Ref. Ares(2022)5447096 - 28/07/2022

EuroHPC-01-2019

DEEP-SEA

DEEP – Software for Exascale Architectures

Grant Agreement Number: 955606

D1.2

Application use cases and traces

Final

Version:	1.1
Author(s):	J.H. Meinke (FZJ), M.E. Holicki (FZJ)
Contributor(s):	J. Amaya (KUL), M.I. Andersson (KTH), N. Arul Murugan (KTH), D. Caviedes Voullieme (FZJ), P. Carribault (CEA), D. Grünewald (FhG), N. Jansson (KTH), D. Mancusi (CEA), S. Markidis (KTH), O. Marsden (ECMWF), J. de la Puente (BSC), J.E. Rodriguez (BSC), O. Castillo-Reyes (BSC), U. Sinha (FZJ)
Date:	19.07.2022

Project and Deliverable Information Sheet

DEEP-SEA Project	**Project ref. No.:**	955606
	Project Title:	DEEP – Software for Exascale Architectures
	Project Web Site:	`https://www.deep-projects.eu/`
	Deliverable ID:	D1.2
	Deliverable Nature:	Report
	Deliverable Level: PU*	**Contractual Date of Delivery:** 31.12.2021
		Actual Date of Delivery: 22.12.2021
	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: 1.1	**Status:** Final
	Available at: `https://www.deep-projects.eu/`	
	Software Tool: LaTeX	
	File(s): DEEP-SEA_D1.2_Application_use_cases_and_traces.pdf	
Authorship	**Written by:**	J.H. Meinke (FZJ), M.E. Holicki (FZJ)
	Contributors:	J. Amaya (KUL), M.I. Andersson (KTH), N. Arul Murugan (KTH), D. Caviedes Voullieme (FZJ), P. Carribault (CEA), D. Grünewald (FhG), N. Jansson (KTH), D. Mancusi (CEA), S. Markidis (KTH), O. Marsden (ECMWF), J. de la Puente (BSC), J.E. Rodriguez (BSC), O. Castillo-Reyes (BSC), U. Sinha (FZJ)
	Reviewed by:	A. Geiß (TUDA) E. Suarez (FZJ)
	Approved by:	BoP/PMT

Document Status Sheet

Version	Date	Status	Comments
1.0	22.12.2021	Final Version	EC submission
1.1	03.06.2022	Final Version	New Section 2.7 — Benchmarking Collaboration

Document Keywords

Keywords:	DEEP-SEA, HPC, Exascale, Software, Applications, Benchmarks, Co-design

Acknowledgements:

The DEEP Projects have received funding from the European Commission's FP7, H2020, and EuroHPC Programmes, under Grant Agreements n° 287530, 610476, 754304, and 955606. The EuroHPC Joint Undertaking (JU) receives support from the European Union's Horizon 2020 research and innovation programme and Germany, France, Spain, Greece, Belgium, Sweden, United Kingdom, Switzerland.

EuroHPC
Joint Undertaking

SPONSORED BY THE
Federal Ministry
of Education
and Research

Swedish
Research
Council

Copyright notice:

© 2021-2022 DEEP-SEA Consortium Partners. All rights reserved. This document is a project document of the DEEP-SEA Project. All contents are reserved by default and may not be disclosed to third parties without written consent of the DEEP-SEA partners, except as mandated by the European Commission contract 955606 for reviewing and dissemination purposes.

All trademarks and other rights on third party products mentioned in this document are acknowledged as own by the respective holders.

Contents

List of Figures

List of Tables

Executive Summary

The applications from work package 1 (WP1, Co-Design Applications) provide use cases and requirements as co-design input to the other work packages. They demonstrate the capabilities of the DEEP-SEA software stack and evaluate its performance and usability for a wide range of scientific applications from molecular dynamics to space weather. The results feed back into the development cycle of the project in close collaboration with all work packages. To cover as many aspects of the software stack as possible WP1 also maintains a set of additional benchmarks.

This deliverable focuses on benchmarks and traces. It provides benchmark setups and metrics of the co-design applications and introduces the additional benchmarks. Each benchmark setup is described using a JUBE (JÜlich Benchmarking Environment) file and available to all members of the project in a GitLab repository hosted at the Jülich Supercomputing Centre (JSC). The applications also provide traces that can be used for analysis and future reference in a common file system location on the DEEP system.

The benchmarks and traces together with this document offer a baseline for determining the effects of changes in the software and hardware ecosystem, as well as changes to the application codes themselves.

1. Introduction

The previous deliverable D1.1 [1] presented the co-design process of seven applications that are extensively used in their respective research fields. The applications described their workflows and parallelization scheme in addition to their software and hardware requirements.

This deliverable presents a combination of synthetic and use-case benchmarks together with the initial performance analysis of each of the applications. A unified development model exploring the points of common interest is adopted such that the applications benefit from each other. In this regard, benchmarking scripts have been developed in an integrated manner using the JÜlich Benchmarking environment (JUBE) [2], which enables each of the applications to carry out automated benchmarking of their use cases. To study profiles and traces application teams used the performance tracing tools Score-P [3] or Extrae [4]. Visualization of the traces was done using Vampir [5] or Paraver [6].

This document is divided into four parts. Part I of this document outlays the importance of benchmarking in the co-design process and describes the JUBE, benchmarking environment. A benchmarking workflow designed as a combination of GitLab runners and JUBE scripts is described. This allows users to build, run, analyse, and archive the benchmark results. The synthetic benchmarks used to test infrastructure, communication, memory hierarchy, big data, data access patterns, and the processing power of scalability are described in Part II. In Part III, each of the use cases separately describes their JUBE scripts and the benchmark metrics obtained using them. In addition, all these applications have been successfully instrumented, and profiles and traces have been generated. The summary is presented in Part IV.

Part I.

Benchmarking

2. Benchmarking

In the computer sciences, benchmarking refers to running software with the intention to collect performance metrics on its execution. This is done to establish a performance baseline for the software or to compare the performance to an already existing baseline, not necessarily one from the software under investigation. Performance metrics are measurable quantities that can be 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 DEEP-SEA project many metrics are of interest, among them network communication times, I/O times and time spent in specific compute kernels.

Regular benchmarking as part of the DEEP-SEA project is necessary to ensure continued improvement in relevant metrics on a software level, as changes in either the software itself or its backbone affect the performance of the benchmarked applications. Therefore, benchmarking is an integral part in making sure project participants remain on track and to demonstrate the continual effort made to improve performance over the project duration.

We use a combination of synthetic and use-case–software benchmarks. The use-case–software benchmarks will allow us to demonstrate how changes made to the underlying software improve application performance as part of the DEEP-SEA project. The role of the synthetic benchmarks is to measure the effects of the hardware and software optimizations made during the project. This allows us to distinguish between variations in the application performance due to changes in the use-cases themselves, and variations due to either hardware or software changes in the underlying backbone.

2.1. The JÜlich Benchmarking Environment (JUBE)

We use the JÜlich Benchmarking Environment (JUBE) [2]. JUBE is a set of Python scripts that simplifies the organization of benchmarks by providing a consistent framework and keeping track of benchmark runs. It also delivers tools to aggregate benchmark results to aid analysis. We make use of these features to organize our benchmarks and to aggregate the results into tables, which we later upload to GitLab for easy access for our project partners.

Project partners were introduced to JUBE in a cross–SEA-project JUBE benchmarking workshop on the 1st of July 2021, which involved 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) described the role of Benchmarking within the SEA projects and its benefits for all involved partners. After that, participants were aided in setting up initial JUBE benchmarking scripts for their own software. All interested participants left the event with initial prototypes and the workshop was considered a success.

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.

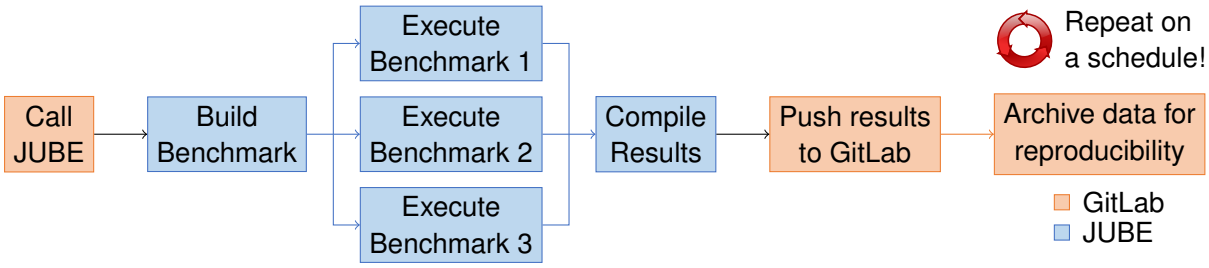


Figure 1: Workflow illustrating a GitLab runner starting JUBE, which in turn first builds the benchmark code and then uses the built binaries to execute three related benchmarks in parallel. Once these are finished, JUBE aggregates the results into a table, which the GitLab runner pushes to the relevant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

2.2. The Benchmarking Workflow

To successfully benchmark on-a-schedule, a benchmarking workflow is necessary. 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, for pushing the results up to GitLab and for archiving the benchmark run for later reference. JUBE executes the benchmark run, which generally consists of a build stage and an execution stage. During the build stage, the to-be-benchmarked software is built or, alternatively, binaries may be derived from elsewhere. In the second stage, JUBE executes the benchmark, which may consist of multiple steps running either serially or in parallel. Finally, JUBE can also collect results, if desired.

For the application 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. This version will only be updated if necessary when, for example, important bugs are resolved. 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 hardware.

2.3. DEEP-SEA Metrics

In the DEEP-SEA project we will look at a plethora of different metrics, among them will be the following:

- Runtime: The runtime of a software is a good measure of its general performance. It is also the easiest to measure. However, interpreting the cause of runtime changes is fraught with difficulties as changes to software and hardware tend to directly affect the runtime. Hence, it is a good indicator of performance improvements, but of poor informational value.
- Network-communication metrics: Network communication plays an important role for many HPC applications as they tend to use several nodes. This is even more important for the DEEP

system, which consists of multiple modules that must communicate efficiently in order to be utilized efficiently. This means that we want not only high bandwidths for inter-module and intra-module communication, but also low latencies. Wherever possible, computation should be overlapped with communication.

- I/O times: I/O is crucial for many HPC applications that deal with vast amounts of data. Keeping data-hungry CPUs and their executing software running efficiently depends on how fast data can be loaded. For local storage, this depends on the performance of the local storage, for network-attached storage, this additionally depends on the network capabilities. For DEEP-SEA, the use of tiered storage solutions, like the available non-volatile memory caches, can speed up applications.
- Kernel performance: Kernel execution speed is crucial for any application. Benchmarking with standardized kernels allows us to compare compute system performance and to monitor the health of a system. As many scientific applications spend the majority of their time in small kernels these are the most prone to optimization and therefore their performance is a good benchmarking metric.
- Memory speed: Network communication and I/O have been mentioned above. Intra-node communication, which refers basically to how fast processors can access memory, is also important because data intensive applications are often memory bound. Hence, optimizing memory usage directly affects performance. Therefore, benchmarking memory performance, in terms of bandwidth or latency, should be considered.

Please note that all of these metrics fundamentally depend on measuring some runtime, usually of a part of an application, from which the desired metric can be determined with the help of additional information.

The optimization cycles being developed as part of DEEP-SEA all directly affect one or more of the above metrics. Optimization cycles are workflows that involve users iteratively tracing relevant parts of their code to better understand the parts and to then optimize, those code parts. In this context a tracing of a code can be thought of as benchmarking these relevant parts thourougly by timing all associated functions calls. For more information please see DEEP-SEA Deliverable 3.1 [7].

2.4. DEEP System

The main hardware platform used in DEEP-SEA for software development and benchmarking is the DEEP system (Figure 2), a prototype modular supercomputer deployed within the DEEP-EST project. It consist of three compute modules and two storage modules, as summarized in Table 1 and Table 2. Note that the tables display the final configuration of the DEEP system, after the maintenance performed in December 2021. Before that time, the DAM module and one small ESB partition featured an Extoll interconnect, and gateway nodes were used to exchange messages between Extoll and InfiniBand. As for the storage modules, the SSSM was connected via a 40 Gb/s Ethernet network. The system has been now reconfigured with a uniform, InfiniBand-only interconnect across all modules.

Figure 2: Picture of the DEEP System, the prototype modular supercomputer built in the DEEP-EST project (predecessor of DEEP-SEA).

DEEP system	**Cluster Module (CM)**	**Data Analytics Module (DAM)**	**Extreme Scale Booster (ESB)**
Time of deployment	2019	2019	2020
Node count	50	16	75
CPU type	Intel Xeon 6146	Intel Xeon 8260M	Intel Xeon 4215
CPU codename	Skylake	Cascade Lake	Cascade Lake
Cores @ frequency	12 @ 3.2 GHz	24 @ 2.4 GHz	8 @ 2.5 GHz
Accelerators per node	n.a.	1× Nvidia V100 GPU 1× Intel Stratix10 FPGA	1× Nvidia V100 GPU
DDR4	192 GB	384 GB + 32 GB (FPGA)	48 GB
HBM	n.a.	32 GB (GPU)	32 GB (GPU)
NVMe	n.a.	3 TB Intel Optane	n.a.
Node max. mem BW	256 GB/s	900 GB/s (GPU)	900 GB/s (GPU)
Storage	1× 512 GB NVMe SSD	2× 1.5 TB NVMe SSD	1× 512 GB NVMe SSD
Network technology	EDR-IB (100 Gb/s)	EDR-IB (100 Gb/s) Ethernet (40 Gb/s)	EDR-IB (100 Gb/s)
Topology	fat-tree	tree	tree
Power draw per node	500 W	1600 W	500 W
Cooling	warm-water	air	warm-water

Table 1.: Key hardware features of the DEEP system (after network upgrade on 16.12.2021).

DEEP system	Scalable Storage Service Module (SSSM)	All-Flash Storage Module (AFSM)
Time of deployment	2019	2021
Metadata Storage Server	2	2
CPU type	2× Intel Xeon 4114	2×Intel Xeon 6246
CPU codename	Skylake	Cascade Lake
Cores @ frequency	20 @ 2.2 GHz	12 @ 3.3 GHz
DDR4	96 GiB	384 GiB
Storage (metadata)	2× 480 GB SAS SSD PX05SV (RAID1)	8×3.2 TB Intel SSD DC P4610
Network technology	EDR-IB (100 Gb/s)	EDR-IB (100 Gb/s)
Object Storage Server	4	6
CPU type	2× Intel Xeon 4114	2×Intel Xeon 6226R
CPU codename	Skylake	Cascade Lake
Cores @ frequency	20 @ 2.2 GHz	16 @ 2.9 GHz
DDR4	96 GiB	384 GiB
Network technology	EDR-IB (100 Gb/s)	EDR-IB (100 Gb/s)
Storage (object data)	2 servers share a RAID enclosure	24×15.36 TB Intel SSD DC P4326
RAID enclosure	2	
HDD	24×8 TB SAS Nearline (2 RAID6 volumes)	
Network technology	2×16 Gb/s fibre channel	

Table 2.: Key hardware features of the storage modules, SSSM and AFSM, of the DEEP system (after network upgrade on 16.12.2021).

2.5. Data Management

Benchmarking always generates data, which is used to either establish a baseline or allows for comparison to a baseline. Additional data generated usually consist of logs, input data and additional output data that can help diagnose unexpected benchmark behaviour and allows for reproducibility.

For example benchmarking the runtime of a scientific application results in two data types:

(1) Benchmark metrics, like the runtime, which we are ultimately interested in.

(2) Reproducibility information describing the system state at the beginning, during, and at the end of the benchmark run, including the necessary input data and the generated output data not related to benchmakring.

The most valuable data in the benchmarking context are the runtimes, either as explicit times or as benchmark metrics. Therefore, after each benchmark run, the resulting benchmark metrics are uploaded to GitLab in a commit for safekeeping and easy access from project partners.

The data types not directly useful for the benchmarking effort, however, are required for approximate reproducibility given a sufficiently similar benchmarking system. Hence, they need also to be stored, but losing said data is not as catastrophic as losing the actual benchmarking data. As such, we opted to archive these data types where possible. This ensures that the data can be queried at a later stage for forensic analysis of benchmark runs that exhibit anomalous behaviour, or for us to be able to repeat an anomalous benchmark. Note that this includes all input data to a benchmark. These also need to be stored where relevant if they are not readily reproducible.

2.6. Future Plans

Our goal is to regularly run our synthetic and use-case benchmarks throughout the life of the project on the DEEP system, the main hardware platform in DEEP-SEA, to monitor its health and project developments.

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

We also plan to expand the tools for visualization of the results, and potentially integrate it into the benchmarks to enable easier-to-digest access to benchmarking results. Some visualization 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, visualization is necessary for some benchmarks to easily allow our partners to extract insights from the benchmarking activities.

2.7. 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 [8] 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 3 summarizes the benchmarking collaboration. The Figure also highlights how TSMP, see Chapter 16, 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 3—8, 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.

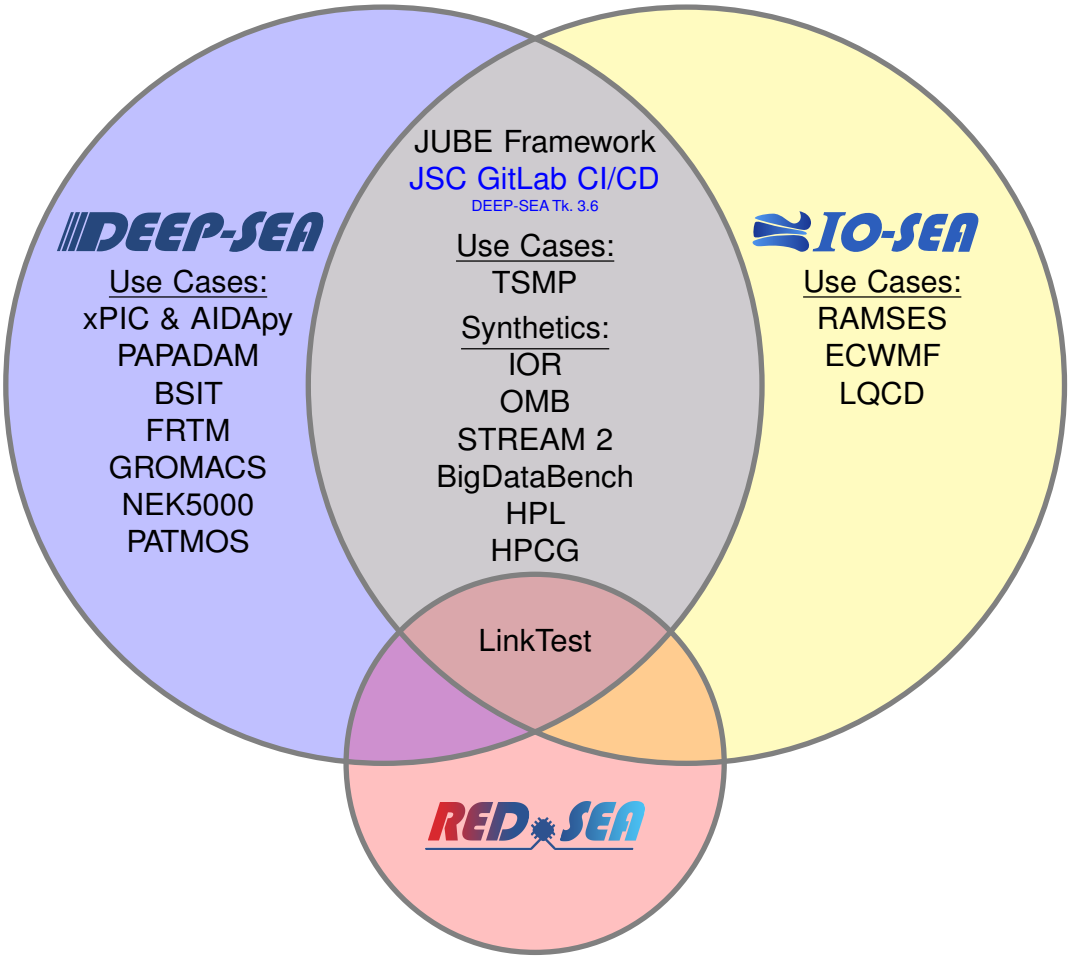


Figure 3: Benchmarking-collaboration Venn diagram that highlights the benchmarking activities of DEEP- and IO-SEA Task 1.2 and how they are spread out across the three SEA projects.

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.

Part II.

Synthetic Benchmarks

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

The Interleaved or Random (IOR) benchmark [9] allows 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 testing the performance of the DEEP system with respect to different types of I/O workloads.

3.1. 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 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 4):

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

(2) a 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 measuring, e.g., the creation and deletion of a defined number of files and a hierarchy of directories.

