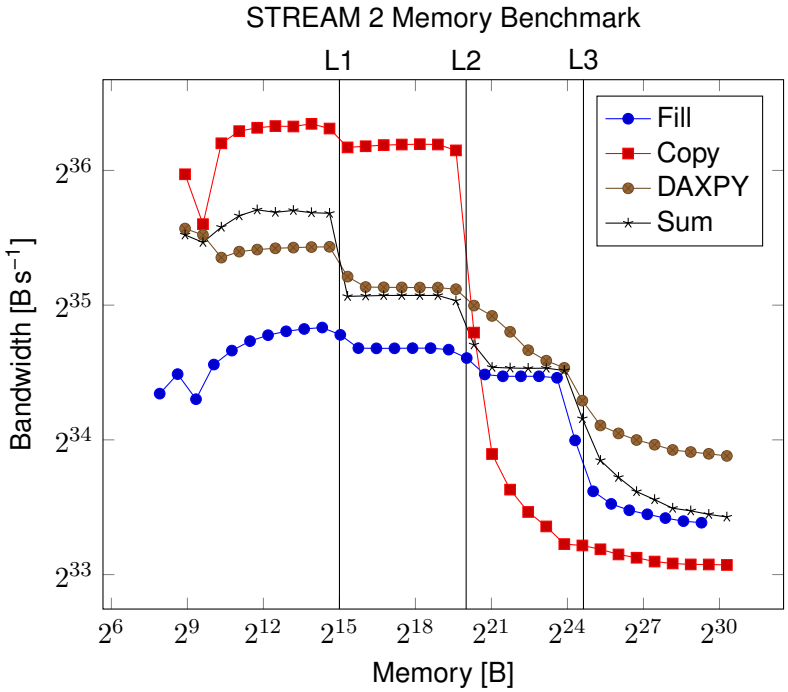


Figure 8.: STREAM 2 benchmark results from an Intel Xeon Silver 4215 CPU in the DEEP cluster partition. The L1, L2 and L3 cache sizes are indicated using vertical lines.

8. The Big-Data Benchmark

The BigDataBench [12] is a suite of representative data-intensive applications and micro benchmarks of multiple workloads. We use the micro benchmarks from this suite to have a representative set of computational benchmarks in the field of big data.

8.1. Description

The BigDataBench is a suite of 27 big-data benchmarks and 13 representative real-world data sets. It aims at covering multiple workload types including online services, offline analysis, graph analysis, data warehouse, NoSQL, and streaming from three important application domains, Internet services, recognition sciences, and medical sciences. It features micro-benchmarks, each of which is a single data motif, as well as components benchmarks, which consist of the data motif combinations, and end-to-end application benchmarks, which are the combinations of component benchmarks. Carefully chosen data sets are either included in the benchmark or generated with the Big-Data–Generator Suite (BDGS) by scaling up a seed of real-world data while preserving their data characteristics.

The micro benchmarks based on MPI, covering offline analysis, are:

- Sort, sorts the input data set into an output data set
- Grep, matches a regular expression against a sequence of characters
- Word Count, counts the number of appearances of each distinct word in a data set
- MD5, calculates a hash of a sequence of characters
- RandSample, collects random samples out of a larger population
- Connected Components, calculates all subsets of connected components in a graph
- FFT, calculates the Fast Fourier Transform (FFT) of a matrix
- Matrix Multiply, multiplies two matrices

The first five work with a seed of 4.3 million Wikipedia articles. The Connected Components benchmark analyses Facebook's Social network. FFT and Matrix Multiply work on a 2-dimensional matrix. Each benchmark represents a different data motif [13], that is an abstraction from a class of computations. A data motif is characterised by its computation, memory access and I/O patterns. The data sets generated by the BDGS will be scaled to match the dimension of our use cases, as long as the total run time of the benchmarks is suitable for regular execution.

8.2. Results

The result of the micro benchmarks are simple run times (see Table 1). We also keep track of the data sizes used for the benchmark, that way we can choose to run bigger or smaller data sets later in the project while still being able to compare with earlier runs.

An example run was conducted on the JUWELS system at JSC. Note that the grep benchmark performance depends not only on the data size, but also on the used regular expression, which was a plain character sequence without operators in this case.

Partition	Nodes	Tasks Total	Data Size [GiB s^{-1}]	Benchmark	Run Time [s]
batch	2	96	10	sort	2 093
batch	2	96	10	wordcount	1 073
batch	2	96	10	md5	24.69
batch	2	96	10	grep	3.371

Table 1.: BigDataBench results for the JSC JUWELS system.

9. High Performance LINPACK (HPL)

The High Performance LINPACK (HPL) software package [14] is a software package that solves linear system. It is commonly used to solve ISO/IEC 60559:2020 [15] 64-bit–precision arithmetic linear systems, or systems using older floating-point standards, on distributed-memory computers. The HPL package provides a testing and timing program to quantify the accuracy of the obtained solution as well as the time it took to compute it. We use HPL to benchmark the processing power and scalability of the individual hardware modules over the evolution of the project. In the future, we might also use it to benchmark the maximum attainable peak performance of the system.

9.1. Description

The HPL benchmark provides an estimate of the achievable theoretical peak performance, which is generally less that the often-unattainable theoretical peak performance provided by manufacturers. The latest version of this benchmark is commonly used to build the TOP500 list [16], which ranks the world's most powerful supercomputers in terms of their peak performance measured in FLoating-point OPerations per Second (FLOPS). The Intel(R) distribution of the LINPACK Benchmark [17] is one such implementation of the Massively Parallel LINPACK benchmark, developed and optimised for Intel processors. This is the version we use for benchmarking our systems, since the processors used in the the DEEP system [18] are from Intel. The most important input parameters for this benchmark are N, P, Q and NB [19]. The parameter N is the problem size, P and Q are the number of rows and columns in the process grid respectively. $P \times Q$ must be equal to the number of MPI processes that HPL is using. The parameter NB determines the block size of the data distribution, and is set to 384 following a recommendation for different processor types by Intel. Usually, HPL runs are setup to have a problem size that is proportional to a fraction of the total available memory of a system. This fraction is typically set to 70 to 90 % of the total available memory.

However, for the benchmarks reported in this deliverable, we set the problem size N to 50 000 to ensure the benchmarks complete within a stipulated amount of time. It is also chosen so that the reported peak performance for a given problem size for the three DEEP system modules (CM, ESB, DAM-ext) can be compared on the same plot over the duration of the project. We do this because the three modules have varying amounts of memory available per node, and a problem size N significantly greater than 50 000 cannot be executed on the ESB module due to memory constraints, especially for small numbers of nodes. The benchmark is set up to use all available cores on each node and compiled using ParaStation MPI version 5.4.9-1.

9.2. Results

We performed strong scaling tests independently on the three hardware partitions of the DEEP system with the same problem size ($N = 50\,000$), for a varying number of a processing elements ranging from 1 to 32, increasing with powers of 2.

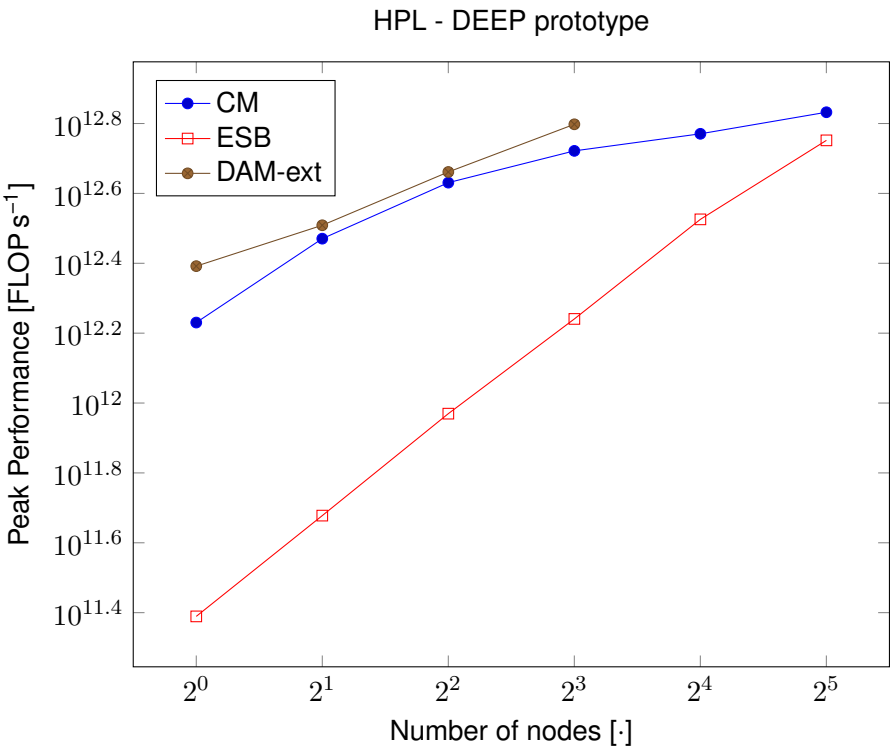


Figure 9.: Strong scaling measurements for Intel Optimised HPL on the major DEEP system modules. The problem size N is set to 50 000, and the block size of the data distribution NB is set to 384. The benchmark is set up to use all available cores on each node and compiled using ParaStation MPI version 5.4.9-1.

