D3.1 Scalability, Optimization and Co-Design Activities (M12)

Date: January 18, 2024

Document Identification			
Status	Final	**Due Date**	31/12/2023
Version	1.1	**Submission Date**	18/01/2024

Related WP	WP3	**Document Reference**	D3.1
Related Deliverable(s)	D2.4, D5.3	**Dissemination Level (*)**	PU
Lead Participant	ICCS	**Lead Author**	Konstantinos Nikas, Petros Anastasiadis (ICCS)
Contributors	MTG, PSNC, SZE, UNISTRA	**Reviewers**	Davide Padeletti (USTUTT)
			Bartosz Bosak (PSNC)

Keywords:
High-Performance Computing (HPC), Benchmarking, Scalability

Disclaimer for Deliverables with dissemination level PUBLIC
This document is issued within the frame and for the purpose of the HiDALGO2 project. Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Poland, Germany, Spain, Hungary, France, Greece under grant agreement number: 101093457. This publication expresses the opinions of the authors and not necessarily those of the EuroHPC JU and Associated Countries which are not responsible for any use of the information contained in this publication. This deliverable is subject to final acceptance by the European Commission.
This document and its content are the property of the HiDALGO2 Consortium. The content of all or parts of this document can be used and distributed provided that the HiDALGO2 project and the document are properly referenced.
Each HiDALGO2 Partner may use this document in conformity with the HiDALGO2 Consortium Grant Agreement provisions.
(*) Dissemination level: **PU**: Public, fully open, e.g. web; **CO:** Confidential, restricted under conditions set out in Model Grant Agreement; **CI:** Classified, **Int =** Internal Working Document, information as referred to in Commission Decision 2001/844/EC.

Document Information

List of Contributors	
Name	**Partner**
Angela Rivera	MTG
Christophe Prud'homme	UNISTRA
József Bakosi	SZE
Konstantinos Nikas	ICCS
László Környei	SZE
Leydi Laura Salazar	MTG
Lukasz Szustak	PSNC
Mátyás Constans	SZE
Michał Kulczewski	PSNC
Petros Anastasiadis	ICCS
Sabela Sanfiz	MTG
Vincent Chabannes	UNISTRA
Wojciech Szeliga	PSNC

Document History			
Version	**Date**	**Change editors**	**Changes**
0.1	23/11/2023	ICCS	Initial version of the document - ToC
0.2	18/12/2023	PSNC, ICCS	Integrated RES input
0.3	19/12/2023	MTG, ICCS	Integrated WildFires input
0.4	20/12/2023	UNISTRA, ICCS	Integrated UB input
0.5	22/12/2023	SZE, ICCS	Integrated UAP input
0.6	28/12/2023	ICCS	Draft for internal review
0.7	28/12/2023	ICCS	Minor fixes of broken references
0.8	01/01/2024	ICCS	Addressed comments from USTUTT & PSNC
0.9	08/01/2024	ICCS	Revised draft sent for internal review
1.0	15/01/2024	ICCS	Final draft sent for review by the QM
1.1	17/01/2024	ICCS	Final version

Quality Control		
Role	**Who (Partner short name)**	**Approval Date**
Deliverable leader	Konstantinos Nikas (ICCS)	15/01/2024
Quality manager (deputy)	Dennis Hoppe (USTUTT)	16/01/2024
Project Coordinator	Marcin Lawenda (PSNC)	18/01/2024

Table of Contents

Document Information 3
Table of Contents 4
List of Tables 5
List of Figures 5
List of Acronyms 6
Executive Summary 7
1. Introduction 8
1.1 Purpose of the document 8
1.2 Relation to other project work 8
1.3 Structure of the document 8
2. HiDALGO2 Benchmarking Methodology 9
2.1 Benchmarking challenges 9
2.2 ReFrame 9
2.3 Using ReFrame in HiDALGO2 10
2.3.1 Script parameterisation and execution 11
2.3.2 Post-processing and storage of benchmarking results 12
3. Access to EuroHPC JU supercomputers 14
3.1 Getting access to EuroHPC JU systems 14
3.2 Awarded EuroHPC JU resources (M1-M12) 15
4. HiDALGO2 pilots' benchmarking 17
4.1 Renewable Energy Sources (RES) 17
4.1.1 Pilot description 17
4.1.2 Benchmarking 18
4.2 Urban Air Project (UAP) 22
4.2.1 Pilot description 22
4.2.2 Benchmarking 22
4.3 Urban Building (UB) 31
4.3.1 Pilot description 31
4.3.2 Benchmarking 32
4.4 Wildfires (WF) 35
4.4.1 Pilot description 35
4.4.2 Benchmarking 36
4.5 Summary & next steps 40
5. HiDALGO2 Co-Design Activities 43

5.1 HiDALGO2 strategy 43
6. Conclusions 44
7. References 45
Annexes – HiDALGO2 ReFrame scripts examples 47
Annex I 47
Annex II 50
Annex III 51

List of Tables

Table 1 Current EuroHPC JU systems coverage matrix. A green cell indicates that access has been awarded; a yellow cell indicates that a partner is waiting for or intends to request access to a system, and a red cell denotes a system that a partner is not planning to request access to, as it is either similar to another system/partition or not suitable for the execution of a pilot. 16
Table 2 Hardware configuration of CPU partitions used during the project's first year 17
Table 3 Hardware configuration of GPU partitions during the project's first year 17
Table 4 Programming & runtime environment for RES benchmarks 18
Table 5 Details of RES-EULAG benchmarking scenarios 19
Table 6 Programming & runtime environment for UAP-OpenFOAM benchmarks 23
Table 7 Programming & runtime environment for UAP-RedSim benchmarks 23
Table 8 Programming & runtime environment for UAP-Xyst benchmarks 23
Table 9 Details of UAP-OpenFOAM benchmarking scenarios 24
Table 10 Details of UAP-RedSIM benchmarking scenarios 24
Table 11 Details of UAP-Xyst benchmarking scenarios 24
Table 12 Benchmarking scenarios for the different implementations of UAP's CFD module 25
Table 13 Programming & runtime environment for UB benchmarks 32
Table 14 Details of Ktirio-UB benchmarking scenarios 33
Table 15 Programming & runtime environment for WF benchmarks 37
Table 16 Details of WF benchmarking scenarios 38
Table 17 Status of benchmarking and optimisation related KPIs in M12 41

List of Figures

Figure 1 RES-EULAG per node speedup for the 1h_r10 scenario 20
Figure 2 RES-EULAG per node speedup for the 1h_r5 scenario 20
Figure 3 RES-EULAG execution breakdown for the 1h_r10 scenario 21
Figure 4 RES-EULAG execution breakdown for the 1h_r5 scenario 21
Figure 5 UAP-OpenFOAM per node speedup for the Győr-728k scenario 25
Figure 6 UAP-OpenFOAM per node speedup for the Győr-3.4M scenario 26
Figure 7 UAP-OpenFOAM per node speedup for the Győr-14M scenario 26
Figure 8 UAP-OpenFOAM execution breakdown for the Győr-728k scenario 27
Figure 9 UAP-OpenFOAM execution breakdown for the Győr-3.4M scenario 27
Figure 10 UAP-OpenFOAM execution breakdown for the Győr-14M scenario 28

Figure 11 Speedup of multi-GPU implementation of UAP-RedSIM — 29
Figure 12 Speedup of multi-CPU implementation of UAP-RedSIM — 29
Figure 13 UAP-Xyst per node speedup for the Taylor-Green-794M scenario — 30
Figure 14 UAP-Xyst per node speedup for the Taylor-Green-144M scenario — 31
Figure 15 Ktirio-UB per node speedup for the 0-M1 scenario — 33
Figure 16 Ktirio-UB per node speedup for the 0-M2 scenario — 34
Figure 17 Ktirio-UB execution breakdown for the 0-M1 scenario — 35
Figure 18 Ktirio-UB execution breakdown for the 0-M2 scenario — 35
Figure 19 WF per node speedup for the small scenario — 38
Figure 20 WF per node speedup for the 2k_test using ndown for dynamic downscaling — 39
Figure 21 WF execution breakdown for the small scenario — 40
Figure 22 WF execution breakdown for the 2k_test scenario — 40

List of Acronyms

Abbreviation / acronym	Description
CFD	Computational Fluid Dynamics
CoE	Centre of Excellence
CPU	Central Processing Unit
EESSI	European Environment for Scientific Software Installations
EULAG	Eulerian/semi-Lagrangian fluid solver
GFS	Global Forecasting System
GIS	Geographic Information System
GPU	Graphics Processing Unit
GRIB	GRIdded Binary or General Regularly-distributed Information in Binary form
HPC	High Performance Computing
I/O	Input/Output
KPI	Key Performance Indicator
LOD	Level of Detail
MPI	Message Passing Interface
RES	Renewable Energy Sources
SIF	Singularity Image Format
UAP	Urban Air Project
UB	Urban Building
WF	Wildfires
WRF	Weather Research and Forecasting Model

Executive Summary

HiDALGO2 aims to follow a systematic and reproducible methodology for collecting and storing benchmarking results for all the HiDALGO2 pilots to enable their development and optimisation towards achieving the highest possible performance when running on the EuroHPC JU supercomputers. For this purpose, Deliverable D3.1 sets the guidelines of the benchmarking methodology that will be followed within the project.

During the first year of the project, the HiDALGO2 methodology has been applied for benchmarking the HiDALGO2 pilots on various EuroHPC JU systems and initial findings are reported in this document. A few bottlenecks have been already identified and constitute the primary target of the optimisation activities that will start in the second year of the project.

Finally, this document discusses the challenges faced in the project's attempts to acquire resources on the EuroHPC JU systems and outlines the project's strategy regarding co-design activities.

1. Introduction

1.1 Purpose of the document

Deliverable D3.1 "*Scalability, Optimization and Co-Design Activities (M12)*" is prepared in the context of WP3, which identifies and tackles the issues that are currently holding HiDALGO2 pilots back from achieving the highest possible performance when running on the EuroHPC JU supercomputers. This document defines the HiDALGO2 benchmarking methodology that will ensure the reproducibility and validation of the gathered results and will reliably assess the performance of applications and identify bottlenecks, providing thus critical input to the optimisation activities.

1.2 Relation to other project work

Deliverable D3.1 summarises initial findings regarding the performance and scalability of HiDALGO2 pilots, which are described in deliverable D5.3 "*Research Advancements for the Pilots*", on the project's HPC infrastructure, as defined in deliverable D2.4 "*Infrastructure Provisioning, Workflow Orchestration and Component Integration*". Deliverable D3.1 drives future activities within WP3 (*Exascale Support for Global Challenges*) and WP5 (*Tackling Global Challenges*). It is the first of a series of reports focusing on scalability, optimisation and co-design activities (D3.1 in M12, D3.2 in M22, and D3.3 in M47).

1.3 Structure of the document

The document is structured as follows:

- ***Chapter 2*** defines the benchmarking methodology that HiDALGO2 has defined to ensure the reproducibility and validation of results.
- ***Chapter 3*** highlights the challenges of acquiring resources on EuroHPC JU systems and presents the current machine coverage.
- ***Chapter 4*** describes the benchmarking configuration used for each HiDALGO2 pilot and presents the initial findings of their benchmarking.
- ***Chapter 5*** discusses the status of HiDALGO2 KPIs related to benchmarking and optimisation activities.
- ***Chapter 6*** describes the HiDALGO2 strategy regarding co-design activities.
- ***Chapter 7*** concludes the document.

2. HiDALGO2 Benchmarking Methodology

The purpose of benchmarking in HiDALGO2 is to continuously evaluate the efficiency of the various components of the HiDALGO2 pilots, identify performance bottlenecks and feed these results to the optimisation activities within the project. Motivated by the difficulties and pitfalls faced by its predecessor, the HiDALGO Centre of Excellence (CoE) [1], HiDALGO2 has defined a benchmarking methodology that focuses on the reproducibility and validation of the collected measurements.

2.1 Benchmarking challenges

Reproducibility is considered as a core requirement for benchmarking activities. However, the complex nature of HPC deployments makes the execution and repetition of benchmarks tedious and error-prone. The HiDALGO CoE faced this exact problem; the diversity of hardware architectures, system and programming environments, application codes, input datasets and output file formats complicated significantly the process of managing and deploying executions, and collecting and reporting results. To mitigate this problem, HiDALGO set up a formal, manual, cumbersome procedure for tracking, organising and logging executions and their results that, nevertheless, allowed the emergence of a few reproducibility issues and led to some inconsistencies in the reporting of the project.

To avoid these problems, the HiDALGO2 consortium has agreed on a uniform approach to benchmarking for all HiDALGO2 pilots that will ensure the ***reproducibility*** and ***validation*** of results. First, we have defined a set of fundamental benchmarking metrics that will be used across all HiDALGO2 pilots and will establish a common ground for performance comparisons. Second, we have agreed on employing ReFrame [2], an easy-to-use, powerful and efficient framework for managing and deploying the pilots' runs across all EuroHPC JU systems.

2.2 ReFrame

ReFrame is a framework dedicated to creating system regression tests and benchmarks, developed and maintained by the Swiss National Supercomputing Centre (CSCS). It is specifically tailored to HPC systems, offering many capabilities that significantly enhance testing methodologies and benchmarking pipelines. Besides CSCS [3], ReFrame is also used for testing the EPCC systems [4] and the Swedish HPC2N and C3SE clusters [5]. Additionally, it is employed for running the tests that comprise the European Environment for Scientific Software Installations (EESSI) test suite [6] and for executing the performance benchmarks and regression tests for the ExCALIBUR project [7].

ReFrame employs an easy-to-use Python-based design, enabling users to define tests and benchmarks as functions and implement complex pipelines as simple workflows. At the same time, ReFrame abstracts away the system interaction details, allowing the users to focus solely on the logic of their tests and, hence, create portable scripts that can be deployed on different HPC systems; this is critical for HiDALGO2, as we must deploy and benchmark all pilots on all EuroHPC JU systems, which differ in supported libraries, modules, parallel frameworks, etc. Finally, ReFrame facilitates the establishment of a standard reporting layout and post-processing of log files for all pilots, making comprehensive reporting feasible.

To employ ReFrame, users need to define two files: the *configuration file* and the *test file*. The former defines the supercomputing systems and their corresponding environments (compilers, modules, libraries) that will be used for running the tests or the benchmarks. Thus, deploying the same run on a different HPC system requires only the creation of a new configuration file (or the extension of an existing configuration file) with the new system's details without any modifications to the test file.

On the other hand, the test file encapsulates a class for testing that serves as a wrapper for multiple tests and benchmarks and supports the creation of complex pipelines with multiple stages and intra-stage dependencies. Additionally, users can define in the test file Python functions employed for pre- and post-processing, result validation and benchmarking. Consequently, deploying the execution of a new or modified pipeline on a specific HPC system requires only the creation or modification of the test file without any modifications to the configuration file.

2.3 Using ReFrame in HiDALGO2

We have developed a ReFrame code template that serves as a common framework among all HiDALGO2 pilots. Following ReFrame's modular approach, the test file has been partitioned into two independent files to separate the definition of the test pipeline from the definition of the benchmarking parameters. Specifically, the HiDALGO2 ReFrame code template comprises three core files, each assuming a unique role as follows:

- ***cluster_config_file.py***: This is the ReFrame configuration file and contains the description of the target HPC system and its partitions, along with access information for each system. It also defines the environments that will be utilised, i.e., the set of modules that need to be loaded before the execution starts. An example configuration file (used by the UB pilot for Altair, Discoverer, Karolina and MeluXina) is given in Annex I.
- ***benchmarking.py***: This file configures test-specific parameters, such as paths and input files. Additionally, it contains scheduler-specific variables that are necessary for launching and managing the execution of individual runs, such as

system partition, and nodes and cores configuration. An example (used for benchmarking the UB pilot) is given in Annex II.

- ***main.py***: This file contains the definition of the pipeline that will be executed. It includes sanity functions for validation and functions for results logging that can be tailored to the specific needs of the different pipelines of each pilot. An example (used for executing the UB pilot) is given in Annex III.

2.3.1 Script parameterisation and execution

The workflows of the HiDALGO2 pilots differ in terms of length and number of stages. Therefore, a different ReFrame pipeline has been developed to benchmark each pilot.

Renewable Energy Sources (RES)

RES already employed an in-house, PSNC-developed, Python-based scheduler for submitting and running the pilot on HPC systems prior to HiDALGO2. Therefore, for simplicity a ReFrame wrapper has been created around this scheduler. This wrapper sets all the necessary variables for the runs loading the input files of the simulated scenarios, while the different stages of the RES workflow are managed by the pre-existing scheduler.

Urban Air Project (UAP)

As detailed in Section 4.2.1, the UAP pilot employs three different codes for CFD. The implementation based on OpenFOAM [8] uses a pipeline with three stages, constant across all scenarios: *data import*, *domain decomposition* and *simulation*. The ReFrame test file has been adapted to implement this pipeline taking into account that the data import stage is executed only once for each distinct input, unlike the other two stages, which are executed multiple times for each input configuration. Further, the created script ensures that the simulation stage always runs on the nodes defined during the domain decomposition stage using a number of tasks equal to that of the decomposed OpenFOAM subdomains.

The integration of ReFrame and the OpenFOAM-based UAP has been a two-step process. First, it has been successfully completed and used on Altair and LUMI. In parallel, to speed up the benchmarking process for the purposes of this deliverable, runs on other EuroHPC JU systems reported in D3.1 have been performed using an in-house, SZE-developed, bash-based tool that has been used prior to HiDALGO2. The integration with ReFrame on these systems will be completed and tested in the second year of the project; similarly for the integration of ReFrame with the other two implementations of UAP.

