HORIZON-EUROHPC-JU-2021-COE-01

The European Centre of Excellence for Engineering Applications

Project Number: 101092621

D3.2 First Updated Report on Exa-Enabling Methodologies

The EXCELLERAT P2 project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101092621. The JU receives support from the European Union's Horizon Europe research and innovation programme and Germany, Italy, Slovenia, Spain, Sweden, and France.

Work Package:	WP3	Exa-HPC Methodologies and Technologies
Author(s):	Ansgar Niemöller, Matthias Meinke	RWTH
	Joeffrey Legaux, Mohamed Ghenai	CERFACS
	Michael Wagner	DLR
	Tufan Arslan, Herbert Owen	BSC
	Matic Brank	UL
	Francesco Salvadore	CINECA
	Sergio Pirozzoli, Giulio Soldati	URMLS
	Mattia Paladino	E4
	Roberto Rocco	E4
	Davide Padeletti, Gregor Weiss	USTUTT
	Tuan Anh Dao	KTH
	Alexis Laplanche, Etienne Renault	SiPearl
Approved by	Executive Centre Management	27-06-2025
Reviewer	Florent Duchaine	CERFACS
Reviewer	Janez Povh	UL
Dissemination Level	Public	

Date	**Author**	**Comments**	**Version**	**Status**
21-05-2025	Tufan Arslan, Herbert Owen	First draft	V0.1	Draft
24-06-2025	Tufan Arslan, Alexis Laplanche, Etienne Renault	Second draft for final check	V0.2	Draft
27-06-2025	Alexis Laplanche, Etienne Renault	Final version	V1.0	Final

List of abbreviations

ADR	Advection-Diffusion-Reaction
ALCF	Argonne Leadership Computing Facility
AMD	Advanced Micro Devices
APU	Accelerated Processing Unit
BiCGSTAB	Biconjugate Gradient Stabilized Method
BMBF	Bundesministerium für Bildung und Forschung
BSC	Barcelona Supercomputing Center
CAA	Computational Aeroacoustic
CE	Computational Efficiency
CERFACS	Centre Européen de Recherche et de Formation Avancée en Calcule Scientifique
CFD	Computational Fluid Dynamics
CG	Conjugate Gradient
CGNS	CFD General Notation System
CINECA	CINECA Consortium
CoE	Center of Excellence
CPU	Central Processing Unit
CRM	Common Research Model
CSR	Compressed Sparse Row
DDR	Double Data Rate
DG	Discontinuous Galerkin
DLR	German Aerospace Center
DMR	Dynamic Malleability Runtime
DoA	Description of the Action
DoF	Degree of Freedom
DSL	Domain Specific Language
EdF R&D	Energie de France research & development
FLOPs	Floating Point Operations Per Second
GCD	Graphics Compute Die
GMRES	Generalized Minimal Residual Method
GPU	Graphical Processor Unit
HBM	High Bandwidth Memory
HIP	Heterogeneous-computing Interface for Portability
HLRS	High-Performance Computing Center Stuttgart
HPC	High Performance Computing
I/O	Input - Output
ITER	International Thermonuclear Experimental Reactor
KTH	Kungliga Tekniska högskolan, Royal Institute of Technology
LES	Large-Eddy Simulation
MPI	Message Passing Interface
NASA	National Aeronautics and Space Administration
NCCL	NVIDIA Collective Communications Library
OpenACC	Open Accelerators (directive-based parallel programming model)

OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
OpenMPI	Open Message Passing Interface
PDE	Partial Differencial Equation
PU	Processing Unit
RAM	Random Access Memory
RANS	Reynolds-averaged Navier-Stokes
RCB	Recursive Coordinate Bisection
RWTH	Rheinisch-Westfälische Technische Hochschule
SAneg	Spalart-Allmaras one-equation turbulence model in its negative form
SIMD	Single Intruction Multiple Data
STL	Standard Template Library (C++)
SVE	Scalable Vector Extension
TEBE	TEsting Benchmarking Engineering
TFLOP/s	Tera Floating Point Operations per second
UC	Use Case
UL	University of Ljubljana
URMLS	University of Rome LA SAPIENZA
USTUTT	University of Stuttgart
WENO	Weighted Essentially Non-Oscillatory
WEST	Tungsten (W) Environment in Steady-state Tokamak