Figure 9 shows that the CM module reports higher peak performance numbers than the ESB module, and that the DAM-ext module reports slightly higher peak performance than the CM module. The former can be explained due to the availability of much more powerful CPUs on the CM module when compared to the ESB module. The latter observation is most likely due to the larger L3 Cache present within the CPUs on the DAM-ext module, when compared to the CPUs within the CM module.

Figure 9 also shows that for the given problem size, the ESB and DAM-ext modules exhibit almost perfect scaling, and that the obtained speedup for the CM module drops as we increase the number of available nodes. This behaviour can be attributed to the fact that the chosen size of the problem is not large enough to saturate the available memory on the nodes of the CM module. As we increase the number of nodes, this leads to an increasing amount of time spent on communication between the cores of a node, as well as the nodes themselves. Note, the absolute numbers and scaling behaviour reported here are a function of the problem size. The reported peak performance numbers do not reflect the highest possible performance attainable on each of the modules, rather they reflect only the peak performance attainable for the chosen input parameters. For a larger problem size, the CM module scales almost perfectly and reports higher peak performance. However, an HPL run with a larger problem size could not be executed on the ESB due to memory constraints. This is the primary reason we limit ourselves to a smaller problem size in these benchmarks.

10. High Performance Conjugate Gradient (HPCG)

HPCG [20][21] was intentionally developed as a complement to the High Performance LINPACK (HPL) benchmark. HPCG focuses on computation and data access patterns, which frequently occur in scientific software, to better reflect the expected real world performance of HPC systems.

10.1. Description

HPL mainly favours HPC systems which excel at dense-matrix computations and at the streaming of coalescent memory regions. HPCG on the other hand is mainly concerned with other frequent computation and data access patterns, which are characterised by high rates of often irregular memory accesses and also fine-grained recursive calls. A well-balanced HPC system should be capable of handling both types of contrasting patterns described [22].

10.2. Results

The scaling behaviour of HPCG on all major DEEP system modules was benchmarked on up to 32 compute nodes. The node-local problem size was selected such that at least 25 % of RAM is occupied as recommended to prevent the local data from fitting into the cache [23]. Official benchmarking runs require a runtime of at least half an hour, but the results do not change significantly when the benchmark is executed longer than 30 seconds [24]. As a compromise between saving of CPU core hours and the reliability of the results, we decided on a runtime of 15 minutes.

The results show that HPCG performance scales linearly as the number of compute nodes is increased (Figure 10). The achieved performance is almost doubled when twice the number of nodes is used. At the scale of up to 32 nodes inter-node communication via a high-speed, low-latency Infiniband network is not limiting the performance on the CM and ESB. In contrast, HPCG does not scale well on DAM, where nodes are interconnected via an Extoll Tourmalet network.

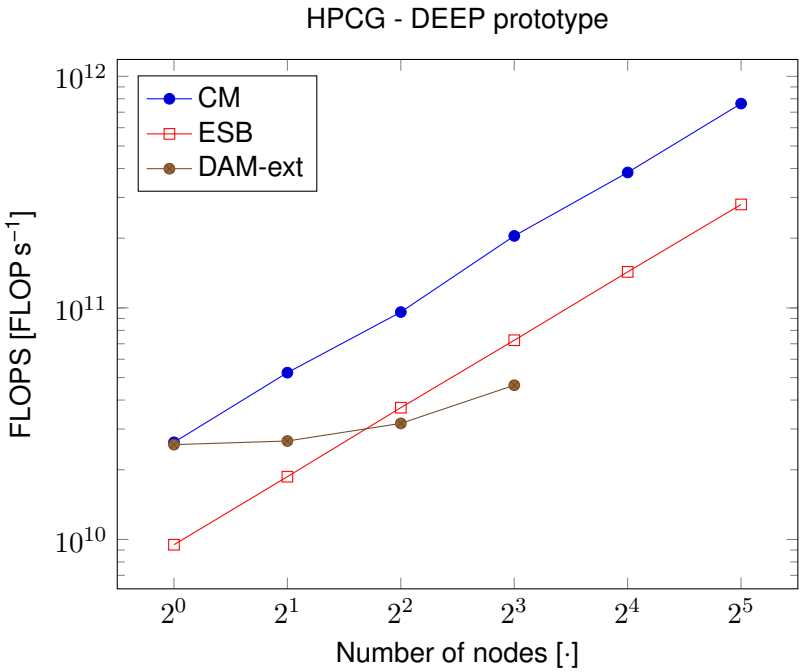


Figure 10.: Scaling demonstration for HPCG from the major modules of the DEEP system. The node-local problem dimension was set to x=256, y=128, z=128. The number of MPI tasks per node was equal to the number of CPU cores per node. For this weak scaling test HPCG was run on up to 32 nodes in parallel. The target runtime was set to 900 s. The Intel compiler version 2021.2.0 20210228 and ParaStation MPI version 5.4.9-1 were used. Network interconnect was either Infiniband (CM, ESB) or Extoll Tourmalet (DAM-ext).

Part III.

Use-Case Benchmarks

11. Benchmarking RAMSES

11.1. Overview

The RAMSES benchmark suite was designed within the RAMSES code to evaluate the RAMSES I/O throughput for both the legacy RAMSES I/O engine (POSIX calls) and for the new engine based on the Hercule parallel I/O library. The introduction of the Hercule library [25] split the single legacy checkpoint/restart dataflow into two distinct dataflows: checkpoint/restart and post-processing. The benchmark suite can evaluate both the legacy RAMSES engine and the new Hercule engine. As detailed in the IO-SEA D1.1 report [1], the RAMSES workflow is quite simple and can be summarised in three steps: read input data or checkpoint, output checkpoint data, and output post-processing data. Those three steps are shown using magenta arrows on Figure 11. Post-processing tools were designed to read post-processing output only and produce images using state of the art algorithms (Gaussian blur in AMR grid techniques, ray-tracing, etc.) or different kinds of reduced data ingested later by other codes. While this dataflow, shown with an orange arrow on Figure 11, will not be benchmarked for I/O performance, it is highly important to maintain it functional.

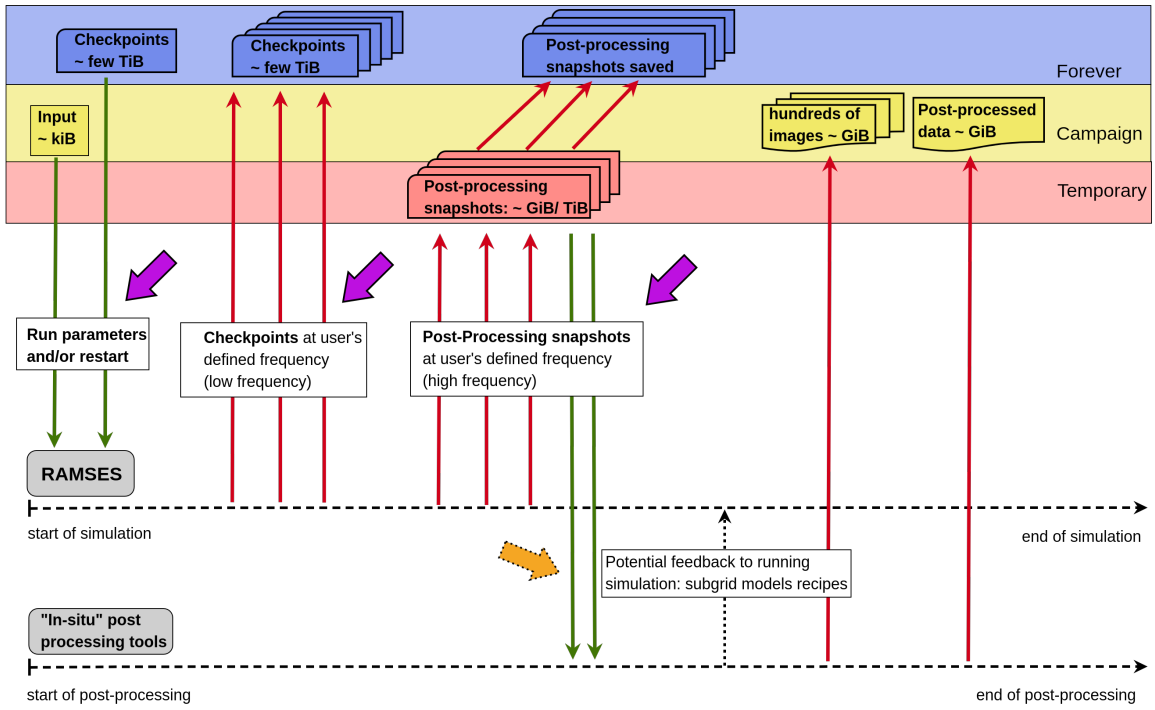


Figure 11.: The RAMSES workflow. Magenta arrows point to data flows targeted by the benchmark suite. The orange arrow points to the data flow that must remain operational.

11.2. Benchmark details

Checkpoint read

Represented by the first magenta arrow from the left on Figure 11, this dataflow occurs once at the beginning of the run. This phase is generally slow and can last for few hours on very large RAMSES run configurations but is done only once per run. The size of the input data is, at worst, the size of the checkpoint data to read and is very astrophysical object dependent. During the benchmark of this dataflow, we will gather metrics regarding the I/O throughput and the total time to read the data.

Checkpoint write

The second magenta arrow in Figure 11 points to the checkpoint dump and happens several times during a RAMSES run. Although the checkpoint dump frequency will now be reduced, thanks to the integration of the Hercule I/O library and the new I/O engine, the volume of data remains quite large, from hundred of gigabytes up to terabytes. The frequency of this dataflow must be adjusted regarding the expected total time of a RAMSES run, the estimated data volume, and available disk space. In general, the checkpoint dump occurs a few dozen times per run. The gathered metrics for this benchmark are the I/O throughput and the total time to write the data.