Urban Building (UB)

The UB pilot uses Apptainer [9] to containerise its workflow. ReFrame is capable of launching containerized applications, hence the integration of UB and ReFrame has

been straightforward. A simple ReFrame test script with one stage has been created, that takes as input a SIF file and an Apptainer command, and launches the run using MPI on the requested number of nodes.

Wildfires (WF)

The WF pilot has been the most complex to integrate with ReFrame. This is because their code pipelines vary per simulated scenario, which leads to creating different ReFrame test files for each scenario. As further detailed in Section 4.4.1, WF consists of a *pre-processing* and *simulation* part. Pre-processing comprises three stages, while the simulation part consists of a dynamically adjusted number of stages based on the scale of the simulated problem.

The integration of these pipelines with ReFrame presented several challenges. The primary hurdle revolved around the configuration of temporary folders for the execution of each run and managing I/O between the different stages of the pipelines, which is done via copying or linking several intermediate files and directories.

2.3.2 Post-processing and storage of benchmarking results

The execution of multiple pilots simulating different scenarios and producing output with different data formats across multiple supercomputing centres complicates results' processing and storage. To address this issue, we have defined a uniform post-processing and log storing procedure and implemented it with ReFrame.

First, we have created pilot-specific post-processing functions in the ReFrame test files, which are executed at the end of each pilot pipeline. These functions parse the output data logs generated by the pilot and transform them to a csv-like format with a predefined layout (columns, datatypes, etc.). The data is stored in a file together with metadata regarding the HPC system including details such as the number of nodes and cores employed at each pipeline stage. Post-processing through ReFrame ensures that data regarding multiple executions of the same pilot for a specific benchmarking scenario using different node configurations will be collated and stored in a single file using the appropriate format.

The files generated on each HPC centre when benchmarking a pilot for a specific simulation scenario are stored in a central repository. More specifically, we have established a central repository for benchmarking (*hid-benchmarking*) in the HiDALGO2 Bitbucket server hosted by PSNC. There, we have established individual repositories for the benchmarking of each pilot and connected them as sub-modules of *hid-benchmarking*.

Each pilot repository uses a predefined directory structure to enable the efficient storing and retrieval of benchmarking data. At the top level, data is grouped by system and then by run type (i.e., benchmarking, profiling, etc.). At the last level, data is organised by simulation scenario; each directory is named after the scenario that was

used for benchmarking and contains the log files created by the post-processing functions explained previously, with the execution date and time embedded in the log file name.

As uploading these logs to the repository manually is time-consuming and error-prone, we have implemented a function that is executed at the end of our ReFrame scripts and extracts the necessary upload information, such as directory path and log name and initiates their push to the central repository. Hence, when benchmarking is completed, our ReFrame scripts parse, concatenate and upload the logs to the appropriate folder of the HiDALGO2 Bitbucket repository.

3. Access to EuroHPC JU supercomputers

HiDALGO2 aspires to deploy, benchmark and optimise the HiDALGO2 pilots on all EuroHPC JU systems. For this purpose, HiDALGO2 requires simplified access to the appropriate amount of resources on all EuroHPC JU supercomputers. Unfortunately, code owners have to apply for limited resources individually, making the entire process cumbersome. The situation is further complicated by the fact that each Hosting Entity has its own user management and accounting as well different security processes in place.

The following subsections detail the procedure followed by the HiDALGO2 partners in order to acquire resources and outline the current access status to the EuroHPC JU systems.

3.1 Getting access to EuroHPC JU systems

For the first year of the project, HiDALGO2 partners have acquired access to the EuroHPC JU systems through the "*Benchmark and Development Access Calls*". According to the calls, these are continuously open with a "***maximum time-to-resources-access of two weeks after the cut-off date***". However, according to our experience, this is not always the case, as detailed below:

- ***RES pilot***: PSNC applied for access to the CPU partitions of Karolina, LUMI, and MeluXina on 1st May 2023. While the proposal for LUMI was accepted after around three weeks, no feedback was provided for the other two systems. Almost two months after the initial submission, PSNC resubmitted its proposal for acquiring resources on MeluXina and Karolina and was immediately accepted for the former. However, for the latter, the PSNC proposal was finally accepted around 7 months after the original application.
- ***UAP pilot***: SZE applied for access to the CPU partitions of Discoverer, LUMI, and MeluXina, and the GPU partitions of Karolina and Vega on 1st February 2023. The proposal for Discoverer, MeluXina and Vega was accepted in the next two weeks. On the contrary, the proposals for accessing Karolina and LUMI were accepted around four and six weeks after the original application, respectively.
- ***WF pilot***: MTG applied for access to the CPU partitions of Discoverer, Karolina, LUMI, MeluXina and Vega on the cut-off on 1st August 2023. The proposal for Karolina and Vega was accepted in the next two weeks, and for LUMI in around four weeks. However, as there was yet to be a decision made for Discoverer and MeluXina, MTG resubmitted its application for both systems on 26th October. While MeluXina accepted the resubmitted proposal almost immediately, MTG has not received any feedback regarding Discoverer yet, almost 5 months after the initial application.

- ***Benchmarking***: ICCS, as the leader of benchmarking activities, applied on the 1st August cut-off for access to all EuroHPC JU systems available at the time. The application was accepted for Vega, Karolina and LUMI in around three, five and six weeks, respectively. As no feedback was provided for the rest of the systems, ICCS contacted PRACE at the start of October inquiring about the status of the application. Following this enquiry, the application was accepted for MeluXina and Leonardo in one and two weeks, respectively, and for Discoverer almost two months later. Finally, ICCS applied to access the GPU partition of LUMI on the 1st November cut-off, which was accepted in around three weeks.

In general, the entire process of getting access to the EuroHPC JU systems has not been straightforward. The "maximum time-to-resources-access of two weeks after the cut-off date" rule of the Benchmark & Development Access Calls seems to be essentially void, hindering the timely access of the project to the required resources.

3.2 Awarded EuroHPC JU resources (M1-M12)

Table 1 provides the EuroHPC JU systems coverage at the end of the first year of the HiDALGO2 project. Access to systems has been requested in such a way that HiDALGO2 does not focus on a subset of supercomputers and works on as many systems as possible, taking into account of course the implementations of the pilots. Specifically:

- No pilot has an FPGA-based implementation.
- RES, UB and WF are currently implemented only for execution on multiple CPUs.
- UAP uses three different codes. Two of those are implemented solely for execution on multiple CPUs, while the third has also a multi-GPU implementation, targeting NVIDIA GPUs.
- WF is interested on developing a version that can employ multiple GPUs.

Based on the above, no resources have been requested in the FPGA partition of MeluXina. Between Vega and MeluXina, where each node in the GPU partition has four NVIDIA A100 GPUs, SZE opted for VEGA for the deployment of the GPU implementation of UAP, together with Karolina, where each node has eight NVIDIA A100 GPUs. Finally, all pilot owners are targeting the pre-exascale machines, i.e. LUMI and Leonardo.

It should be noted that in order to kick-start the benchmarking activities, UNISTRA, the owner of the UB pilot, has been accessing EuroHPC systems through the ICCS grants. UNISTRA will apply for its own resources in the next cut-off date of the Benchmark and Development Access Call.

Table 1 Current EuroHPC JU systems coverage matrix. A green cell indicates that access has been awarded; a yellow cell indicates that a partner is waiting for or intends to request access to a system, and a red cell denotes a system that a partner is not planning to request access to, as it is either similar to another system/partition or not suitable for the execution of a pilot.

System	Partition	RES	UAP	UB	WF	Benchmarking
Discoverer	CPU					
Karolina	CPU					
	GPU					
LUMI	CPU					
	GPU					
Meluxina	CPU					
	GPU					
	FPGA					
Vega	CPU					
	GPU					
Leonardo	CPU					
	GPU					
MareNostrum5		*System not yet available*				
Deucalion		*System not yet fully accessible*				

4. HiDALGO2 pilots' benchmarking

This section reports the benchmarking activities that took place during the first year of the HiDALGO2 project. As the pilots have been executed on various systems, Table 2 and Table 3 provide details regarding the hardware platforms of the CPU and GPU partitions of the used supercomputing centres, respectively.

Table 2 Hardware configuration of CPU partitions used during the project's first year

	CPU/node	Cores/ node	Memory	Interconnect
Altair	2x INTEL Xeon 8268	48	192GB	InfiniBand @ 200 Gb/s
Discoverer	2x AMD EPYC 7H12	128	256GB	InfiniBand @ 200 Gb/s
Karolina	2x AMD EPYC 7H12	128	256GB	InfiniBand @ 200 Gb/s
LUMI	2x AMD EPYC 7763	128	256GB	Slingshot-11 @ 200 Gb/s
MeluXina	2x AMD EPYC 7H12	128	512GB	InfiniBand @ 200 Gb/s
Vega	2x AMD EPYC 7H12	128	256GB	InfiniBand @ 200 Gb/s

Table 3 Hardware configuration of GPU partitions during the project's first year

	CPU	Memory	GPU	GPU Memory
Karolina	2x AMD EPYC 7763	1 TB	8x NVIDIA A100	40 GB HBM2
Vega	2x AMD EPYC 7H12	512 GB	4x NVIDIA A100	40 GB HBM2

4.1 Renewable Energy Sources (RES)

4.1.1 Pilot description

A detailed presentation of the Renewable Energy Sources (RES) can be found in Deliverable D5.3 "*Research Advancements for Pilots*". In short, this pilot deals with different scenarios: i) prediction of *energy produced by wind farms*, ii) prediction of *energy produced by photovoltaic systems*, and iii) prediction of the *damages to the overhead electrical network*.

All these use cases use multiscale weather prediction models, namely WRF [10] and EULAG [11][12], which are coupled to each other. As WRF is the primary focus of another HiDALGO2 pilot (Wildfires), the benchmarking and optimisation activities of the RES pilot focus mainly on **EULAG**, an all-scale geophysical flow solver, written in Fortran and parallelised using message passing. The pilot is built in a modular way, such that every component can be executed and, hence, benchmarked independently: pre-processing for obtaining initial boundary conditions; pre-processing for mesoscale

weather prediction and actual run; coupling between the models; pre-processing for EULAG model and actual run, and post-processing of simulation results including visualisation. The execution of each module in an HPC environment is orchestrated by a framework written in Python.

4.1.2 Benchmarking

Systems & Environment

RES has been benchmarked on two supercomputers, the EuroHPC JU LUMI system and PSNC's Altair. The programming and runtime environments used in each system are given in Table 4.

Table 4 Programming & runtime environment for RES benchmarks

	Altair	LUMI
Compiler	GNU Fortran v.6.2.0	GNU Fortran v.10.3.0
Parallel framework	Open MPI v.4.1.0	Cray MPICH v.8.1.27
Libraries		
NetCDF-C	4.8.1	4.9.2
NetCDF-Fortran	4.5.3	4.6.1
HDF5-C	1.12.1	1.14.1
HDF5-Fortran	1.12.1	1.14.1
Python	3.10.11	3.9.17
Python modules		
cartopy	0.21.1	0.21.1
imageio	2.31.1	2.31.5
matplotlib	3.7.1	3.7.1
netcdf4	1.6.0	1.6.0
numpy	1.22.3	1.22.3
pandas	1.5.3	2.0.3
pyproj	3.4.1	3.4.1
pytest	7.4.0	7.4.3
retry	0.9.2	0.9.2
scipy	1.8.1	1.8.1
setuptools	68.0.0	68.2.2
wrf-python	1.3.4.1	1.3.4.1
xarray	2023.6.0	2023.11.0

Benchmarking configuration

The benchmarks presented in this deliverable are based on the third scenario described in Section 4.1.1, i.e., the prediction of damages. The analysis is conducted for the electrical overhead network over a 3.03km x 2.39km area; however, as this scenario is based on sensitive data provided by a Polish Distribution System Operator, more details regarding the simulated area cannot be disclosed. Initial conditions are

taken from the Global Forecasting System (GFS) [13] and are further elaborated by WRF executions on nested domains of 3,600m, 600m and 100m horizontal resolution.

In order to get an insight into how the number of grid points per core affects the performance of **RES-EULAG** while increasing the accuracy of the results, the domain used by the EULAG model covers the 3.03km x 2.39km area with two variants of horizontal resolution used for benchmarking purposes: 10m and 5m. The vertical resolution of 230m domain height is kept at the level of 5m and the simulated time is equal to one hour. The details of the two simulation scenarios are given in Table 5.

Table 5 Details of RES-EULAG benchmarking scenarios

Scenario	Horizonal mesh resolution	Grid resolution	Timestep	Simulated time
1h_r10	10m	320 x 252 x 46	0.05 s	1 hour
1h_r5	5m	608 x 472 x 46	0.02 s	1 hour

As the grid is divided between MPI tasks, each task of the 1h_r5 scenario receives 3.5x more data to be computed within a single timestep compared to the 1h_r10 scenario. As the horizontal grid gets denser, there is a need to decrease the timestep in order to preserve the numerical stability of the solver. Therefore, 1h_r5 requires 2.5x more timesteps to be computed.

Results & Analysis

Figure 1 and Figure 2 depict the speedup achieved on LUMI and Altair for the 1h_r10 and 1h_r5 scenarios, respectively. On Altair the pilot was executed with up to 85 nodes, i.e, 4080 cores; on LUMI it was executed with up to 10 nodes, i.e., 1,280 cores. Obtaining results for more than 10 nodes in LUMI for this deliverable was not possible due to the many of jobs waiting in the system's queue; we will include them in the forthcoming relevant deliverables (e.g. D3.2 in M22). In both figures, we present results for the total execution (*end-to-end*) of the pilot as well as the iterative RES-EULAG computational part (*simulation*).

For both systems and scenarios the code scales linearly or better up to 10 nodes. For the 1h_r10 scenario, RES-EULAG scales better than linear for 2-6 nodes because of the best fit of data size to be computed within a single CPU. Adding more nodes affects the speedup, but it still remains close to linear up to 10 nodes; running on more than 10 nodes, allows to obtain results in less amount of time, however the speedup tends to flatten.

As explained before, for the 1h_r5 scenario, each task receives a 3.5x larger subdomain to be computed compared to the 1h_r10 scenario, i.e., 1h_r5 is more compute-intensive than 1h_r10; hence, the speedup is expected to be better. Indeed,

as shown in Figure 2, the speedup achieved for 1h_r5 drops slower compared to 1h_r10. It is estimated that linear speedup could be achieved for 32 nodes on LUMI.

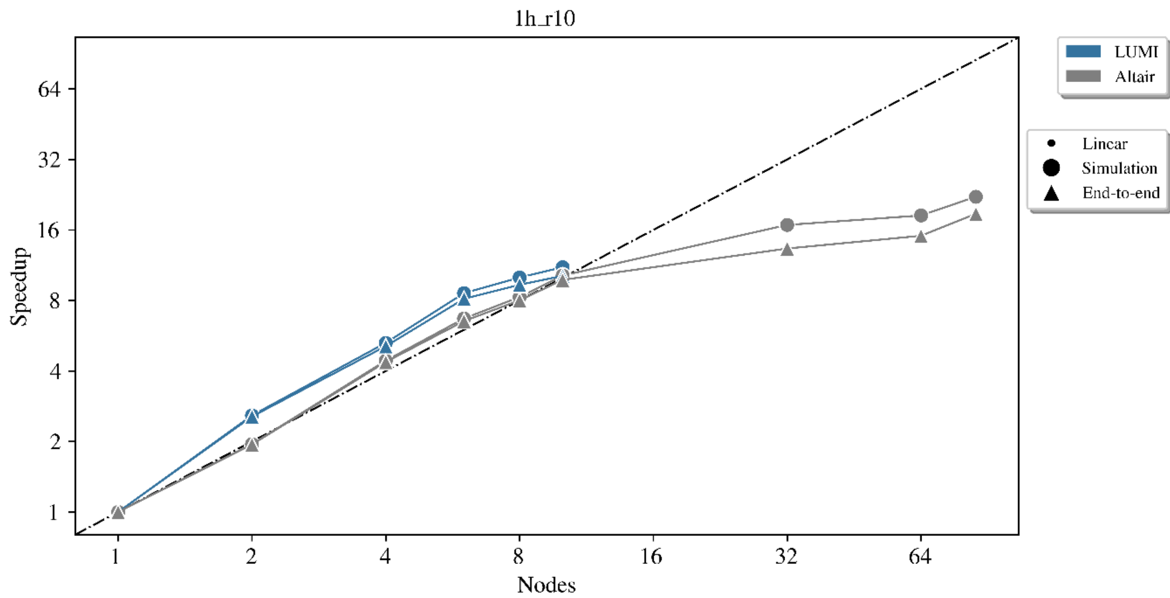


Figure 1 RES-EULAG per node speedup for the 1h_r10 scenario

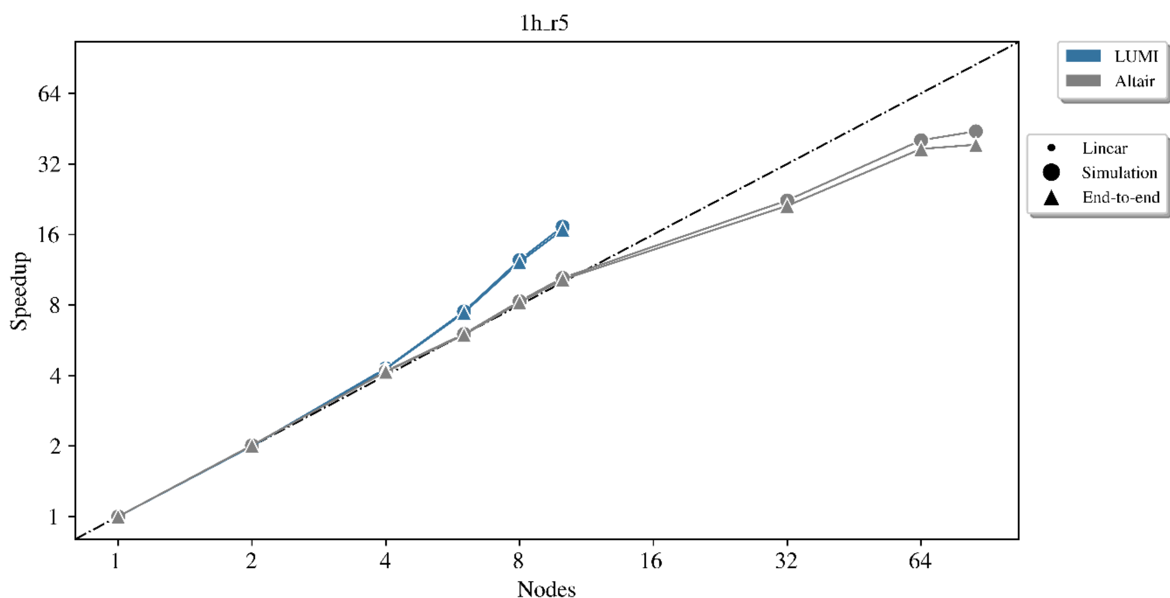


Figure 2 RES-EULAG per node speedup for the 1h_r5 scenario

Figure 3 and Figure 4 show how the execution is split between the computational part and the pre- and post-processing stages. Pre-processing is mainly focused on reading initial simulation conditions from files and distributing them among processes, while post-processing on processing output data and generating images.