3.2. Results

For our IOR benchmarks (Figure 5) using the parallel API MPI-I/O, we have set block size and transfer size equal to the chunk size of the storage system. The aggregate file size has been set by defining the number of segments. Thereby, we allow well aligned striping access to the storage system, which is reflected by good performance values. The measured average read throughput using a single, two, or four nodes of the Cluster Module (CM) demonstrate that the flash-based storage module is capable of saturating the available bandwidth of up to 400 Gbs^{-1}. At eight nodes the read throughput peaks at about 1.73×2^5 $GiBs^{-1}$ (corresponding to 55.3 GiB/s) and falls off beyond 16 nodes, which is probably due to the high number of MPI processes competing for resources at the All-Flash Storage

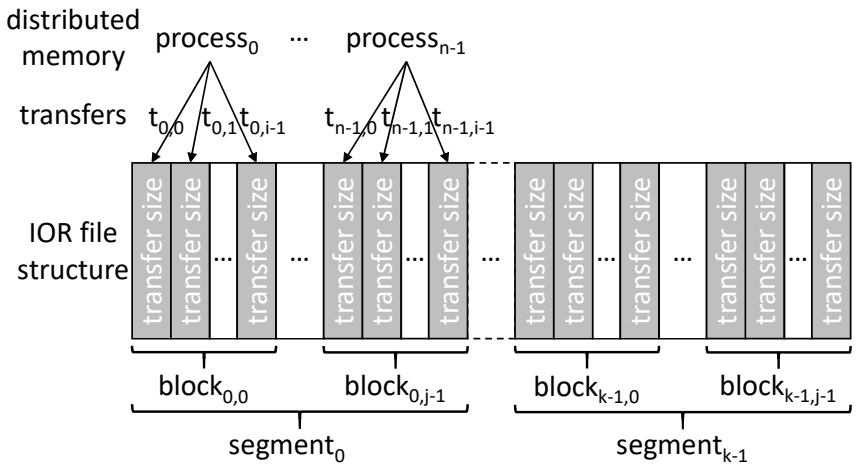


Figure 4: IOR file structure and access pattern. An IOR data stream consists of one or more segments. Each allocated process in distributed memory accesses exactly one block per segment. Each block consists of either a single or multiple chunks. The size of these chunks is equal to the transfer size and corresponds to the amount of data transferred per I/O function call. The figure was adapted from [10, 11].

Module (AFSM). The average write throughput using a single or two nodes shows that the AFSM can fully saturate the aggregate bandwidth of up to 200 Gbs^{-1}. Using 4, 8 or 16 nodes, write throughput peaks at about 1.59×2^4 GiBs^{-1} (corresponding to 25.4 GiB/s) not saturating the available bandwidth. However, the measured peak read and write data rates fit well to the technical specification of the used flash storage devices, which states that their write speed is only half of their read speed [12].

In contrast to the AFSM the performance of the Scalable Storage Service Module (SSSM) is much lower and already peaks using a single node only with a write speed of about 1.0×2^2 GiBs^{-1} (corresponding to 4 GiB/s) and a read speed of 1.34×2^0 GiBs^{-1} (equivalent to 1.34 GiB/s). Surprisingly, writing is consistently faster than reading across all node configurations tested. One possible explanation is that, when reading data, access to chunks is not evenly distributed among the storage servers, which lowers the effective aggregate bandwidth available. Another reason could be that jobs from other users running at the same time on the DEEP system can have a noticeable influence on our benchmarks.

The results of the metadata benchmarks underline the performance advantage of current flash-based mass storage devices. (Figure 6). 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 (Figure 6C,D).

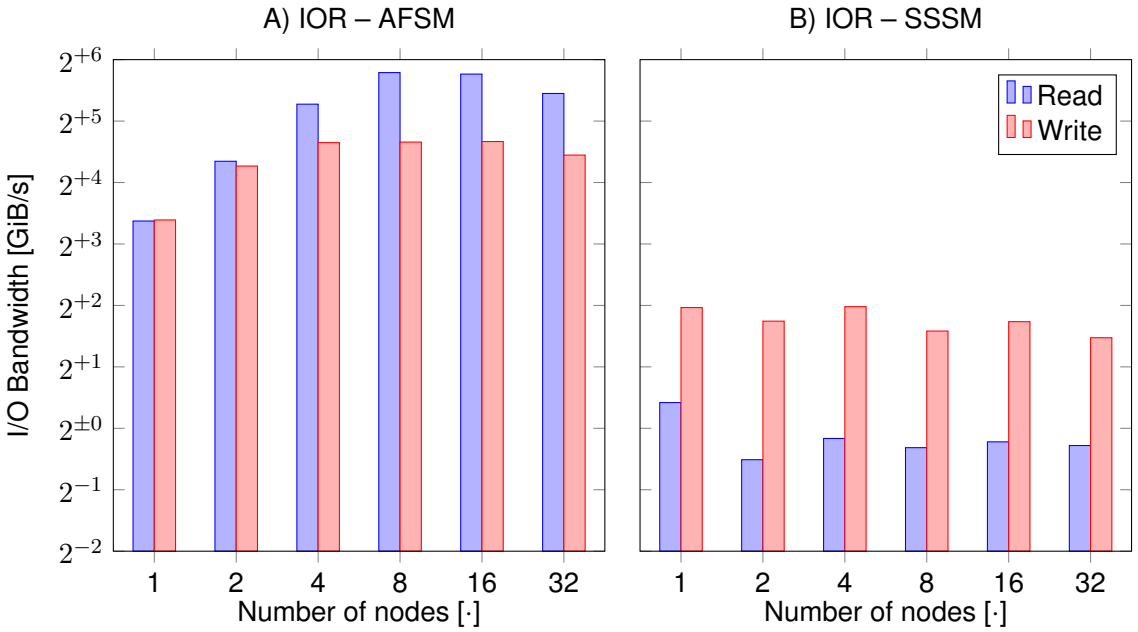


Figure 5: Strong scaling of write and read performance with IOR on (A) the AFSM and (B) the SSSM of the DEEP system is shown. Throughput was measured with a transfer size of 512 KiB (equal to chunk size of storage system), the block size was set to correspond to the transfer size. Each MPI process wrote 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 4608 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 Ethernet network.

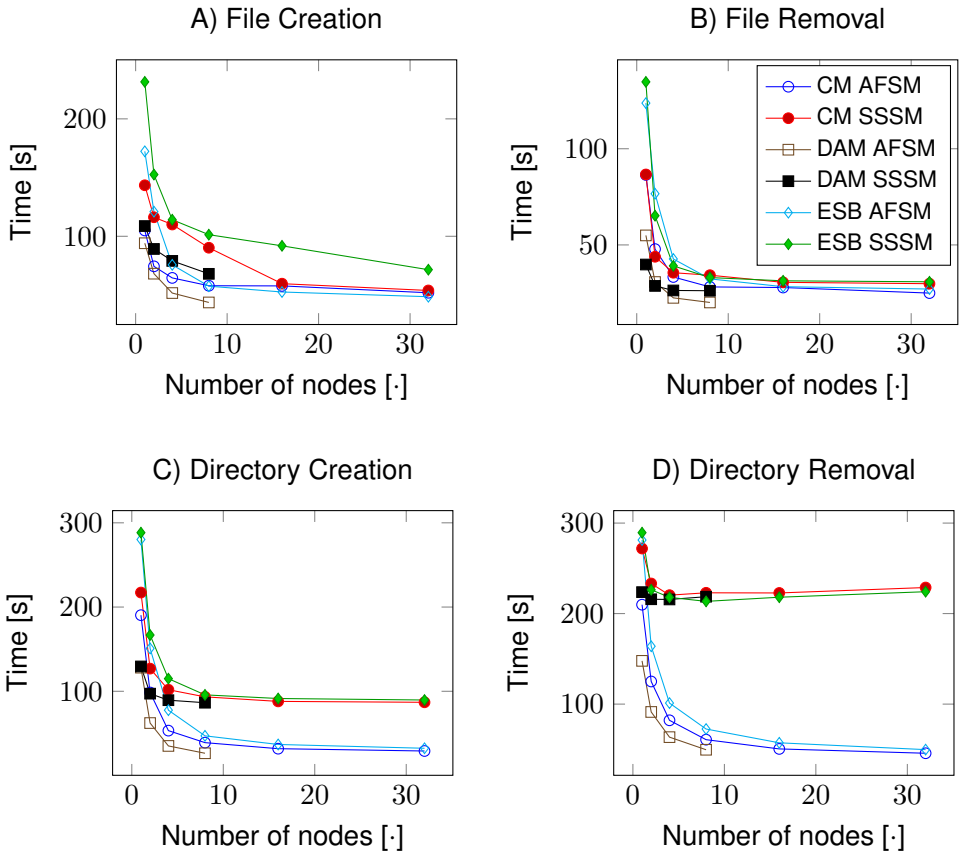


Figure 6: Strong scaling of file and directory creation and removal operations on the AFSM and the SSSM of the DEEP system is shown. Separate tests using up to 32 nodes were carried out one after the other to avoid mutual interference. Each test was repeated five times by mdtest and the average values 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 Ethernet network.

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

The OMB [13] is a suite of Message-Passing Interface (MPI) benchmarks aimed at testing all available MPI functions. It consists of many small executables designed to test one aspect of MPI, each. The goal of this synthetic benchmark is to ascertain the communication performance of the DEEP-SEA test infrastructure.

4.1. 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 are the number of messages to pass and the message size. The number of messages defines the number of communications over which the results are averaged; the message size influences what exactly is measured, as MPI behaviour strongly depends on the message size passed to it. Furthermore, a 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.

4.2. Results

The OMB can be used to generate a plethora of graphs of a system's MPI performance. We have chosen to restrict ourselves to the most readily understandable results: MPI latency and bandwidth. MPI communication latency describes the inherent unavoidable time delay associated with the communication. In general, the MPI communication time t consists 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 latter is referred to as t_b, where the b stands for bandwidth, because it directly depends on the bandwidth of the network over which the message is communicated. Mathematically the total MPI communication time can be written as:

$$t = t_l + t_b(m), \tag{4.1}$$

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.

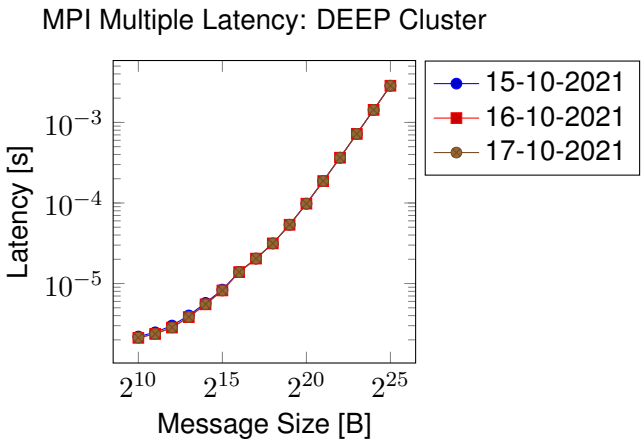


Figure 7: A comparison of multi-latency OMB tests run on the DEEP Cluster Module (Intel Xeon Skylake CPUs).

Figure 7 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 8 shows a comparison of multiple multi-bandwidth tests run on the DEEP Cluster Module. Again we see excellent run-to-run consistency with a small degree of variance. There is a kink in the bandwidth, which is hardly visible in the message rate due to scaling, for message sizes around 8–64 kiB (2^{13}–2^{16} B), presumably, due to CPU related cache effects. The peak bandwidth caps out at around 10 $MiBs^{-1}$. As the bandwidth approaches the cap, the message rate decays linearly.

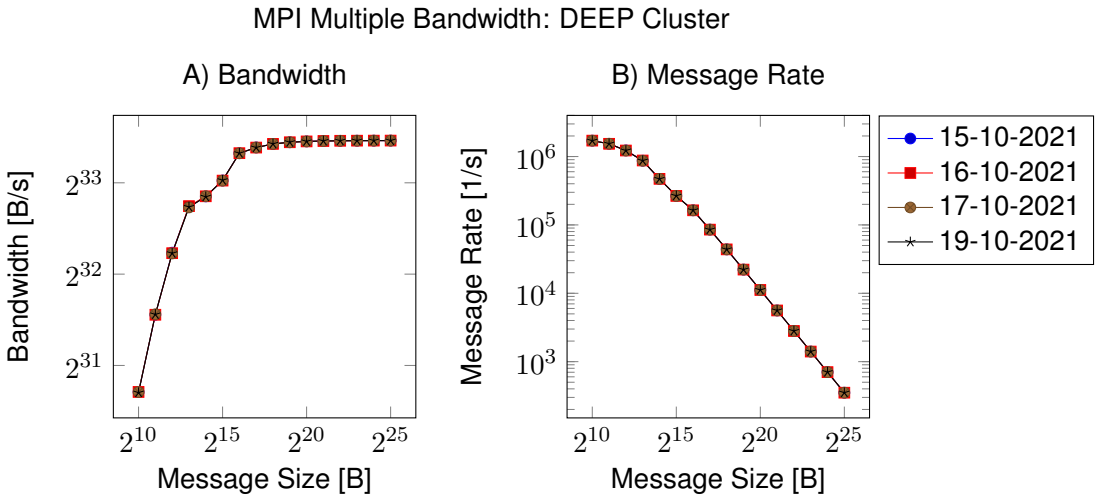


Figure 8: A comparison of multi-bandwidth OMB tests run on the DEEP Cluster Module (Intel Xeon Skylake CPUs).

5. Linktest

5.1. 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 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 messages are stored can be controlled. They can reside either in CPU RAM 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 a 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 program's output is a full timing communication matrix of the message-transmission times for all pairs of tasks written to a SION file using the SIONlib package [14]. A standard-out log that summarizes 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 DEEP-SEA project, Linktest is used to benchmark communication, an important factor in the performance of HPC applications. The MPI benchmarks are used here because MPI is the most common communication API in HPC, and for easy comparison with the benchmark results of OMB Chapter 4. This helps us to corroborate results. Note that using multiple different types of benchmarks that measure the same parameter makes it more likely to pick up on errors for which one of the benchmarks is more susceptible. Furthermore, 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.

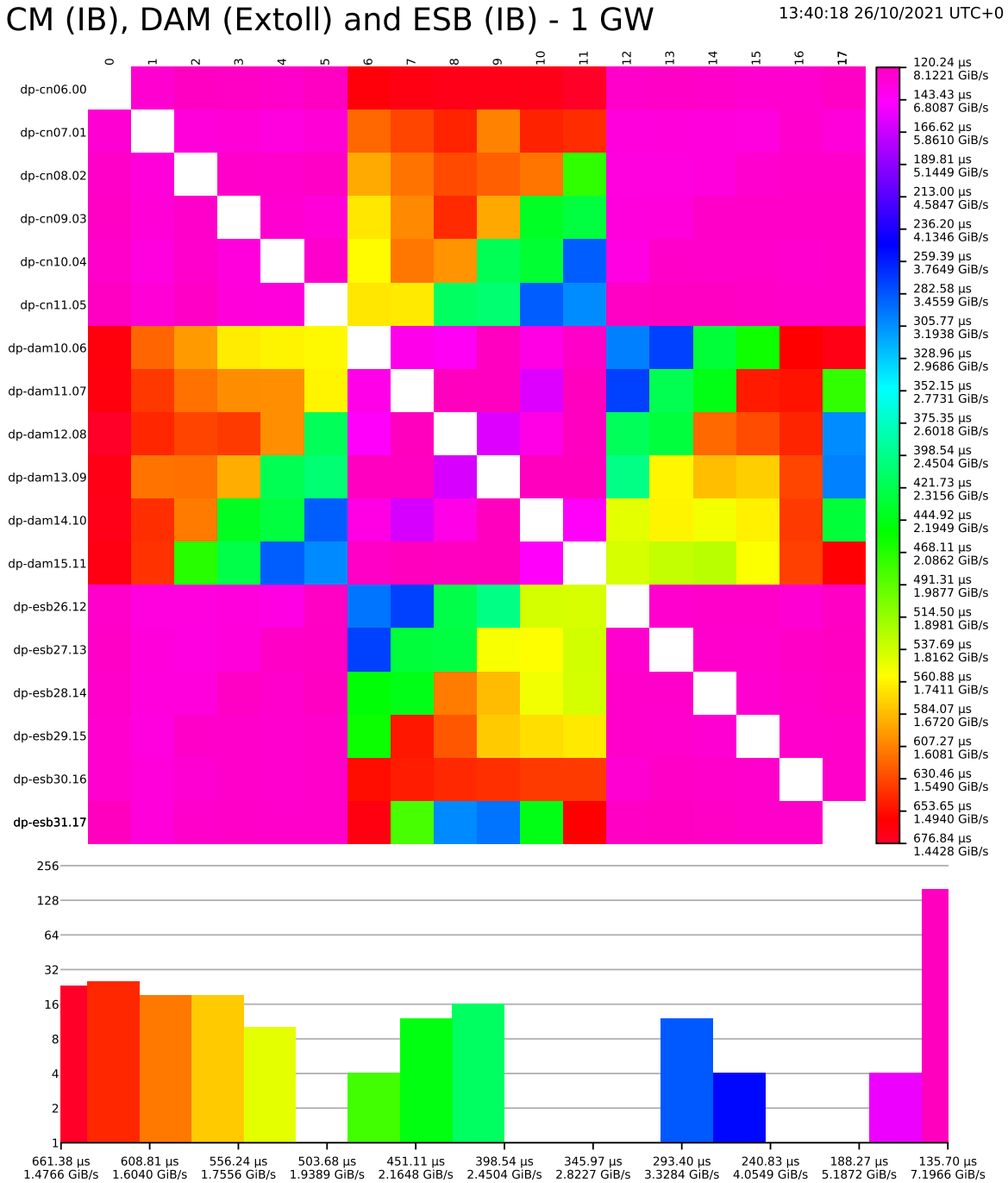
5.2. Results

Page 30 shows a one-page Linktest report from a benchmark run on the CM, DAM, and ESB modules. The top indexed-colour plot shows the average measured communication time to send one thousand 1 MiB messages back and forth using MPI. Six nodes per module with one task per node were used for this test, resulting in 18 tasks total. The fastest communication times were recorded within modules or between the CM and ESB module: the pink blocks. We see the same performance inside and between the CM and ESB because they both used the same InfiniBand fabric and are connected through it. A gateway node was used to translate messages between the Extoll network of the DAM and the InfiniBand network of the CM or the ESM, when communicating from the DAM to either the CM or the ESM. This means that the performance of a given connection through the lone gateway depended on how many connections for a given step in the parallel Linktest test had to go

through said gateway at the same time. For the dark-blue pixels in the image, only one of the tested connections was routed through the gateway, while for the red pixels six connections were routed through the gateway, resulting in poorer performance. This is the reason for the interesting structure in the indexed image plot.

In the histogram, the dominant pink peak corresponds to intra-module or CM to ESB communication. The rest is attributable to cross-module communication involving the DAM, and therefore network gateways.

Note that after the system maintenance on 16.12.2021 the network configuration has been homogenized to use InfiniBand in all modules.



Communication API:	mpi	Number of Tasks:	18	Number of Hosts:	18
Number of Messages:	1000	Message size:	1,048,576 B (1.0000 MiB)		
# Warm-Up Messages:	10	# Serial Retests:	10	Domain:	
Execution Order:	Parallel	All-to-All:	No	Mixed Ranks:	No
Test Configuration:	Semidirectional	Bisection:	No	Use GPUs:	Yes
Min. Value:	120.24 µs 8.1221 GiB/s	Average Value:	320.19 µs 3.0500 GiB/s	Max. Value:	676.84 µs 1.4428 GiB/s

Worst Connections Serially Retested:
dp-dam12.08 to dp-cn06.00: 676.84 µs (1.4428 GiB/s) → 262.78 µs (3.7163 GiB/s)
dp-cn06.00 to dp-dam15.11: 676.46 µs (1.4436 GiB/s) → 263.00 µs (3.7131 GiB/s)
dp-cn06.00 to dp-dam14.10: 670.52 µs (1.4564 GiB/s) → 262.01 µs (3.7271 GiB/s)
dp-cn06.00 to dp-dam12.08: 670.22 µs (1.4571 GiB/s) → 262.86 µs (3.7151 GiB/s)
dp-cn06.00 to dp-dam13.09: 670.22 µs (1.4571 GiB/s) → 262.69 µs (3.7176 GiB/s)

Linktest Version: 2.1.12 Report generated by the FZJ Linktest Python Report Tool 2.1.12, Forschungszentrum Jülich GmbH

6. STREAM 2

The STREAM and STREAM 2 benchmarks [15] measure sustainable memory bandwidth and corresponding computation rate for simple vector kernels. As a result they are used for benchmarking the memory hierarchy of modern compute systems.

6.1. 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 reading and writing. 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, due to the widening gap between CPU performance and memory bandwidth. Benchmarking performance across memory tiers, therefore, gives us direct insight into how certain operations may perform.

STREAM 2 offers four different vector kernels for benchmarking:

(1) *Fill* fills 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 exhibit the smallest memory bandwidth for small array sizes as it only writes to memory, 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 the 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* boasting the larger memory bandwidth for very small array sizes. For larger array sizes the aggregated memory bandwidth achieved by the *DAXPY* kernel is greater due to the presence of specialized instructions for the kernel in many modern CPUs.