Post-processing write

The third magenta arrow in Figure 11 represents the new RAMSES dedicated post-processing dump dataflow. This new dump is at the origin of the whole scientific analysis pipeline, thus it has a much higher output frequency than the checkpoint dataflow. Its frequency also varies depending on the user requirements, the size of the simulation, the available disk space, etc. The introduction of physical field selection and several new state-of-the-art algorithms such as AMR tree pruning, AMR grid compact description, and AMR-specific lossless/lossy data compression, significantly reduced the data volume and thus should reduce the I/O time. Thus we expect to achieve a frequency of several hundred or several thousand outputs per run (depending on the size of the simulation). The gathered metrics for this benchmark are the same as for checkpoint dumps. In addition, we will vary the number of exported physical fields as well as the activation/deactivation of the lossless/lossy compression to see how much it impacts the I/O throughput and time.

11.3. Benchmark parameters

There are three kinds of parameters that can be used: RAMSES parameters, Hercule parameters and filesystem parameters. While RAMSES parameters are set within the Fortran input namelist, Hercule and the filesystem parameters are set through command line or environment variables before the run. It is also important to mention that RAMSES and Hercule parameters do not depend on the filesystem.

RAMSES parameters	Value	Comment
benchmark_type	'write' or 'read'	Type of benchmark
hprot_benchmarks	True or False	Activate checkpoint/restart
hdep_benchmarks	True or False	Activate post-processing dump
hdep_nvars	n	Number of field to dump
hdep_savevars	$1 - n$	Field indices
hdep_compression	True or False	Activation of data compression
Hercule parameters	**Value**	**Comment**
GME_PARITION	x GB	Maximum filesize
HER_CONTRIB_PER_FILE	1, 2, 4, 8, 16 and 32	Size of MPI group for I/O
Filesystem parameters	**Value**	**Comment**
stripe_size	1, 2, 4, 8 and 16 MB	Depends on filesystem
stripe_count	N	Depends on filesystem

Table 2.: Parameters of the RAMSES benchmark.

11.4. Baseline results and future work

The study of Strafella & Chapon, [26], of the RAMSES Hercule I/O engine can be considered as baseline results. Indeed, benchmarks were run in a user-like environment with no specific privileges and during a normal load of the Irene Joliot-Curie TGCC supercomputer of CEA. The RAMSES 3D test case Sedov3D (explosion in a cubic box) using a domain size of 2 048 cells in each direction of space was used. AMR and time integration were deactivated during the execution, thus the load was perfectly balanced between all the MPI processes and benchmark runs do not consume extra computational time. The output bandwidth of RAMSES, namely the data volume in gigabytes written by the code by unit of time was measured. The number of MPI processes ranged from 2 048 to 8 192, while keeping the size of the problem constant. All benchmarks were run five times in order to average results. The RAMSES I/O capacity with the Hercule engine was able to scale from 50 to 110 GB s^{-1}, while increasing the number of MPI process from 2 048 to 8 192, where the legacy POSIX engine rapidly reach a plateau at 50 GB s^{-1}. For future benchmarks we will use the same test case to compare the evolution of the I/O capacity of checkpoint and extend it to the post-processing dump. In addition, a read benchmark from real astrophysical datasets with real data load balance between MPI processes (factor 10 between the most populated process and least populated one) will be run as well, as a checkpoint and post-processing dump based on those datasets.

12. Benchmarking the Analysis of Electron Microscopy Images

The benchmark is based on a simplified version of the general cryo–electron–image-analysis workflow described in D1.1 [1]. Its purpose is to estimate available bandwidth between the two involved sites and obtain baseline performance figures of selected pipeline stages running on the HPC infrastructure at IT4I.

12.1. Input data

For benchmarking purposes two full datasets have been transferred from CEITEC infrastructure to IT4I using IRODS distributed storage system [27]. This test has been performed with default settings of the IRODS client to get a baseline results. Raw test of network bandwidth have also been performed using the iperf3 tool [28], which shows that stable a bandwidth of 10 Gbit s^{-1} is available between the IRODS client machine at CEITEC and the target server in IT4I. Results of this test are presented in Table 3.

Dataset name	Size [GB]	Image count	Avg. speed [Mbit s^{-1}]
210201_IR_trim	888	7 551	569.7
210311_nucLEDGF	1 500	13 050	620.1

Table 3.: Benchmark data sets transfer from CEITEC to IT4I.

The measurement shows a lot of room for improvement, particularly in fine tuning of the IRODS client settings and in the usage of a storage resource with a higher number of I/O operations per second available due to the high amount of small files that are transferred at once.

Each dataset consists of a number of Tag–Image–File-Format (TIFF) images accompanied by a small plain-text file which contains specific metadata, as shown in Figure 12.

12.2. Pipeline stages in benchmark

Different stages of the pipeline are executed as pre-compiled CUDA executables. Process execution and synchronization is implemented using Python `multiprocessing` module. The real-time analysis of the cryo-EM data comprises utilization of several third-party programs which need to be installed on the computational cluster first. The program dm2mrc, which is part of the IMOD software package [29], is used for the conversion of the gain reference image from .dm4 format (native format of the camera producer) to the more general format used during data analysis. Programs MotionCor2 [30] and GCTF [31] are used for the compensation of the sample drift present in the raw data (movies) and estimation of the contrast transfer function parameters, respectively. All programs run on GPU and are only available as pre-compiled and free to use binaries with no access to the source code. The simplified pipeline is shown in Figure 13.

```
T = SerialEM: CEITEC: K2 on Titan Krios 12-Mar-21 12:18:28
Voltage = 300

[FrameSet = 0]
GainReference = CountRef_mic_Mar12_12.18.13.dm4
DefectFile = defects_mic_Mar12_12.18.13.txt
TiltAngle = 0.00349957
StagePosition = -835.928 345.294
StageZ = -39.8434
Magnification = 165000
Intensity = 0.115082
...
```

Figure 12.: Except of a mdoc file assigned to each TIFF image.

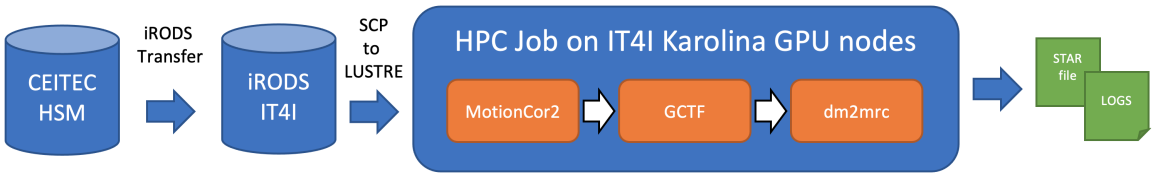


Figure 13.: Simplified Cryo-EM image analysis pipeline for benchmarking.

12.3. Baseline results

A workflow with several stages, including data transfer from the acquisition centre (CEITEC) to the remote data analysis centre (IT4I) was tested on the two datasets and contained the following three stages:

1. Data transfer
2. Drift correction (MotionCor2)
3. Contrast transfer function estimation (GCTF)

Table 4 shows run times of the second and third stage of the provided image processing pipeline on CEITEC and IT4I infrastructures. CEITEC ran the benchmark on dedicated bare-metal compute node accessed directly. It has two 16 core Intel Xeon CPUs (Broadwell microarchitecture), 124 GB of RAM and 4 NVIDIA GeForce GTX 1080 GPUs. At IT4I, single accelerated node from the Karolina cluster has been used in a batch job. Each node has two 64 core AMD Zen 2 EPYC CPUs, 1 024 GB of RAM and 8 NVIDIA A100 GPUs with 40 GB of second generation High Bandwidth Memory (HBM2). As expected, all the tools run slightly faster on the IT4I infrastructure with newer GPUs.

Site	Pipeline task	Runtime [s]
IT4I	MotionCor2	6.26
CEITEC	MotionCor2	8.58
IT4I	GCTF	2.92
CEITEC	GCTF	3.82

Table 4.: Pipeline run times.

12.4. Future outlook and roadmap

The following steps in the data analysis workflow will be gradually implemented. CEITEC MU is currently developing an in-house AI-based program for particle picking. We will also try to mitigate the issues encountered due to the utilization of the closed third-party software by developing open software tools for data analysis.

Two execution modes will be investigated. The first one is the continuous analysis of the produced images to provide near real-time feedback for the ongoing imaging process. This mode will require efficient and simple data transfer protocol of the individual TIFF images and accompanying mdoc files as well as a service which will run the entire pipeline with the least amount of overhead. The second one is the batch processing which will be used to analyse the entire dataset (usually approximately 15 000 images) in a highly parallel HPC batch job.

The benchmark will be further extended with I/O tracing to cover all parts of the execution. There are several parts of the pipeline where unnecessary overhead is generated, specifically in repeated execution of CUDA binaries. Optimisation of the pipeline execution will be explored as well.

13. Benchmarking the ECMWF Weather Forecasting Workflow

13.1. Overview

The operational weather forecasting workflow at ECMWF is a complex pipeline that includes observational data acquisition, assimilation into the forecasting model, forecasting, product generation, and dissemination to the downstream users. In this pipeline, the most computationally intensive part is the forecasting model, coupled to product generation (re-gridding and extracting regions of interest, according to downstream requirements). This part of the workflow is shown in Figure 14 and is run four times a day in a one-hour time-critical window. As a consequence of the time constraints, the model output is read as soon as possible by the product generation software, causing substantial I/O contention.