In both scenarios, post-processing becomes more significant as the number of nodes is increased. This is due to the computational time decreasing while processing output

data is fairly constant and independent of the number of nodes used. It is evident that it will be beneficial to parallelise the I/O, in particular for longer production runs.

On the other hand, pre-processing stage is short and almost negligible for up to 10 nodes. However, some unexpected long runtimes were observed on Altair when executing the 1h_r10 scenario with 32 and 64 nodes. We believe that these are due to some temporary issues with the storage of the system, as we did not observe something similar when executing the 1h_r5 scenario. Nevertheless, these runs need to be re-executed to confirm that this is indeed the case.

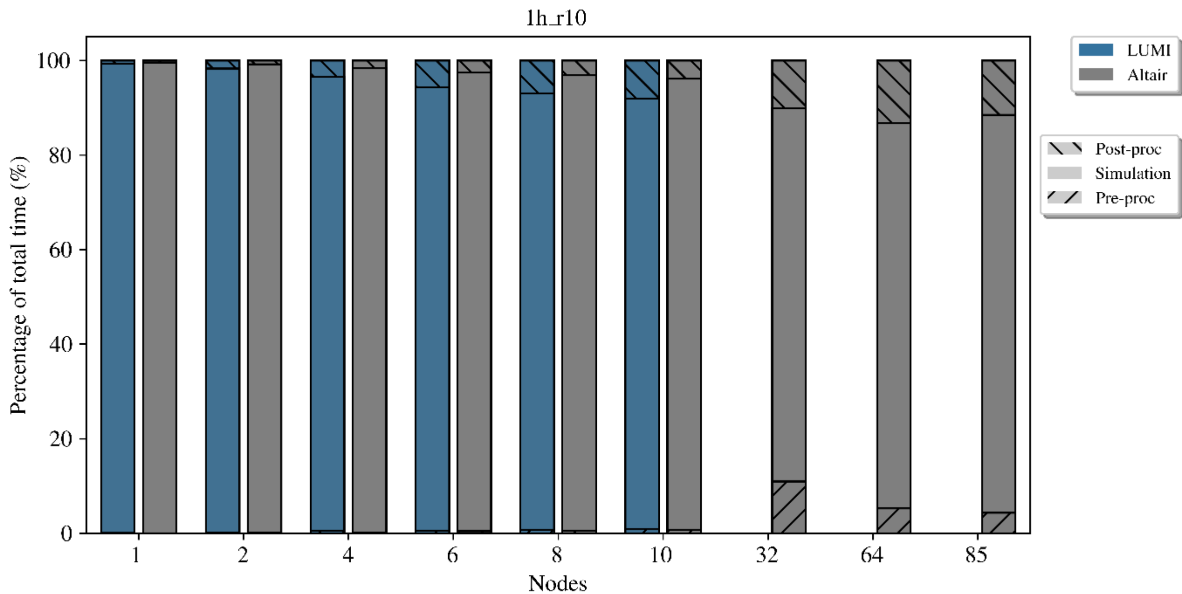


Figure 3 RES-EULAG execution breakdown for the 1h_r10 scenario

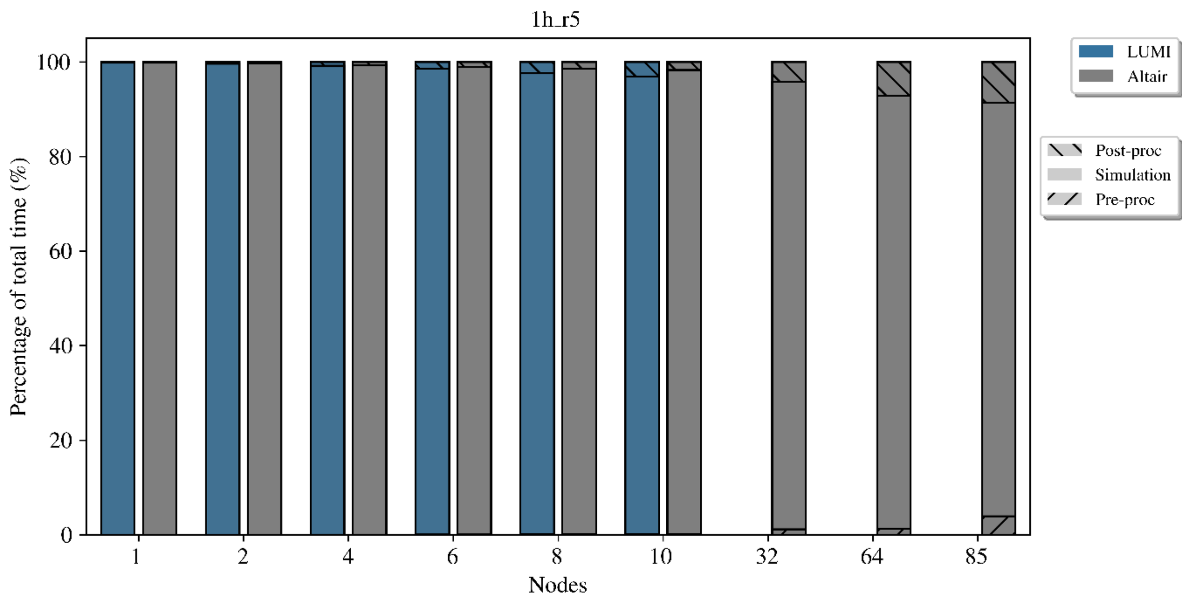


Figure 4 RES-EULAG execution breakdown for the 1h_r5 scenario

4.2 Urban Air Project (UAP)

4.2.1 Pilot description

The CFD module of the Urban Air Project (UAP) workflow calculates airflow within a city. It is the most computationally demanding module of the workflow and will be the focus of the benchmarking activities reported in this deliverable.

UAP uses three different codes for CFD: **OpenFOAM**, **RedSIM**, and **Xyst**.

The first implementation is based on OpenFOAM [14], with custom modules developed by SZE for imposing external, time-dependent atmospheric boundary conditions and time-dependent pollution sources. It uses MPI for inter-process communication and its pipeline consists of four stages:

1. First, external data, including an unstructured mesh, boundary condition tables and source term tables, all provided by the UAP pipeline, are converted to OpenFOAM format.
2. Next, the calculation domain is decomposed into several subdomains that correspond to the number of processes that will be executed.
3. Then, a parallel section follows, where a steady state of the incompressible Navier-Stokes equations is calculated using simpleFoam [15] with the initial set of constant parameters and conditions.
4. Finally, the produced result serves as an initial condition for the unsteady part solving the unsteady incompressible Navier-Stokes equations with time varying boundary conditions with pimpleFoam [16].

The second code is RedSIM, an in-house code developed by SZE that solves the compressible Navier-Stokes equations on unstructured grids with boundary conditions provided by the UAP pipeline. SZE has implemented a multi-CPU and a multi-GPU version of RedSIM. The former leverages OpenMP and MPI and works in a master-slave configuration, with one node handling administrative tasks and I/O, while the latter relies on CUDA targeting primarily NVIDIA GPUs.

Finally, the third code is Xyst [17], another in-house code developed by SZE that is open source and contains multiple finite element solvers using unstructured tetrahedron grids. Instead of MPI, Xyst relies on the Charm++ runtime system [18]. Charm++'s execution model is asynchronous by default and enables automatic redistribution of computational load based on real-time CPU measurements.

4.2.2 Benchmarking

Systems & Environment

UAP has been benchmarked on five EuroHPC JU systems. Each system's programming and runtime environments used in each system are detailed in Table 6 for UAP-OpenFOAM, Table 7 for UAP-RedSim and Table 8 for UAP-Xyst, respectively.

Table 6 Programming & runtime environment for UAP-OpenFOAM benchmarks

	Discoverer	LUMI-CPU	Meluxina-CPU
Compiler	GNU GCC 11.3.0	Cray clang 14.0.2	GNU GCC 11.3.0
Parallel framework	Open MPI 4.1.4	Cray MPICH 8.1.18	Open MPI 4.1.4
Libraries			
OpenFoam	2206	2112	2206

Table 7 Programming & runtime environment for UAP-RedSim benchmarks

	Karolina-GPU	Vega-GPU	LUMI-CPU
Compiler	GNU GCC 12.2.0	GNU GCC 12.3.0	GNU GCC12.2.0
Compiler-GPU	CUDA 11.3	CUDA 12.2.2	-
Parallel framework	-	-	Cray MPICH 8.1.27
Libraries			
zlib	-	1.2.13	-

Table 8 Programming & runtime environment for UAP-Xyst benchmarks

	Discoverer	LUMI-CPU	Meluxina-CPU
Compiler	GNU GCC 11.3.0	g++ (SUSE Linux) 7.5.0	GNU GCC 11.3.0
Parallel framework	Open MPI 4.1.4	Open MPI 4.1.4	Open MPI 4.1.4
Libraries			
ninja	-	-	1.11.1
netCDF	4.9.0	4.9.0	4.9.0
netlib-lapack	3.10.1	3.10.1	3.10.1

Benchmarking configuration

The benchmarking of UAP-OpenFOAM is performed for a 3D simulation of the city of Győr on multiple unstructured grids with varying sizes, number of iterations in simpleFoam and simulated times in pimpleFoam. More specifically, three scenarios are used, which are presented in Table 9.

The multi-GPU implementation of UAP-RedSIM is benchmarked for a 3D simulation based on a tetrahedral unstructured mesh using the geometry of the city of Győr. Additionally, it is benchmarked using Karman vortex calculations in 2D for various mesh sizes. These 2D scenarios are also used to benchmark the multi-CPU implementation of RedSIM. In total, 7 and 4 scenarios are used for benchmarking the multi-GPU and multi-CPU implementations of UAP-RedSIM, respectively. Their details are presented in Table 10.

Finally, for the benchmarking of UAP-Xyst, the Taylor-Green problem, which is widely used in fluid dynamics for verification, is used. The system of equations solved is 3D the Euler equations, augmented by a source term of the energy equation to ensure a stationary 2D periodic vortical flow. The simulation domain is a cube centred around

the point {0,0,0}. The initial conditions are sampled from the analytic solution at t=0. We set Dirichlet boundary conditions on the sides of the cube, sampling the analytic solution. The numerical solution does not depend on time and approaches a steady state due to the source term which ensures equilibrium in time. As the numerical solution approaches a stationary state, the numerical errors in the flow variables converge to stationary values, determined by the combination of spatial and temporal errors, which are measured and assessed.

The simulation is run for a single time unit to assess numerical errors. For the benchmarking, only 10 time steps are taken, since the time taken by a single time step is verified to be approximately equal for any time step. In total, 2 scenarios of varying mesh sizes are used that are presented in Table 11.

Table 12 summarises the various scenarios used for benchmarking the three different implementations of UAP's CFD module.

Table 9 Details of UAP-OpenFOAM benchmarking scenarios

Scenario	Mesh size	Iterations	Simulated time
Győr-728k	728,000	600	3600 s
Győr-3.4M	3,400,000	600	900 s
Győr-14M	14,000,000	400	100 s

Table 10 Details of UAP-RedSIM benchmarking scenarios

Scenario	Mesh size	Iterations	UAP-RedSIM versions
Győr-2.1M	2,100,000	5000	multi-GPU
Győr-10.1M	10,100,000	1000	
Karman-3.7M	3,700,000	5000	multi-GPU, multi-CPU
Karman-33.7M	33,700,000	500	
Karman-60.0M	60,000,000	25	
Karman-184M	184,000,000	125	
Karman-375M	375,000,000	50	multi-GPU

Table 11 Details of UAP-Xyst benchmarking scenarios

Scenario	Mesh size	Iterations
TaylorGreen-144M	144,000,000	10
TaylorGreen-794M	794,000,000	10

Table 12 Benchmarking scenarios for the different implementations of UAP's CFD module

Code	Scenario	Mesh sizes
UAP-OpenFOAM	Győr (3D)	728k, 3.4M, 14M
UAP-RedSIM	Győr (3D)	2.1M, 10.1M
	Karman vortex (2D)	3.7M, 33.7M, 60M, 184M, 375M
UAP-Xyst	Taylor-Green (3D)	144M, 794M

Results & Analysis

UAP-OpenFOAM

Figure 5, Figure 6 and Figure 7 depict the speedup achieved on LUMI, MeluXina and Discoverer for the Győr-728k, Győr-3.4M and Győr-14M scenarios, respectively. In all figures, we present results for the total execution (*end-to-end*) and the OpenFOAM computational part (*simulation*).

For low mesh cell counts, multi node execution is not advantageous, as additional nodes provide a slight benefit compared to single-node runs. For the medium-size mesh, the performance is improved up to 16 nodes, and better efficiency is achieved for up to 4 nodes. For the largest mesh, the efficiency peaks for 8 nodes for all three EuroHPC machines; however, performance improvements slow down for more nodes.

Finally, the observed superlinear speedup for the larger meshes (Győr-3.4M and Győr-14M) is due to memory bottlenecks suffered by the baseline execution on a single node. Specifically, the data cannot fit in the L3 cache of the EuroHPC machines' AMD cores, impacting the performance of the single node. However, when more nodes are used and the mesh is partitioned between them, data allocated to each node fits better in the memory hierarchy of each CPU, leading to superlinear speedups.