6.2. Results

This kernel behaviour can easily be corroborated by running the benchmark. Figure 9 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

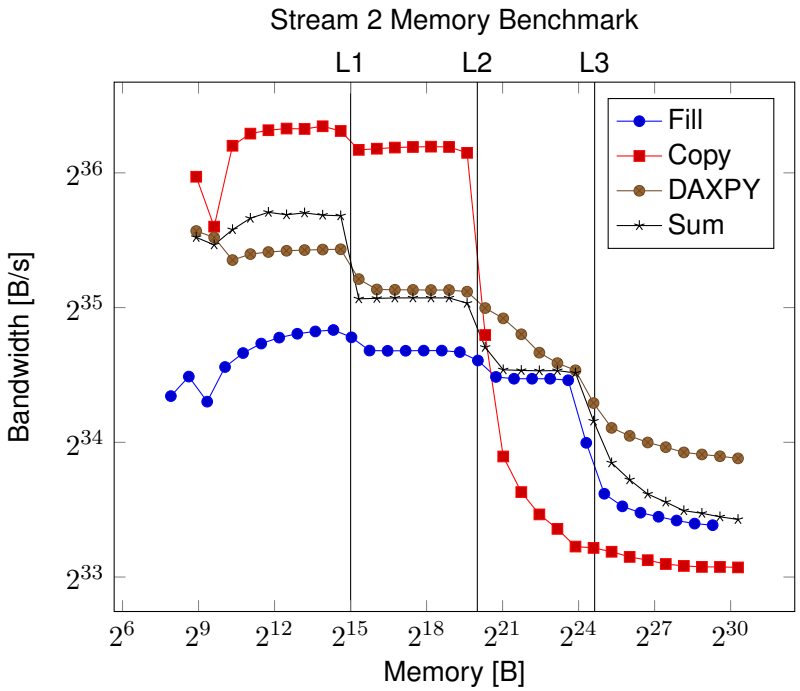


Figure 9: Stream 2 benchmark results from an Intel Xeon Gold 6146 CPU in the DEEP Cluster partition. The L1, L2 and L3 cache sizes are indicated using vertical lines.

changes in memory caches. The change in copy performance is especially apparent as soon as the array size exceeds half the L1-cache size.

7. High Performance Conjugate Gradient (HPCG)

HPCG [16, 17] was developed as a complement to the High Performance LINPACK (HPL) benchmark (see also Chapter 8). HPCG focuses on computation and data access patterns that frequently occur in scientific software, to better reflect the expected real world performance of HPC systems.

7.1. Description

HPL mainly favours HPC systems that excel at dense-matrix computations and the streaming of coalescent memory regions. HPCG, on the other hand, is mainly concerned with other frequent computation and data access patterns, which are characterized 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 [18].

7.2. Results

The scaling behaviour of HPCG on the DEEP system was benchmarked on up to 32 compute nodes using CPUs only. 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 [19]. 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 sec [20]. As a compromise between saving 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.

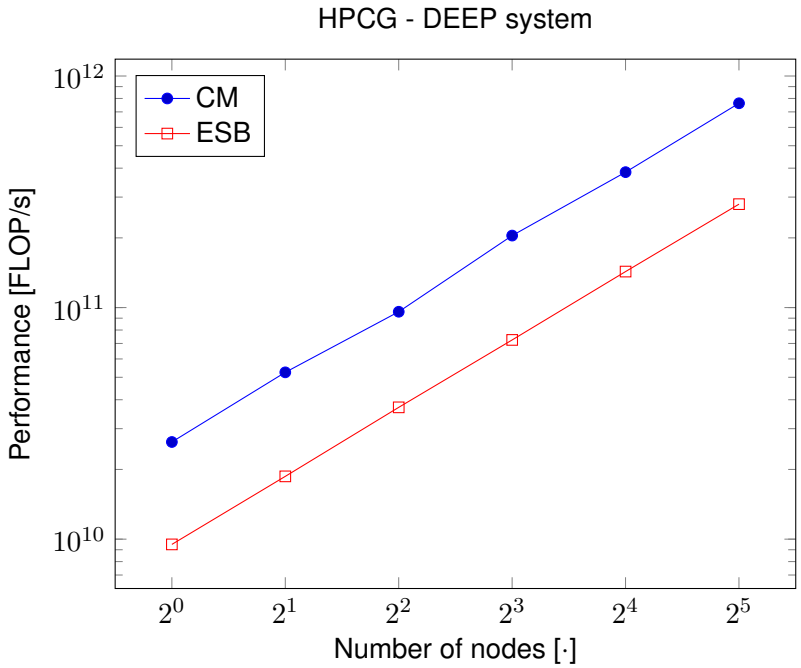


Figure 10: Scaling demonstration for HPCG from the major modules of the DEEP system. The node-local problem dimension was set to x=256, y=128, z=128. The number of MPI tasks per node was equal to the number of CPU cores per node. For this weak scaling test, HPCG was run on up to 32 nodes in parallel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

8. High Performance LINPACK (HPL)

The HPL software package [21] solves linear equation systems. It is commonly used to solve ISO/IEC 60559:2020 [22] 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.

8.1. Description

The HPL benchmark provides an estimate of the achievable peak performance, which is generally less than the often-unattainable theoretical peak performance provided by manufacturers. The latest version of this benchmark is commonly used to build the TOP500 list [23], which ranks the world's most powerful supercomputers in terms of their peak performance measured in Floating Point Operations per Second (FLOP/s). The Intel(R) distribution of the LINPACK Benchmark [24] is one such implementation of the Massively Parallel LINPACK benchmark, developed and optimized for Intel processors. This is the version we use for benchmarking our systems because the CPUs used in the DEEP system are from Intel.

The most important input parameters for this benchmark are N, P, Q, and NB [25]. 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 set up 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 70-90% of the total available memory.

However, for the benchmarks reported in this deliverable, we set the problem size to $N = 50000$ to ensure the benchmarks complete within a stipulated amount of time. It is also chosen so that the reported performance for a given problem size for the two modules (CM, ESB) can be compared on the same plot over the duration of the project. This is done because both modules have varying amounts of memory per node, and a problem size significantly greater than $N = 50000$ 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.

8.2. Results

We performed a strong scaling test independently on the two hardware partitions of the DEEP system with the same problem size ($N = 50000$), for a varying number of nodes ranging from 1 to 32, increasing in powers of 2.

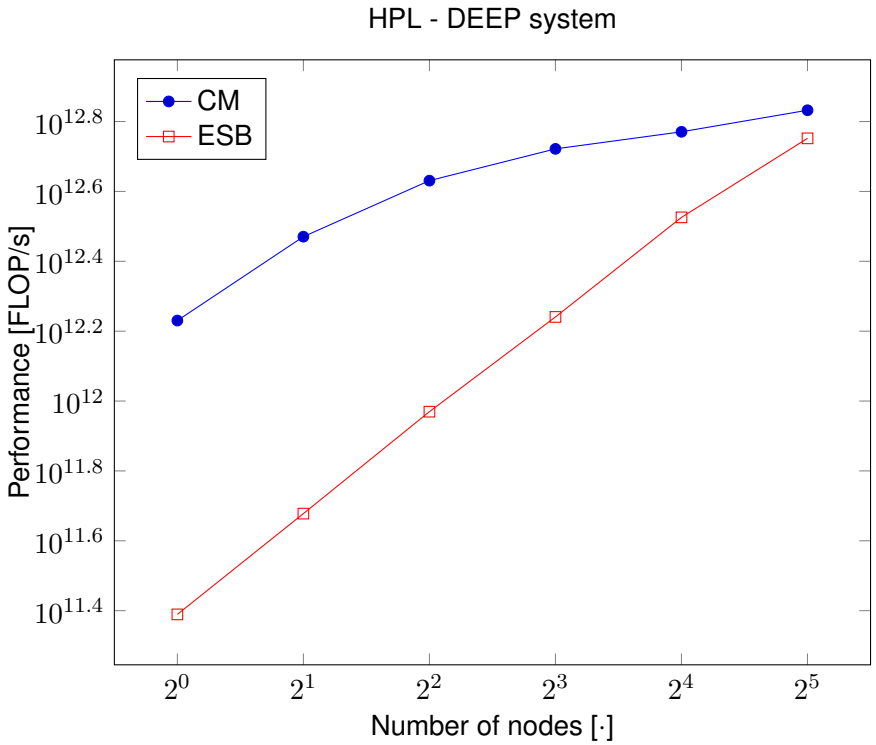


Figure 11: Strong scaling measurements for Intel optimized HPL on the major DEEP system modules. The problem size is set to $N = 50000$, and the block size of the data distribution NB is set to 384. The benchmark is set up to use all available cores on each node and compiled using ParaStation MPI version 5.4.9-1.

Figure 11 shows that the CM reports higher performance than the ESB. This can be explained due to higher number of powerful CPUs on the CM, when compared to the ESB module.

Figure 11 also shows that for the given problem size, the ESB module exhibits almost perfect scaling, and that the obtained speedup for the CM module saturates 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 performance numbers do not reflect the highest possible performance attainable on each of the modules, rather they reflect only the performance attainable for the chosen input parameters. For a larger problem size, the CM module scales almost perfectly and reports higher performance.

Part III.

Use-Case Benchmarks and Traces

9. Space Weather

Space weather studies the physics of the energetic activity of the Sun, the propagation of solar plasma through the solar system, its interaction with the complex plasma environment that envelops all the planets. Space weather studies the effects of these interactions on the Earth's atmosphere, on human life and technology. Space weather has a high societal relevance for its impact on multiple industries (satellite operations, telecommunications, geolocation, electric distribution, space exploration, among others). During the DEEP projects, KU Leuven has developed the particle-in-cell code xPic, which studies the effects of solar plasma on the environment of different planets in the solar system. A suite of Data Analysis (DA) and Machine Learning (ML) techniques are employed around xPic to set up the initial conditions of the code and its boundary conditions and to analyse the generated data. This space weather workflow already uses the MSA and runs on the DEEP system hosted at JSC.

The code xPic has two solvers: a field solver for the computation of electromagnetic fields (which runs on the CM), and a particle solver for the transport of plasma ions and electrons (which runs on the ESB). Terabytes of data are produced in the particle solver. The code AIDApy creates initial conditions for the simulation (using Deep Learning, surrogate models, or Self-Organizing Map techniques) and analyses the velocity distribution data of the particles (using Gaussian Mixture Models).

9.1. xPic: plasma physics HPC code

The code xPic is based on the Particle-in-Cell (PIC) algorithm. The physical phenomenon that xPic studies is the dynamics and transport of space plasmas. Plasma is the fourth state of matter: when very high pressures or temperatures are applied to matter, electrons are peeled from atoms producing two types of elements of different mass and different electric properties. These two types of particles (ions and electrons) carry opposite electric charges (positive and negative). The movement of each individual particle is then governed by electromagnetic forces. At the same time, the movement of the particles causes changes in the total electric charge of the environment and can produce currents that modify in turn the electric and magnetic fields. In a plasma the charged particles and the electromagnetic fields are tightly coupled.

Figure 12 shows the general structure of a PIC code. The algorithm is divided into four main phases:

(1) a *field solver* calculates the evolution of Maxwell's equations of electromagnetism using a numerical solver,

(2) a *particle solver* transports billions (or trillions) of individual and independent particles of different electric charge (ions and electrons),

(3) the movement of the particles depends on the forces interpolated from the field solver to the particle solver,

(4) and finally statistical information is gathered from the particles by integration of different moments of the velocity distribution function.

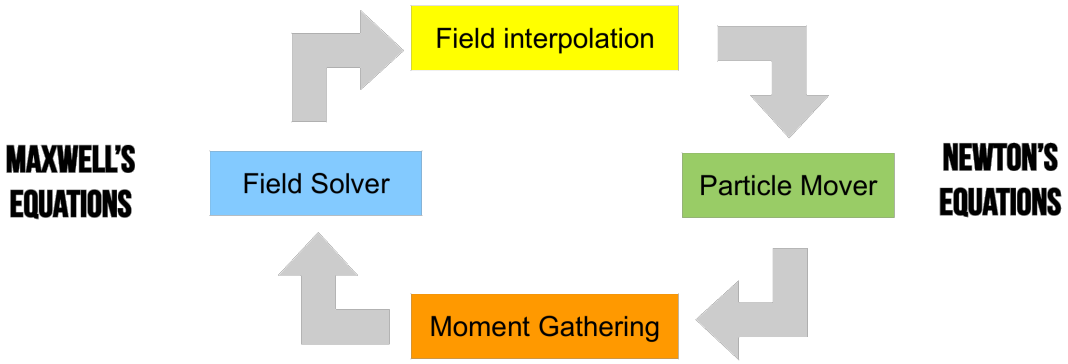


Figure 12: Four phases of the PIC algorithm used in the code xPic (KU Leuven).

9.2. JUBE scripts

The code xPic contains a suite of JUBE scripts that can be used to perform benchmarks and microbenchmarks in order to analyse the performances of the code or the effects of modifications in the sources. The majority of these scripts were created during the DEEP-EST project and still contain examples used for the benchmarking of different architectures, including the Intel KNL processor and a 64 core AMD Epyc processor. The git repository of xPic contains also the JUBE scripts for benchmarking the code on the DEEP and the JURECA systems of JSC.

The scripts contain weak scaling benchmarks of problems in 1D and 2D. During previous developments, we have included also what we call *microbenchmarks* allowing us to reduce the total runtime of the benchmarking, increase the quality of the benchmarking statistics, and perform traces using the tools developed by the BSC (Extrae and Paraver).

Up until now, our developments have been focused on the acceleration and optimization of the particle mover. Simple plasma simulations, where the electromagnetic fields are quasi-constant, are a good setup to analyse particle algorithms. However, we have noticed more recently that the field solver has become more and more important in simulations where strong changes in the electromagnetic fields are involved. We are currently developing and integrating benchmarks that demand a more realistic effort of the field solver.

9.3. Extrae/Paraver traces

In this section, we present the performance results of the code xPic under basic run conditions. The code was run on a single node of the JURECA Cluster at JSC. We used the code in the MONO mode, i.e., all the phases of the code are executed by the CPUs of the node. We ran the code in two different configurations: in the first case we allocated 24 MPI processes with 1 OpenMP thread per MPI process, and in the second we did the opposite: keeping a single MPI process with 24 OpenMP threads. In this section, we report the classical execution situation based on the former case (using 24 MPI processes).

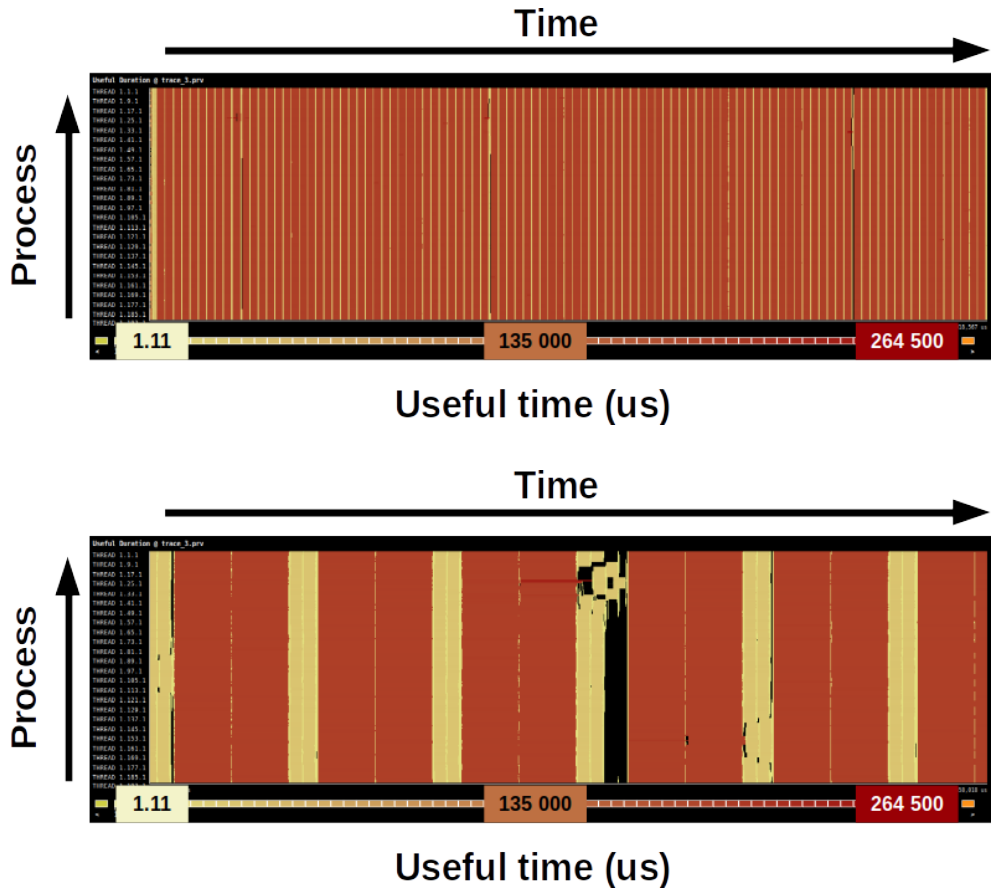


Figure 13: *Useful duration* of the code xPic. The horizontal axis represents the execution of the code in time. The vertical axis is the MPI process. The colours represent the time in μs spent outside the MPI calls. Top panel: full execution of the benchmark. Bottom panel: enlarged section showing five iterations of the code (KU Leuven).

Figure 13 shows the traces obtained for a run with 100 iterations on a 1D simulation composed of 147456 cells, with 1024 particles per cell of two different species (negative electrons and positive ions). The tool Extrae, developed by the BSC, measures multiple hardware and software counters during the execution of the code. It keeps a record of the moments where the MPI or the OpenMP libraries have been called. Extrae generates a trace (a set of measurements over time and over multiple processes) that can be analysed in the Paravver GUI (also developed by BSC). A trace is presented as a 2D image, where the horizontal axis shows the execution time, the vertical axis the thread (or process) number, and the color represents any given metric like the execution time of a section of code, the IPC, the memory access, the MPI communication times, etc. The top panel of Figure 13 presents the total *useful duration* of the full execution, defined in Extrae/Paraver as the time spent by the code on regions outside MPI calls.

The bottom panel of Figure 13 shows an enlarged section where the code presents a strange behaviour. In this panel it is possible to observe five full cycles of the code. The colours represent the duration of the useful instructions of the code. The colors yellow and brown correspond to approximately one thousand (1000) and two hunderd thousand (200000) microseconds of *useful* execution time. Each cycle is mainly composed of four clearly defined parts: the two larger parts, indicated with the color brown, correspond to the mover phase of the code for each one of the two particle species, while the two shorter sections, colored in yellow, correspond to the moment gathering of the two species. In this zoomed-in section of the code we observe that in the middle of the window a single core in the mover phase of the second species is delaying the normal execution of the code. These sudden execution errors have been identified in the past as run-to-run performance variabilities due to hardware issues. In this particular run, this small bottleneck did not produce an unbalance in the code. An analysis of the *MPI call profile* provided by Extrae, shows that the code in the current run presents a very high balance, with a parallel efficiency of 97.55%, a communication efficiency of 98.56% and a load balance ratio of 0.99.

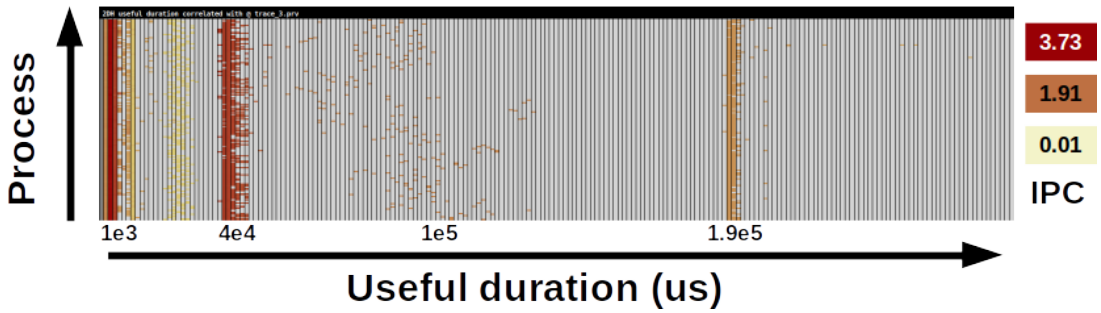


Figure 14: Instructions per cycle (IPC) of xPic during the useful duration of the code. The horizontal axis corresponds to the duration of the useful sections of the code. The vertical axis corresponds to the MPI process. The colours correspond to the IPC (colour scale from low values in green to high values in blue) (KU Leuven).