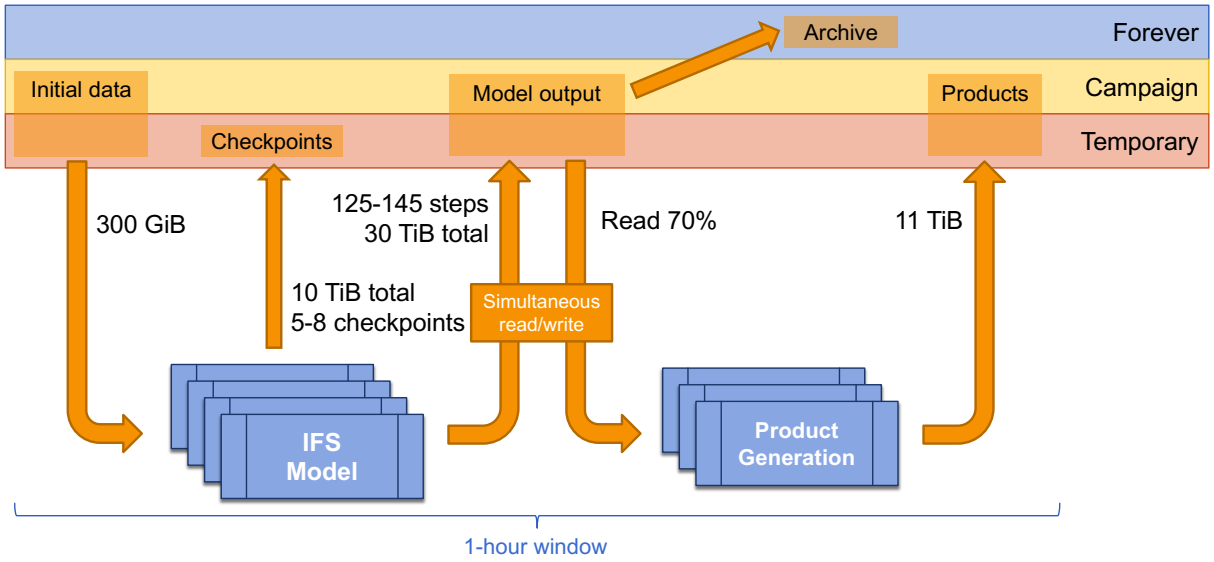


Figure 14.: Weather forecasting data flow. One instance of the IFS model is run at high resolution and 51 instances are run as an ensemble at coarser resolution. Each step of the high-resolution run and each common step of the ensemble trigger one product generation run.

Since the purpose of the benchmark is to assess I/O performance, all the components have been stripped down to a minimum: the forecasting model has been replaced by a simulator that generates mock data and pushes them through the Integrated Forecasting System (IFS) I/O software stack, into ECMWF's FDB meteorological object store. This "model" output is immediately read by the operational product generation software, with requirements tailored for maximal use of the data with very limited computation. In accordance with the operational workflow, about 70 % of the model data is consumed by the product generation steps. This reduced workflow is shown in Figure 15. During the course of the project, the object store will be changed to take advantage of the solutions developed by the other work packages.

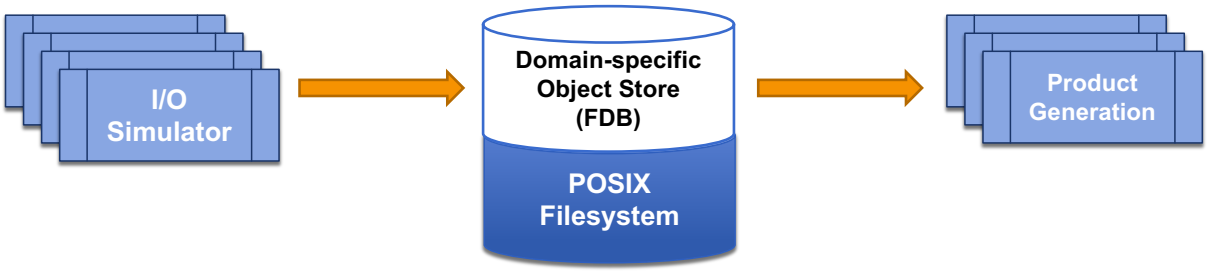


Figure 15.: Reduced version of the ECMWF workflow used for benchmarking, in its initial version. The I/O simulator is a thin wrapper around the operational IFS I/O software stack.

The scheduling of the producer (I/O simulator) and consumer (product generation) steps is handled by the Kronos[1] workflow simulator. First, a number of I/O simulators are started, mimicking the ensemble meteorological model, which produce data in time steps. Each "ensemble member" emits notifications when it has written all the data corresponding to a time step. These notifications are picked up by Kronos, and once all the producers have notified the availability of a given time step, the corresponding product generation step will be submitted to the queueing system. A timeline of the workflow is pictured in Figure 16.

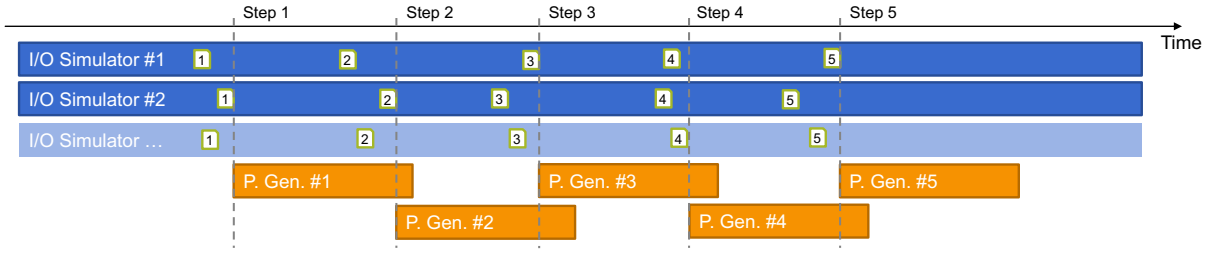


Figure 16.: Timeline of the ECMWF workflow used for benchmarking. Each box is a job submitted to the queueing system. A time step is deemed complete when all the I/O simulator instances have issued the corresponding notification, in turn triggering the submission of the corresponding product generation job.

13.2. Behaviour and parameters

The FDB object store handles *fields* as its basic objects. A field is a two-dimensional slice of the atmosphere. Each field in the store is uniquely identified by its metadata. The I/O simulator takes one sample field as input and generates mock output by changing the metadata. The output data is organised to mimic an operational forecast: each of the N_e instances of the simulator acts like an ensemble member, generating N_s time steps. Each time step consists of N_l vertical model levels and N_p parameters. Therefore, each instance generates $N_s \times N_l \times N_p$ fields. The actual writing is done by N_w writer tasks. On the product generation side, one time step is read per product generation job. The work is distributed across N_r reader tasks, and consists of $N_e \times N_l \times N_p$ fields. The values of these parameters is given in Table 5. To check for potential contention effects due to the simultaneous

[1] `https://github.com/ecmwf/kronos`

read and write from the FDB, the benchmark will be run in two configurations, one with product generation, ant the other without product generation.

Parameter		Value
Number of writers per ensemble member	N_w	32
Number of readers per product generation instance	N_r	64
Number of ensemble members	N_e	5
Number of steps	N_s	10
Number of levels	N_l	200
Number of parameters	N_p	11
Input field size		803 KiB
Total number of fields		110 000
Total output size		84 GiB

Table 5.: Parameters of the ECMWF benchmark.

13.3. Metrics and first results

Both the I/O simulator and the product generation code print detailed statistics at the end of the run. Among these, the most important are the throughput on the writing end of the I/O simulator, and the read rate of the product generation, and will be the main metrics of a benchmark run. The other statistics will be kept for further analysis if needed. The results of the first few runs on the ECMWF HPC system are shown in Table 6. Since the work performed by this benchmark is almost exclusively I/O, there is a significant jitter due to the usage of the file system, which makes it difficult to interpret the differences between the runs with and without product generation. A detailed assessment of the performance of an alternative I/O solution like the one that IO-SEA aims to build will require running in an isolated context with a heavier workload, which can be achieved by increasing the values of the input parameters, but is not suitable for regular runs of the benchmark.

Product generation	Write rate [MiB s^{-1}]			Read rate [MiB s^{-1}]		
	min	avg	max	min	avg	max
On	22 603	29 484	48 068	350	473	746
Off	22 807	27 486	30 889			

Table 6.: First runs of the ECMWF benchmark. Each configuration is run five times to assess the reproducibility of the results.

14. Benchmarking TSMP

The Terrestrial Systems Modelling Platform (TSMP) is a fully coupled, scale consistent, highly modular and massively parallel regional Earth System Model. TSMP (v1.2.3) is a model interface which couples three core model components: the COSMO (v5.01) model for atmospheric simulations, the CLM (v3.5) land surface model and the ParFlow (v3.7) hydrological model. Coupling is done through the OASIS coupler. TSMP allows to simulate complex interactions and feedbacks between the different compartments of terrestrial systems. Specifically, it enables the simulation of mass, energy and momentum fluxes and exchanges across land surface, subsurface and atmosphere [32]. TSMP is maintained by the Simulation and Data Lab Terrestrial Systems (SDLTS) at JSC, and is an open source software publicly available in GitHub [1].