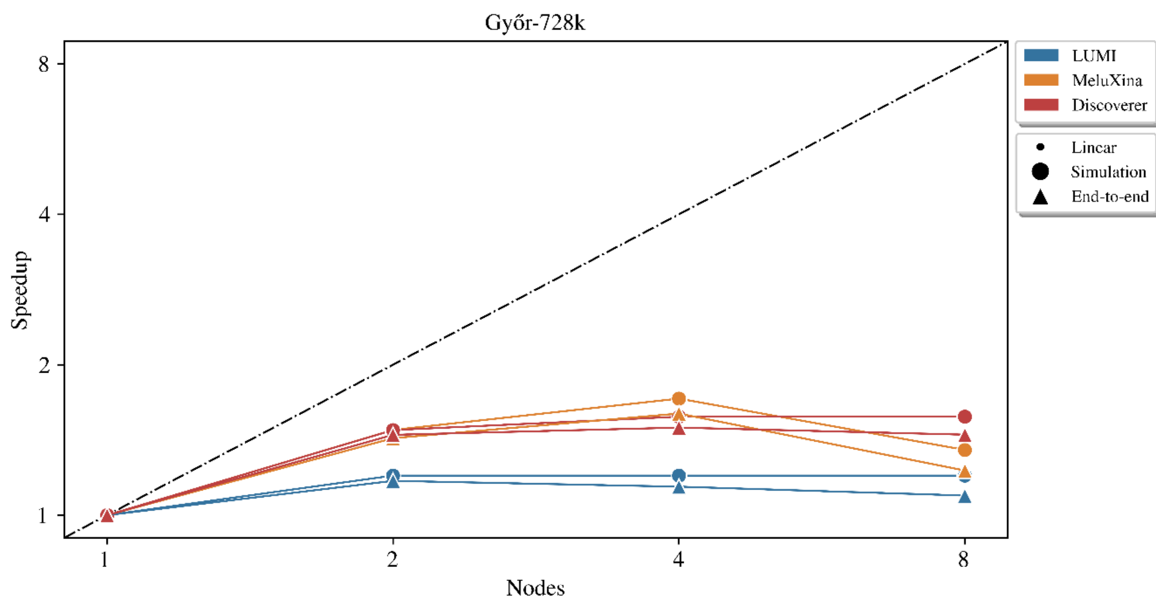


Figure 5 UAP-OpenFOAM per node speedup for the Győr-728k scenario

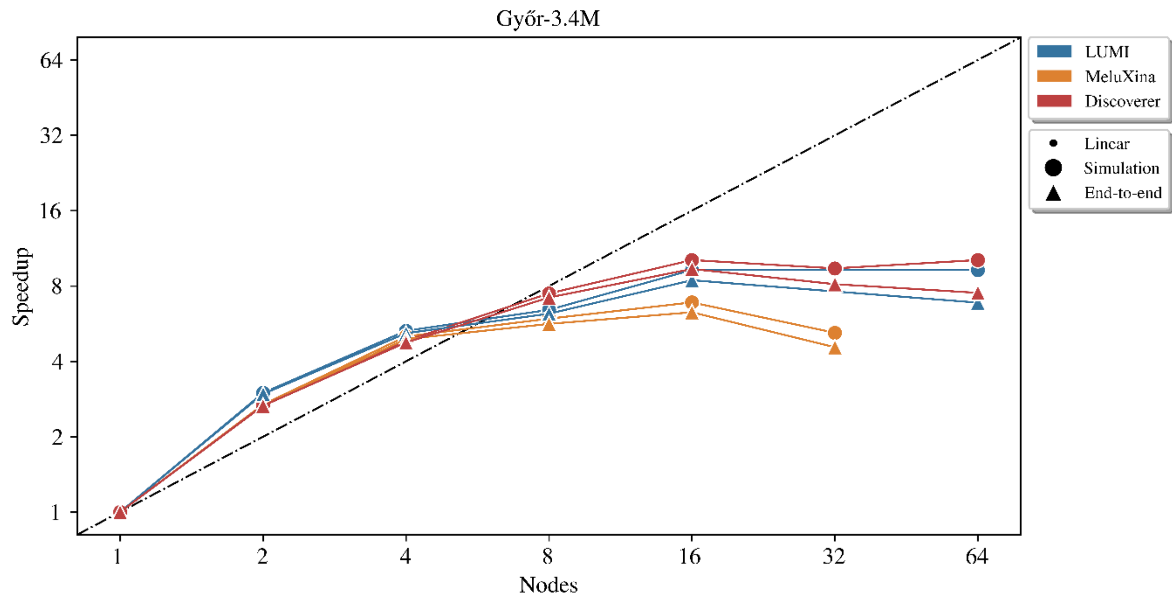


Figure 6 UAP-OpenFOAM per node speedup for the Győr-3.4M scenario

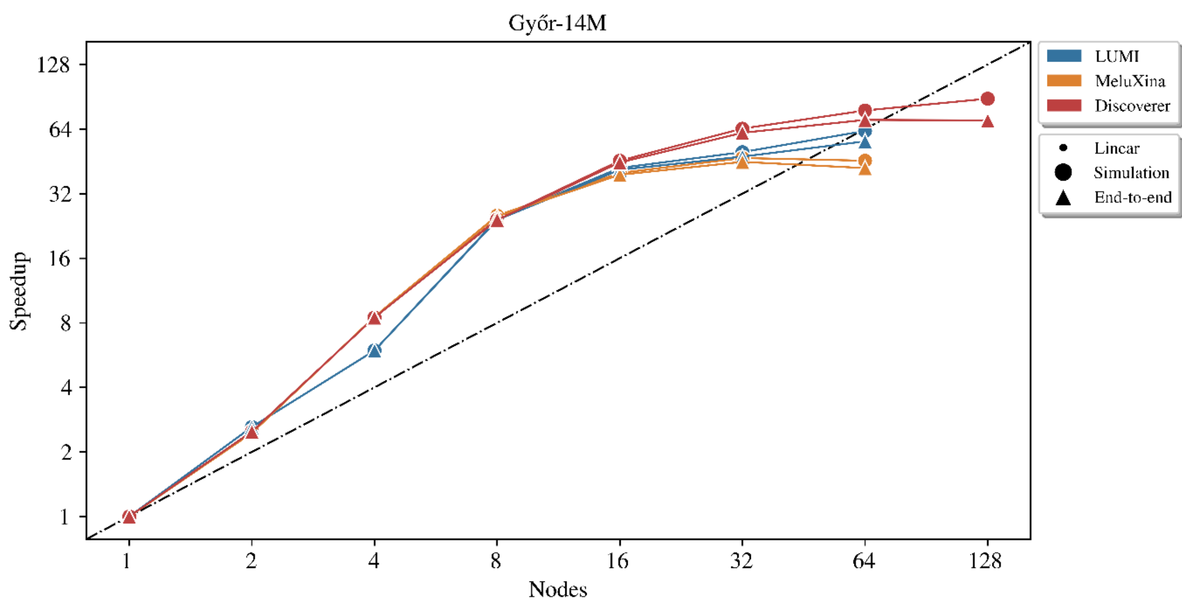


Figure 7 UAP-OpenFOAM per node speedup for the Győr-14M scenario

Figure 8, Figure 9 and Figure 10 show how the execution of UAP-OpenFOAM is split between the computational part and the pre-processing part of the simulation. As the length of the pre-processing stage depends on the mesh size, to compare the two stages appropriately, we scale the simulated time to that of a production run for which the simulated time is equal to one day or 86400 seconds and calculate the simulation time accordingly.

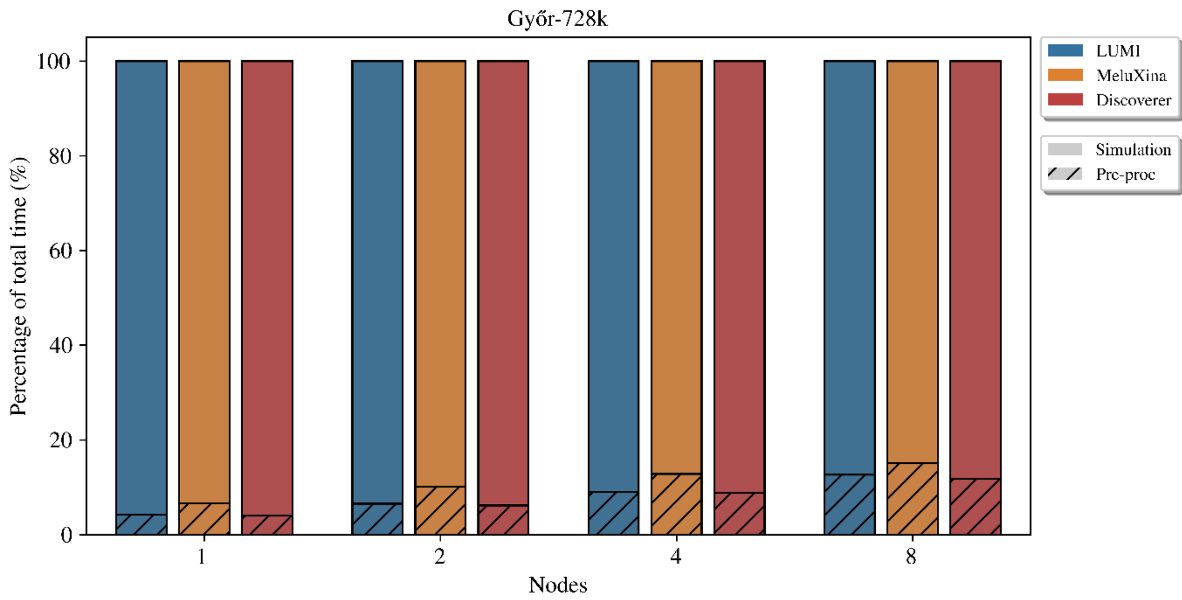


Figure 8 UAP-OpenFOAM execution breakdown for the Győr-728k scenario

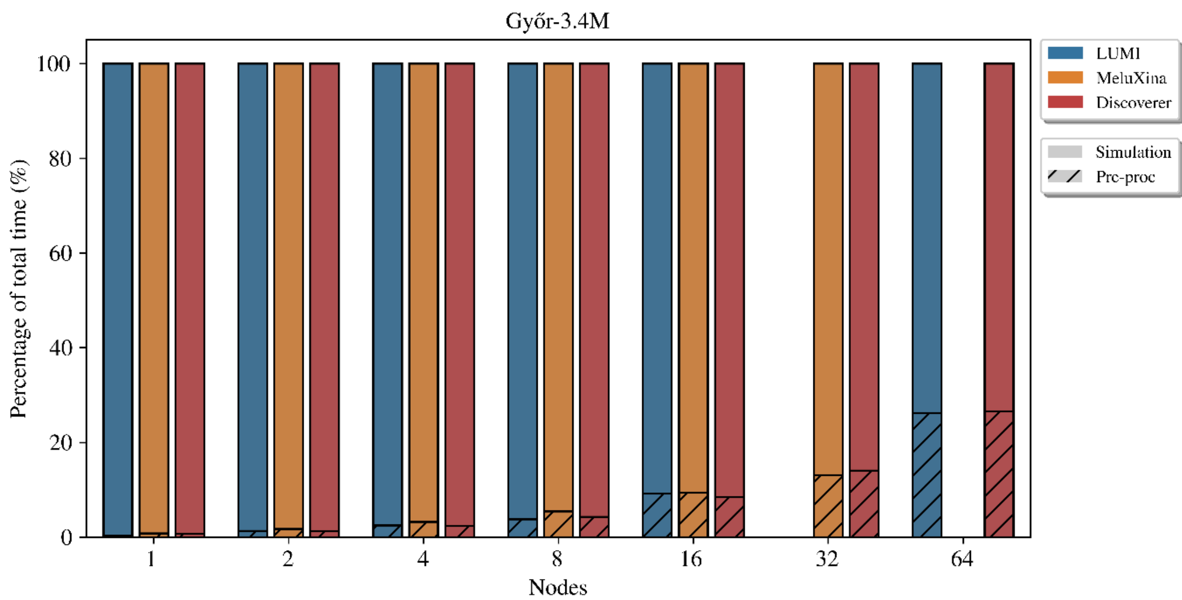


Figure 9 UAP-OpenFOAM execution breakdown for the Győr-3.4M scenario

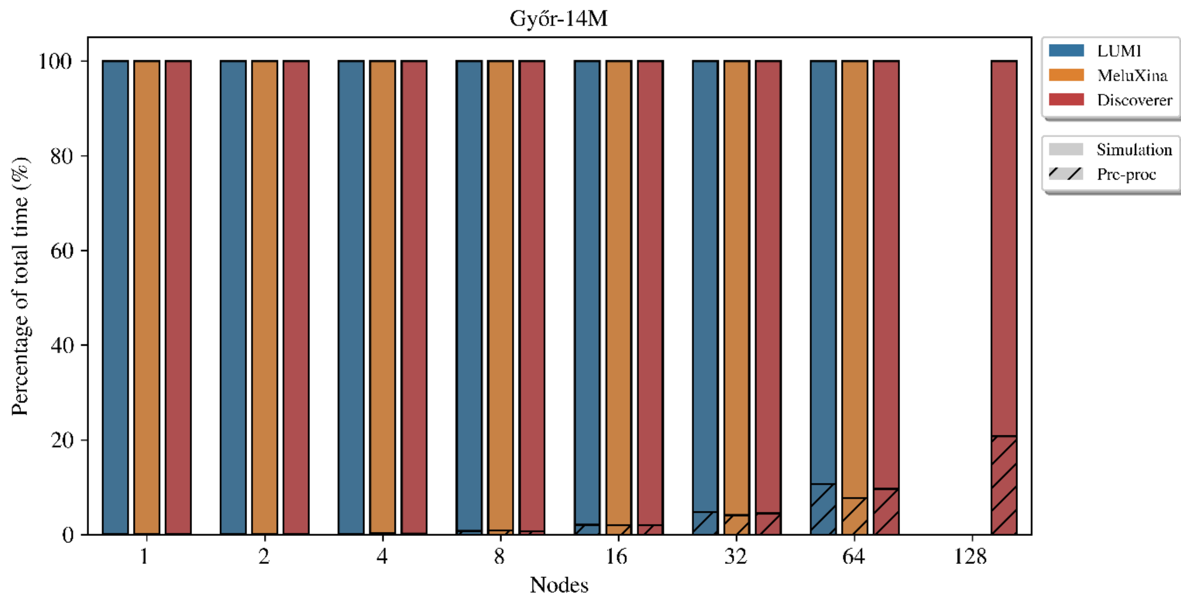


Figure 10 UAP-OpenFOAM execution breakdown for the Győr-14M scenario

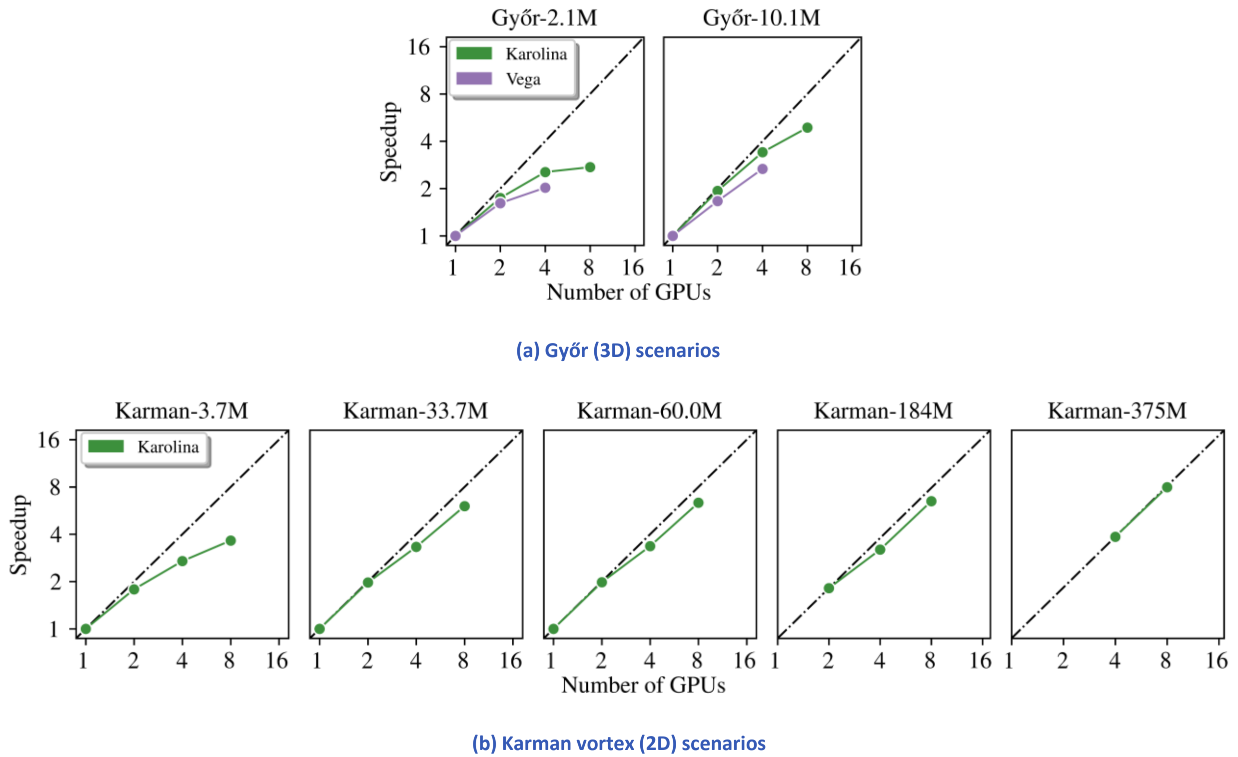
For small meshes, pre-processing can take up a significant portion of the execution. However, as the size of the mesh increases, pre-processing becomes less significant compared to the actual simulation, at least for low node counts. As the nodes increase to 32 or 64 for Győr-3.4M and to 64 or 128 for Győr-14M, the simulation time is decreased, and pre-processing becomes significant again.

UAP-RedSIM

For the benchmarking of the multi-GPU implementation of UAP-RedSIM, two different problems of various sizes were used. Figure 11a presents the achieved speedup on Karolina and Vega for the 3D simulations (Győr-2.1M and Győr-10.1M scenarios), while Figure 11b shows the achieved speedup for the 2D simulations on Karolina (Karman-3.7M, Karman 33.7M and Karman-60M scenarios).

For both problems, as the size of the mesh gets larger, UAP-RedSIM scales better, achieving almost linear speedup for 8 GPUs on Karolina. At the same time, the larger sizes cannot be executed for a small number of GPUs, as they do not fit in the memory and the executions fail with an out-of-memory error (for 1 GPU running Karman-184M and for 1 and 2 GPUs running Karman-375M).