Executive Summary

DLR has continued to improve performance and scalability of the Computational Fluid Dynamics (CFD) software CODA, the FlowSimulator framework and the sparse linear systems solver Spliss. This includes an evaluation of CODA's improved scalability, of the newly introduced mixed-precision mode in Spliss, and of the newly developed hierarchical mesh partition method in FlowSimulator. Next to that CODA's containerised delivery was studied and CODA was ported and tested on various upcoming Central Processing Unit (CPU) and Graphics Processing Unit (GPU) architectures.

CERFACS has worked on expanding the existing GPU port of AVBP in terms of use cases coverage, supported architectures (with a strong focus on Advanced Micro Devices (AMD) GPUs) and general optimisation of the structure of the code to make it more efficient when offloaded to GPUs.

RWTH continued to improve the performance and parallel efficiency for large-scale multiphysics simulations with the code m-AIA. A large-scale use case has been executed with high efficiency on the 4096 compute nodes demonstrating exascale readiness of the m-AIA code on CPU-based High-Performance Computing (HPC) system. Porting efforts to adapt m-AIA to GPU/Accelerated Processing Unit (APU) architectures are advancing at a high pace. Benchmarking on four EuroHPC systems has been carried out.

BSC has focused on the GPU offloading of Alya using directive-based programming with OpenACC to minimise code changes while maintaining a unified codebase for both CPU and GPU targets. A first version of the code that can run incompressible Navier-Stokes problems fully on the GPU was obtained. The GPU performance was analysed and improved. In task 3.4, we pursued further work on the integration of Alya with the malleability framework Dynamic Malleability Runtime (DMR) to enable physics simulations which can resize at runtime to operate inside a desired efficiency range.

CINECA and URMLS have completed the rewriting of the FLEW code as part of the STREAmS-2 code. STREAmS-2 is based on an object-oriented architecture with support for different computational backends. The code for the different computational backends is generated through an in-house portability library that has been extended to integrate the new code features. The code has been benchmarked on different HPC systems with special focus on Leonardo and LUMI clusters. An initial pipeline for Continuous Benchmarking was also implemented. Several features for workflow improvement in exascale perspective were also implemented.

The focus of Neko was on improving the compressible solver and enabling efficient GPU-to-GPU communication using the NCCL library. Strong scaling tests on AMD and NVIDIA GPUs showed good parallel efficiency. Neko also benefited from vectorisation optimisations and memory access tuning, which demonstrated strong performance potential on architectures with high-bandwidth memory.

During the period, the UL team extended the Further Application FA-1 case to use distributed memory architectures by porting L2G and Raysect with OpenMPI library. L2G uses hybrid parallelisation with OpenMPI and OpenMP, and Raysect employs OpenMPI and Python multiprocessing. Strong scaling benchmarks were performed for all three codes (focusing on ITER and WEST reactor scenarios). Some scalability is observed, but performance is not optimal and the future work will focus on the optimisation improvement.

Preliminary results on co-design indicate that High Bandwidth Memory (HBM) can significantly benefit certain codes within EXCELLERAT P2, particularly when combined with Double Data Rate (DDR) memory on systems like Rhea. Ongoing and future work focuses on

code classification via Roofline analysis, with the goal of enabling targeted optimisations based on performance profiles.