14.1. Benchmark cases

Two cases have been selected as benchmarks to use with the fully coupled TSMP, i.e., with COSMO, CLM and ParFlow, based on several criteria. Firstly, they should allow to identify improvements in the I/O processes in response to developments in IO-SEA. Second, we selected an idealised case configuration (*A*) which is of interest only for correctness and computational benchmarks. This is a very adaptable problem, which can be run as a very small configuration, or can be scaled indefinitely and used as a stress test both for computations and I/O. For the initial benchmarking we use a problem defined on a 100×100 element mesh. Finally, we selected a short version of a typical production run for our second benchmark test (*B*). This case consists of a fully coupled land surface, atmosphere and hydrological dynamics over Europe, specifically using the EURO-CORDEX [33] domain, with a resolution of $0.11°$ (approx. 12.5 km). In the case of production runs, multidecadal periods are simulated [e.g. 34]. For the benchmark, only 12 hours of simulated time are computed. The interest of this short benchmark is that the I/O metrics captured can be extrapolated to estimate the requirements of the multidecadal runs.

Both cases are part of the typical and publicly available set of tests for TSMP. Both have been tested with CPU only and heterogeneous configurations (ParFlow running on GPUs). Currently, the benchmark configuration is for CPUs only but can be easily changed. Case *A* is run on three nodes, and Case *B* on six nodes (JUWELS @ JSC).

14.2. Benchmarking workflow

A JUBE benchmarking workflow was developed to systematically manage the two benchmarking cases. The JUBE workflow is illustrated around the TSMP workflow diagram (Figure 17) originally introduced on Deliverable 1.1 [1], with small modifications to account for the actual benchmark workflow. The benchmarking workflow deals only with the core of the simulation pipeline, for both benchmark tests. Although post-processing and visualisation steps may be of interest due to the

[1] https://github.com/HPSCTerrSys/TSMP

potentially intensive I/O operations they entail, there is currently no single typical in-situ workflow for these post processing stages (and many ad-hoc offline post-processing workflows).

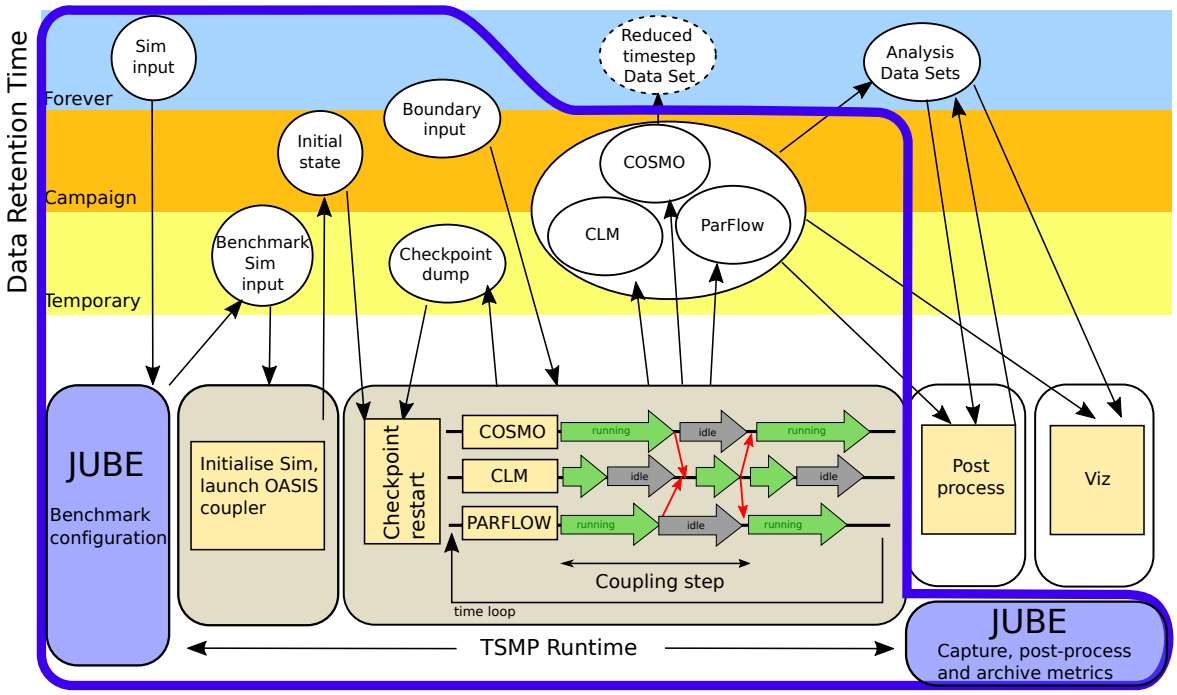


Figure 17.: JUBE-TSMP workflow for a single simulation including all three component models.

The JUBE-TSMP workflow makes use of both native JUBE operations and also relies on a variety of TSMP shell scripts. The workflow automatically submits a pre-configured benchmark case as a TSMP job, and minor configuration changes are enforced, specifically for the benchmarking process. Due to the comparably long time to compile and build TSMP, pre-compiled binaries are used. A number of log files reporting native instrumentation in the component models are generated during the TSMP runtime. These contain timers and simple performance metrics (e.g., total I/O time per component model, among many others). Additional metrics are post-processed by assessing timestamps in the output files. The full set of metrics and logs is pushed to a GitLab repository for archiving.

14.3. Metrics and preliminary results

A first set of metrics is targeted in the benchmarking workflow, as shown in Table 7. Currently, the metrics are mostly collected from the native timing files that all component models provide. More detailed metrics are likely to be implemented in the future. Basic metrics are provided for the full TSMP instance, although it must be stressed that TSMP relies on the component models for I/O. Consequently, the time required for I/O is provided per component. Additionally, since the component models can be running concurrently or staggered at any given time, it is relevant to consider the time between file writes, per component.

It is important to highlight that the total TSMP I/O time T_{IO} only aggregates the I/O time of the components $T_{\mathrm{IO}} = C_{\mathrm{I}} + C_{\mathrm{O}} + L_{\mathrm{IO}} + P_{\mathrm{IO}}$, but it does not necessarily match the actual walltime since the component I/O processes may happen in parallel and are non-blocking for the other component models. The ratio metrics ($C_{\mathrm{I/r}}$, $C_{\mathrm{O/r}}$, $L_{\mathrm{IO/r}}$ and $P_{\mathrm{IO/r}}$) are defined as the ratio of the corresponding I/O times to the runtime of the component model.

Finally, the native timers in the component models allow for many other metrics, which may enrich the first set present here, which will be evaluated as more specific information is required.

Metric	Description	Units
T_{r}	total TSMP runtime	s
C_{I}	COSMO input time	s
C_{O}	COSMO output time	s
L_{IO}	CLM I/O time	s
P_{IO}	ParFlow I/O time	s
T_{IO}	total TSMP I/O time	s
C_{Op}	COSMO output period	s
L_{Op}	CLM output period	s
P_{Op}	ParFlow output period	s
$C_{\mathrm{I/r}}$	COSMO input-to-runtime ratio	%
$C_{\mathrm{O/r}}$	COSMO output-to-runtime period	%
$L_{\mathrm{IO/r}}$	CLM I/O-to-runtime period	%
$P_{\mathrm{IO/r}}$	ParFlow I/O-to-runtime period	%

Table 7.: Metrics captured in the TSMP-JUBE benchmarks.

Table 8 summarises preliminary results for the metrics described in Table 7. These results so far correspond to CPU-only TSMP runs, although investigations will also be performed with heterogeneous configurations (i.e., CLM+COSMO on CPUs, and ParFlow on GPUs). These runs were computed in JUWELS (JSC).

Metric	T_{r}	T_{IO}	C_{I}	C_{O}	L_{IO}	P_{IO}	C_{Op}	L_{Op}	P_{Op}	$C_{\mathrm{I/r}}$	$C_{\mathrm{O/r}}$	$L_{\mathrm{IO/r}}$	$P_{\mathrm{IO/r}}$
Case *A*	195	11.41	0.01	7.11	1.78	2.51	27.7	29.0	32	0.005	3.64	0.92	1.3
Case *B*	423	210	12.4	75.6	10.7	111.4	31	31	33	2.97	18.1	2.8	26

Table 8.: Preliminary results for all metrics and both TSMP benchmark cases.

The results in Table 8 show that CLM has the lowest ratio of I/O time to runtime, and the lowest absolute I/O time in both cases. For Case *A*, COSMO has the highest ratio, but for Case *B* it is ParFlow. ParFlow output files are smaller than COSMO's (see Tables 9 and 10), but their writing, according to these metrics, appears less efficient. This signals a first point to investigate in detail.

Tables 9 and 10 show finer-grained statistics of the output for all component models for both benchmark cases. Both COSMO and CLM write NetCDF files (nc), whereas ParFlow writes in ParFlow Binary format (pfb). The component models write different numbers of files. Only the main file types

are differentiated in Tables 9 and 10, and a number of smaller files are lumped together. These correspond to a variety of processes, most relevant of which are single outputs occurring during initialisation. These outputs may benefit from improvements in IO-SEA, but are comparably less important, as longer simulations will mostly multiply the number of main output files detailed in the tables.