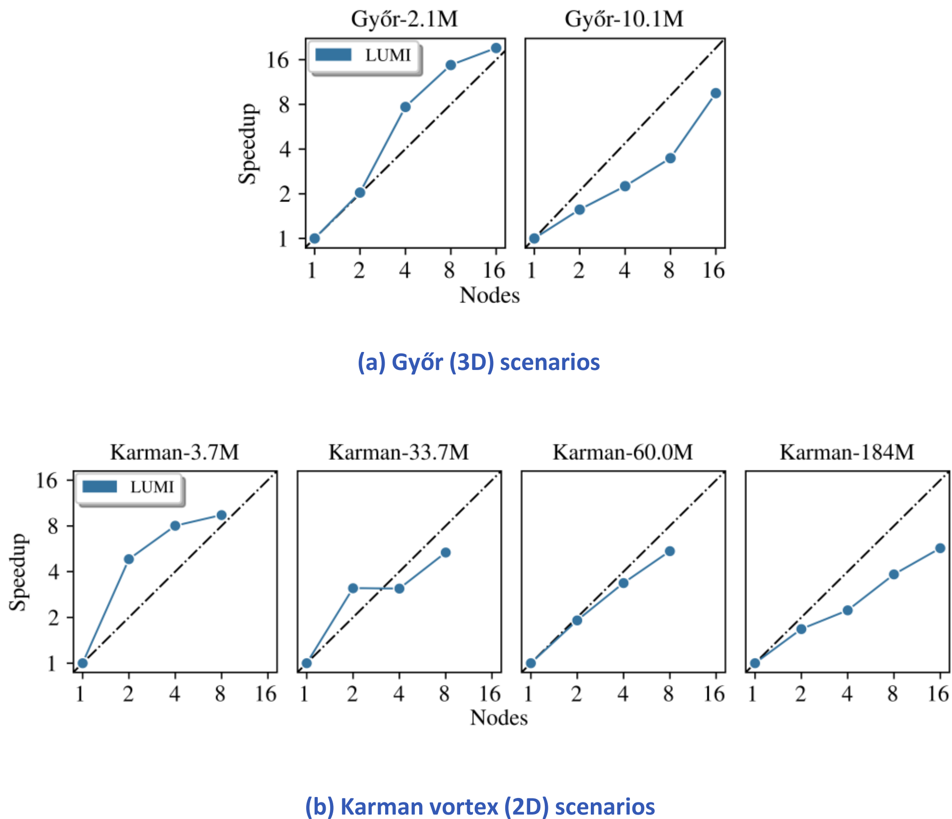
The same two problems have also been used for the benchmarking of the multi-CPU implementation of UAP-RedSIM on LUMI. Figure 12a presents the achieved speedup for the 3D simulations (Győr-2.1M and Győr-10.1M scenarios), while Figure 12b shows the achieved speedup for the 2D simulations (Karman-3.7M, Karman 33.7M and Karman-60M scenarios). The causes for sublinear and superlinear speedup behaviour are still under investigation.



(a) Győr (3D) scenarios

(b) Karman vortex (2D) scenarios

Figure 11 Speedup of multi-GPU implementation of UAP-RedSIM



(a) Győr (3D) scenarios

(b) Karman vortex (2D) scenarios

Figure 12 Speedup of multi-CPU implementation of UAP-RedSIM

UAP-Xyst

Figure 13 and Figure 14 depict the speedup achieved for UAP-Xyst when running the Taylor-Green-794M scenario on LUMI, MeluXina and Discoverer, and the scenario Taylor-Green-144M scenario on MeluXina, respectively.

In general, UAP-Xyst scales very well. The superlinear speedup observed in the case of MeluXina can be attributed to the larger amount of memory per node compared to the other two EuroHPC systems (512MB on each MeluXina node versus 256MB on each LUMI and Discoverer node), which combined with the partitioning of the problem between the nodes causes the data to fit into the processors' memory hierarchy a lot quicker when scaling the number of nodes. The lack of scalability beyond 32 nodes on Discoverer is currently under investigation.

Finally, regarding the smaller scenario illustrated in Figure 14, UAP-Xyst's strong scalability plateaus at about 9K elements per compute core (at 128 nodes), beyond which point communication becomes a bottleneck limiting the scalability.

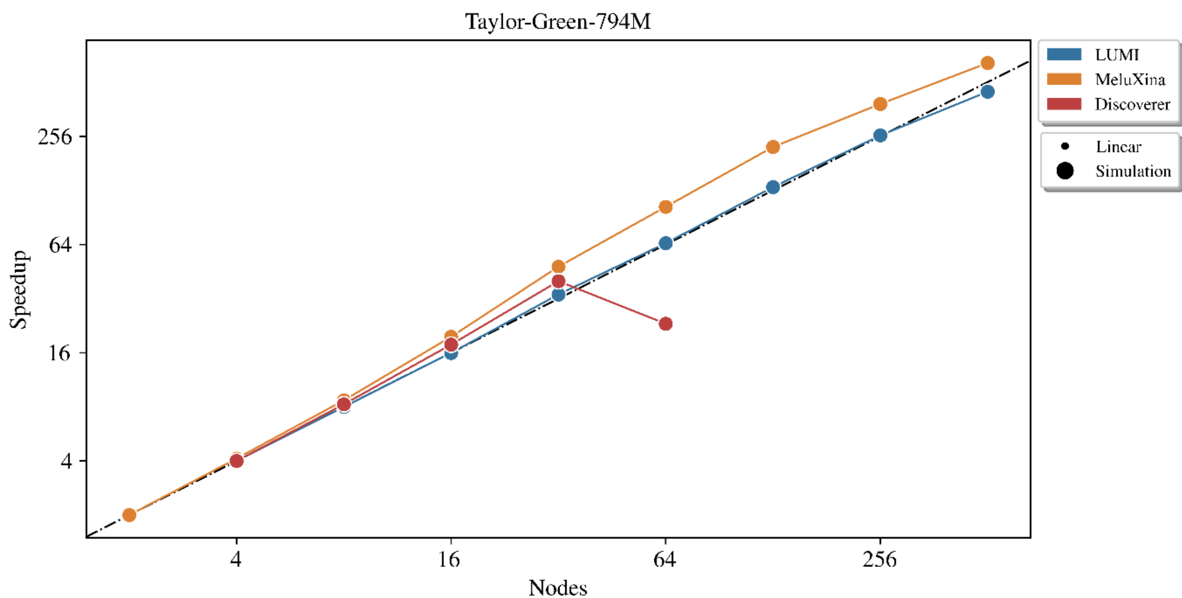


Figure 13 UAP-Xyst per node speedup for the Taylor-Green-794M scenario

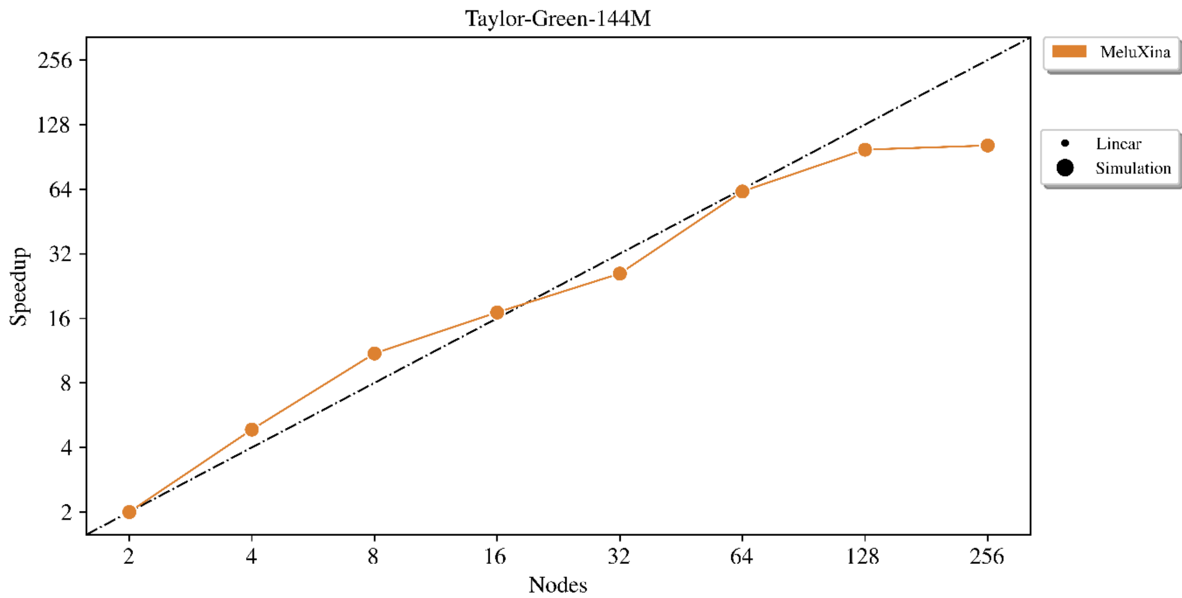


Figure 14 UAP-Xyst per node speedup for the Taylor-Green-144M scenario

4.3 Urban Building (UB)

4.3.1 Pilot description

The Urban Building (UB) pilot aims to simulate the energy behaviour of buildings from a neighbourhood to the city level and beyond. The simulation outcomes encompass each building's thermal comfort, energy consumption, and air quality. We aim to obtain these predictions over a span ranging from one month to an entire year, reflecting a realistic environment by considering factors like weather, occupancy, and surrounding vegetation.

UB examines several building and district models with varying degrees of accuracy. We refer to this differentiation as the Level of Detail (LOD), and the current classification is as follows:

- LOD-0: Buildings are represented as oriented bounding boxes.
- LOD-1: Buildings are depicted as (multi-)polygonal extrusions, optionally including roof shapes.
- LOD-2: Buildings are detailed from an Industry Foundation Classes (IFC) [19] description encompassing many intricate details.

The pilot is developed within the Ktirio-UB framework [20]. Its main components that enable the execution of the UB simulation workflow are the following:

- *GIS data generation*: A geographic area containing buildings, e.g. a district or a city, provides the input data required by the city energy simulation. The implemented solution employs open databases on the web, such as

OpenStreetMap [19], and allows the generation of building entities (*terrain, buildings, vegetation,* etc.) through JSON files and mesh files (*LOD-0* and *LOD-1*). It is written in C++ and uses multiple threads.

- *Building Model definition*: UB uses the Modelica language [22] to model physical systems, which are translated to C++ applications using the Functional Mock-up Interface (FMI) [23].
- *City energy simulator*: A C++ library designed to compute the solution of a city energy model parametrised by GIS data, LOD, building models and scenarios (time period based on which weather forecast and solar direction are provided). It is based on the Feel++ library [24] and produces several output values using various formats. It is currently parallelised based on a distributed approach using MPI and leverages a data partitioner to allocate data to each computing core. Support for multithreading is currently in progress.

4.3.2 Benchmarking

Systems & Environment

UB has been benchmarked on three EuroHPC JU systems: Discoverer, Karolina and MeluXina. To move from one HPC system to another without requiring installations of additional packages to satisfy its dependencies, the pilot leverages containers through Apptainer. The containers ensure the UB programming environment (including backend packages) is reusable at any time and independent of the underlying machine, guaranteeing the reproducibility of results. Consequently, across the different EuroHPC systems, only the hardware layer (and tightly coupled libraries like MPI) varies, and pilot deployment is done by updating a SIF image. The programming and runtime environments used in each system are detailed in Table 13.

Table 13 Programming & runtime environment for UB benchmarks

	Discoverer	Karolina	MeluXina
Compiler (container)	Clang 14	Clang 14	Clang 14
Compiler (MPI)	GCC 12.3.0	GCC 12.2.0	GCC 12.3.0
Parallel framework	Open MPI 4.1.5	Open MPI 4.1.4	OpenMPI 4.1.6
Libraries			
Apptainer	1.2.4	1.1.5	1.2.4
Python	3.10.4	3.10.4	3.9.7

Benchmarking configuration

For the benchmarking activities reported in this deliverable, UB has selected two scenarios that are presented in Table 14. In order to get an insight into how the size of the city area under study affects the performance, benchmarking has been performed using two different area sizes.

Table 14 Details of Ktirio-UB benchmarking scenarios

Scenario	Area Location	Period	Building Model	LOD	Area size
0-M1	Square centred in Strasbourg city	July	Radiative & convective heat transfer	LOD-0	2km square side (~6K buildings)
0-M2					4km square side (~ 17K buildings)

Results & Analysis

To study the scalability of Ktirio-UB, we deploy the full Ktirio-UB pipeline using 1-32 nodes and 128 processes per node. Figure 15 and Figure 16 depict the speedup achieved on Discoverer, Karoling and MeluXina for the 0-M1 and 0-M2 scenarios respectively. In both figures, we present results for the total execution (*end-to-end*) of the pipeline as well as the simulating component (*simulation*).

The simulation part of the pipeline scales almost linearly, which is expected as in the model currently used for the simulation the buildings are not coupled together. On the other hand, the total execution of the pilot's pipeline does not scale; as more nodes are employed the performance degrades.

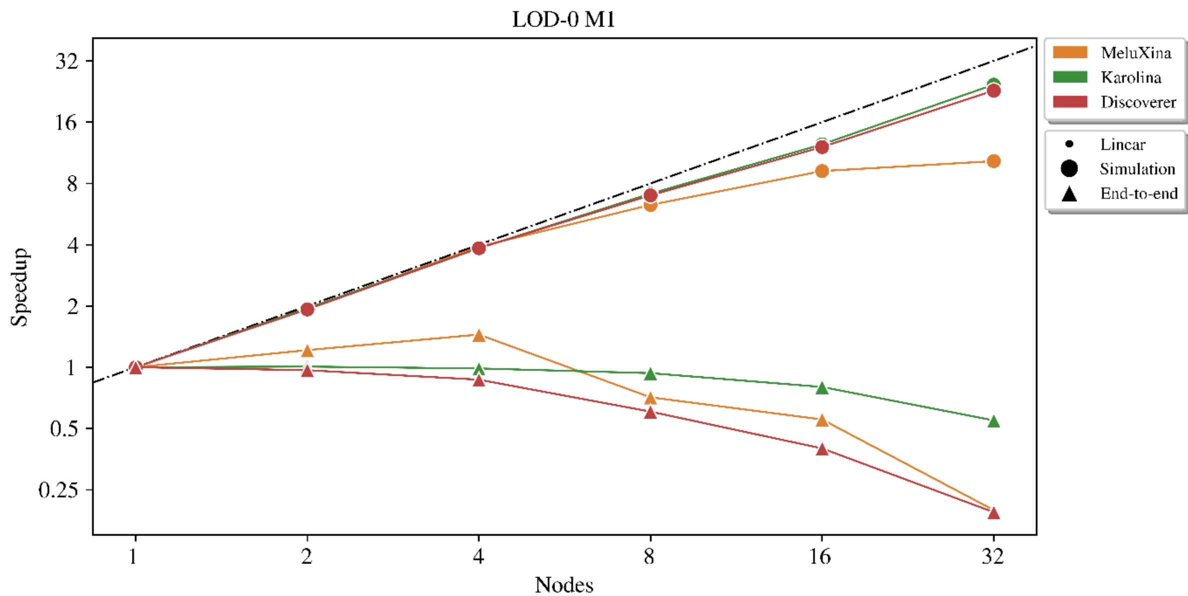


Figure 15 Ktirio-UB per node speedup for the 0-M1 scenario

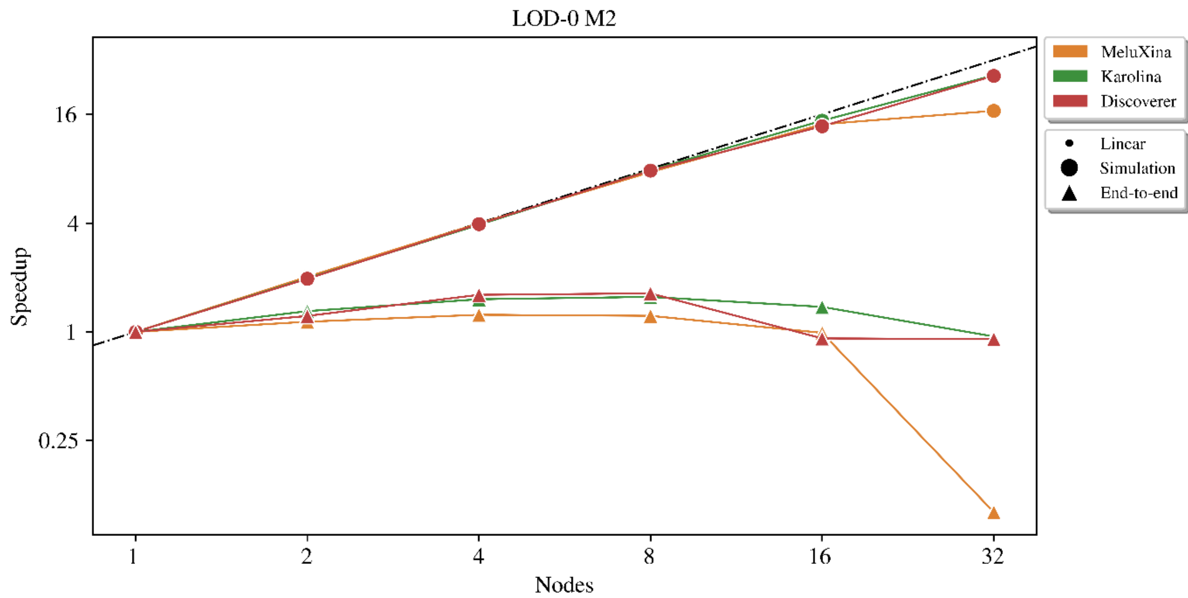


Figure 16 Ktirio-UB per node speedup for the 0-M2 scenario

To better understand what causes this degradation, we measure the computing times of the different stages of the pipeline and present the execution breakdown in Figure 17 and Figure 18 for scenarios 0-M1 and 0-M2 respectively. Both figures report the portion of the total execution taken by:

- Pre-processing (*Pre-proc*): The time elapsed in initialization before entering the time loop of the simulation
- Simulation (*Simulation*): The cumulative time spent calculating the new solution at each time step
- Post-processing (*Post-proc*): The cumulative time spent for exporting results, i.e., generating files containing the output of the UB model.

Pre-processing does not scale. However, it occupies only a small part of the total execution, and thus it is not performance-critical. On the other hand, as more nodes are employed and the time spent in the actual simulation is decreased, the post-processing stage dominates the execution. It becomes the main bottleneck, causing the previously observed performance degradation.