Figure 14 shows a histogram that summarizes the performance of the code. This histograms are a second method used by Paraver to present the information extracted (and processed) from the Extrae traces. In this particular case the horizontal axis represents the useful duration, and the colour of the histogram represents the total Instructions Per Cycle (IPC). The almost vertical lines show that there is a very good load balancing of the code: almost all processors execute the same number of instructions during the same amount of time. The largest amount of time, around $1.9 \times 10^5 \mu s$, is spent

on sections of the code where every processor has an IPC of just above one (see color scale; actual values can be observed in the Paraver GUI). This is not an optimal solution and requires optimization.

Figure 15 shows that the particle mover is responsible for these sections of the code. A new version of the code that offloads the particle mover to GPUs has already been developed. We are improving the software in order to accelerate both the particle mover and the moment gathering. Looking again at Figure 14, the points in the histogram with an useful duration between $5\times10^4\mu s$ and $3\times10^5\mu s$, are unbalanced, i.e., there is no vertical line which indicates that each core performs operations over different periods of time. However, we identified that this section of the code corresponds to the initialization previous to the execution of the main 100 cycles. From this we can conclude that our application presents very good parallel load-balancing.

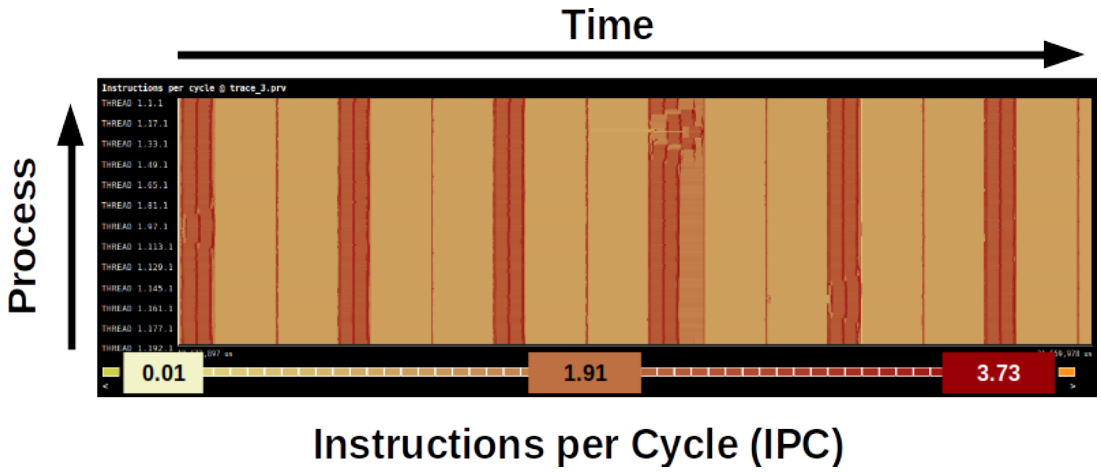


Figure 15: Instructions per cycle (IPC) for slightly over five iterations of the code. The yellow sections correspond to an IPC of 1.14 on average while the red sections present values of around 2.2. (KU Leuven).

These are the first steps in the optimization of our codes. During the following months, we will include additional benchmarks and produce more detailed traces of complex simulations. We are in the process of instrumenting the newer version of the code which runs in the Cluster–Booster mode with GPU offloading.

10. Weather Forecast and Climate

10.1. The IFS

The Integrated Forecasting System (IFS) is the weather forecasting suite jointly developed by ECMWF and Météo-France. It is run operationally at ECMWF, currently on a Cray XC40 system, in Reading, UK, with migration of operations in 2022 to an AMD-based Atos BullSequana XH-2000 machine, in Bologna, Italy. It is recognized as capable of generating global forecasts of world-leading accuracy in a computationally efficient manner on current CPU-based systems. However it is currently not able to make good use of heterogeneous compute ressources, be it intra-node heterogeneity in the form of compute accelerators on every node, or hardware heterogeneity across nodes, such as exposed by MSA-style systems.

As described in Deliverable D1.1 [1] section 3.1, coupled, global forecasts carried out with the IFS suite consist of 3 main components, namely the atmosphere, covered by the IFS itself, the ocean, represented with NEMO, ("Nucleus for European Modelling of the Ocean"), and the ocean-atmosphere interface, covered by WAM, the ECMWF WAve Model. As a reminder, dependencies between the three components are illustrated in Figure 16.
Improvements to the technical implementation of the coupling will be one focus of DEEP-SEA work for the IFS.

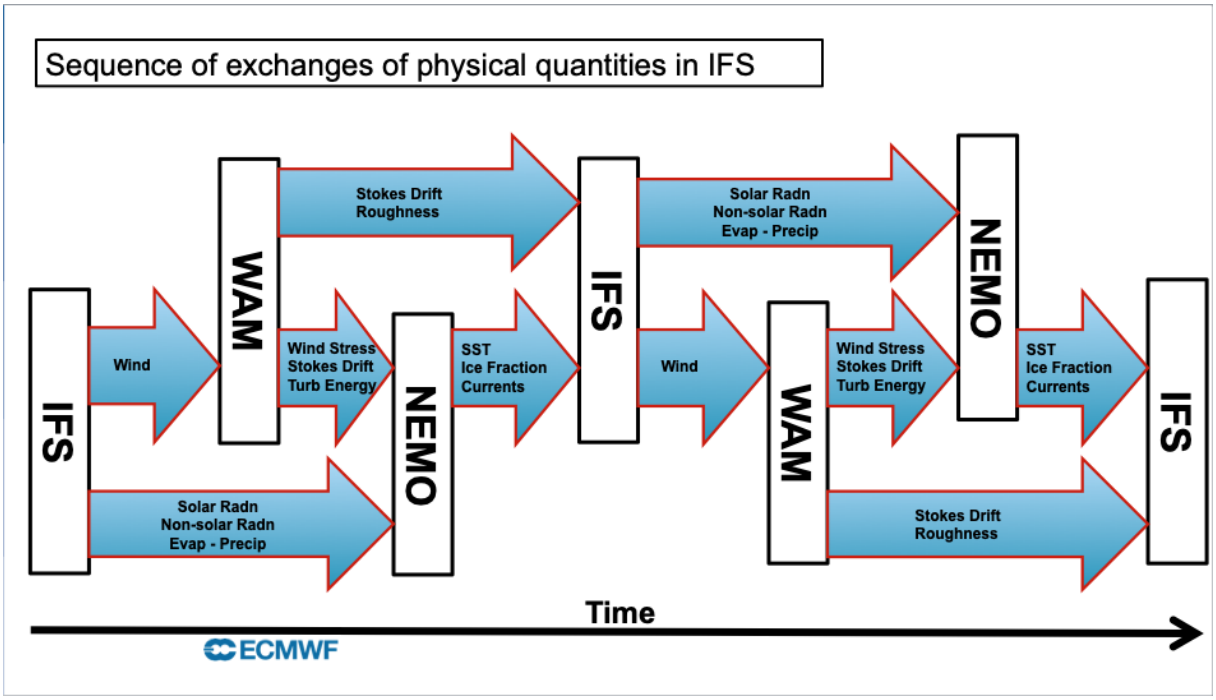


Figure 16: IFS two–time-step view of coupling sequence and exchange of physical quantities between the atmospheric, ocean wave and ocean models

10.2. DEEP-SEA first use case and the PAPADAM mini-app

The coupling of the three main IFS components allows no flexibility in allocating different resource characteristics across the components. This is a limitation even within a single homogeneous CPU allocation, as the number of MPI tasks is fixed by the main IFS setup, which is typically run with a large number of OpenMP threads. Components that have poorer OpenMP scaling than the main IFS are nevertheless required to run with the same number of MPI tasks, leading to an inefficient execution on these components. For example, NEMO is run with the same number of OpenMP threads as the main atmospheric dynamical core, despite exhibiting poor OpenMP scaling.

In DEEP-SEA, the first effort that will be undertaken is to increase flexibility in the mapping of IFS components to hardware resources. Components will be able to use their preferred ratio of MPI tasks to OpenMP threads, thereby improving execution efficiency of components with differing scaling characteristics. This should be achieved without changing the numerical coupling, allowing scientific results to be unchanged. On an MSA-enabled platform, it should also allow different components to be run on different hardware partitions, enabling an incremental route to full GPU execution.

To this end, a small proxy application, PAPADAM (**P**roxy **A**pp to **P**lay **A**round with **D**EEP-SEA **A**PI for **M**PI), has been written that couples two meteorological components, namely the IFS spectral transforms package and an implementation of the MPDATA gridpoint advection scheme.

The PAPADAM mini-app time step consists of a spectral transform of wind field from spectral space to gridpoint space, followed by an MPDATA gridpoint advection of a tracer by the wind field, and a final spectral transform of the wind field from gridpoint space back to spectral space, as illustrated in Figure 17. Although in PAPADAM the spectral transforms do not result in actual modifications to the wind field, as no algorithmic operations are carried out in spectral space, the transforms represent the communication-intensive characteristics of the IFS' dynamical core, whereas the gridpoint advection resembles the denser compute components.

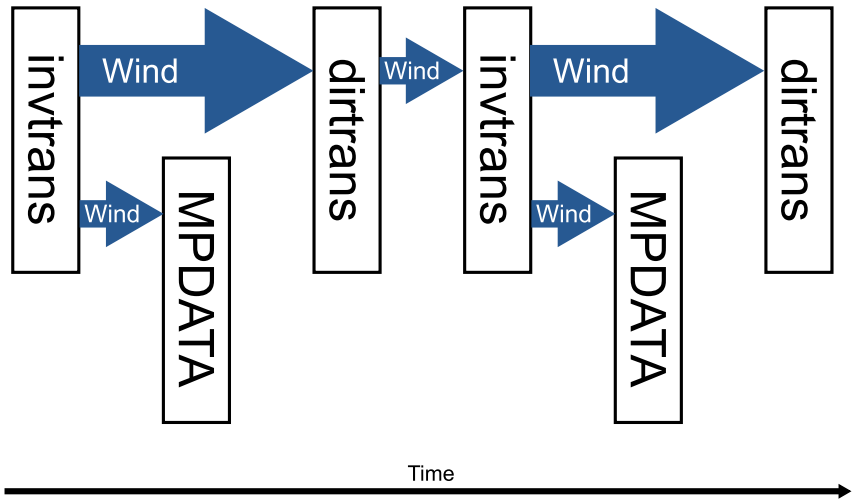


Figure 17: PAPADAM mini-app two–time–step view of coupling sequence and data flow between inverse and direct spectral transforms, and advection component.

The OpenMP thread count of each component can be adjusted separately in order to mimic behaviour of the IFS, in which not all components scale equally well with thread count. Both of the components, namely the spectral transforms and the gridpoint advection, will be of interest individually for certain DEEP-SEA optimization cycles, but the first work to be carried out will be to develop a blueprint for flexible MPI coupling of the components, allowing:

- optimal use by both components of a single hardware allocation, by allowing different MPI$\times$OpenMP combinations, or
- execution of each component on hardware allocations with different characteristics.

PAPADAM relies on a number of libraries developed at ECMWF: *atlas* for grid handling, *trans* for the spectral transforms, *eccodes* for GRIB file handling, and the utility libraries *eckit* and *fckit*.

10.3. Trace generation with JUBE and Score-P

A JUBE execution script as well as a GitLab runner have been prepared, allowing the application to be added to a continuous-integration style workflow. The JUBE script manages the checking out of the source code for the mini-app and its dependencies, the build of the PAPADAM thanks to the ECMWF CMake-based ecbuild infrastructure, and the submission of the application run to the DEEP system's job scheduler.

Additionally, the application build process has been modified to allow instrumentation with the Score-P infrastructure, which can be requested in the JUBE script. The instrumentation covers all the compute and communication parts of the mini-app. In the process of setting up the Score-P instrumentation, a number of issues related to the package were uncovered, and communicated to the Score-P team. The JUBE script also handles trace collection during an application run, if requested.

10.4. Trace visualization

Trace files have been generated on the Cluster module of the DEEP-EST machine for two configurations, one with 4 nodes and the other with 8 nodes, and both with 6 MPI tasks per node and 4 OpenMP threads per task. The grid used for these cases was a cubic-truncation octahedral grid of maximum spectral wavenumber 640, and 137 vertical levels. This corresponds to the grid used operationally at ECMWF for ensemble forecasts.

The two components of the PAPADAM mini-app can clearly be observed in the Vampir visualization of the 8-node trace, in Figure 18, which shows one time step. The MPI_Alltoallv sections correspond to spectral transform data transpositions, while the OpenMP-heavy sections correspond to the advection phase.

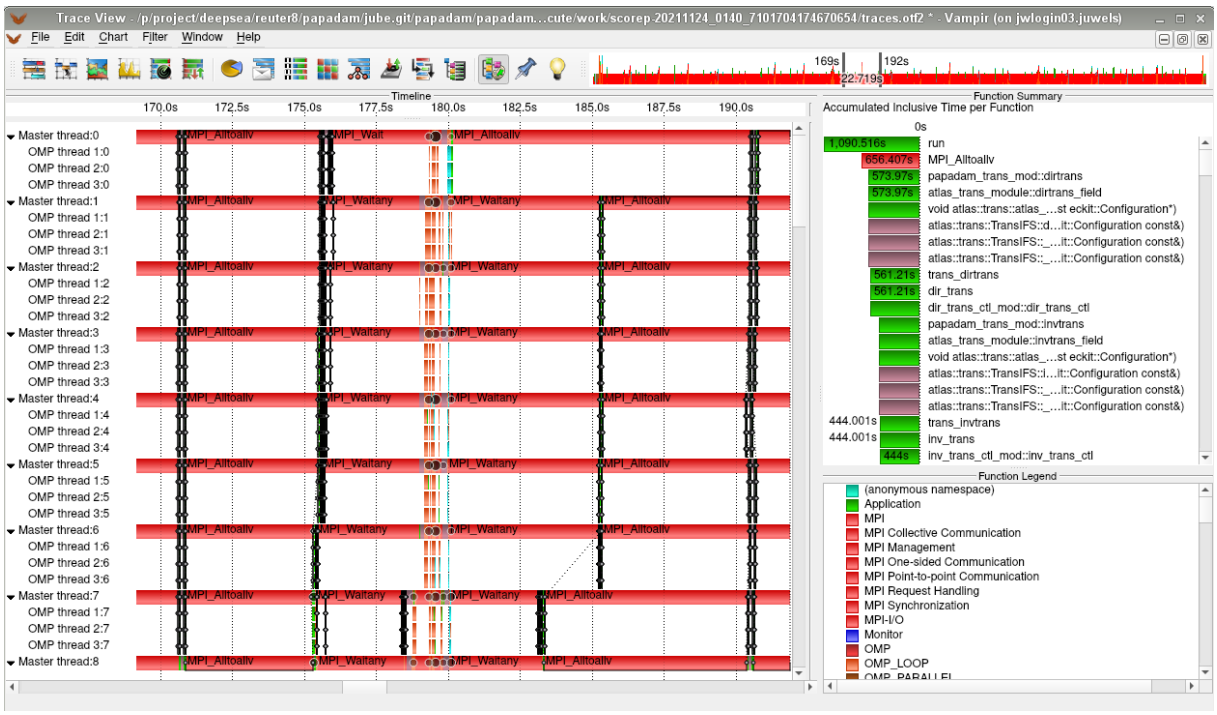


Figure 18: Vampir visualization of one time step of the PAPADAM mini-app (8-node trace, 6 MPI tasks per node, 4 OpenMP threads per task).

11. Barcelona Subsurface Imaging Tools

Full-waveform inversion (FWI) solves an inverse problem iteratively to retrieve physical values from the subsurface, typically velocity and density. As described in Deliverable D1.1 [1], the workflow includes several inversion steps and frequency bands following a multi-scale approach for the adjoint method. Algorithmically, inside each of these iterations and frequency bands, the problem resembles that of Fraunhofer Reverse Time Migration (FRTM) (c.f. Chapter 12). Within DEEP-SEA we use the Barcelona Subsurface Imaging Tools (BSIT) Mockup to solve FWI using different parallelization strategies and optimization approaches.

11.1. BSIT

The BSIT Mockup package is an analog of the BSIT proprietary package for RTM and FWI. An example of BSIT application for FWI is shown in Figure 19.

Of the complete runtime in FWI, most is taken by single-shot iterations. There can be tens of thousands of these in a single FWI run, in production. A single-shot iteration is performed at kernel level and involves computing a gradient, similar to FRTM, and one or several modelling runs. These steps fundamentally rely on solving the wave equation for a given grid several times. Therefore we focus on monitoring and optimizing a single iteration in order to improve the overall BSIT performance. The Mockup is easily configurable and modifiable, thus providing an ideal environment to investigate different optimization and porting strategies. Specifically, during the project we will explore storage and memory usage technologies on novel architectures, e.g., Post-K (Fugaku), Cascade Lake with Optane, or servers with heterogeneous memory.

11.2. Trace generation with JUBE and Extrae

We have set up a test for BSIT using JUBE. The test script manages download, compilation, and execution of the test. The code is instrumented so that Extrae can be used to analyse the obtained traces. This first test produces one inversion iteration and is set up to use OpenMP parallelism only. In its current configuration, this first test performs the computation of a gradient (one forward and a backward run) and single line search or test run (forward run) for a single shot. Future tests will involve an increasing number of shots, frequencies, iterations and line search runs.

11.3. Trace visualization

We present an example of the traces obtained at the hmem/EPEEC cluster at BSC. They have been obtained using the JUBE script discussed in the last section. The cluster has been used because of the installation of 256 GB OptaneDC NVMEM nodes that can help to test such memories and reduce the memory bandwidth limitations of the code prior to testing it in larger environments. The node has two Xeon Gold 5218 CPUs @ 2.3 GHz as processing units and 64 GB DDR4 @ 2666 Mbps.

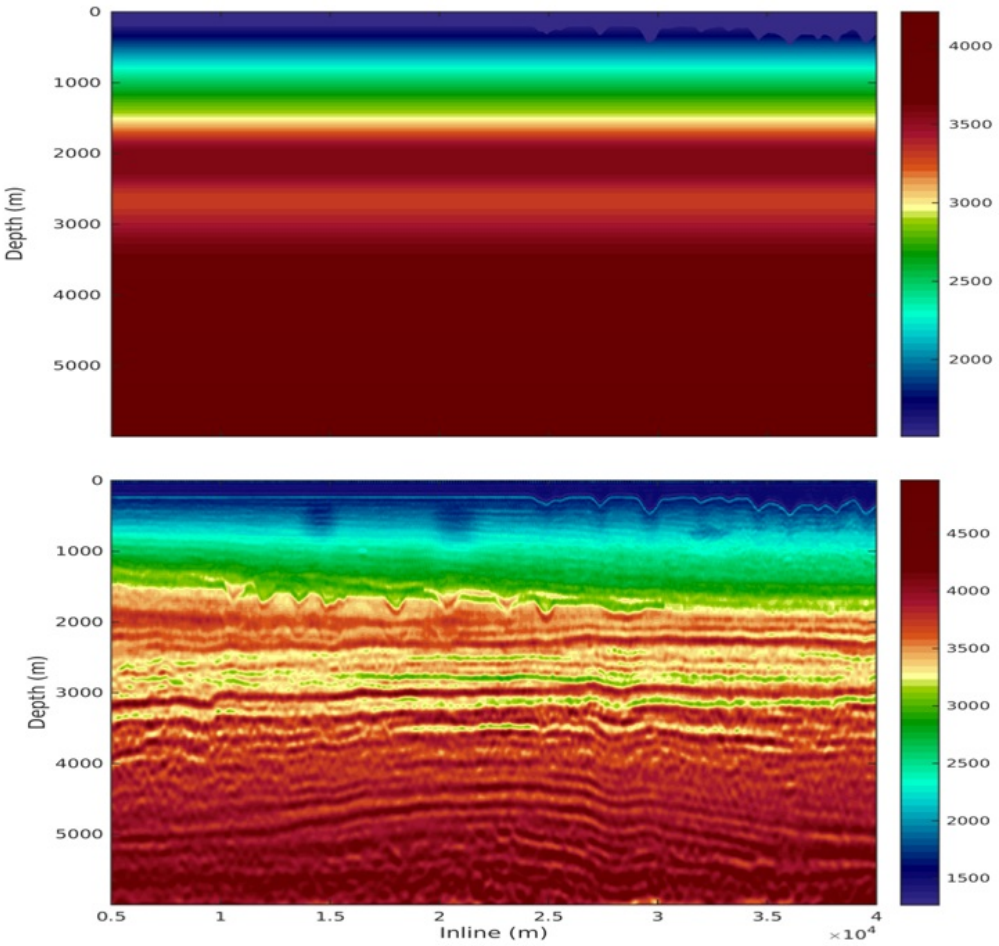


Figure 19: Top: Initial velocity model. Bottom: Result of BSIT's FWI.