File	Size [MB]	Number	Average time between writes [s]
COSMO output (.nc)	27.7	7	27.7
CLM output *r* (.nc)	10.8	7	26.7
CLM output *h0* (.nc)	3.67	6	31.8
ParFlow output *press* (.pfb)	2.29	8	32
ParFlow output *satur* (.pfb)	2.29	8	32
Other (lumped size)	29.8	51	
Total	358	87	$\sim$30

Table 9.: Output files for TSMP benchmark *A* (ideal). File sizes are reported per file (except for "Other").

File	Size per file MB]	Number	Average time between writes [s]
COSMO output (.nc)	463	15	31.6
CLM output *r* (.nc)	0.41	13	31.0
CLM output *h0* (.nc)	67	13	30.9
ParFlow output *press* (.pfb)	21.2	14	32.6
ParFlow output *satur* (.pfb)	21.2	14	32.5
Other (lumped size)	746	72	
Total	9 161	135	$\sim$31

Table 10.: Output files for TSMP Benchmark *B* (CORDEX). File sizes are reported per file (except for "Other").

15. Benchmarking LQCD

15.1. LQCD workflow

In preparing benchmarks for lattice quantum-chromodynamics (LQCD), we examined the LQCD workflow for steps that should be particularly affected by I/O improvements developed as part of the IO-SEA project. As detailed in the report for IO-SEA Deliverable 1.1 [1], the LQCD workflow has, in general, four steps which we label *A* — *D*. These are illustrated in Figure 18.

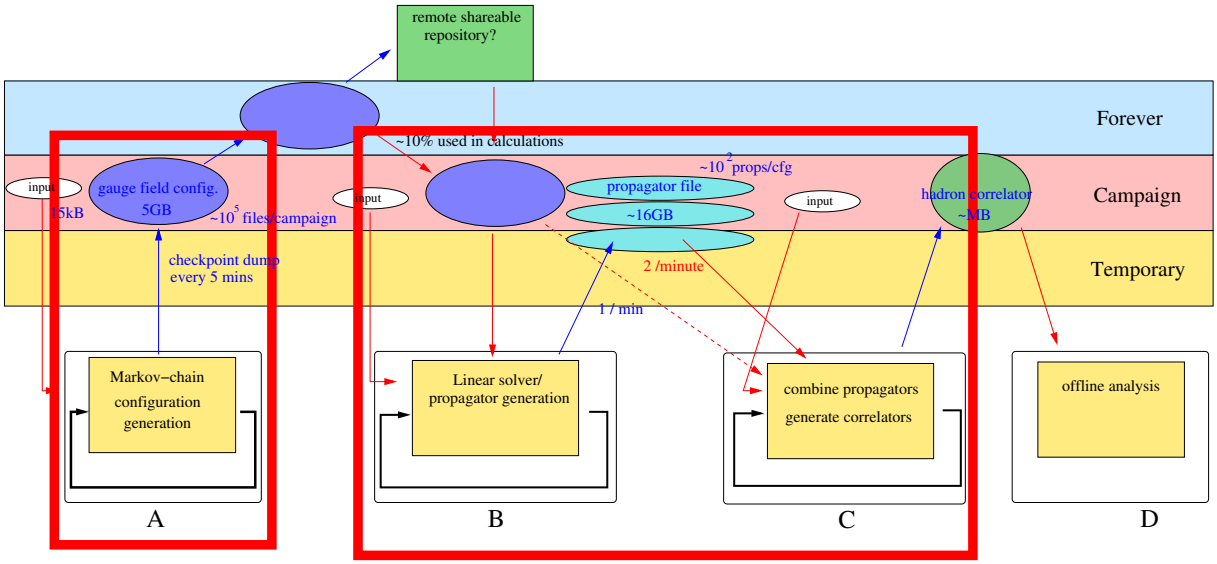


Figure 18.: LQCD workflow diagram, highlighting the steps included in the LQCD benchmark suite. The first benchmark includes workflow Step *A*, generation of gauge field configuration files. The second workflow combines Steps *B*, generating propagators, and Step *C*, combining propagators to form hadron correlators. Horizontal bands represent different storage tiers.

In Step *A* we use a Monte Carlo algorithm to generate a Markov chain of lattice gauge field ensemble files. These are stored on disk to be used as the background fields for physics calculations in the later steps. They are also generally put in long-term tape storage for future physics calculations. For each job in Step *B*, one of the gauge field configuration files is read in and used to construct a large, sparse linear system. The solutions to this system are vectors called quark propagators which are saved to disk. In Step *C* the gauge field configuration and the propagators are read in again, and the propagators are combined in element-wise products to form *hadron correlators*. For memory reasons, the propagators may be read in small groups in Step *C* . In some simple cases, that we do not consider here, Steps *B* and *C* might be combined into a single step without saving propagators to disk. The hadron correlators are small arrays that are analysed offline in Step *D*.

From this workflow, we have developed two JUBE benchmark scripts. The first covers Step *A*, gauge field configuration generation, which we expect to benefit from IO-SEA technology when data nodes are used as burst-buffers. A second benchmark includes Steps *B* and *C* running sequentially. We

similarly suspect these combined steps to benefit from the use of data nodes. Both benchmarks are based on the CHROMA QCD software system [35]. Below, we describe the two benchmarks.

15.2. LQCD benchmark *A*

In LQCD Benchmark *A*, the configuration-generation benchmark, the application reads an input file which holds the path to an initial configuration file. The input file also contains instructions to perform a number of updates, as well as parameters specifying the details of the updates. After every update, a Metropolis test is performed, accepting or rejecting the update. Regardless of the outcome, the state is check-pointed after the test, and the configuration is saved to disk. The problem size is chosen to be typical of a realistic, medium-sized LQCD simulation. Update parameters, however, are chosen to give faster-than-realistic updates so that the benchmark does not require too much CPU or wallclock time. Table 11 summarises the input and output files for LQCD Benchmark *A*.

File type	Size	Number	Time between reads or writes
Input			
input file	15 kB	1	
gauge field configuration	1.2 GB	1	
Output			
gauge field configuration	1.2 GB	5	$\sim$30 s

Table 11.: Input and output files for LQCD Benchmark *A*. Output frequency is for two nodes of the JUWELS cluster at JSC.

In LQCD Benchmark *A* we record four metrics, listed in Table 12. In capturing I/O times, we encounter a curious issue. CHROMA uses a specialised library, QIO [36], to handle I/O. QIO is capable of reading and writing in parallel, with all nodes or only a subset of nodes being designated I/O nodes. In all cases, except for the case of every node reading or writing its own data, communication and re-shuffling the data structure is required. For unknown reasons, QIO reports read times broken into read and communication time. However, QIO does not report timings for writes and we must rely on timing information from CHROMA itself. Consequently, we cannot separate communication time from the write time. This is reflected in our benchmark metric definitions.

Metric	Description
T^A_{traj}	Average time to generate and checkpoint an update. First update is disregarded.
$T^A_{\mathrm{save\text{-}cfg}}$	Average time to save a configuration (including communication)
$B^A_{\mathrm{save\text{-}cfg}}$	Save bandwidth for check-pointing configurations (including communication)
T_A	Total run time of step A

Table 12.: Metrics captured in the LQCD Benchmark *A*.

15.3. LQCD benchmark *BC*

In LQCD Benchmark *BC* we measure performance of workflow Steps B and C. This combination of solving for propagators and then combining them to make hadron correlators is sometimes done in a single workflow step. However, the high memory demands of step *C* mean that it is often necessary to split the step into two jobs. For the Benchmark *BC* we execute steps *B* and *C* as separate jobs within a single Slurm batch script. We anticipate that, with the benefit of specialised data nodes, the propagators produced in step *B* will be directed to fast storage and the input and output times will decrease. Table 13 lists the input and output files for LQCD Benchmark *BC*.

File type	Size	Number	Time between reads or writes
Input *B*			
input file	26 kB	1	
gauge field configuration	1.2 GB	1	
Output *B*			
propagator file	4.6 GB	8	∼30 s
Input *C*			
input file	13 kB	1	
gauge field configuration	1.2 GB	1	
propagator file	4.6 GB	8	∼15 s
Output *C*			
hadron correlator files	300 kB	4	∼15 s

Table 13.: Input and output files for LQCD Benchmark *BC*. Output frequency is for two nodes of the JUWELS cluster at JSC.

Table 14 lists the metrics recorded by LQCD Benchmark *BC*. The issue of separating write times from the related inter-node communication again affects our metric definitions.

15.4. Test platform and baseline results

The benchmarks are integrated into JUBE and stored in a repository on GitLab. Due to the time required to build CHROMA and its supporting libraries, the build step is not executed through JUBE. Rather, the executable is manually built and resides in an accessible directory on the target system. The GitLab repository has a CI/CD pipeline and a GitLab runner is enabled on the target system to allow automated benchmarking and archiving of results in the repository.

Both benchmarks use the same initial gauge configuration file as input, and this resides at a fixed location on the target cluster on a disk accessible to the compute nodes. Likewise, the large files produced in the benchmark runs, such as gauge configuration files for Benchmark *A* or propagator files for Benchmark *BC*, are saved outside of the directory structure generated by JUBE and overwritten on subsequent runs of the benchmarks. Except for timestamps in the metadata, these files are expected to be identical on subsequent runs.