This behaviour is caused by the multiple files being written in parallel on the shared file system. More specifically, most of the writing time is spent in opening and closing files in parallel. We are investigating potential solutions, such as asynchronous writes, data caching, etc. Finally, as the project progresses, we expect the urban building models used in the simulation to become more complex, leading to an increase in the time occupied by the simulation part and, hence, to a reduction of the impact of post-processing on the total execution time of the pilot.

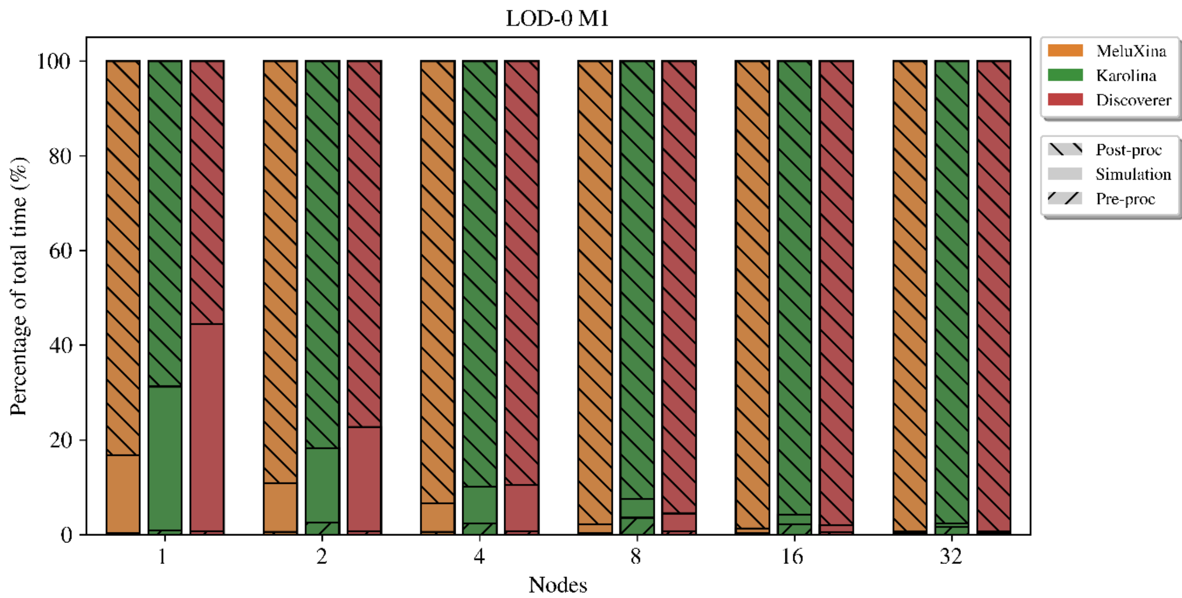


Figure 17 Ktirio-UB execution breakdown for the 0-M1 scenario

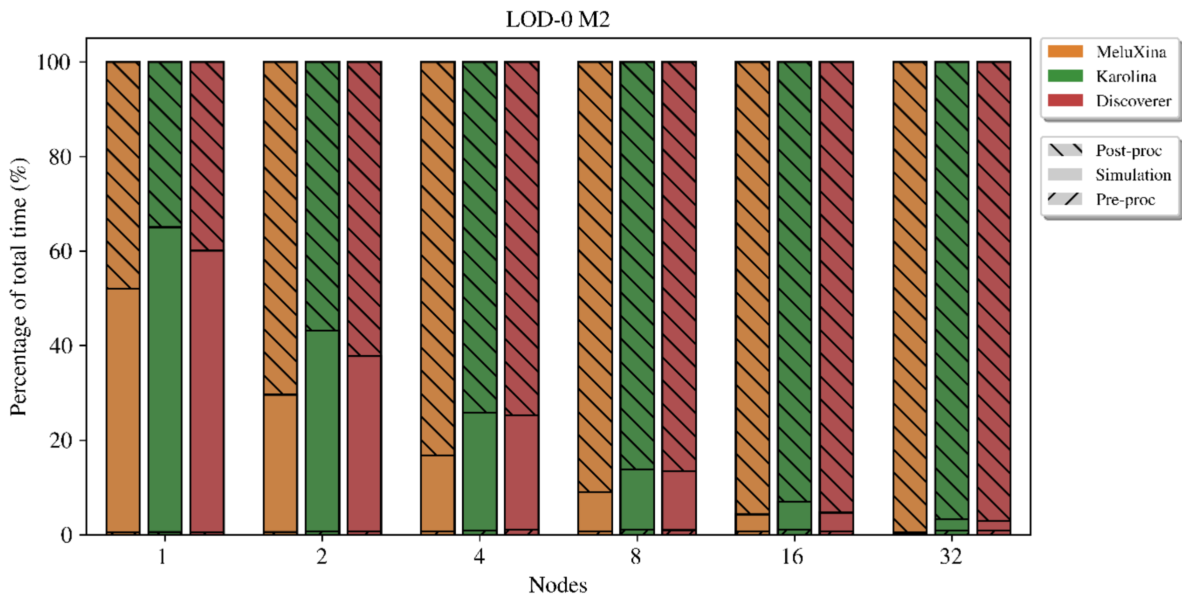


Figure 18 Ktirio-UB execution breakdown for the 0-M2 scenario

4.4 Wildfires (WF)

4.4.1 Pilot description

The WF pilot operates at two levels: the *landscape scale* and the *local urban scale*. In the former, WF simulates large forest fires under various meteorological configurations, with particular attention to the variables that have the most significant influence on the development of fire spread, such as wind speed and direction, as well as the interactions between the lower atmosphere and the fire. In the latter, WF aims

to incorporate in the outputs of the landscape scale level simulations of combustion results from a wildfire affecting and progressing within an urbanised area. It includes the generation and dispersion of smoke plumes using OpenFOAM and fireFOAM [25], employs a much finer spatial and temporal resolution than the landscape scale using sub-meter spatial resolution so that the working area is smaller, and uses as boundary conditions the outputs of the landscape scale.

As the local urban scale is planned to be developed and integrated during the next period of the project, the focus of the benchmarking activities presented in this deliverable is the landscape scale. For this, to capture the dynamics of the atmosphere and its influence and interaction with the spread of forest fires, WRF is combined with two additional modules, SFIRE [26] for fire modelling and CHEM [27] for chemistry. This way, the numerical model can simulate the emission, transport, and deposition of air pollutants resulting from wildfires based on actual atmospheric conditions.

The WF workflow currently comprises two main parts:

- Pre-processing: The WRF Pre-processing Systems (WPS) prepares the input for real-data simulations. It consists of three separate stages: first, the definition of model domains and the interpolation of static geographical data to the grids (*stage 1 – geogrid.exe*); second, the extraction of meteorological fields from GRIB-formatted files [28] (*stage 2 – ungird.exe*), and finally the horizontal interpolation of the extracted meteorological fields in the second stage to the model grids defined in the first stage (*stage 3 – metgrid.exe)*. Each stage is executed only once for each run of the pilot.
- Simulation: The simulation component consists of three stages as well: first, the vertical interpolation of 3D meteorological fields and sub-surface soil data using the 2D output of the pre-processing part and the creation of the boundary and initial conditions that are fed into the next stages (*stage 1 – real.exe*); second, the execution of the WRF model (*stage 2 – wrf.exe*), and third, the dynamical downscaling using ndown (*stage 3 – ndown.exe*). In the latter stage, ndown is used for one-way nesting and obtains the initial and lateral boundary conditions for the fine-resolution domain from the coarse-resolution domain with input from higher resolution terrestrial fields (e.g., terrain, land use, etc.) and masked surface fields, such as soil temperature and moisture. Stage 1 is executed only once for each pilot run, while stages 2 and 3 can be executed multiple times.

4.4.2 Benchmarking

Systems & Environment

WF has been benchmarked on the EuroHPC JU Vega system. The programming and runtime environments used are detailed in Table 15.

Table 15 Programming & runtime environment for WF benchmarks

	Vega
Compiler	GCC-12.2.0
Parallel framework	Open MPI v4.1.4
Libraries	
zlib	1.2.12
jasper	2.0.33
libpng	1.6.37
HDF5	1.14
NetCDF-C	4.9.2
WRF-SFIRE	4.4
WPS	4.4

Benchmarking configuration

The primary benchmarking constraint of WF is that, depending on the spatial decomposition, WRF imposes boundaries to the number of processes that can be employed for the simulation. More specifically, the number of processes is chosen taking into account their decomposition in relation to the size of the domains.

For processing, the domains are divided into tiles, the total number of which depends on the total number of processors used (e.g. 1 tile per processor). Each tile has a minimum of 5 rows/columns on each side (called "*halo*" regions), which pass information from each cell/processor to the neighbouring tile. Additionally, the entire tile should not only comprise halo regions, as there must be some space left for computation in the middle of each tile; otherwise, the model will crash, or its output will be unrealistic. To avoid this, the model divides the total number of grid spaces in the west-east direction (e_we) by the number of tiles in the x-direction and tests that the result is greater than 10. The same constraint must be true also for the south-north direction (e_sn).

In order to get an insight into the performance of WF, benchmarking was performed using two scenarios with different resolutions and different model pipelines. In terms of simulation configuration, the two scenarios are defined in Table 16. For the small scenario, all stages of the simulation are executed exactly only once. In contrast, for the 2k_test scenario the second and third stages of the simulation are executed multiple times. More specifically:

- WRF is run for D01 and D02.
- ndown is used to remap the output of D02 to input for D03.
- WRF is run for D0.
- ndown is used to remap the output of D03 to input for D04.
- WRF model is run for D04.

In both scenarios, the first 3 domains simulate atmospheric processes, while in the fourth domain SFIRE is activated to simulate the interaction between the fire and the atmosphere.

Table 16 Details of WF benchmarking scenarios

Scenario	D01	D02	D03	D04	Number of points per domain	
					e_we	e_sn
small	9km	3km	1km	333.333m	D01 = 188 D02 = 202 D03 = 349 D04 = 277	D01 = 167 D02 = 172 D03 = 310 D04 = 286
2k_test				200m	D01 = 379 D02 = 520 D03 = 670 D04 = 636	D01 = 331 D02 = 373 D03 = 589 D04 = 576

Results & Analysis

To study the scalability of WF, we deploy its pipeline using 128 processes per node. Due to the boundaries imposed by WRF, the small scenario can be deployed from 1 up to 6 nodes, i.e., using 128-768 processes, and the 2k_test scenarion from 2 up to 16 nodes, i.e., using 256-2048 processes. Figure 19 and Figure 20 depict the speedup achieved in Vega for the two scenarios for both the total execution (*end-to-end*) of the pilot as well as the WRF part(s) (*simulation*).

For the small scenario, the simulation scales almost linearly for up to 4 nodes and flattens out for 6 nodes due to limited parallelism in the decomposition. However, as the pre-processing is fixed, the end-to-end speedup is already limited at 4 nodes.

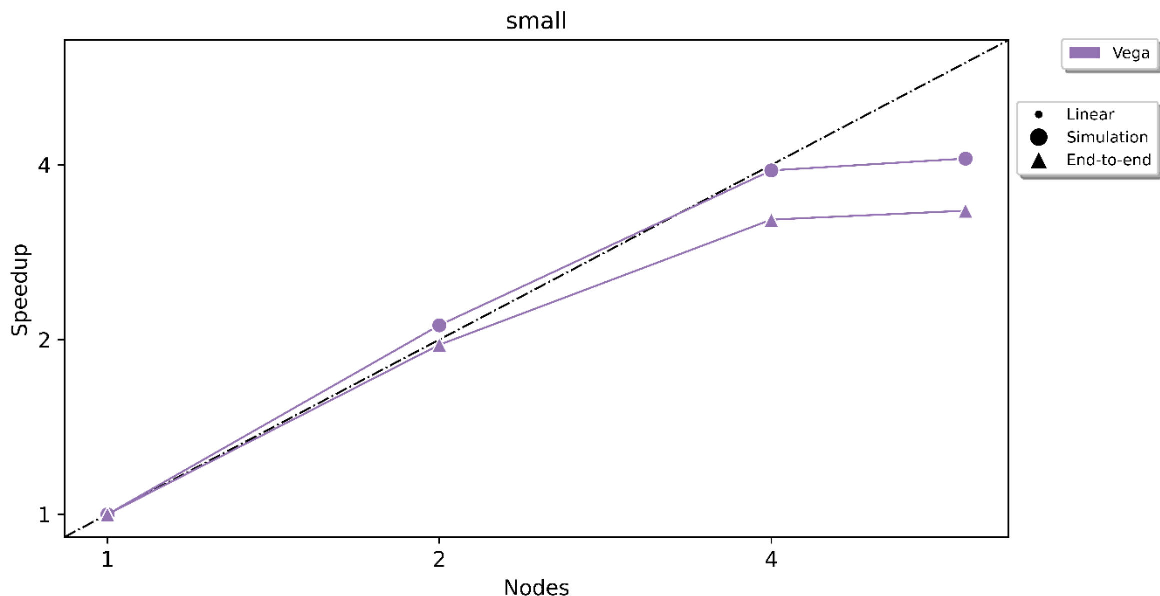


Figure 19 WF per node speedup for the small scenario

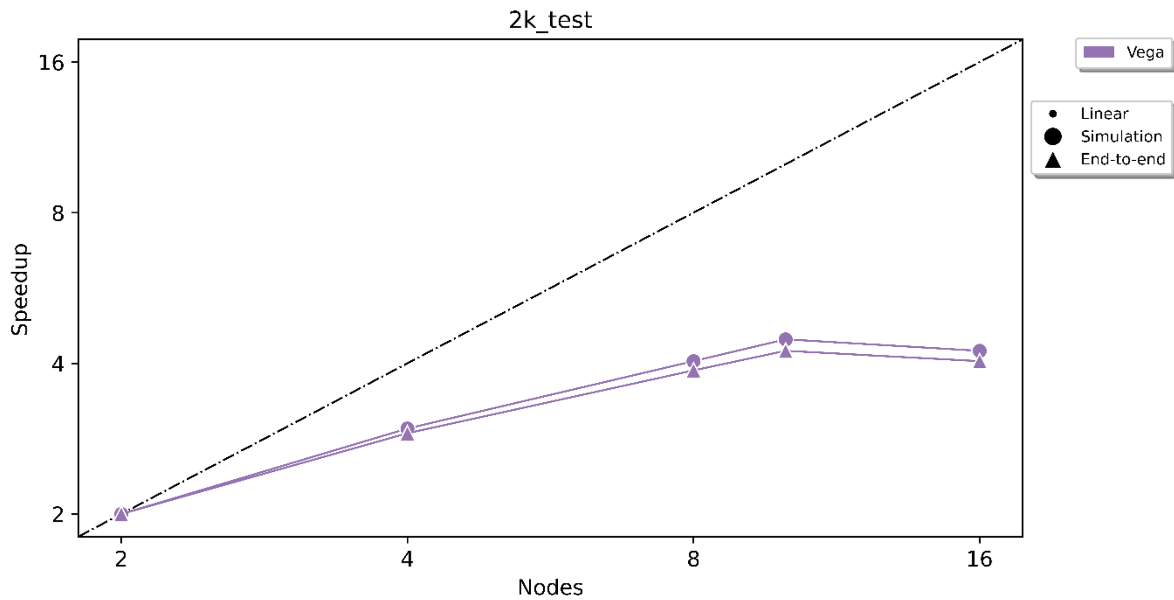


Figure 20 WF per node speedup for the 2k_test using ndown for dynamic downscaling

On the other hand, the simulation cannot scale linearly for the 2k_test scenario. In this case, the complex pipeline comprises multiple WRF invocations with subsequent storing and loading data to and from the disk, limiting the scalability and achieving a maximum 5x speedup for 10 nodes, with the performance degrading for a larger number of nodes. At the same time, as the simulation is much more complex, pre-processing becomes less significant in this case, and the total execution speedup follows closely the simulation speedup.

To confirm the observations regarding pre-processing and its impact on the total execution speedup, we present the execution breakdown in Figure 21 and Figure 22 for the small and 2k_test scenarios, respectively.

In both scenarios, the pre-processing stage occupies a larger part of the execution as the number of employed nodes increases. This is because the computational time decreases when running on more nodes, while pre-processing is always performed on one node. However, as it is evident when comparing the small and 2k_test scenarios, pre-processing becomes less significant as the simulation becomes more complex and computationally intensive. Hence, we expect that pre-processing will not be a bottleneck as the WF pilot matures and more complex scenarios are simulated. On the other hand, as discussed before for Figure 20, I/O between multiple invocations of the WRF model affects the performance and must be optimised to achieve better scalability for complex, multi-staged pipelines.