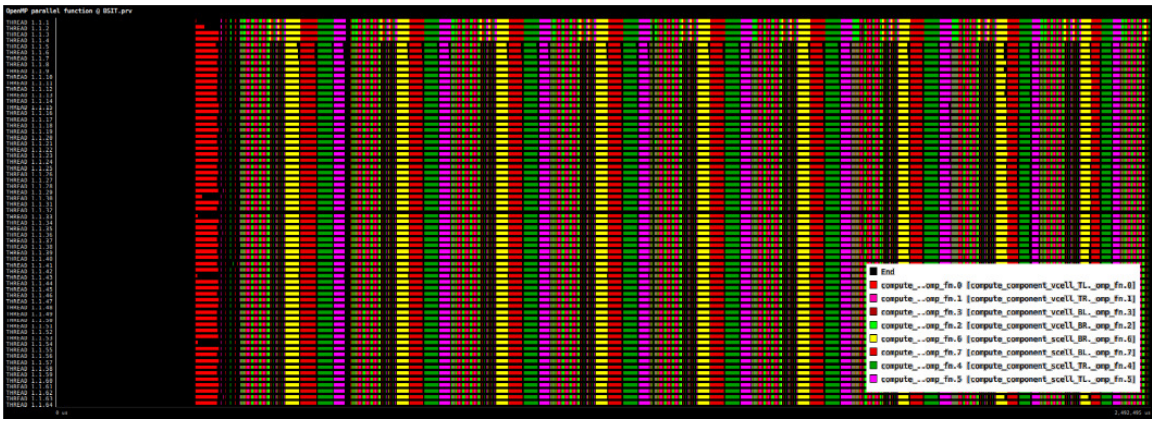


Figure 20: Traces showing, for the first 2500 ms, different iteration steps of BSIT using OpenMP

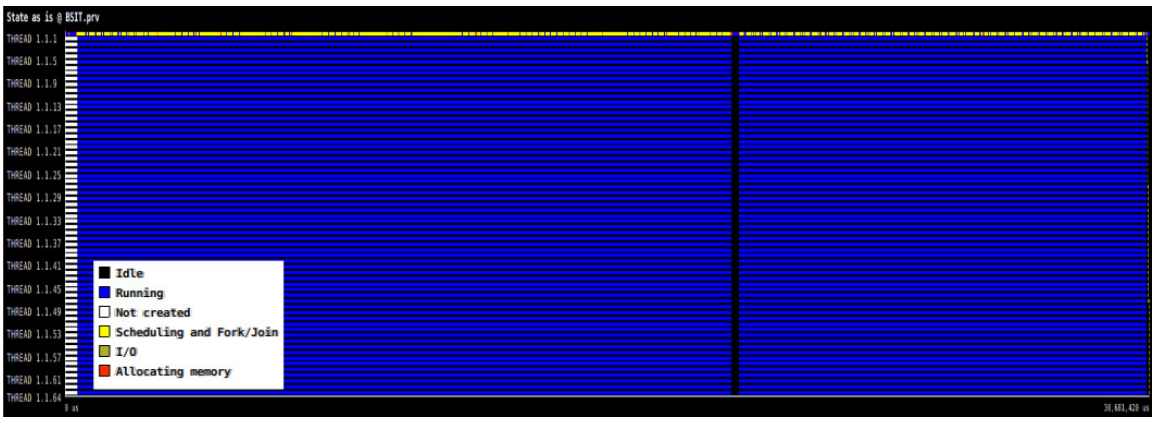


Figure 21: Traces showing, for the complete execution, the state of each thread. Notice the longer time scale than in the previous Figure, for the same case

Traces are generated using Extrae v3.8.3 and visualized with Paraver, both of them open source. The total runtime of the test is 31.54 s. In Figure 20 we analyse the first 2500 ms of the run, which covers roughly nine iterations of the gradient computation as well as the initialization. At each iteration, we observed the instrumented functions that are called, which compute a particular stencil of the simulation. We can observe the synchronization at the end of each iteration and the associated load imbalances as segments in black.

In Figure 21 we observe the performance for all threads. We can clearly differentiate two blocks: gradient and test – gradient is longer regarding execution time. We can observe an overall well-balanced application, together with the setup time (in white) and the scheduling tasks managed by thread 0. Paraver estimates the average overall efficiency for this test at 62.4%.

12. Fraunhofer Reverse Time Migration

The Reverse Time Migration (RTM) method images seismic data and is based on the discretization of the full wave equation. It is employed for oil & gas exploration and allows for accurate imaging of complex subsurface structures. The Fraunhofer Reverse Time Migration (FRTM) implements the RTM method in a robust and massively parallel way by using proprietary HPC tools (c.f section 12.1).

12.1. FRTM proxy application

The FRTM proxy application has been designed in order to allow for easy benchmarking. It performs the computation of a single shot, which is the basic computational step of the RTM method.

The proxy application allows to configure, at runtime, the size of the problem to be migrated, the number of processes to be used, and the number of threads to be used. In order to be independent of any external input data, the proxy application generates internally the seismic reflection data for a set of horizontal reflector planes, and migrates the model afterwards. The migration consists of a forward propagation step in which the source signal is propagated forward in time. This uses a random velocity perturbation boundary condition. Due to the special boundary treatment, the energy is approximately preserved along the propagation path, as a result the propagation path can be inverted. This allows to avoid the checkpointing by a common backward propagation of the source and receiver signals to which the cross-correlation can be directly applied to obtain the image. Once the imaging is complete, the final image is written to disk.

The proxy application is built out of the standard FRTM modules used by FRTM itself. As such, it is a hybrid parallel GASPI based application using two-level domain decomposition. On the first level, the simulation domain is partitioned on the distributed memory level across the processes and data is exchanged using GASPI. On the second level, the process local partitions are further partitioned in order to allow for a task-based execution on the shared memory level.

12.2. JUBE scripts and Score-P instrumentation

The provided JUBE script instruments the FRTM proxy application. It comes as part of the FRTM git repository, which contains all the required files. The JUBE scripts measure the setup time, the pure computation time, as well as the I/O time in a strong scalability setup, i.e., the problem size is fixed whereas the number of processes may vary within a configurable amount of nodes (1-8). One process is started per NUMA domain and a configurable number of Asynchronous Constraint Execution (ACE) threads is started within each of the processes. ACE is a pthread based task scheduler.

The given time measurements have been chosen so that they allow us to deduce the different contributions (e.g., compute, I/O) to the turnaround time and the scalability of the implementation. The turnaround time is what finally matters to the end-user. The scalability measures the efficiency of the implementation, and in particular also the energy efficiency.

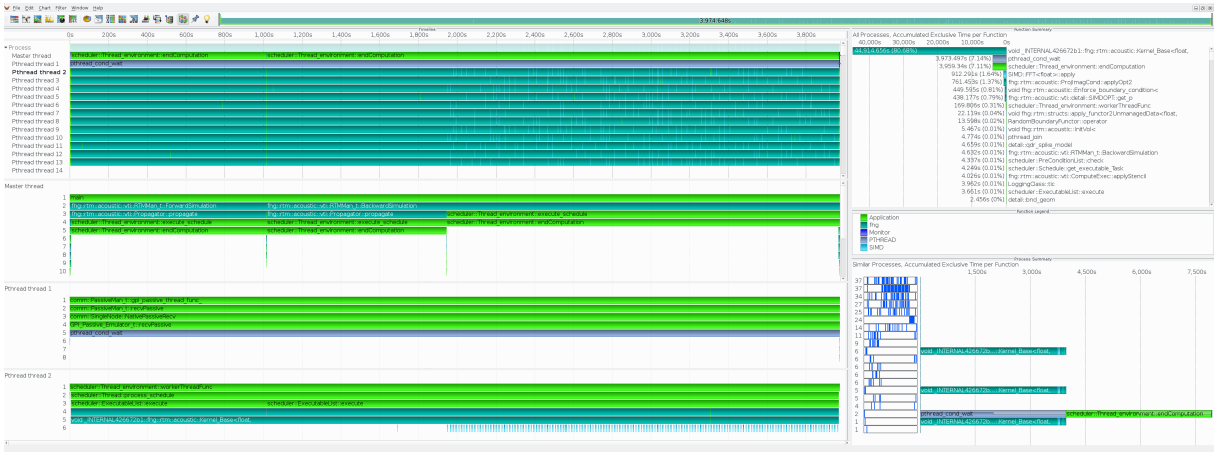


Figure 22: Visualization of a trace for a single shot computation in FRTM

As a use case, we use a problem on a cubic grid having an extension of $N = 500$ in every direction. This corresponds to an extension of the simulation volume of $(10km)^3$ which is the typical size for real data computations.

Tracing and profiling are performed using Score-P. As Score-P has no native GASPI support, profiling is done using a single process only. The size of the problem is chosen in such a way that it corresponds to the amount of work executed by a single process when running the use case on eight nodes (16 processes), i.e., $N = 500$ for the depth direction and $N = 125$ for the two lateral directions.

12.3. Trace Visualization

The trace generated for the FRTM proxy application is shown in Figure 22. The master thread starts the forward and backward propagation and waits for their completion, respectively. The worker threads of the ACE thread pool (e.g., pthread thread 2) are performing the actual computations, i.e., they perform the stencil operations and apply the imaging condition if required.

13. Molecular Dynamics

GROMACS is a widely used molecular dynamics (MD) code used to study the structure and dynamics of biological systems, including recently the COVID-19 virus. GROMACS is an open-source framework written in C++ and runs on various systems, from laptops to heterogeneous supercomputers [26]. GROMACS utilizes parallelisms on multiple levels, from handwritten SIMD intrinsics to an MPI-based Multiple-Program Multiple-Data (MPMD) design. Each MPI rank performs either short-range particle-particle (PP) or long-range Particle Mesh Ewald (PME) related work in the MPMD design. GROMACS can also utilize GPUs to distribute the MPMD workload heterogeneously.

Within DEEP-SEA, our goal is to improve the performance of the PME calculations. In particular, we aim at utilizing work on a 3D FFT based on the DaCe framework developed in DEEP-SEA WP4, as the 3D FFT is a central component in the PME calculations. To evaluate the performance impact and the development opportunities, we do a profiling and tracing study. The profiling analysis has been carried out using Score-P, Vampir, and GROMACS's built-in performance counters. For the scalability analysis, we study three systems: one system with approximately 30,000 atoms and two systems with approximately 0.85 million atoms (a homogeneous system of liquid water and a heterogeneous system of spike:hACE2 complex in a water solvent, as described in deliverable D1.1 [1]).

The three cases can be summarized as:

- Lysozyme – a common first system for GROMACS users
- Spike protein:hACE2 complex – a relevant production case for domain scientists
- Water – a load balanced system for performance modeling

This performance study will provide a performance baseline which we will use for comparing the performance of 3D FFT developed in WP4.

13.1. Molecular Dynamics

In MD simulations, we solve Newton's equation of motion for an N-particle system for a timescale ranging from nanoseconds to microseconds. The usual time step employed for integrating the equation of motion is 1–2 fs. In a naive implementation of MD codes, the field calculation requires N^2 calculations where N is the number of particles in the system. For this reason, MD calculations can mainly be classified as compute-driven. However, for systems with large sizes, e.g., with millions of atoms, parallel communication also becomes an equally demanding segment.

The MD algorithm can be improved. For instance, it is possible to divide the energy calculation into short-range calculations and long-range calculations. For the short-range computation, we can consider only the energy and force contributions of particles within a cut-off distance, reducing the number of short-range interactions drastically. For the long-range field calculations, we can use the PME approach. In the PME approach, a grid is introduced in the simulations, and an electrostatic potential is calculated on it. For this, the FFT is used.

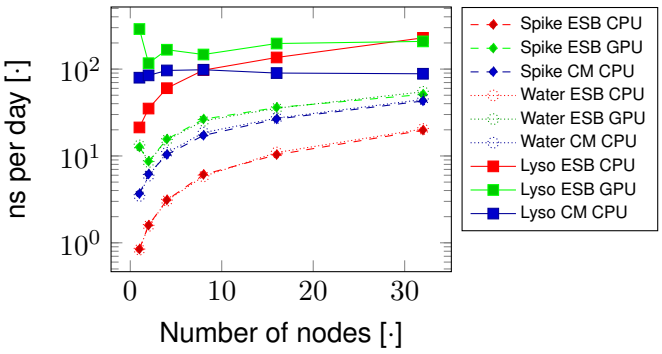


Figure 23: Performance analysis of GROMACS on the DEEP system for three test systems namely lysozyme in water, liquid water, and spike:hACE2 complex in water.

Using PME makes the cost associated with electrostatic calculations $\mathcal{O}(N_g \log N_g)$, where N_g is the number of grid points. In this way, the actual computational cost is not of the order of N^2. For the calculations of short-range interactions, the computing nodes need to know the coordinates of the particles' positions and those within a cut-off distance (usually 15–20 Å). The calculations of the long-range part of the interactions require coordinates of *all* particles. The PME method also uses parallel interpolation techniques requiring communication.

GROMACS allocates the computing associated with the short-range and long-range interactions to a separate set of processes, which are referred to as Particle-Particle (PP) ranks and PME ranks, respectively. In DEEP-SEA, we focus on improving the performance of the PME calculations requiring the usage of 3D FFT. The all-to-all type of communication in the 3D FFT drastically affects GROMACS performance and its parallel scaling, especially going towards exascale. Typically, the PME calculations are carried out on one-third of the total number of ranks available. The typical size of the data generated during MD (the positions of all the particles and velocities are stored as a function of time to carry out analyses of various structural and dynamical properties) can be from a few tens to a few hundred GiB, depending upon the time interval defined for storing the coordinates/velocities.

13.2. Benchmarking and trace generation with JUBE and Score-P

We have carried out GROMACS calculations for the aforementioned biological and solvent systems on multi-CPU and CPU-GPU machines (Figure 23). A JUBE script for GROMACS has been developed for this purpose. The results from this study are discussed below. The numbers of nodes allocated were 1, 2, 4, 8, 16, and 32; in this specific case only the performance of MKL FFT library (`DGMX_FFT_LIBRARY=mkl`) has been studied. Further calculations were run using three options: Cluster Module (CM) with CPU only (will be referred to as CM CPU), as well as on the Extreme Scale Booster (ESB) with GPU and CPU (will be referred to as ESB GPU and ESB CPU, respectively).

As we see in Figure 23, the performance of the lysozyme case running on the ESP CPU improves when increasing the number of nodes, which is not seen when running the same benchmark on the CM CPU and ESB GPU.

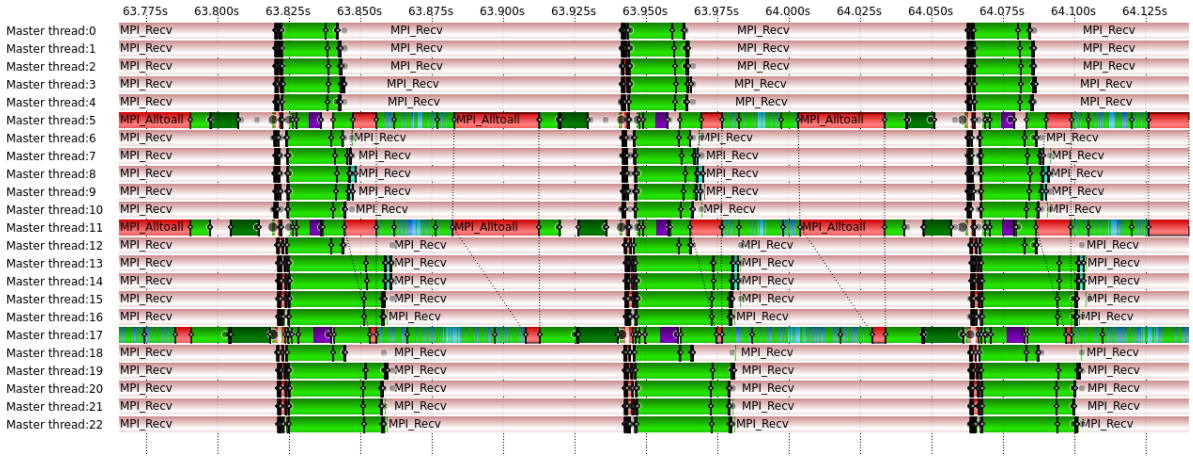


Figure 24: GROMACS run of the spike case with 24 ranks: 4 PME ranks and 20 PP ranks. We have highlighted five major parts of the PME calculation. Purple is the GROMACS phase spread, blue is the 1D FFT calculation, light blue is the electrostatic calculation, dark green is the gather, and dark red is the 3D FFT all-to-all communication. We can see that the setup is load-imbalanced in between the PME ranks and the PP ranks.

Interestingly, the scaling behaviour for the liquid water and spike protein system is exactly the same (the data overlaps in the plot), which suggests that the heterogeneity in the latter system does not affect the performance significantly. For these two systems (water and spike), the performance improves when increasing the number of nodes on both the CM CPU and the ESB CPU. On the ESB GPU the performance drops when using 2 nodes, and only with at least 4 nodes the performance increases beyond single-node performance for water and spike. Overall, the calculations with the GPUs of the ESB nodes show superior performance.

There are many parameters available for tuning such as the number of threads to be used, the number of nodes to be allocated for PME or PP calculations, the allocation of GPUs for PP or PME. More detailed analysis will be carried out in the future.

13.2.1. Score-P tracing

The traces were generated using GCC version 10.3.0, ParaStationMPI version 5.4.9-1-mt and Score-P version 7.0. We use the installed Vampir 9.10.0 in the DEEP system and ran the Spike protein case on two cluster nodes. Figure 24 highlights the MPMD design of GROMACS. In Figure 25 we see a breakdown of a PME rank into its components.

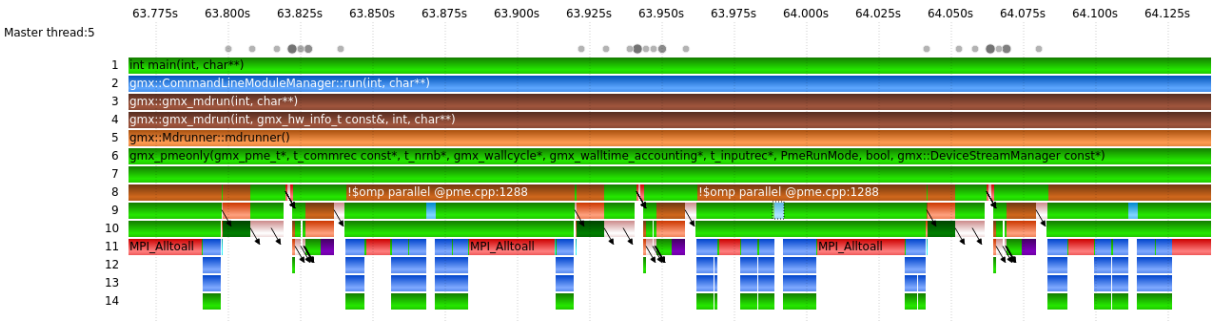


Figure 25: The format of a PME rank. The colour scheme for the labels is the same as in the figure above.

14. Computational Fluid Dynamics

Nek5000 is an open source computational fluid dynamics code based on the spectral element method. Nek5000 solves the incompressible Navier–Stokes equations, together with a number of additional physics (heat transfer, magneto-hydrodynamics, low Mach number, electrostatics) on general hexahedral spectral elements. For a more detailed description of Nek5000 and computational fluid dynamics position in the exascale computing landscape, please refer to deliverable D1.1 [1].

In Nek5000, special focus is laid on single-core efficiency via fast tensor product operator evaluations. For high-order methods, assembling either the local element matrix or the full stiffness matrix is prohibitively expensive. Therefore, a key to achieving good performance in spectral element methods is to consider a matrix-free formulation, where one always works with the unassembled matrix on a per-element basis. Gather–scatter operations are used to ensure continuity of functions on the element level, operating on both intra-node and inter-node element data.

14.1. Generating Traces with Score-P

For generating traces, we used Nek5000's proxy-app Nekbone [27] instead of the proposed use cases, since Score-P did not support the Fortran 2008 MPI wrappers necessary to run the modernized Nek5000 version used in the project. Nekbone captures the basic structure of the main code but is limited to solving a standard Poisson equation in a box using a conjugate gradient method together with a spectral element multigrid preconditioner. Furthermore, Nekbone exposes the principal computational kernels and the essential algorithmic design in Nek5000, thus it is well suited as a first step during the optimization cycles in the project.

The traces were generated using GCC version 10.3.0, ParaStationMPI version 5.4.9-1-mt and Score-P version 7. We ran Nekbone's `nek_mgrid` case using 8th, 10th, and 12th order polynomials with 128 elements per MPI rank, corresponding to typical production cases. Figure 26 presents weak scaling results for Nekbone running on one to eight nodes with 24 MPI ranks per node on the DEEP system's cluster module. Performance is much lower than standard Nekbone using a naive preconditioner, but the spectral element multigrid formulation better illustrates the work performed in the actual use case. Furthermore, the experiments also demonstrate the performance drop when the small matrix–matrix does not fit in cache any more (somewhere between 10th and 12th order polynomials).

14.2. Trace Visualization

We present an example of traces obtained from Nekbone on the DEEP system's cluster module visualized using Vampir version 9. Figure 27 illustrates the different phases in one iteration of the conjugate gradient method in Nekbone. Here, `solvem` is the spectral element multigrid solver, `ax` the matrix-free evaluation of Ax in each Krylov iteration, `mxm` are the various small matrix–matrix kernels used to assemble and evaluate a spectral element and `gs` the gather–scatter operation necessary to ensure C^0 continuity across elements.