Metric	Description
$T^{B}_{\text{read-cfg}}$	Average read time for a configuration. (without communication)
$T^{C}_{\text{read-cfg}}$	Average read time for a configuration. (without communication)
$B^{B}_{\text{read-cfg}}$	Read bandwidth for configuration (without communication)
$B^{C}_{\text{read-cfg}}$	Read bandwidth for configuration (without communication)
$T^{B}_{\text{read-cfg-comm}}$	Average read time for a configuration (including communication)
$T^{C}_{\text{read-prop-comm}}$	Average read time for a propagator (including communication)
$T^{B}_{\text{save-prop-comm}}$	Average save time for propagator (including communication)
$T_{\text{read-prop}}$	Average read time for propagator (without communication)
$B^{C}_{\text{read-prop}}$	Read bandwidth for propagator (without communication)
$B^{C}_{\text{write-prop-comm}}$	Write bandwidth for propagator (including communication)
T_{BC}	Total run time of step BC

Table 14.: Metrics captured in the LQCD benchmark *BC*.

In the initial period of use-case benchmark development, we have designed the benchmarks to run on GPU-accelerated nodes of the JUWELS cluster at JSC. Both the LQCD Benchmark *A* and LQCD Benchmark *BC* are set up to run on two nodes with four Nvidia V100 GPUs per node. The runtime for Benchmarks *A* and *BC* on this platform are about four minutes and seven minutes, respectively. Table 15 summarises the complete preliminary results for both LQCD benchmarks on two nodes of the JUWELS cluster.

In the future, as we test the use case with IO-SEA technology on other platforms, we will add new platform-specific branches to the git repository. This may require tuning the exact details of the benchmark runs to meet the demands of the hardware.

Metric	Result
T^A_{traj}	42 s
$T^A_{\text{save-cfg-comm}}$	5.7 s
$B_{\text{save-cfg-comm}}$	211.5 MB s^{-1}
T_A	248.7 s
$T^B_{\text{read-cfg}}$	0.1 s
$T^C_{\text{read-cfg}}$	0.4 s
$B^B_{\text{read-cfg}}$	20.133 3 GB s^{-1}
$B^C_{\text{read-cfg}}$	2.946 3 GB s^{-1}
$T^B_{\text{read-cfg-comm}}$	1.1 s
$T^C_{\text{read-prop-comm}}$	15.2 s
$T^B_{\text{save-prop-comm}}$	14.7 s
$T_{\text{read-prop}}$	2.4 s
$B^C_{\text{read-prop}}$	1.973 3 GB s^{-1}
$B^C_{\text{write-prop-comm}}$	328.6 MB s^{-1}
T_{BC}	446 s

Table 15.: Preliminary results for the LQCD benchmarks on the JSC JUWELS cluster.

Part IV.

Summary

16. Summary

Continuous benchmarking is an integral part of any development cycle. The work done in the past eight months months serves as an important stepping stone in realising the IO-SEA project. Integration of the use cases and synthetic benchmarks into JUBE, in conjunction with GitLab, gives us a basis for regularly benchmarking future performance developments arising from the IO-SEA project. This will ease integration with the other work packages as they now have a standardised way to execute the use cases.

Work for this deliverable was officially kicked-off on the first of July 2021 with a JUBE workshop held across the three SEA projects. In this, the methodology to achieve this deliverable was outlined and partners were given the tools and knowledge to work towards its realisation. Afterwards those in Task 1.2 integrated the synthetic benchmarks, from the first half of this report, into JUBE while the other use-case tasks integrated their own use cases. This was supervised by Task 1.1.

Now we have a GitLab repository from which partners can automatically execute their benchmarks, as well as those of others, and have the results uploaded back to GitLab for inspection. Once a prototype system is available for IO-SEA this will aid collaboration with other work packages as they will be able to test their developments using the provided benchmarks. This will also help the optimisation of the work done by the other work packages, as well as easing the integration of said work into the use cases.

As described in IO-SEA's Data Management Plan (D7.5) [4], certainly not all, but some of the data generated via our benchmarking activities will be made publicly available. The partners will jointly identify which data to publicly release. Such data will then be pushed into the public and open GitLab or GitHub repositories.

It is our hope that when a suitable IO-SEA test bed is available, continuous benchmarking will help this project stay on track to reach its goals, and allow us to demonstrate improvements over the project lifetime.

List of Acronyms and Abbreviations

A

AFSM	The All-Flash Storage Module is a purely flash-based storage module of the DEEP system.
AI	Artificial Intelligence.
AMR	Adaptive Mesh Refinement.
API	An Application Programming Interfaces (API) allows software to communicate with other software which support the same API.
ATOS	ATOS is Europe's largest digital services deliverer.

B

BDGS	The Big Data Generator Suite efficiently generates scalable big data while employing data models derived from real data to preserve data veracity [37].

C

CEA	The French Alternative Energies and Atomic Energy Commission.
CEITEC	Central European Institute of Technology, Masaryk University, Brno, Czech Republic.
CHROMA	The Chroma software system for lattice QCD .
CI/CD	Continuous Integration/Continuous Deployment is an automated system for the testing, integration and deployment of software.
CLM	Community Land Model.
CM	Cluster Module - the general purpose compute module of the DEEP System.
COSMO	Consortium for Small-scale Modeling.
CPU	Central Processing Unit.
CUDA	The Compute Unified Device Architecture is a parallel computing platform as well as an API that allows for the communication with certain types of graphics-processing units.

D

DAM	Data Analytics Module of the DEEP System focused on data-intensive applications.

DEEP The Dynamical Exascale Entry Platform (DEEP) is a multi-component project project to prepare for upcoming exascale HPC systems. The DEEP-SEA project is latest component of this project. It is also used to refer to the prototype developed in the DEEP projects based on context..

E

ECMWF European Centre for Medium-Range Weather Forecasts.

ESB Extreme Scale Booster - module of the DEEP System focused on compute intensive applications with high scalability.

F

FDB Fields DataBase, the ECMWF meteorological object store..

FFT The Fast Fourier Transform was originally an optimized algorithm proposed for Fourier transforming discrete data, nowadays it more commonly refers to a suite of better-optimized algorithms that perform said transform.

FLOPS FLoating-point OPerations per Second (FLOPS) is the number of floating-point operations achieved in a second.

FPGA Field-Programmable Gate Array.

FZJ Forschungszentrum Jülich, in Jülich, Germany, is one of the largest research centres in Europe and a member of the Helmholtz Association.

G

GitLab GitLab is an open-source software-development–and–operations platform that uses Git to organize software development.

GitLab Runner A GitLab Runner is software that connects to GitLab servers for remote execution of CI/CD tasks.

GPFS The General Parallel File System (GPFS) is an IBM developed high-performance clustered file system.

GPU Graphics Processing Unit.

H

HBM2 The second generation of High Bandwidth Memory using synchronous dynamic random-access memory stacked in three dimensional space.

Hercule Parallel I/O and data management library developed at CEA.

HPC High-Performance Computing.

HPCG The High Performance Conjugate Gradient benchmark is a benchmark based on a conjugate-gradient kernel.

HPL The High Performance LINPACK is a performant software package for solving linear system.

I

I/O Input/Output is either a noun referring to the action of doing either input and/or output, generally either reading or writing memory, or is an adjective or adverb that describes that the following operation does input and/or output.

IB verbs The API for communication using InfiniBand, a communication hardware.

IFS Integrated Forecasting System, ECMWF's operational weather forecasting system.

IOR The Interleaved Or Random (IOR) benchmark is benchmark for I/O.

iRODS Integrated Rule-Oriented Data System.

IT4I IT4Innovations National Supercomputing Centre at VSB Technical University of Ostrava, Czech Republic.

J

JSC The Jülich Supercomputing Centre at the Forschungszentrum Jülich is one of the IO-SEA partners.

JUBE The JÜlich Benchmarking Environment is a script based framework to easily create benchmark sets, run those sets on different computer systems and to evaluate the results.

JURECA-DC The JÜlich Research on Exascale Cluster Architectures – Date Centre module is one of the modules of the JÜlich Research on Exascale Cluster Architectures (JURECA) system.

JUWELS The JÜlich Wizard for European Leadership Science is one of the super computers at JSC.

K

Kronos The ECMWF workload simulator.

L

Linktest Linktest is a communication-API benchmark developed at JSC.

LINPACK LINPACK is a software package for solving linear systems.

LQCD Lattice quantum-chromodynamics is a numerical framework for calculating physical properties of hadrons, composite particles composed of quarks.

LUSTRE LUSTRE is an open-source parallel file system.

M

mdtest Included with the IOR benchmark, the mdtest benchmark is for benchmarking metadata creation.

MPI The Message Passing Interface is a common API for communication between tasks running on one or more computers.

MU Masaryk University, Brno, Czech Republic.

N

NetCDF NETwork Common Data Form is a community standard, machine-independent data format that support the creation, access, and sharing of array-oriented scientific data. It is extensively used in Earth system modelling.

NoSQL NOn relational SQL databases are SQL databases that can efficiently handle huge amounts of unstructured rapidly changing data. NoSQL unlike SQL does not refer to a language and is generally an adjective.

NVLink NVLink describes the suite of tools for communicating between NVIDIA GPUs. It includes an API that requires physical NVLink bridges between GPUs to use.

O

OASIS OASIS3-MCT is a software allowing synchronized exchanges of coupling information between numerical codes representing different components of the Earth System.

OMB The Ohio State University (OSU) MicroBenchmarks is a suite of MPI benchmarks that has been extended to other communication APIs.

P

ParFlow A physically-based and spatially distributed hydrological model solving surface and subsurface flows in a massively parallel computational framework.

PCIe Peripheral Component Interconnect Express is a high-speed serial-bus expansion standard designed to unify and replace older standards.