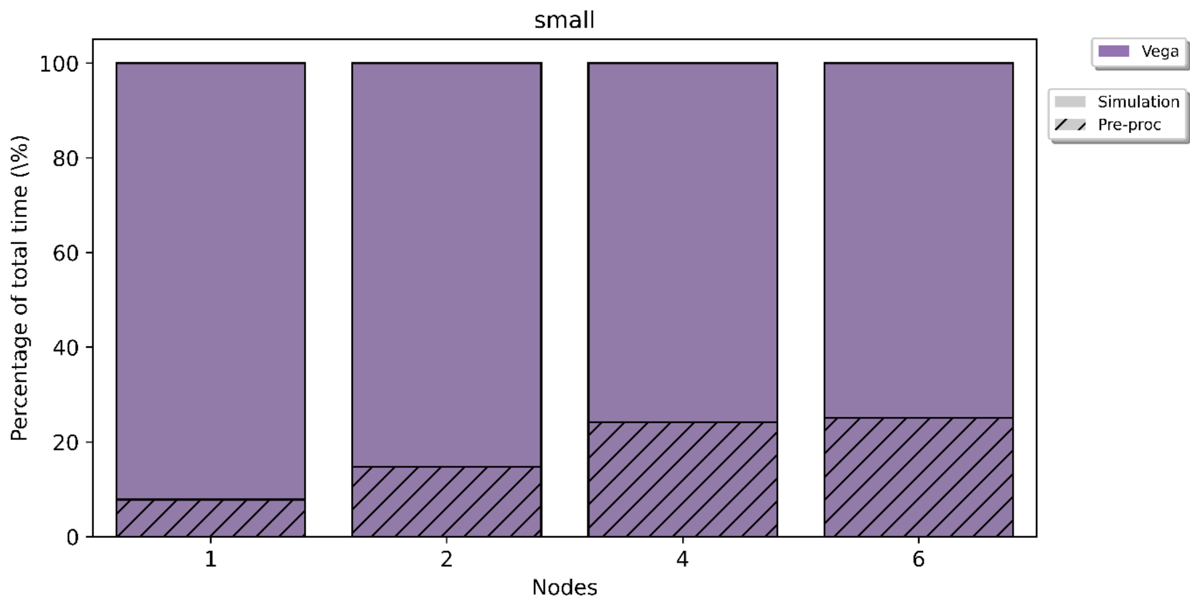


Figure 21 WF execution breakdown for the small scenario

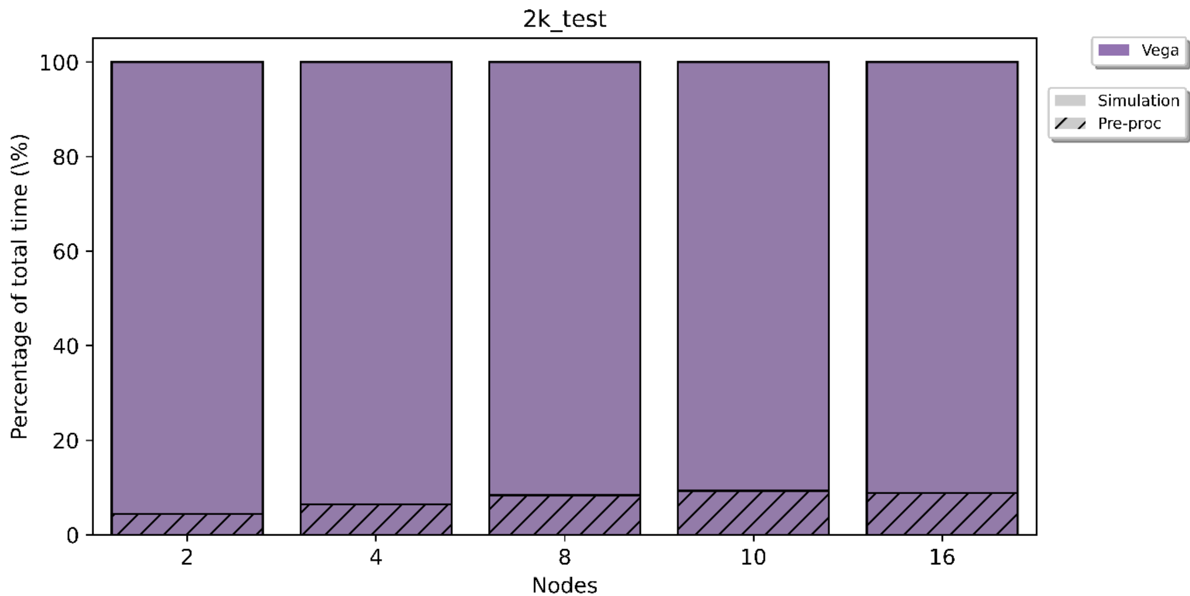


Figure 22 WF execution breakdown for the 2k_test scenario

4.5 Summary & next steps

During its first year, HiDALGO2 has successfully kick-started the benchmarking of its pilots on the HiDALGO2 HPC infrastructure, i.e. PSNC's Altair and the EuroHPC JU systems. More specifically:

- **RES** has been benchmarked on Altair and LUMI. In both systems the code currently scales linearly up to 10 nodes, and on LUMI it is estimated that linear speedup could be achieved for up to 32 nodes as the simulation becomes more compute intensive for denser data inputs. Besides optimising the computational

part of the pilot and improving its scalability, it has been deduced that the I/O of the post-processing part needs to be parallelised as well.

- The three different codes of **UAP** have been benchmarked on five EuroHPC JU systems. **UAP-OpenFOAM** has been executed on Discoverer, LUMI and MeluXina, and for the larger meshes it currently scales up to 64 nodes. The multi-GPU version of **UAP-RedSIM** has been benchmarked on Karolina and Vega, and it currently scales up to 8 GPUs as the size of its input mesh gets larger. The multi-CPU version of **UAP-RedSIM** has been executed on LUMI and it currently exhibits sublinear speedups for larger inputs. Finally, **UAP-Xyst** has been benchmarked on Discoverer, LUMI and MeluXina, and it currently scales linearly up to 512 nodes.
- **UB** has been benchmarked on Discoverer, Karolina and MeluXina. In all three systems, the code currently scales linearly up to 32 nodes. However, its total execution does not scale, due to the post-processing stage which becomes the main bottleneck by writing in parallel on multiple files.
- **WF** has been benchmarked on Vega. It currently scales linearly up to 4 nodes only for the smaller, simpler benchmarking scenario and does not scale for the larger, more complex scenario that involves more pipeline stages leading to multiple invocations of the WRF model.

HiDALGO2 has identified multiple KPIs related to benchmarking and optimisation activities and is committed to achieve ambitious targets. Table 17 presents the current status of these KPIs based on the benchmarking activities that have been reported in Section 4.

Table 17 Status of benchmarking and optimisation related KPIs in M12

KPI	Target	M12	Comments
Applications with a scalability of 50k cores in a single run	≥ 3	1	Scalable runs with more than 50k cores (512 nodes with 128 cores each) have already been achieved by UAP-Xyst.
Applications with a scalability of 200k cores in a single run	≥ 1	0	We expect to reach our target by around the end of the project's third year.
Applications with a scalability of 80k cores in ensemble runs	≥ 3	0	No work has been performed yet for ensemble runs.
Applications with parallel efficiency improved by 30%	≥ 3	0	Activities in the first year of the project have focused on benchmarking, i.e., on setting the baseline for all pilots. Optimisation activities will essentially start in the second year of the project, and we expect to meet our KPI target around the end of the third or the start of the fourth year of the project.

The targets set for the scalability-related KPIs have not been met yet. This is expected, though, as in the first year of the project, we focused on deploying the pilots on the EuroHPC JU systems and benchmarking them in order to set their baseline. As optimisation activities will start in the project's second year, we expect to improve these KPIs in the forthcoming periods.

More specifically, during the project's second year, the benchmarking of the pilots will continue as they advance and are optimised. The main next steps can be summarised as follows:

- ***Complete integration with ReFrame***: All pilots must finalise their integration with ReFrame. RES is considering replacing their internal, in-house scheduler with ReFrame. UAP needs to finalise the integration of all UAP implementations with ReFrame on all EuroHPC JU systems, and for WF I/O issues need to be solved, especially for complex pipelines with multiple resolution enhancing steps.
- ***Extend deployment on EuroHPC JU systems***: As Table 1 in Section 3.2 highlights, pilots currently have access to a subset of the available EuroHPC JU systems and intend to acquire access to more.
- ***Profiling and bottleneck analysis***: The initial benchmarking reported in this deliverable has already identified a few performance issues. It is essential to focus on these, employing profiling and tracing to detect and analyse the bottlenecks that are responsible for them. Towards this end, we plan to rely on tools that are readily available or can be easily installed on all EuroHPC JU systems. More specifically, we plan to collect profiles and traces using Vampir [29] and Score-P [30], that were also used successfully in the HiDALGO project.

5. HiDALGO2 Co-Design Activities

This section tackles the issues related to the scalability of solutions and the optimal adaptation of the software to the infrastructure (*co-design*) by using the appropriate benchmarking methodology and algorithmic optimisation methods. One of the main goals is to determine general strategies for co-designing the HiDALGO2 pilots.

5.1 HiDALGO2 strategy

HiDALGO2 aims to explore current and next-generation HPC architectures to identify promising technologies that can impact the project's goals, by enabling efficient, highly performant systems to simulate complex structures with much higher accuracy, performance and energy efficiency. Our strategy is not to limit our investigation to the different computing systems offered by the EuroHPC JU centres, but to leverage our close cooperation with IT vendors such as Intel, AMD, and NVIDIA and acquire access to systems equipped with a various processors and accelerators.

In particular, we plan to explore HPC solutions based on cutting-edge x86 processors from vendors, such as Intel and AMD, as a starting point for the co-design activity. Specifically, a series of top-of-the-line AMD EPYC CPUs with a wide range of CPU products based on Rome, Milan, Milan-X, Genoa, Genoa-X, and Bergamo architectures are under investigation. In the successive periods of the project, we plan to extend our investigation to the newest ARM-based product lines for HPC solutions. At the same time, we focus on novel GPU accelerators from NVIDIA and AMD.

One of the main targets of co-design activities is utilising current and future computing systems better and more efficiently. To that end, our activities will focus on the required adaptations of the software in order to leverage the capabilities of the underlying hardware platforms. In that context, we distinguish different targets of the co-design process, including but not limited to optimisation aspects for intra- and inter-node communication, as well as data read from or written to HPC storage systems.

In general, the co-design activities will be carried out for both CPU and GPU-based applications. Our main goal is to understand the correlation between the application and the underlying platform and to determine appropriate synergies between the hardware and a given parallel code. We will analyse trade-offs between performance, memory, and energy consumption to determine the relation between application requirements and system capabilities in terms of balance between, e.g., memory capacity, bandwidth and computing resources. To achieve that, we will follow a robust benchmarking methodology, assessing the performance of applications and acquiring reliable activity profiles. Finally, once performance bottlenecks and associated trade-offs have been established, we will apply a wide range of known optimisation techniques to increase execution efficiency.

6. Conclusions

This deliverable has defined the HiDALGO2 benchmarking methodology, drawing lessons from the HiDALGO project. This generic, systematic methodology for collecting benchmarking information and storing benchmarking results is centred around the usage of ReFrame, which, besides managing the deployments and executions on the various HPC centres, will help ensure the reproducibility and validation of gathered results.

During the first year of the HiDALGO2 project, we focused primarily on deploying the HiDALGO2 pilots on the EuroHPC JU systems. The process of acquiring resources was cumbersome, but at the end of the first reporting period, all pilots have been deployed and benchmarked on at least one EuroHPC JU system. Deployment onto more systems will continue in the second year of the project as well, targeting especially the pre-exascale machines, i.e., Leonardo, that became available recently, and Marenostrum 5, which is expected to become available in 2024.

Initial benchmarking has revealed that one of the HiDALGO2 codes (UAP-Xyst) scales linearly up to 512 nodes, while the others currently scale up to a few dozen nodes. A few potential bottlenecks related to I/O have been identified and will serve as the first target for the optimisation activities, which will start in the project's second year. Finally, we have defined the HiDALGO2 strategy for the project's co-design activities, which will be further elaborated and presented together with initial findings in deliverable D3.4 "*Innovative HPC Technologies and Benchmarking (M15)*".

7. References

[1] (2023), *Homepage of HiDALGO CoE*, https://hidalgo-project.eu/ [retrieved: 2023-12-31]

[2] (2023), *Homepage of ReFrame*, https://reframe-hpc.readthedocs.io/en/stable/ [retrieved: 2023-12-28]

[3] (2023), *ReFrame test suite of CSCS*¸ https://github.com/reframe-hpc/cscs-reframe-tests [retrieved: 2023-12-31]

[4] (2023), *EPCC ReFrame repository*, https://github.com/reframe-hpc/epcc-reframe-tests [retrieved: 2023-12-31]

[5] (2023), *ReFrame tests for HPC2N and C3SE*, https://github.com/reframe-hpc/hpc2n-tests [retrieved: 2023-12-31]

[6] (2023), *A portable test suite for software installations*, https://github.com/EESSI/test-suite [retrieved: 2023-12-31]

[7] (2023), *ExCALIBUR tests,* https://github.com/ukri-excalibur/excalibur-tests [retrieved: 2023-12-31]

[8] (2023), *Homepage of OpenFOAM*, https://www.openfoam.com/ [retrieved: 2023-12-28]

[9] (2023), *Homepage of Apptainer*, https://apptainer.org/ [retrieved: 2023-12-28]

[10] (2023), *WRF wiki page*, https://en.wikipedia.org/wiki/Weather_Research_and_Forecasting_Model [retrieved: 2023-12-28]

[11] M. Z. Ziemiański, M.J. Kurowski, Z.P. Piotrowski, B. Rosa, O. Fuhrer, “Toward very high horizontal resolution NWP over the Alps: Influence of increasing model resolution on the flow pattern”, Acta Geophysica, 59, 2011, 1205 – 1235

[12] Z. Piotrowski, A. Wyszogrodzki, P. Smolarkiewicz, Piotr, “Towards Petascale Simulation of Atmospheric Circulations with Soundproof Equations”, Acta Geophysica, 59, 2011, 1294-1311.

[13] (2023), *Homepage of Global Forecasting System*, https://www.ncei.noaa.gov/products/weather-climate-models/global-forecast [retrieved: 2023-12-31]

[14] L. Környei, Z. Horváth, A. Ruopp, A. Kovacs, and B. Liszkai. “Multi-scale Modelling of Urban Air Pollution with Coupled Weather Forecast and Traffic Simulation on HPC Architecture”. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Companion (HPC Asia 2021), 2021

[15] (2023), *SimpleFoam documentation,* https://doc.openfoam.com/2212/tools/processing/solvers/rtm/incompressible/simpleFoam/ [retrieved: 2023-12-31]

[16] (2023), *PimpleFoam documentation*, https://doc.openfoam.com/2212/tools/processing/solvers/rtm/incompressible/pimpleFoam/ [retrieved: 2023-12-31]

[17] J. Bakosi, "Open-source complex-geometry 3D fluid dynamics for applications with unpredictable heterogeneous dynamic high-performance-computing loads", Computer Methods in Applied Mechanics and Engineering, Volume 418, Part B, 2024.

[18] (2023), *Homepage of Charm++*, https://charmplusplus.org/ [retrieved: 2023-12-28]

[19] (2024), *IFC: Industry Foundation Classes standards*, https://technical.buildingsmart.org/standards/ifc/ [retrieved: 2024-07-01]

[20] (2024), *Ktirio Urban Building Framework*, https://feelpp.github.io/ktirio-urban-building/ktirio-urban-building/index.html [retrieved: 2024-07-01]

[21] (2023), *OpenStreetMap Wiki*, https://wiki.openstreetmap.org/wiki/Main_Page [retrieved: 2023-12-31]

[22] (2023), *Homepage of Modelica association*, https://modelica.org/ [retrieved: 2023-12-28]

[23] (2023), *Homepage of FMI standard*, https://fmi-standard.org/ [retrieved: 2023-12-28]

[24] (2023), *Github page of feel++*, https://github.com/feelpp/feelpp [retrieved: 2023-12-28]

[25] (2023), *FireFoam wiki page*, https://openfoamwiki.net/index.php/FireFoam [retrieved: 2023-12-28]

[26] (2023), *Open wildland fire modelling Wiki,* https://wiki.openwfm.org/wiki/WRF-SFIRE [retrieved: 2023-12-31]

[27] (2023), *Official WRF-Chem web page*, https://ruc.noaa.gov/wrf/wrf-chem/ [retrieved: 2023-12-31]

[28] (2023), *GRIB definition in Wikipedia*, https://en.wikipedia.org/wiki/GRIB [retrieved: 2023-12-31]

[29] (2023), *Vampir wiki page*, https://hpc-wiki.info/hpc/Vampir [retrieved: 2023-12-28]

[30] (2023), *Score-P wiki page*, https://hpc-wiki.info/hpc/Score-P [retrieved:2023-12-28]