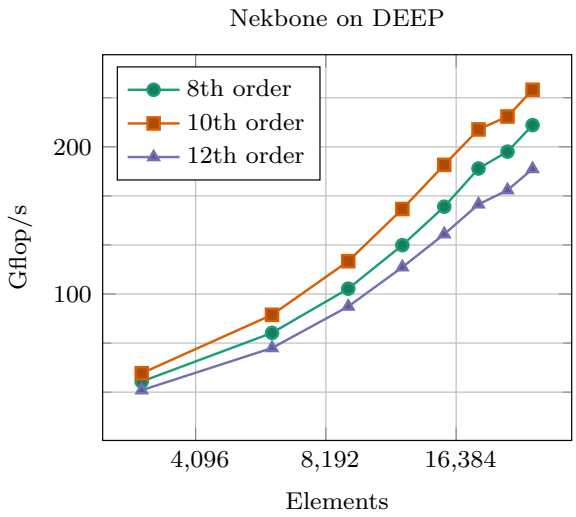


Figure 26: Weak scaling results for Nekbone using the cluster module in the DEEP system with 128 elements per MPI rank.

Traces for one conjugate gradient iteration are given in Figure 28, where it is visible that most communication is due to the gather–scatter operations between 1.0275 s and 1.0375 s, and after 1.04 s. The first batch of gather–scatter operations inside the multigrid preconditioner is also more unbalanced than the gather–scatter operation after the matrix–vector product, illustrating why the `nek_mgrid` case performs worse than the regular Nekbone case often used in benchmarking.

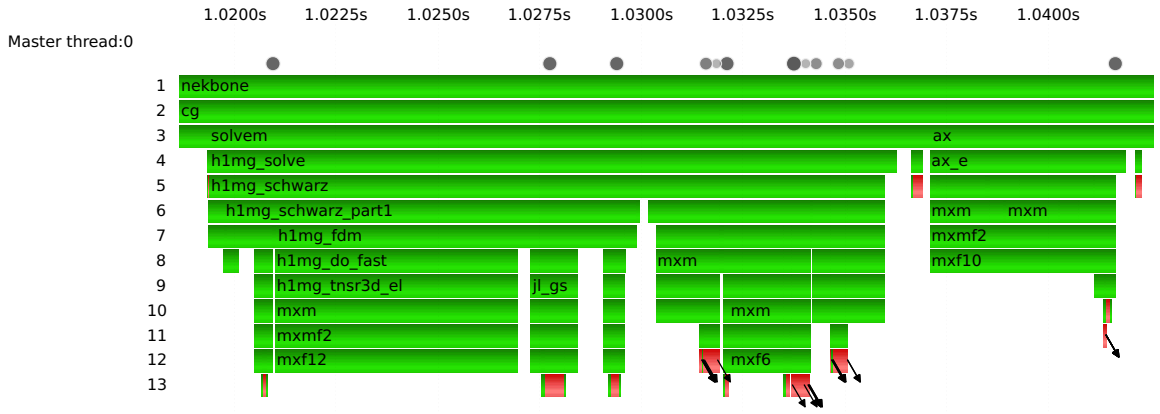


Figure 27: An illustration of the different phases in one conjugate gradient iteration inside Nekbone.

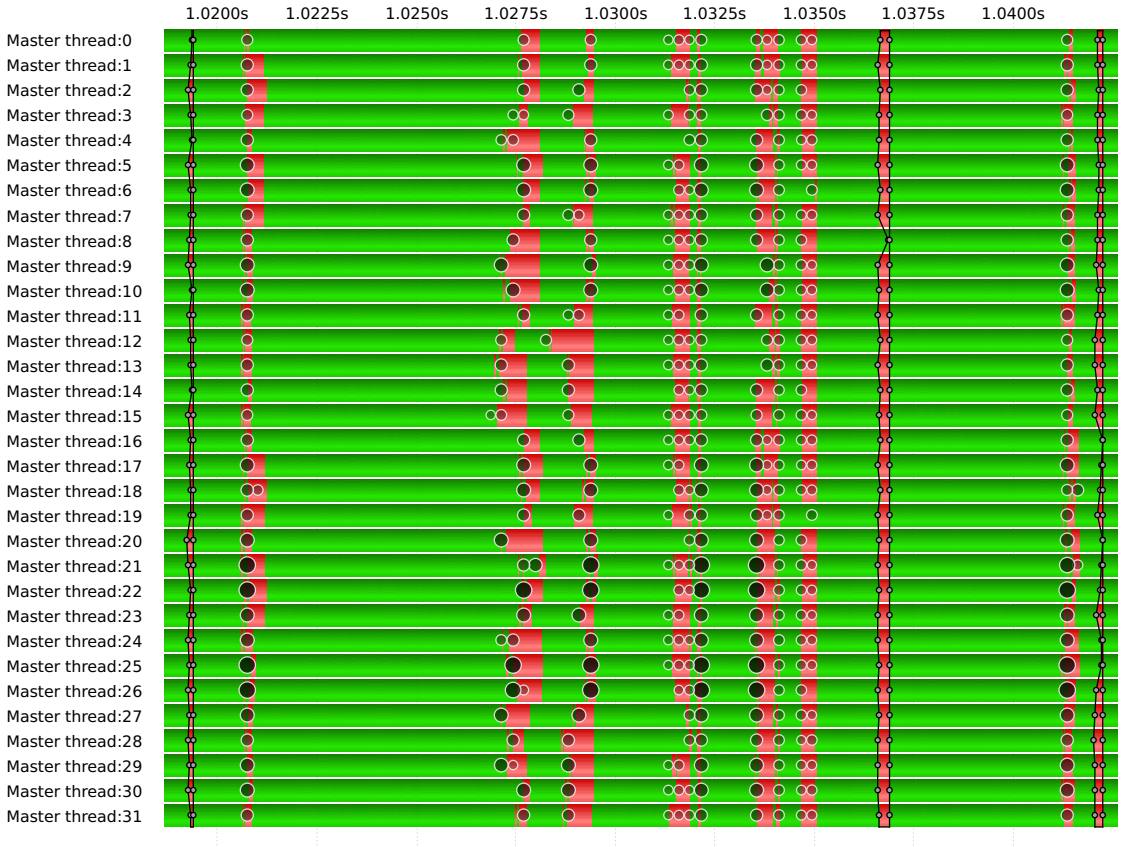


Figure 28: Visualization of Nekbone traces for one conjugate gradient iteration.

15. Neutron Monte-Carlo Transport for Nuclear Energy

The solution of the linear Boltzmann equation by the Monte Carlo method is based on the simulation of a great deal of random particle trajectories within the considered system. The ensemble averages of the simulated trajectories provide estimates for the physical observables of interest. The Monte Carlo trajectories describe random walks whose mathematical properties are chosen to take into account the physical laws of particle–matter interaction, to the best of our knowledge. Since the method does not require any space, energy, or angle discretization, Monte Carlo has always been considered as the *golden standard* for the study of nuclear reactors. This desirable property comes at the price of a somewhat slow statistical convergence with uncertainties scaling as the inverse of the square root of the number of simulated histories. In turn, the basic Monte Carlo algorithms lend themselves well to massive parallelization, which can help mitigate the slow convergence.

Until very recently, the large CPU requirements of Monte Carlo simulation have limited its use almost exclusively to the study of stationary problems within systems with a predetermined set of temperatures and compositions (i.e., without any physical feedback loop). However, the computing power and available storage have now grown to the point where it is possible to run true “numerical experiments” in realistic configurations using Monte Carlo.

Today, the goal of Monte Carlo neutron transport is the simulation of full-core configurations, in non-stationary conditions (quasi-static depletion calculations and true dynamic calculations with physical feedback). This requires a deep revision of the architecture of Monte Carlo codes; indeed, it is crucial to efficiently exploit the massive parallelism and vectorization which characterize modern machines.

The PATMOS code, developed at CEA, has been designed to explore the requirements of next-generation Monte Carlo neutron transport in terms of code architecture, parallelism, algorithms, memory, and CPU time. The goal of PATMOS is to demonstrate the feasibility of Monte Carlo calculations for the depletion of a full-scale pressurized water reactor, taking into account thermo-hydraulics and thermo-mechanical feedbacks.

This chapter describes the traces obtained from the PATMOS application on a typical use case to highlight the main behaviour of the parallel application.

15.1. Use Case and Parameter Space

The first step to generate traces is to choose a relevant use case and the parameters that should be studied. The use case chosen for this deliverable is the typical use case described in D1.1 [1].

Figure 29 illustrates the typical use case used for generating the traces. Within this input set, multiple parameters can be changed to evaluate the behaviour of the parallel application. Thus, it is possible to modify the number of MPI ranks, the number of OpenMP threads, and the amount of work per MPI rank (to evaluate the load balancing among ranks and parallel regions). This set of parameters

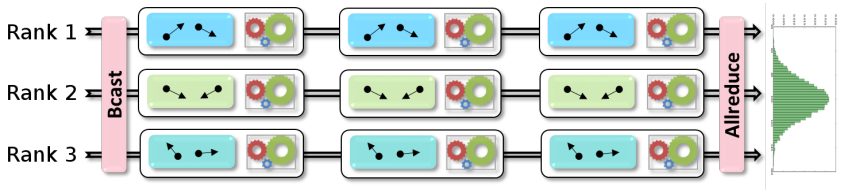


Figure 29: Typical use case with 3 MPI ranks and OpenMP parallel regions.

comes in addition to the choice of the number of particles per MPI rank and the number of batches (i.e., the overall number of gear boxes on Figure 29 within each MPI rank).

Because the main developments within this project will be based on porting to heterogeneous modular architecture (including GPU porting, multiple memory support, and MPI gateway usage), the number of MPI ranks is set to 2 nodes, with 1 or 2 ranks per node. This will allow evaluating the amount of communication between compute nodes and within a node. The number of OpenMP threads was set to fill up spare cores to exploit all compute resources. Finally, the parameter `adjust_fuel` is set to different values to change the load balancing between MPI ranks.

This parameter space leads to 8 different runs with one trace generated per execution.

15.2. Generating Traces with JUBE and Score-P

Trace generation relies on two different tools: Score-P 7.0 (to instrument and generate the OTF2 traces) and JUBE (to manage source compilation and application execution with the correct set of parameters) The workflow consists of one JUBE XML file (named `patmos.xml`) with the following steps:

1. Checkout: this step clones the main PATMOS repository with sub-folders.
2. Build: it compiles the whole application with GCC 10.3 and ParaStationMPI 5.4.9 through CMake 3.18. This compilation step relies on the Score-P wrappers for MPI (`scorep-mpicxx` and `scorep-mpicc`).
3. Run: this step runs the application with different parameters (number of MPI ranks, number of OpenMP threads, etc.). To keep the memory consumption of the tools below a few gigabytes, the runs are based on a small set of particles (1000 per MPI rank). Furthermore, this step uses Score-P filtering to keep information related to MPI and OpenMP only.

The output of running this XML file with JUBE is located inside the `bench_run` directory with one OTF2 trace file for each run.

15.3. Trace Visualization

We use Vampir to visualize the OTF2 traces generated by the previous script. For example, Figure 30 illustrates the trace overview for 2 MPI ranks (one per compute node). It shows that the amount of work is the same for the 2 MPI ranks, but the load balancing within each parallel region is not well

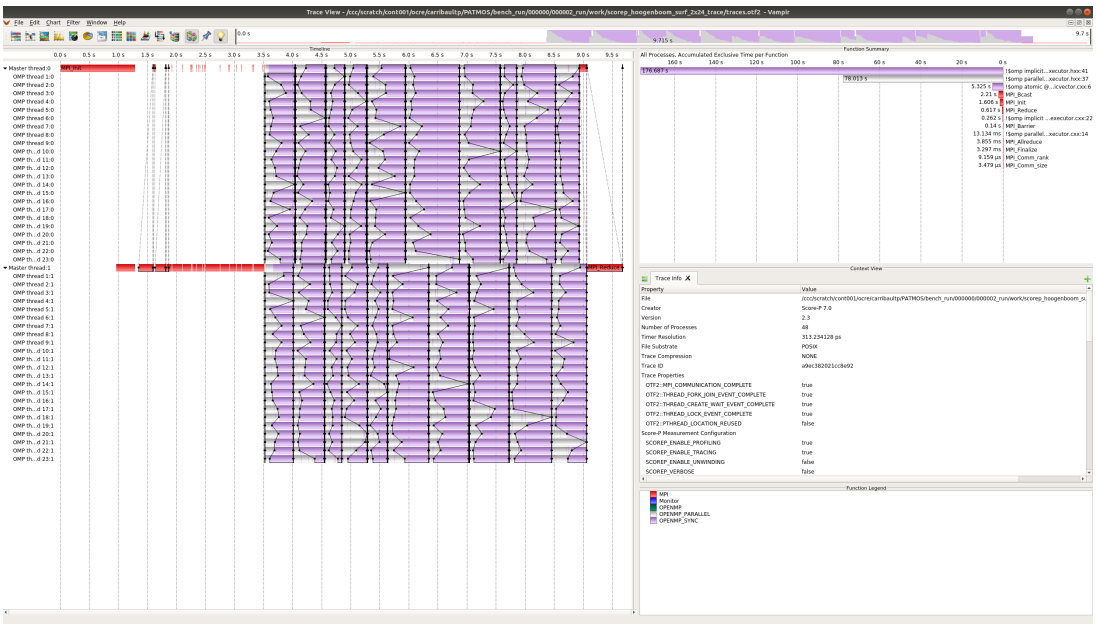


Figure 30: Visualization of a PATMOS trace with 2 MPI ranks and 24 OpenMP threads on a balanced typical use case.

managed. Most of the time, OpenMP threads are waiting on a barrier for another thread to finish its work. It means that some work can be done to improve the dynamic behaviour of the OpenMP parallel regions. Finally, the iteration ends with a reduction operation performed in MPI.

To compare this result, Figure 31 presents the same trace with a more unbalanced work among the MPI ranks. While the behaviour within each rank is the same (many idle times for the OpenMP threads), one MPI rank finishes ahead of the other.

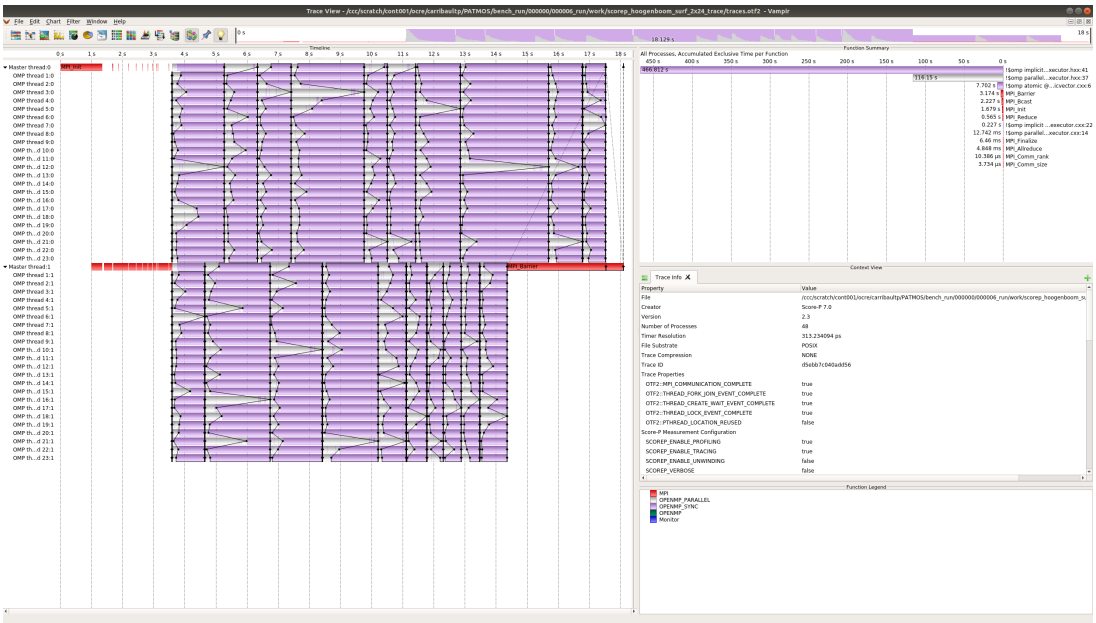


Figure 31: Visualization of a PATMOS trace with 2 MPI ranks and 24 OpenMP threads on unbalanced typical use case.

16. Earth System Modelling

A summarized introduction (based on the full description in Deliverable 1.1) to the aspect of Terrerstrial Systems and the Terrestrial Systems Modelling Platform (TSMP) is presented here and throughout Section 16.1 for the purposes of completeness. Readers familiar with TSMP may prefer to skip to Section 16.2.

Terrestrial Systems are conceptualized as a subset of the Earth System integrating interactions and feedback among processes occurring on the land surface, the subsurface, and the lower atmosphere. Processes in these Earth System compartments are typically non-linear and span a large range of spatial and temporal scales. Moreover, the physical processes, typically controlled by small scale variability, tend to manifest in emergent behaviour on larger scales. This multiscale nature implies that it is complex and difficult to predict how the emergent behaviours arise from a multitude of small scale interactions. Consequently, physics-based representations of single processes, which are well-understood at small scales are key to develop an integrated understanding of Terrestrial Systems. In the context of Earth System Science, the modelling of Terrestrial Systems is key to assess the effects of environmental change on the soil–vegetation–atmosphere continuum, and the effects of land-use changes on regional climate.

Physics-based modelling of Terrestrial Systems poses significant computational challenges. Firstly, it is an intrinsically multi-physics problem, with various and very different domains (porous media, land surface, atmosphere). Fluxes in these domains are described by systems of partial differential equations of different nature and thus require different numerical and computational solutions. The spatial scales range from the process and spatial heterogeneity scales (meters), to continental scales (thousands of kilometres). The temporal scales range from the dynamic process scales (seconds) to climate and ecological time scales (decades and centuries).

16.1. TSMP: Terrestrial System Modelling Platform

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 OASIS3-MCT coupler. TSMP is also enabled for Data Assimilation (DA) through the Parallel Data Assimilation Framework (PDAF). TSMP allows simulating complex interactions and feedback between the different compartments of terrestrial systems. Specifically, it enables the simulation of mass, energy, and momentum fluxes and exchanges across the land surface, the subsurface, and the atmosphere [28]. TSMP is maintained by the Simulation and Data Laboratory Terrestrial Systmes (SDLTS) at JSC, and is an open source software publicly available on GitHub [1].

The coupling design is inherently modular, allowing to build all combinations of component models or only build one of them. This design also leads to a Multiple-Program Multiple-Data (MPMD) execution model and operational flexibility, allowing the different model components to run at different spatial

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

and temporal resolutions. In fact, within TSMP different versions of the component models are supported for both legacy and experimental purposes.

The component models — mostly developed by third parties — are written using different programming languages, use different parallelization and acceleration schemes (can exploit different hardware) and show different scaling behaviour. The diversity in features and responses is partly what motivates the modular design of TSMP.

TSMP is mostly a compute-driven application. The core computational effort comes from solving large sets of partial differential equations. The computational requirements are higher for COSMO and ParFlow, and considerably lower for CLM. Additionally, DA ensemble runs are somewhat data-driven, as observational data needs to be assimilated into the workflows.

16.2. Benchmarking TSMP

16.2.1. Benchmarking cases

Two cases have been selected as benchmarks to be used with the fully coupled TSMP (consisting of COSMO, CLM and ParFlow). The following two benchmark cases allow us to identify performance and scalability improvements in response to developments in DEEP-SEA.

- (*A*): An idealized case configuration, 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 for performance and scalability studies. For the initial benchmarking, we use a problem defined on an $N \times N$ element mesh, and N can be arbitrarily chosen. In this report, two configuration are used $N = 100$ (*A.100*) and $N = 300$ (*A.300*). Simulation time is set to 4 hours, with a coupling time step of 36 seconds.
- (*B*): A fully coupled land surface, atmosphere, and hydrological dynamics over Europe, specifically using the EUR11-CORDEX pan-European domain. It has a resolution of 0.11° (approx. 12.5 km) and conforms to the specifications of the EUR-11 model grid, as defined by the World Climate Research Program's Coordinated Regional Downscaling Experiment (CORDEX) [29, 30]. This domain is roughly 5450 $\times$ 5300 km in extension, including Europe and parts of northern Africa, western Russia and western Asia. The horizontal grid consists of $436 \times 424 = 184864$ elements. Each of the component models requires different vertical resolutions. ParFlow is discretized with 15 vertical elements($\sim 2.8 \times 10^6$ total elements), CLM with 10 ($\sim 1.9 \times 10^6$ total elements) and COSMO with 50 ($\sim 9.2 \times 10^6$ total elements) [31, 32]. Parametrization, initial conditions, boundary conditions (and other necessary data) for this typical use case are well established and publicly available. Typical production jobs run for months to years for forecasting, and also into multiple decades of simulation time for climate analysis (e.g., [33]). For benchmarking, only 12 hours of simulated time are computed, which is also a standard test configuration for TSMP. The interest of this short benchmark is that the metrics captured can be extrapolated to estimate the requirements of the multidecadal runs, at a low cost. Changing the duration of the simulation is not trivial, but possible if longer statistic become important. This case is also typically used for strong scaling tests, but is ill-suited for weak scaling tests

because scaling the domain (and the parametrization, initial and boundary conditions it entails) is non-trivial. An increased resolution CORDEX case is planned during DEEP-SEA, as kilometre-scale simulations are planned as the exascale target simulations.