POSIX Portable Operating System Interface is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

PSM2 Performance Scaled Messaging 2 is the second generation of the PSM API for communication.

Q

QCD Quantum-ChromoDynamics (QCD).

QIO QCD Input/Output Applications Programmer Interface developed under the auspices of the U.S. Department of Energy Scientific Discovery through Advanced Computing (SciDAC) program.

R

RAM Random Access Memory is memory optimised for random access, often by a compute device.

RAMSES Code to model astrophysical systems, featuring self-gravitating, magnetised, compressible, radiative fluid flow, using AMR technique..

RDBMS A Relational-DataBase Management System (RDBMS) is a management system for relational databases.

RDBMS A Relational-Data–Stream Management System (RDSMS) is a management system for relational data streams.

RED-SEA EuroHPC project focused on network solutions for exascale architectures.

S

SDLTS Simulation and Data Laboratory: Terrestrial Systems.

SEA The Software/Solutions for Exascale Architectures project is a joint combination of three separate projects DEEP-, IO-, and RED-SEA.

Slurm Slurm is an open-source cluster-management and job-scheduling system.

SQL The Structured Query Language is a domain-specific language for accessing data in a RDBMS or in a RDBMS.

SSSM Scalable Storage Service Module - conventional, spinning-disk-based storage module of the DEEP System.

STREAM 2 A benchmark for measuring CPU-cache and CPU-to-RAM latencies and bandwidth.

T

TCP The Transmission Control Protocol is one of the main communication protocols of the internet protocol.

TGCC The *Très Grand Centre de Calcul* is CEA's very large computing centre in Bruyères-le-Chêtel, France.

TIFF The Tag Image File Format is an image file format.

TSMP Terrestrial System Modelling Platform is an open source scale-consistent, highly modular, massively parallel regional Earth system model.

U

UCP The Unified Communication Protocol is an API aimed at unifying different communication APIs, similiar in that sense to MPI.

Bibliography

[1] E. B. Gregory, P. Couvée, and M. Golasowski. *IO-SEA D1.1 Application and co-design input*. Tech. rep. IO-Software for Exascale Architectures, July 2021.

[2] J. Meinke and M. Holicki. *DEEP-SEA D1.2: Application use cases and traces*. Tech. rep. Dec. 2021.

[3] J. S. Centre. *JUBE — Jülich Benchmarking Environment*. http://www.fz-juelich.de/jsc/jube. 2008. (Visited on 2021).

[4] M. Gilliot. *IO-SEA D7.5: Data management plan*. Tech. rep. IO-Software for Exascale Architectures, Oct. 2021.

[5] E. C. Project. *Jacamar CI*. https://gitlab.com/ecp-ci/jacamar-ci. 2022.

[6] LLNL. *HPC IO Benchmark Repository – IOR and mdtest parallel I/O benchmarks*. https://github.com/hpc/ior. 2021. (Visited on 2021).

[7] H. Shan and J. Shalf. "Using IOR to Analyze the I/O Performance for HPC Platforms". In: *In: Cray User Group Conference (CUG'07)*. 2007.

[8] LLNL. *HPC IO Benchmark Repository – IOR and mdtest parallel I/O benchmarks*. https://ior.readthedocs.io/en/latest/userDoc/tutorial.html. 2021. (Visited on 2021).

[9] O. S. Univsersity. *OSU-Microbenchmarks*. https://mvapich.cse.ohio-state.edu/benchmarks/. Nov. 8, 2021. (Visited on 11/19/2021).

[10] W. Frings et al. *Linktest*. https://gitlab.jsc.fz-juelich.de/cstao-public/linktest. Aug. 29, 2021. (Visited on 11/19/2021).

[11] J. D. McCalpin. http://www.cs.virginia.edu/stream/stream2/. 2003. (Visited on 11/01/2021).

[12] W. Gao et al. "BigDataBench: A Dwarf-based Big Data and AI Benchmark Suite". In: *CoRR* abs/1802.08254 (2018). arXiv: 1802.08254. URL: http://arxiv.org/abs/1802.08254.

[13] W. Gao et al. "Data Motifs: A Lens towards Fully Understanding Big Data and AI Workloads". In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. PACT '18. Limassol, Cyprus: Association for Computing Machinery, 2018. ISBN: 9781450359863. DOI: 10.1145/3243176.3243190. URL: https://doi.org/10.1145/3243176.3243190.

[14] J. J. Dongarra, P. Luszczek, and A. Petitet. "The LINPACK Benchmark: past, present and future". In: *Concurrency and Computation: Practice and Experience* 15.9 (2003), pp. 803–820. DOI: https://doi.org/10.1002/cpe.728. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.728. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.728.

[15] "ISO/IEC/IEEE International Standard - Floating-point arithmetic". In: *ISO/IEC 60559:2020(E) IEEE Std 754-2019* (2020), pp. 1–86. DOI: 10.1109/IEEESTD.2020.9091348.

[16] J. Strohmaier et al. *Top500 List*. https://www.top500.org/. 2021. (Visited on 2021).

[17] I. Corporation. *The Intel(R) Distribution for LINPACK Benchmark*. https://www.intel.com/content/www/us/en/develop/documentation/onemkl-linux-developer-guide/top/intel-oneapi-math-kernel-library-benchmarks/intel-distribution-for-linpack-benchmark-1/overview-intel-distribution-for-linpack-benchmark.html. 2021. (Visited on 2021).

[18] D. P. EU. *Research Prototypes Developed within the DEEP set of projects: DEEP-EST Prototype Specification*. https://www.deep-projects.eu/hardware/prototypes.html. 2021. (Visited on 2021).

[19] I. Corporation. *Developer Guide for Intel(R) oneAPI Math Kernel Library for Linux*. https://www.intel.com/content/www/us/en/develop/documentation/onemkl-linux-developer-guide/top/intel-oneapi-math-kernel-library-benchmarks/intel-distribution-for-linpack-benchmark-1/configuring-parameters.html. 2021. (Visited on 2021).

[20] ICL and SNL. *HPCG Benchmark*. https://www.hpcg-benchmark.org. 2021. (Visited on 2021).

[21] J. Dongarra, P. Luszczek, and M. A. Heroux. "HPCG technical specification". In: *Sandia National Laboratories, Technical Report SAND2013-8752* (2013).

[22] M. A. Heroux and J. Dongarra. "Toward a New Metric for Ranking High Performance Computing Systems". In: *Sandia National Laboratories, Technical Report SAND2013-4744* (2013).

[23] ICL and SNL. *How big must the problem size be when running HPCG?* https://www.hpcg-benchmark.org/faq/index.html#360. 2021. (Visited on 2021).

[24] AMD. *AMD – Developer Central Spack HPCG*. https://developer.amd.com/spack/hpcg-benchmark/. 2021. (Visited on 2021).

[25] Bressand et al. "Hercule: a library of scientific data management for numerical simulation". In: *CHOCS* 41, 29-37 ().

[26] L. Strafella and D. Chapon. "Boosting I/O and visualization for exascale era using Hercule: test case on RAMSES". In: *CoRR* abs/2006.02759 (2020). arXiv: 2006.02759. URL: https://arxiv.org/abs/2006.02759.

[27] D. Lecarpentier et al. "EUDAT: a new cross-disciplinary data infrastructure for science". In: *International Journal of Digital Curation* 8.1 (2013), pp. 279–287.

[28] J. Dugan et al. "iperf3, tool for active measurements of the maximum achievable bandwidth on ip networks". In: *URL: https://github. com/esnet/iperf* (2014).

[29] D. N. Mastronarde and S. R. Held. "Automated tilt series alignment and tomographic reconstruction in IMOD". In: *Journal of structural biology* 197.2 (2017), pp. 102–113.

[30] S. Q. Zheng et al. "Anisotropic correction of beam-induced motion for improved single-particle electron cryo-microscopy". In: *bioRxiv* (2016), p. 061960. URL: https://emcore.ucsf.edu/ucsf-software.

[31] K. Zhang. "Gctf: Real-time CTF determination and correction". In: *Journal of structural biology* 193.1 (2016), pp. 1–12.

[32] P. Shrestha et al. "A Scale-Consistent Terrestrial Systems Modeling Platform Based on COSMO, CLM, and ParFlow". In: *Monthly Weather Review* 142.9 (Sept. 2014), pp. 3466–3483. DOI: 10.1175/mwr-d-14-00029.1.

[33] D. Jacob et al. "EURO-CORDEX: new high-resolution climate change projections for European impact research". In: *Regional Environmental Change* 14.2 (July 2013), pp. 563–578. DOI: 10.1007/s10113-013-0499-2.

[34] C. Furusho-Percot et al. "Pan-European groundwater to atmosphere terrestrial systems climatology from a physically consistent simulation". In: *Scientific Data* 6.1 (Dec. 2019). DOI: 10.1038/s41597-019-0328-7.

[35] R. G. Edwards and B. Joo. “The Chroma software system for lattice QCD”. In: *Nucl. Phys. B Proc. Suppl.* 140 (2005). Ed. by G. T. Bodwin et al., p. 832. DOI: 10.1016/j.nuclphysbps.2004.11.254. arXiv: hep-lat/0409003.

[36] *QIO, the QCD Input/Output Applications Programmer Interface*. http://usqcd-software.github.io/qio/.

[37] Z. Ming et al. “Bdgs: A scalable big data generator suite in big data benchmarking”. In: *Advancing big data benchmarks*. Springer, 2013, pp. 138–154.