Annexes – HiDALGO2 ReFrame scripts examples

Annex I

cluster_config_file.py

```
# Directory path of reframe_feelpp folder
start_path = '/home/users/xexpanas/atheodor/cemosis'

# Directory where ReFrame will make temporary copies of the sourcedir for
each test.
temp_directory = '/home/users/xexpanas/atheodor/cemosis/stage'

# Directory where ReFrame will copy its output files and the files
contained in output_files variable.
output_directory = start_path+'/output'

#############################
# Site Configuration Object #
#############################

max_cores_per_node=48
import socket

hostname = socket.gethostname()

site_configuration = {
    'systems': [
        {
            'name': 'eagle',
            'descr': 'eagle',
            'hostnames': ['eagle'],
            'modules_system': 'tmod32',
            'prefix': start_path,
            'stagedir': temp_directory,
            'outputdir': output_directory,
            'partitions': [
                {
                    'name': 'altair',
                    'scheduler': 'slurm',
                    'launcher': 'custom_mpiexec',
                    'access': ['--partition=altair'],
                    'environs': ['cemosis'],
                    'container_platforms':[
                        {
                            'type': 'Apptainer',
                        }
                    ],
                    'max_jobs': 8,
                },
            ]
        },
        {
            'name': 'discoverer',
            'descr': 'discoverer',
            'hostnames': ['login\d+.discoverer.sofiatech.bg','cn*'],
            'modules_system': 'tmod4',
```

```
            'prefix': start_path,
            'stagedir': temp_directory,
            'outputdir': output_directory,
            'partitions': [
                {
                    'name': 'cn',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'access': ['--partition=cn --account=ehpc-dev-2023d08-
025 --qos=ehpc-dev-2023d08-025'],
                    'environs': ['env_discoverer'],
                    'container_platforms':[
                        {
                            'type': 'Singularity'# 'Apptainer',
                        }
                    ],
                    'max_jobs': 8
                }
            ]
        },
        {
            'name': 'karolina',
            'descr': 'karolina',
            'hostnames':
['login\d+.karolina.it4i.cz','cn\d+.karolina.it4i.cz'],
            'modules_system': 'lmod',
            'prefix': start_path,
            'stagedir': temp_directory,
            'outputdir': output_directory,
            'partitions': [
                {
                    'name': 'qcpu',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
                    'access': ['--partition=qcpu --account DD-23-129'],
                    'environs': ['env_karolina'],
                    'container_platforms':[
                        {
                            'type': 'Singularity'# 'Apptainer',
                        }
                    ],
                    'max_jobs': 8
                }
            ]
        },
        {
            'name': 'meluxina',
            'descr': 'meluxina',
            'hostnames': [f'{hostname}'],
            'modules_system': 'lmod',
            'prefix': start_path,
            'stagedir': temp_directory,
            'outputdir': output_directory,
            'partitions': [
                {
                    'name': 'cpu',
                    'scheduler': 'slurm',
                    'launcher': 'srun',
```

```
                    'access': ['--partition=cpu --account p200229 --
time=02:00:00 --qos=default'],
                    'environs': ['env_meluxina'],
                    'container_platforms':[
                        {
                            'type': 'Apptainer',
                        }
                    ],
                    'max_jobs': 8
                }
            ]
        }

    ],
    'environments': [
        {
            'name': 'cemosis',
            'modules': ['openmpi/4.0.0_gcc620'],
            'cc': 'gcc',
            'cxx': 'g++',
            'target_systems': ['eagle:altair']
        },
        {
            'name': 'env_discoverer',
            'modules': ['openmpi/4/gcc/latest'],
            'target_systems': ['discoverer:cn']
        },
        {
            'name': 'env_karolina',
            'modules': ['OpenMPI/4.1.4-GCC-12.2.0','apptainer'],
            'target_systems': ['karolina:qcpu']
        },
        {
            'name': 'env_meluxina',
            'modules': [' env/staging/2023.1', 'Apptainer/1.2.4-GCCcore-
12.3.0', 'OpenMPI/4.1.5-GCC-12.3.0'],
            'target_systems': ['meluxina:cpu']
        }
    ]
}
```

Annex II

benchmarking.py

```
import os

from cluster_config_file import start_path

#############################
# System specific variables #
#############################

platform_used = ['Apptainer']
system=['eagle:altair']
prog_environment=['cemosis']

# Commands to run before execution
prerun_commands=[]

# Different input files. Must be list of lists.
input_files=[[]]

#  Source  directory  containing  all  files  needed  for  execution.  Its  the
directory
# that will be copied.
source_directory=None

# Container commands
sif_path                                                                     =
'/home/users/xexpanas/atheodor/cemosis/singularity/feelpp_v0.111.0-
preview.7-focal.sif'
apptainer_home_general                                                       =
'/home/users/xexpanas/atheodor/cemosis/apptainer_home'
select_case = 'Case3'
command_for_container = "feelpp_toolbox_heat \
                        --config-file
/usr/share/feelpp/data/testcases/toolboxes/heat/cases/Building/ThermalBridg
esENISO10211/case3.cfg \
                        --case.discretization=P2 \
                        --heat.scalability-save=1"

# Import slurm variables.
nodes_pre=[1]
tasks_per_node_pre=[1, 2, 4, 8, 12, 24, 36, 48]
exclusive_access_pre=True
# Not required.
tasks_pre=-1 # list of tasks per run.
cores_per_task_pre=-1 # int

# Output files
output_files_pre=[]

######################
## Bitbucket upload ##
######################

scenario_name = 'test_scenario'
```

Annex III

main.py

```
import os
import csv
import json
import reframe as rfm
import reframe.utility.sanity as sn
import reframe.utility.udeps as udeps
import sys, getopt

from datetime import datetime
from reframe_lib import *
from benchmarking import *
from cluster_config_file import *

@rfm.simple_test
class Feelpp(rfm.RunOnlyRegressionTest):
    '''
    Run feelpp using apptainer.
    '''
    platform = platform_used
    valid_systems = system
    valid_prog_environs = prog_environment
    number_of_nodes = parameter(nodes_pre)
    tasks_per_node = parameter(tasks_per_node_pre)
    exclusive_access=exclusive_access_pre

    sourcesdir = source_directory
    keep_files = output_files_pre

    inputs = parameter(input_files)
    input_file=variable(str)

    prerun_cmds = prerun_commands

    @run_after('init')
    def setup_container_platf(self):
      '''
      Setup Apptainer.
      '''
        self.apptainer_home       =       os.path.join(apptainer_home_general,
self.short_name)
        self.descr = f'Run commands inside a container using {self.platform}'
        self.container_platform.image = sif_path
        self.container_platform.command = command_for_container
        self.container_platform.options=[f'--home {self.apptainer_home}']
        self.container_platform.workdir=None

        os.mkdir(self.apptainer_home)
        self.logs_path                                                         =
self.apptainer_home+'/feelppdb/toolboxes/heat/ThermalBridgesENISO10211/'+
select_case

    @run_before('run')
    def set_resources(self):
        '''
```

```
        Sets resource variables for Slurm.
        '''
        if tasks_pre==-1 and cores_per_task_pre==-1:
            self.num_tasks = self.tasks_per_node * self.number_of_nodes
            self.num_cpus_per_task = 1
            self.num_tasks_per_node = self.tasks_per_node
        elif tasks_pre==-1:
            self.num_tasks = self.tasks_per_node * self.number_of_nodes
            self.num_tasks_per_node = self.tasks_per_node
            max_cores = altair_harware_cores_per_node
            if max_cores < self.tasks_per_node * cores_per_task_pre:
                raise BadSlurmVariablesException("Feelpp: Each node has only 
"+str(max_cores)+" cores.")
            else:
                self.num_cpus_per_task = cores_per_task_pre
        elif cores_per_task_pre==-1:
            idx = tasks_pre.index(self.tasks_per_node)
            if tasks_pre[idx] != nodes_pre * tasks_per_node_pre:
                raise  BadSlurmVariablesException("Feelpp:  Tasks  must  be 
equal to nodes * tasks_per_node.")
            self.num_tasks = tasks_pre[idx]
            self.num_cpus_per_task = 1
            self.num_tasks_per_node = self.tasks_per_node
        else:
            idx = tasks_pre.index(self.tasks_per_node)
            if tasks_pre[idx] != nodes_pre * tasks_per_node_pre:
                raise  BadSlurmVariablesException("Feelpp:  Tasks  must  be 
equal to nodes * tasks_per_node.")
            self.num_tasks = tasks_pre[idx]
            self.num_tasks_per_node = self.tasks_per_node
            max_cores = altair_harware_cores_per_node
            if max_cores < self.tasks_per_node * cores_per_task_pre:
                raise BadSlurmVariablesException("Feelpp: Each node has only 
"+str(max_cores)+" cores.")
            else:
                self.num_cpus_per_task = cores_per_task_pre

    @run_before('performance')
    def set_git_data(self):
      '''
      Prepare for git upload.
      '''
        self.upload_data = upload_data
        self.upload_data['system_name'] = str(self.current_system)
        self.upload_data['processes'] = self.num_tasks
        # Timestamp in UTC
        self.upload_data['date']                                          = 
datetime.utcfromtimestamp(int(self.job.completion_time)).strftime('%Y-%m-
%d_%H:%M:%S')
        # Find log file
        self.upload_data['repo_filename']     =     "dt-%s_procs-%d.log"     % 
(self.upload_data['date'], self.upload_data['processes'])
        self.upload_data['logfile_path']  = self.prefix+'/logs/'
        self.upload_data['repo_destination_path']   =   "%s/%s/%s/%s/%s/"   % 
(self.upload_data['local_repo_path'],       self.upload_data['pilot_name'], 
self.current_system.name,                    self.upload_data['file_type'], 
self.upload_data['scenario_name'])
```

```
    #------- HeatConstructor -------#
    @performance_function('')
    def extract_HeatConstructor_names(self, pos=1):
        '''Name of performance variables for HeatConstructor data file.'''

        return    sn.extractsingle(rf'nProc[\s]+([a-zA-z\-]+)[\s]+([a-zA-z\-
]+)[\s]+([a-zA-z\-]+)[\s]+([a-zA-z\-]+)[\s]+([a-zA-z\-]+)[\s]+([a-zA-z\-
]+)[\s]+([a-zA-z\-]+)[\s]+([a-zA-z\-]+)[\s]+',
         self.logs_path+'/heat.scalibility.HeatConstructor.data', pos, str)

    @performance_function('sec')
    def extract_HeatConstructor(self, nProc=48, pos=1):
        '''Performance extraction function for HeatConstructor data file.'''

        return  sn.extractsingle(rf'{nProc}[\s]+([0-9e\-\+\.]+)[\s]+([0-9e\-
\+\.]+)[\s]+([0-9e\-\+\.]+)[\s]+([0-9e\-\+\.]+)[\s]+([0-9e\-
\+\.]+)[\s]+([0-9e\-\+\.]+)[\s]+([0-9e\-\+\.]+)[\s]+([0-9e\-\+\.]+)[\s]+',
         self.logs_path+'/heat.scalibility.HeatConstructor.data',       pos,
float)

    #------- HeatPostProcessing -------#
    @performance_function('')
    def extract_HeatPostProcessing_names(self, pos=1):
        '''Name of performance variables for HeatPostProcessing data file.'''

        return sn.extractsingle(rf'nProc[\s]+([a-zA-z\-]+)',
         self.logs_path+'/heat.scalibility.HeatPostProcessing.data',    pos,
str)

    @performance_function('sec')
    def extract_HeatPostProcessing(self,  nProc=48, pos=1):
        '''Performance  extraction  function  for  HeatPostProcessing  data
file.'''

        return sn.extractsingle(rf'{nProc}[\s]+([0-9e\-\+\.]+)',
         self.logs_path+'/heat.scalibility.HeatPostProcessing.data',    pos,
float)

    #------- HeatSolve -------#
    @performance_function('')
    def extract_HeatSolve_names(self, pos=1):
        '''Name of performance variables for HeatSolve data file.'''

        return      sn.extractsingle(rf'nProc\s+([a-zA-z\-]+)[\s]+([a-zA-z\-
]+)[\s]+([a-zA-z\-]+)[\s]+([a-zA-z\-]+)',
         self.logs_path+'/heat.scalibility.HeatSolve.data', pos, str)

    @performance_function('sec')
    def extract_HeatSolve(self,  nProc=48, pos=1):
        '''Performance extraction function for HeatSolve data file.'''

        return  sn.extractsingle(rf'{nProc}[\s]+([0-9e\-\+\.]+)[\s]+([0-9e\-
\+\.]+)[\s]+([0-9e\-\+\.]+)[\s]+([0-9e\-\+\.]+)',
         self.logs_path+'/heat.scalibility.HeatSolve.data', pos, float)


    @run_before('performance')
```

```
    def set_perf_variables(self):
        '''Build the dictionary with all the performance variables.'''

        self.perf_variables = {}
        # HeatConstructor
        for v in range(1,9,1):

self.perf_variables.update({str(self.extract_HeatConstructor_names(pos=v)):
self.extract_HeatConstructor(nProc=self.num_tasks, pos=v)})
        # HeatPostProcessing
        for v in range(1,2,1):

self.perf_variables.update({str(self.extract_HeatPostProcessing_names(pos=v
)): self.extract_HeatPostProcessing(nProc=self.num_tasks, pos=v)})
        # HeatSolve
        for v in range(2,5,1):

self.perf_variables.update({str(self.extract_HeatSolve_names(pos=v)):
self.extract_HeatSolve(nProc=self.num_tasks, pos=v)})

    @sanity_function
    def assert_done(self):
        '''
        Asserts correct execution for each step of the test.
        '''
        step1     =     sn.assert_found('\\[32m     \[success\]     \\[00m\|
Normal\_Heat\_Flux\_alpha', self.stdout)
        step2     =     sn.assert_found('\\[32m     \[success\]     \\[00m\|
Normal\_Heat\_Flux\_beta', self.stdout)
        step3     =     sn.assert_found('\\[32m     \[success\]     \\[00m\|
Normal\_Heat\_Flux\_gamma', self.stdout)
        step4     =     sn.assert_found('\\[32m     \[success\]     \\[00m\|
Points\_alpha\_min\_field\_temperature', self.stdout)
        step5     =     sn.assert_found('\\[32m     \[success\]     \\[00m\|
Points\_alpha\_max\_field\_temperature', self.stdout)
        step6     =     sn.assert_found('\\[32m     \[success\]     \\[00m\|
Points\_beta\_min\_field\_temperature', self.stdout)
        step7     =     sn.assert_found('\\[32m     \[success\]     \\[00m\|
Points\_beta\_max\_field\_temperature', self.stdout)
        step8     =     sn.assert_found('\\[32m     \[success\]     \\[00m\|
Statistics\_temperature\_alpha\_min', self.stdout)
        step9     =     sn.assert_found('\\[32m     \[success\]     \\[00m\|
Statistics\_temperature\_beta\_min', self.stdout)

        return step1 and step2 and step3 and step4 and step5 and step6 and
step7 and step8 and step9

    @run_before('cleanup')
    def create_log(self):
      ''' Build the final log. '''
        new_names = []
        new_values = []

        with
open(self.current_system.prefix+'/perflogs/'+self.current_system.name+'/'+s
elf.current_partition.name+'/'+self.short_name+'.log', 'r') as fp_r:
            lines = fp_r.readlines()
            names = lines[0].split(',')
```

```
            values = lines[1].split(',')

            display_name = values[2]
            _,number_of_nodes,tasks_per_node,inputs = display_name.split('
')
            number_of_nodes_name,          number_of_nodes_value          =
number_of_nodes.split('=')
            new_names.append(number_of_nodes_name[1:])
            new_values.append(number_of_nodes_value)
            tasks_per_node_name,           tasks_per_node_value           =
tasks_per_node.split('=')
            new_names.append(tasks_per_node_name[1:])
            new_values.append(tasks_per_node_value)


            new_names.append(names[3])
            new_values.append(values[3])
            new_names.append(names[4])
            new_values.append(values[4])
            new_names.append(names[5])
            new_values.append(values[5])
            for i in [8,13,18,23,28,33,38,43,48,53,58,63]:
                new_names.append(names[i]+'_'+values[i+1])
                new_values.append(values[i])

        with
open(self.prefix+'/temp_logs/'+self.upload_data['repo_filename'],  'w')  as
fp_w:
            csv_file = csv.writer(fp_w)
            csv_file.writerow(new_names)
            csv_file.writerows([new_values])

    @run_before('cleanup')
    def upload_to_bitbucket(self):
        upload_to_git(self.upload_data,
self.upload_data['repo_destination_path'],
create_log(start_path+'/reframe_feelpp'))

##############
#   Git    #
##############

def create_log(run_dir):
    csv_files = glob.glob(run_dir+'/temp_logs/'+'*.{}'.format('log'))
    df_csv_append = pd.DataFrame()
    tmstmp = datetime.now().timestamp()
    date     =     datetime.utcfromtimestamp(int(tmstmp)).strftime('%Y-%m-
%d_%H:%M:%S')
    filename = "dt-%s.log" % (date)
    for file in csv_files:
        df = pd.read_csv(file)
        df_csv_append = df_csv_append.append(df)
    df_csv_append     =     df_csv_append.to_csv(run_dir+'/logs/'+filename,
mode='x', index=False, index_label=False)

    return filename

upload_data={
```

```
    'repo_url':
'ssh://atheodor@cslab.ece.ntua.gr@git.man.poznan.pl/hidalgo2/hid-bench-
ub.git',
    'local_repo_path': start_path+'/HIDALGO2_benchmarking',
    'logfile_path': start_path+'/perflogs',
    'pilot_name': 'UB',
    'system_name': 'PSNC',
    'file_type': 'benchmark',
    'scenario_name': scenario_name,
    'processes': 128,
    'date': "None"
}

def is_git_repo(path):
    try:
        _ = git.Repo(path)
        return True
    except git.InvalidGitRepositoryError:
        return False

def upload_to_git(upload_data, repo_destination_path, repo_filename):

    print ("repo_destination_path = %s" % repo_destination_path)
    print ("repo_filename = %s" % repo_filename)
    print ("logfile_path = %s" % upload_data['logfile_path'])

    if is_git_repo(upload_data['local_repo_path']):
        print("%s is a Git repository. Pulling..." %
upload_data['local_repo_path'], end="")
        repo = git.Repo(upload_data['local_repo_path'])
        repo.remotes.origin.pull()
    else:
        print("%s is not a Git repository. Cloning..." %
upload_data['local_repo_path'], end="")
        repo = git.Repo.clone_from(upload_data['repo_url'],
upload_data['local_repo_path'])

    print("successful.")
    print(repo)

    os.makedirs(repo_destination_path, exist_ok=True)
    shutil.copy(upload_data['logfile_path'] + repo_filename,
repo_destination_path + repo_filename)

    repo.index.add([repo_destination_path + repo_filename])
    repo.index.commit("Added %s to repo." %(repo_destination_path +
repo_filename))
    origin = repo.remote('origin')
    origin.push()
```