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 3 nodes (one per component model), and case *B* on 6 nodes. All jobs reported here were computed on JUWELS Cluster at JSC. The nodes used for CPU tasks in JUWELS Cluster have Intel Xeon Platinum 8168 (2x24 cores, 2.7GHz) CPUs. Accelerated nodes have 4xNvidia V100 (16 GB HBM) GPUs –hosted by Intel Xeon Gold 6148 (2x20 cores, 2.4 GHz) CPUs. Although TSMP was ported to the DEEP system and preliminary heterogeneous jobs were run, recent updates in the platform resulted in issues when building TSMP. It is expected that an update in the software stack will allow to solve these issues.

Additional benchmark cases might be considered in the future, most likely variations of *A* and *B*, in response to specific performance queries. Additionally, a benchmark case is expected to be developed to address poorly balanced ensemble runs, which are likely to benefit specifically from some of the solutions proposed in DEEP-SEA, such as dynamic resources allocation and malleability. This is not included at this stage for two reasons: firstly, investigating single simulations is a necessary previous step; second, because the specific goals and investigations for malleable ensemble runs still need development.

16.2.2. Benchmarking workflow

A JUBE benchmarking workflow was developed to systematically manage the two benchmarking cases. The JUBE workflow is illustrated around the TSMP workflow in Figure 32. The benchmarking workflow deals only with the core of the simulation pipeline for both benchmark tests. The simulation itself starts with the concurrent initializations of the core component models. The different component models solve different equations, with different numerical techniques, all resulting in different computational effort and runtime required to reach the coupling time levels. CLM will typically finish first. Once the two other components reach the desired time level, information is exchanged from COSMO and ParFlow into CLM, which can start computing again, and then exchange the information back again, followed by triggering the other two component models to continue marching forward. This process is repeated throughout the simulation. There are some input operations to include boundary conditions into COSMO, and snapshot outputs from all three models.

Post-processing and visualization steps may be of interest as they may potentially be done in-situ, e.g., making use of idle processor times. However, there is currently no single typical in-situ workflow for these post-processing stages (in fact there are many ad-hoc offline post-processing workflows).

The JUBE–TSMP workflow makes use of both native JUBE operations and 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., the runtime per component model, the runtime of major kernels, the I/O runtime, etc). Additional metrics are post-processed

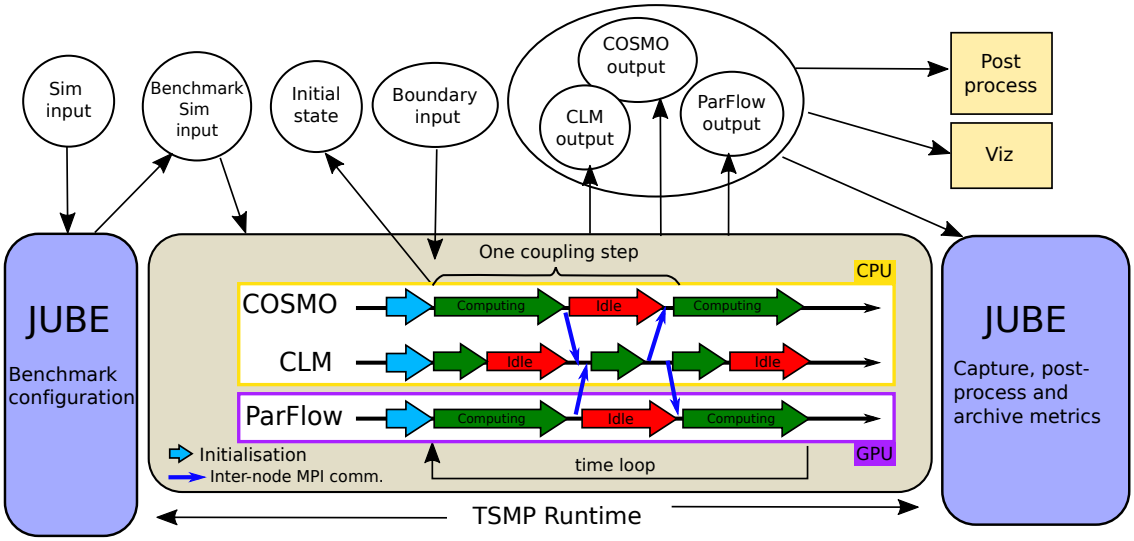


Figure 32: JUBE–TSMP workflow for a single simulation including all three component models and their I/O.

by assessing timestamps in the output files. The full set of metrics and logs is pushed to a GitLab repository for archiving.

16.2.3. Benchmark metrics

A first set of metrics is targeted in the benchmarking workflow, as shown in Table 3. Currently, the metrics are mostly collected from the native timing files that all component models provide. More detailed metrics are likely to be included/implemented in the future. Some metrics are provided per component.

It is important to highlight that the total TMSP I/O time T_{IO} only aggregates the I/O time of the components $T_{\mathrm{IO}} = C_{\mathrm{IO}} + L_{\mathrm{IO}} + P_{\mathrm{IO}}$, but it does not necessarily match the actual wall-time because the component I/O processes may happen in parallel and are non-blocking for the other component models. Ratios of these times relative to the runtimes are also of interest.

Finally, the native timers in the component models allow for many other metrics, which may enrich the first set present here. They will be evaluated as more specific information is required. Additionally, metrics can and will be captured from the profile and traces, although they are currently not fully defined.

16.3. Profiling and tracing

A setup is in place for profiling and tracing using Score-P. Currently, traces have only been generated using the JUWELS Cluster system at JSC, but workflows are in place and preliminary tests have

Metric	Description	Units
T_{r}	total TSMP runtime	s
C_{r}	COSMO runtime	s
L_{r}	CLM runtime	s
P_{r}	ParFlow runtime	s
C_{i}	COSMO initialization time	s
L_{i}	CLM initialization	s
P_{i}	ParFlow initialization	s
T_{comm}	total communication times	s
C_{IO}	COSMO input 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

Table 3.: Metrics captured in the TSMP-JUBE benchmarks.

been carried out in the DEEP system. It is currently possible to do traces on both CPU-only and heterogeneous configurations. Visualizations of profiles and traces shown here are done with Vampir.

Preliminary profiles and traces have been generated for both benchmark cases. The first investigation of the results is still ongoing, and will lead to better focused performance studies.

16.3.1. Profiles

The profiles for case *A.300* are illustrated in Figure 33 (CPU-only), Figure 34 (heterogeneous) and in Figure 35 for case *B* (heterogeneous). They all show a similar behaviour, in which it is evident that MPI operations and in particular `MPI_Waitall` take the longest times. In case *B*, for example, MPI operations account for 76% of the trace runtime. This is however misleading, due to the concurrent MPMD executions. In short, the cumulative `MPI_Waitall` is very large because its execution can be blocking for one component model (typically CLM), although concurrently COSMO and ParFlow continue to perform intensive computations. At certain points, a second component (COSMO or ParFlow) may wait for the third one, but this is non-blocking for the computations of such third component. This concurrent runtime is conceptually sketched in Figure 32. In practice, it is motly CLM that waits. Although not optimal, this is not a severe problem, since CLM is comparatively very cheap and uses the smallest fraction of resources in a TSMP job. Further details are discussed together with the traces (see Subsection 16.3.2).

For the heterogeneous cases (Figure 34 and Figure 35), it is interesting to highlight that aside from MPI operations `cudaStreamSynchronize` appears as a relevant process, which similarly to

`MPIwaitall` is basically a barrier to wait for CUDA processes. Other CUDA operations, namely `cudaLaunchKernel` and `cudaMemcpy` also play a relevant role.

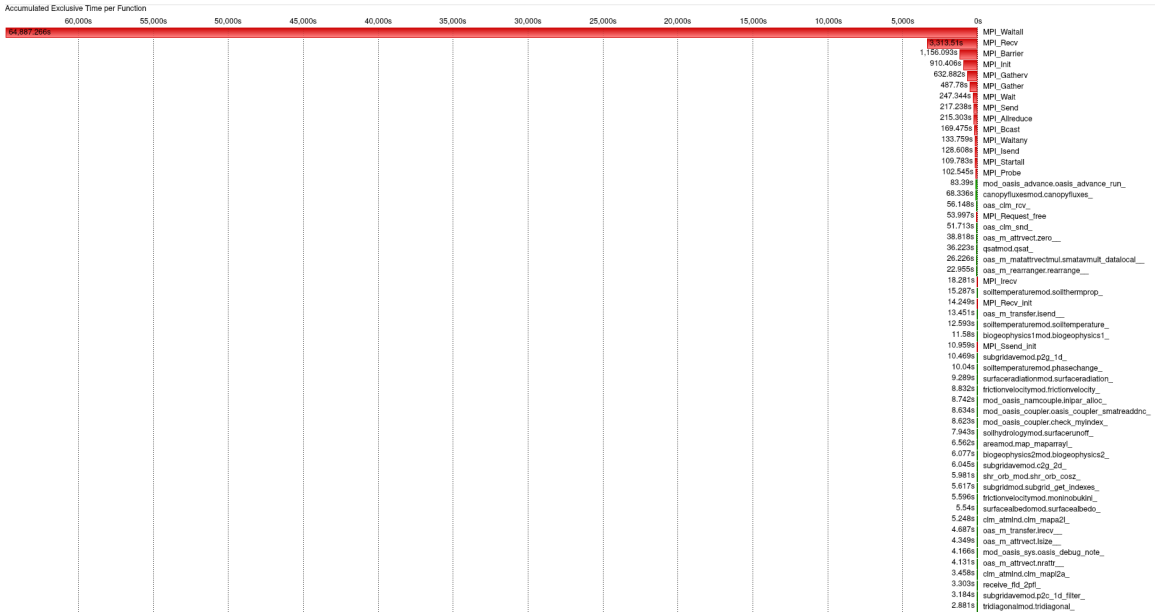


Figure 33: Example profile for the idealised case *A.300* under a CPU-only configuration on JUWELS Cluster.

16.3.2. Traces

Traces for the *A.300* case are shown in Figure 36. Figure 38 shows an abbreviated trace result for case *B*. The abbreviated number of processes is for clarity only; otherwise, visualizing all processes simultaneously becomes hard to read. The traces show the MPMD nature of TSMP, and in particular the concurrent execution of the three component models.

To illustrate and explain the concurrent execution, let us consider Figure 36. Three distinct blocks can be seen. The top block corresponds to COSMO processes, the middle block to ParFlow and the bottom to CLM processes. The overwhelmingly present red bars (in particular in CLM and to a lesser extent in ParFlow) represent MPI operations, of which a detailed analysis shows that they are mostly `MPI_Waitall`, whereas grey circles represent MPI communications. To understand this, it is useful to consider the concurrent operation of the component models illustrated in Figure 32 together with the traces in Figure 36. Additionally, consider Figure 37, in which only the first 30 seconds of the same trace are shown. After initialization (roughly 7 seconds), CLM computations are run, and then ParFlow runs. At approximately 14 s CLM finishes its computations and information is exchanged between the models. Afterwards CLM waits, while COSMO and ParFlow run. ParFlow runs for a short time (approx. 1 second), and then also waits, but COSMO continues to compute, after which communications occur (approx 17 seconds). At this time a full coupling time step has been achieved. The cycle then repeats itself. CLM starts again computations, which are very fast, followed again by ParFlow and COSMO. The recurrent cycles are easy to observe in the figures. It is important to note that the dominance of `MPI_Waitall` is misleading because while `MPI_Waitall` is blocking

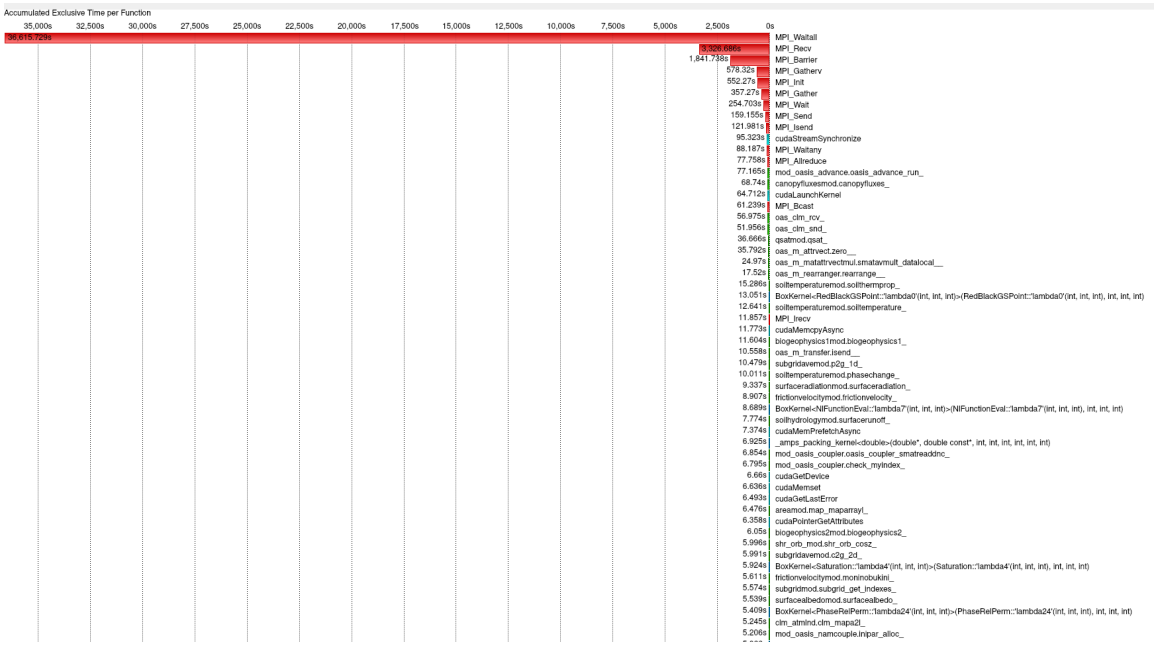


Figure 34: Example profile for the idealised case *A.300* under a heterogeneous configuration on JUWELS Cluster.

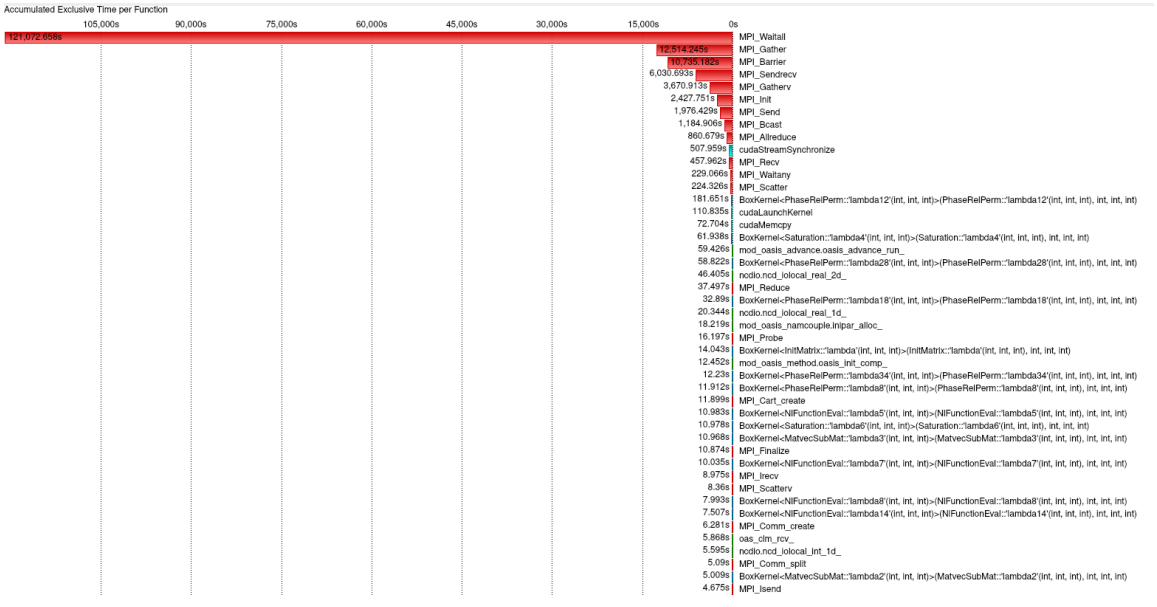


Figure 35: Example profile for case *B* (CORDEX) under a heterogenous configuration on JUWELS Cluster.

one component (typically CLM), other processes in the other two components are not necessarily waiting, but are most likely running. In this particular case CLM experiences very long wait times, and ParFlow has comparatively long wait times too. It is obvious from the trace that COSMO drives the computational cost of this particular case. This is essentially a load balancing issue, which is strongly case dependent (as it becomes evident when comparing with case *B*).

These traces nonetheless point to optimization opportunities in TSMP. The resources assigned to CLM are idle for long times and could be used for additional processes. Some of the solutions proposed in DEEP-SEA may enable a dynamic and malleable use of these resources, enabling, for example to run insitu postprocessing. The potential benefits and complexity of these tasks are yet to be explored.

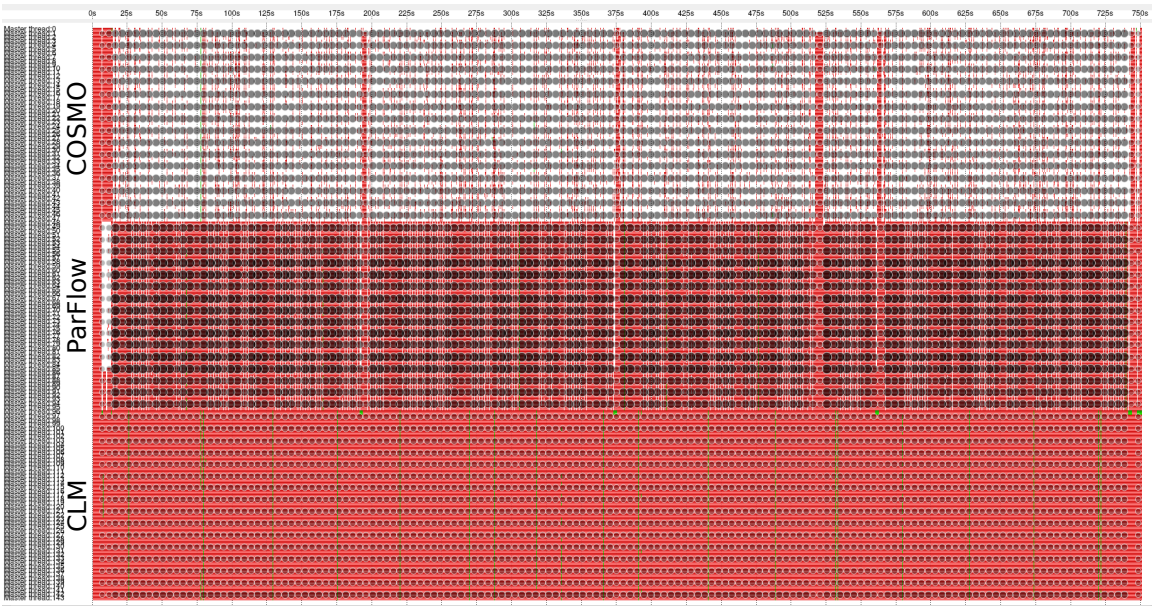


Figure 36: Example trace for case *A.300* under a CPU-only job configuration on JUWELS Cluster, showing all TSMP tasks (48 COSMO + 48 ParFlow + 48 CLM CPU processes).

Traces for case *B* are shown in Figure 38. Note that, for clarity, not all processes are explicitly shown. The number of processes are: 48 CLM CPU processes, 192 COSMO CPU processes, and 4 ParFlow GPU tasks. Figure 38 shows a similar pattern to that of case *A.300*. After initialization, COSMO runs and CLM waits (the `MPI_Waitall` calls can be explicitly seen). After the data exchange between components occurs, there are some wait times for COSMO. All the communications (grey circles) seen during computations (which appear as white space) are COSMO-internal exchanges (that is, responding to halo exchanges and not to model coupling exchanges). In this trace, ParFlow was run on GPUs, and therefore only 4 tasks exist for ParFlow. CUDA kernels appear in blue colours. Figure 39 shows a detailed view of 100 seconds (between 80 and 180 seconds of the job). In this figure it is easier to see the internal communications in COSMO intertwined with its computational kernels, and how COSMO sometimes needs to wait for ParFlow to reach the coupling time level. CLM can be seen again to mostly wait, but with ParFlow and COSMO running intensively. At some stages, ParFlow can also be seen waiting for COSMO computations to finish. This again highlights that the exclusive time reported for `MPI_Waitall` is misleading, and further detailed metrics need to be obtained from these traces. These traces are nonetheless useful and necessary, as the

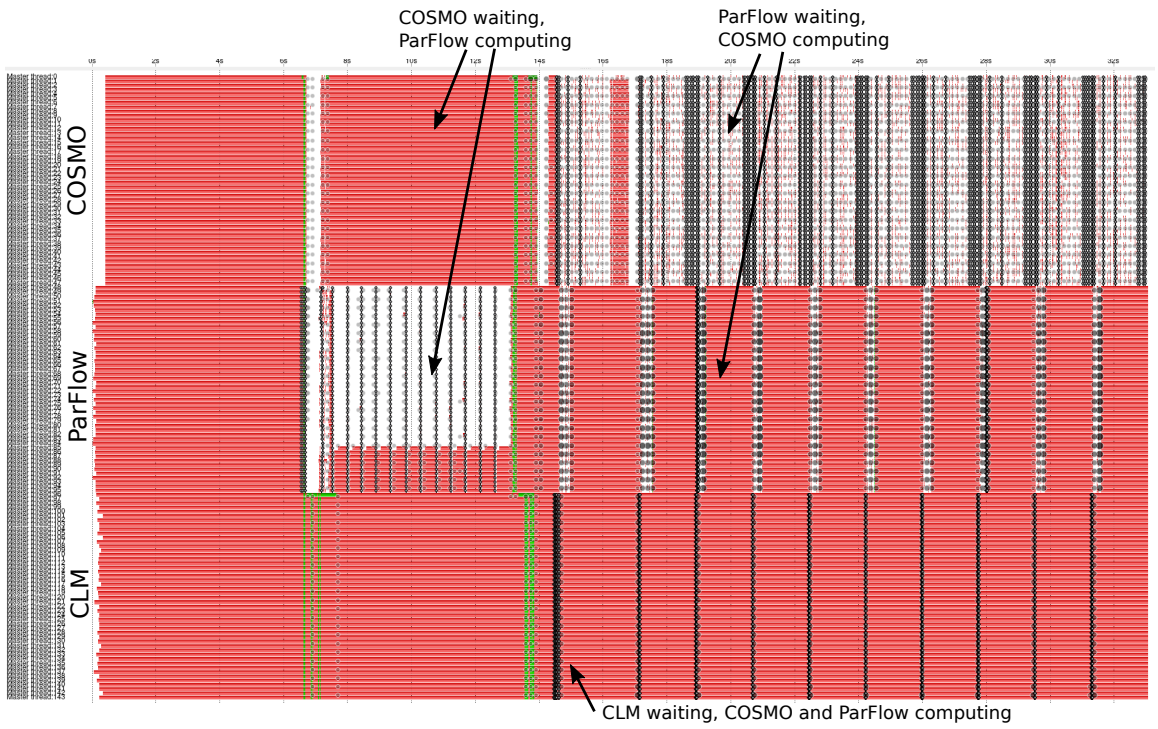


Figure 37: Detailed view of the trace for case *A.300* for the first 30 seconds, under a CPU-only job configuration on JUWELS Cluster, showing all TSMP tasks (48 COSMO + 48 ParFlow + 48 CLM CPU processes).

new heterogeneous configurations need to be inspected in detail to find optimal load balancing configurations for TSMP on heterogeneous hardware. The well-established configurations used on CPU-only runs will no longer hold on heterogeneous and modular sytems.

Figure 38: Example trace for case *B* (CORDEX) under a heterogeneous job configuration on JUWELS Cluster (not all tasks are shown). The configuration has 48 CLM (CPU) + 192 COSMO (CPU) tasks + 4 ParFlow GPU tasks.

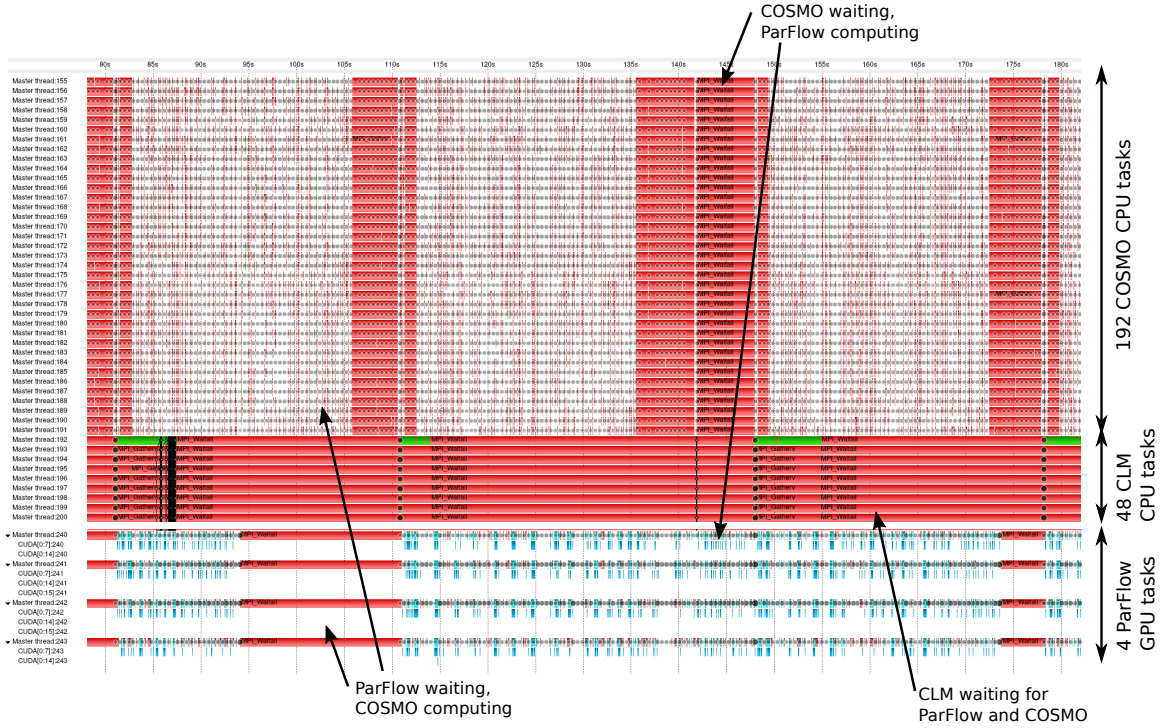


Figure 39: Detailed view of 80 – 100 seconds of the trace for case *B* (CORDEX) under a heterogeneous job configuration on JUWELS Cluster (not all tasks are shown). The configuration has 48 CLM (CPU) + 192 COSMO (CPU) tasks + 4 ParFlow GPU tasks.

Part IV.

Summary

17. Summary

This deliverable describes both the synthetic and application-based benchmarks that will be used in the DEEP-SEA project.

The initial set of synthetic benchmarks is used to monitor the overall system performance and provide references for elements of the DEEP-SEA software stack, not easily tested in the context of a full application. All the benchmarks are described by JUBE files. The files include instructions for building, running and analysing the benchmarks. This improves reproducibility and allows for regular monitoring. The benchmarks can be triggered by GitLab and run automatically making them a valuable tool for Continuous Integration (CI).

Each of the applications provides at least one of its use cases as a benchmark as well. Here, too, the benchmark setup is described using JUBE files and includes instructions for building or in the case of TSMP fetching the binaries, running the benchmark case, and analysing the results. In addition, most of the JUBE files include a mode that generates runtime traces of the application with Score-P or Extrae. All the JUBE files are made available to the project through a GitLab repository hosted at the Jülich Supercomputing Centre (JSC).

Each application provides one or more traces on JSC's shared file system. These traces are available to the members of the project and accessible through the DEEP system. Traces can give deep insights into the behaviour of an application and these insights are an important input for the co-design process.

List of Acronyms and Abbreviations

A

ACE	Asynchronous Constraint Execution, a pthread based task scheduler used by the FRTM application
AFSM	The All-Flash Storage Module (AFSM) is a purely flash-based storage module of the DEEP system
AIDApy	Python package for the analysis of space data developed by the AIDA project
API	An Application Programming Interface (API) allows for a software to communicate with other software that support the same API
ACE	Asynchronous Constraint Execution framework. A task based scheduling system for heterogeneous architectures

B

BDGS	Big Data Generator Suite efficiently generates scalable big data, such as a petabyte (PB) scale, while employing data models to capture and preserve the important characteristics of real data during data generation
BN	Booster Node (functional entity)
BoP	Board of Partners for the DEEP-SEA project
BSC	Barcelona Supercomputing Centre, Spain
BSIT	Barcelona Subsurface Imaging Tools is a software platform, designed and developed to fulfill the geophysical exploration needs for HPC applications.

C

CI/CD	Continuous Integration/Continuous Deployment (CI/CD) is an automated system for the testing, integration and deployment of software
CLM	Community Land Model
CM	The Cluster Module (CM) is one of the DEEP-system modules. It consists of 50 CNs.
CN	A Cluster Node (CN) consists of two high-end general-purpose CPUs
CORDEX	Coordinated Regional Climate Downscaling Experiment

COSMO Atmospheric model - 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

CEA French Alternative Energies and Atomic Energy Commission, France

D

DA Data Analysis

DAM Data Analytics Module: with nodes (DN) based on general-purpose processors, a large amount of (non-volatile) memory per core, and support for the specific requirements of data-intensive applications

DEEP Dynamical Exascale Entry Platform (project FP7-ICT-287530)

DEEP-ER DEEP – Extended Reach (project FP7-ICT-610476)

DEEP/-ER Term used to refer jointly to the DEEP and DEEP-ER projects

DEEP-EST DEEP – Extreme Scale Technologies

DEEP-SEA DEEP – Software for Exascale Architectures

DEEP system Prototype Modular Supercomputer deployed within the DEEP-EST project. It consist of three compute modules and two storage modules.

DN Nodes of the DAM

DaCe Data Centric Parallel Programming is a parallel programming framework that takes code in Python/NumPy and other programming languages, and maps it to high-performance CPU, GPU, and FPGA programs, which can be optimized to achieve state-of-the-art.

E

EC European Commission

ESB Extreme Scale Booster provides an additional 75 power-efficient nodes to the DEEP-EST Prototype, each hosting one Intel Xeon CPU and one NVIDIA V100 GPU, to address the needs of highly scalable codes and adapt them to the computer architectures likely to be used in the Exascale era.

EU European Union

Exascale Computer systems or Applications, which are able to run with a performance above 10^{18} Floating point operations per second

Extrae Extrae is a tool that uses different interposition mechanisms to inject probes into the target application to gather information regarding the application performance.

ECMWF European Centre for Medium-range Weather Forecasts, headquartered in Reading, UK

F

FFT Fast Fourier Transform

FLOP/s FLoating-point OPeration per Second

FP7 European Commission 7th Framework Programme

FPGA Field-Programmable Gate Array, Integrated circuit to be configured by the customer or designer after manufacturing

FRTM Fraunhofer RTM is a RTM software package developed by the Fraunhofer Institute for Industrial Mathematics

FWI Full Waveform Inversion (FWI) is a technique for estimating the medium parameters inside a medium of interest by means of the adjoint method. Obtaining gradients, the building block of FWI, is achieved by means of wave equation modelling. In terms of application design, gradient computation is similar to computing one FWI gradient.

G

GitLab GitLab is a platform for software development and information-technology operations. In this project it is used to organize developed software.

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

GROMACS A toolbox for molecular dynamics calculations providing a rich set of calculation types, preparation and analysis tools

GASPI Global Address Space Programming Interface is a Partitioned Global Address Space (PGAS) API. It aims at extreme scalability, high flexibility and failure tolerance for parallel computing environments.

H

H2020 Horizon 2020

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 (HPL) is a performant software package for solving linear system.

I

IB verbs The API for communication using InfiniBand (IB), a communication hardware

IFS Integrated Forecasting System

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

I/O Input/Output. May describe the respective logical function of a computer system or a certain physical instantiation

IPC Instructions Per Cycle

J

JUBE The JÜlich Benchmarking Environment

JURECA JURECA (Jülich Research on Exascale Cluster Architectures) Supercomputer at FZJ

JUWELS JUWELS (Jülich Wizard for European Leadership Science) Supercomputer at FZJ

K

KNL Knights Landing, second generation of Intel® Xeon Phi (TM)

KU Leuven Katholieke Universiteit Leuven, Belgium

L

LINPACK LINPACK is a software package for solving linear systems

Linktest Linktest is a communication-API benchmark

LLNL Lawrence Livermore National Laboratory

M

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

MPI Message Passing Interface, API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages

MPI-I/O MPI – Input/Output is an extentsion to MPI for I/O

MPMD Multiple-Program-Multiple-Data

MSA Modular Supercomputer Architecture

ML Machine Learning

MPDATA Multidimensional Positive Definite Advection Transport Algorithm, Smolarkiewicz 1988

MD Molecular Dynamics

MKL The Math Kernel Library is a mathematics library provided by Intel®

N

NoSQL NOn relational SQL (NoSQL) 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

NUMA Non-Uniform Memory Access

NEMO Nucleus for European Modelling of the Ocean, ocean forecasting model used in coupled IFS forecasts

Nek5000 Nek5000 is an open-source computational fluid dynamics code based on the spectral element method.

Nekbone Proxy app for Nek5000. It solves the standard Poisson equation using a conjugate gradient iteration with a simple or spectral element multigrid preconditioner on a block or linear geometry.

O

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

OpenMP Open Multi-Processing, Application programming interface that support multi-platform shared memory multiprocessing

OTF2 Open Trace Format Version 2

P

PAPADAM **P**roxy **A**pp to **P**lay **A**round with **D**EEP-SEA **A**PI for **M**PI, ECMWF development mini-app

ParaStation MPI ParaStation MPI is the MSA-enabled MPI library and runtime of ParaStation Modulo. It contains a high-performance communication library especially designed for HPC supporting different communication transports concurrently, and offers a complete process management system integrated with the batch queuing system and job scheduler.

Paraver Paraver is a flexible performance analysis tool.

ParFlow Hydrological model

PATMOS Monte Carlo neutron transport parallel application developped by CEA

PDAF Parallel Data Assimilation Framework

PIC Particle-in-Cell algorithm.

PME Particle Mesh Ewald

PMT Project Management Team of the DEEP-SEA project

Profile Recording of aggregated information, time measurements, and counts for function calls, bytes transferred, and hardware counters.

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

R

RAM Random-Access Memory

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

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

RTM Reverse Time Migration (RTM) is an imaging scheme that uses two related wavefields inside a medium of interest to image said medium. This is commonly achieved by zero-lag cross correlating the two wavefields in time, even though a temporal deconvolution of the two would be theoretically preferred. One of the used wavefields in the medium is commonly estimated from data recorded on the surface of the medium after wavefields were execited in the medium. The other wavefield is often estimated using an approximation of the excitation method that was dominantly responsible for the wavefield that was recorded on the surface of the medium which was used to estimate the other wavefield.

S

Score-P Scalable Performance Measurement Infrastructure for Parallel Codes. It is a software system that provides a measurement infrastructure for profiling, event trace recording, and online analysis of HPC applications.

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

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

SW Software

SDLTS Simulation and Data Laboratory Terrestrial Systems

T

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

Trace Recording detailed information about significant events during execution of the program and save information using a timestamp, location, and event type.

TSMP Terrestrial System Modelling Platform

TDFD Time-Domain Finite Difference

U

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

V

Vampir A framework that enables developers to quickly display and analyze arbitrary program behavior at any level of detail.

W

WP Work package

WAM WAve Model, used in IFS forecasts to predict the ocean-atmosphere interface

X

x86 Family of instruction set architectures based on the Intel® 8086 CPU

Xeon Non-consumer brand of the Intel® x86 microprocessors (TM)

Xeon Phi Brand name of the Intel® x86 many-core processors (TM)

xPic eXascale ready Particle-in-Cell code for space plasma physics.

Bibliography

[1] J. H. Meinke and A. Kreuzer. *DEEP-SEA Deliverable 1.1: Initial Application Co-Design Input*. Tech. rep. July 2021.

[2] Jülich Supercomputing Centre. *JUBE — Jülich Benchmarking Environment*. http://www.fz-juelich.de/jsc/jube. 2008.

[3] A. Knüpfer et al. "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir". In: *Tools for High Performance Computing 2011*. Ed. by H. Brunst et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91. ISBN: 978-3-642-31475-9. DOI: 10.1007/978-3-642-31476-6_7. URL: http://link.springer.com/10.1007/978-3-642-31476-6_7.

[4] Barcelona Supercomputing Center. *Extrae*. https://tools.bsc.es/paraver. 2006.

[5] A. Knüpfer et al. "The Vampir Performance Analysis Tool-Set". In: *Tools for High Performance Computing*. Ed. by M. Resch et al. Berlin, Heidelberg: Springer, 2008, pp. 139–155. ISBN: 978-3-540-68564-7. DOI: 10.1007/978-3-540-68564-7_9.

[6] Barcelona Supercomputing Center. *Paraver*. https://tools.bsc.es/paraver. 2001.

[7] S. Pickartz, M. Marazakis, and N. Eicker. *DEEP-SEA Deliverable 3.1: Software Architecture*. Tech. rep. Nov. 2021.

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

[9] LLNL. *HPC IO Benchmark Repository – IOR and mdtest parallel I/O benchmarks*. 2021. URL: https://github.com/hpc/ior.

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

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

[12] Intel. *Intel SSD D5-P4326 Series – Product Specifications*. 2021.

[13] O. S. Univsersity. *OSU-Microbenchmarks*. https://mvapich.cse.ohio-state.edu/benchmarks/. Nov. 8, 2021.

[14] W. Frings. "Efficient Task-Local I/O Operations of Massively Parallel Applications". RWTH Aachen, Diss., 2016. Dr. Jülich: RWTH Aachen, 2016, xiv, 140 S. ISBN: 978-3-95806-152-1. URL: https://juser.fz-juelich.de/record/811621.

[15] J. D. McCalpin. http://www.cs.virginia.edu/stream/stream2/. 2003.

[16] ICL and SNL. *HPCG Benchmark*. 2021. URL: https://www.hpcg-benchmark.org.

[17] J. Dongarra, P. Luszczek, and M. A. Heroux. *HPCG technical specification*. Technical Report SAND2013-8752. Sandia National Laboratories, 2013. DOI: https://doi.org/10.2172/1113870.

[18] M. A. Heroux and J. Dongarra. *Toward a New Metric for Ranking High Performance Computing Systems*. Technical Report SAND2013-4744. Sandia National Laboratories, 2013. DOI: https://doi.org/10.2172/1089988.

[19] ICL and SNL. *How big must the problem size be when running HPCG?* 2021. URL: https://www.hpcg-benchmark.org/faq/index.html%5C#360.

[20] AMD. *AMD – Developer Central Spack HPCG*. 2021. URL: https://developer.amd.com/spack/hpcg-benchmark/.

[21] 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.

[22] *ISO/IEC/IEEE International Standard - Floating-point arithmetic*. Tech. rep. 2020, pp. 1–86. DOI: 10.1109/IEEESTD.2020.9091348.

[23] J. Strohmaier et al. *Top500 List*. https://www.top500.org/. 2021.

[24] Intel. *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.

[25] Intel. *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.

[26] D. Van Der Spoel et al. "GROMACS: Fast, flexible, and free". In: *Journal of Computational Chemistry* 26.16 (2005), pp. 1701–1718. DOI: https://doi.org/10.1002/jcc.20291.

[27] P. Fischer and K. Heisey. "NEKBONE: Thermal Hydraulics mini-application". In: *Nekbone Release* 2 (2013).

[28] P. Shrestha et al. "A scale-consistent terrestrial systems modeling platform based on COSMO, CLM, and ParFlow". In: *Monthly weather review* 142.9 (2014), pp. 3466–3483.

[29] D. Jacob et al. *EUROCORDEX: New high-resolution climate change projections for European impact research, Regional Environmental Changes, 14, 563–578*. 2013.

[30] D. Jacob et al. "Regional climate downscaling over Europe: perspectives from the EURO-CORDEX community". In: *Regional environmental change* 20.2 (2020), pp. 1–20.

[31] J. Keune et al. "Studying the influence of groundwater representations on land surface-atmosphere feedbacks during the European heat wave in 2003". In: *Journal of Geophysical Research: Atmospheres* 121.22 (2016), pp. 13–301.

[32] S. Kollet et al. "Introduction of an experimental terrestrial forecasting/monitoring system at regional to continental scales based on the terrestrial systems modeling platform (v1. 1.0)". In: *Water* 10.11 (2018), p. 1697.

[33] C. Furusho-Percot et al. "Pan-European groundwater to atmosphere terrestrial systems climatology from a physically consistent simulation". In: *Scientific data* 6.1 (2019), pp. 1–9.