Task 3.3 focused on developing a unified testing platform for validation, deployment, and benchmarking. Key efforts have included integrating tools from the CASTIEL 2 project for Alya and AVBP codes and creating an automated testing pipeline for the previously unsupported STREAmS application.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

Work Package (WP) 3 is intended to support all the algorithmic and computational developments of the different methodologies defined to execute the use cases. It is focused on the appropriate use of software and hardware so the use cases can be executed with exascale workflows. In the context of heterogeneous systems, the best mapping of algorithms and architectures will be analysed in detail, considering both computing time and energy costs. The research and development carried out in this work package will be crystallised into exascale-type workflows for the reference applications. It includes code developments and optimisations of the simulation elements to exploit all levels of parallelism from heterogeneous HPC systems and testing on emerging technologies and cooperation with vendors for co-design. Considering the diversity of use cases in terms of computational methods, discretisation strategies, HPC algorithms and simulation workflows, the activities are grouped into four tasks.

This document reports advances on Exascale enabling methodologies for all codes in the EXCELLERAT P2 project. The report includes advances in Tasks 3.1, 3.3 and 3.4 as established in the Grant Agreement. The first task focuses on the optimisation of the computational efficiency of the simulation methodologies employed in the use case at inter- and intra-node levels. Advances in code scalability, code optimisation and porting to GPU are reported. Task 3.2 focus on co-design (e.g., porting and optimizing) activities for (1) the upcoming Sipearl Rhea CPU, based on ARM micro-architecture, and (2) the Grace-Hopper 200 CPU-GPU and AMD MI300 APU in collaboration with the exaSim project. For Task 3.3 progress on Testing, Validation and Deployment is presented. Task 3.4 on Exascale Engineering deals with the specific developments required to extend the simulations workflows from Task 3.1 to achieve the large-scale readiness required in exascale simulations.

2 Task 3.1 Performance & Efficiency Engineering

This task is focused on the optimisation of the computational efficiency of the simulation methodologies employed in the use case at inter- and intra-node levels. It includes the combination of different parallelisation strategies based on distributed and shared memory, stream processing on GPU accelerators and efficient usage of hierarchical memory systems. Load balancing and communication/synchronisation reduction will be conducted in multiphysics applications and workflows including data-driven methods with Artificial Intelligence and multi-disciplinary analysis and optimisation. Advanced features of Message Passing Interface (MPI) such as non-blocking collectives, fault tolerance and remote memory access will be considered for some use cases. Finally, specific algorithmic modifications and communication strategies will be explored in the workflows and mapped to the supercomputing architectures. Optimisations considering both the algorithmic design and the implementation strategy such as energy efficiency and performance portability will be pursued.

The trend of HPC architectures in recent years and in particular the increasingly pervasive presence of accelerated architectures represents a great opportunity for achieving simulation objectives of great impact on both research and engineering application. To seize these opportunities, however, it is necessary to have software capable of adequately exploiting the hardware resources available. In this sense, the traditional approach to programming, which sees the compiler and the operating system as capable of providing a simple abstraction of the hardware to the developer, is in crisis. In HPC, software architects and developers are supposed to have a substantial knowledge of target hardware and program from that perspective using the adequate programming paradigms.

In a nutshell, this type of interaction can be framed within three main performance-oriented objectives:

1. parallelisation-oriented software design starting from the choice of algorithms that are or remain particularly efficient if parallelised.
2. implementation of algorithms “exposing” the parallel potential as much as possible.
3. choice of suitable programming paradigms to best use the available hardware.

From the point of view 1, particularly in the field of Computational fluid dynamics (CFD), the issues have been the subject of reflection for decades now, even if the balances of the parameters in the field are constantly evolving and can lead to changing conclusions. For example, an implicit algorithm for temporal evolution allows the use of a larger integration step, but the possible parallelisation methods are less efficient. On the contrary, an explicit algorithm, penalised by a very limited time step, can however be overall better due to its optimal versatility from a parallel calculation perspective.

From the point of view 2, it is necessary to remember that the same algorithm can be implemented in different ways and these implementation choices can significantly affect the compiler's ability to translate the source into efficient and truly parallel machine code. The conservative finite difference schemes used in FLEW can be implemented in a more compact, more efficient way in serial optics, or in a more extensive way, which however turns out to be more efficient in parallel optics.

What is expressed in points 1 and 2 strongly depends on the particular type of hardware or generation of hardware considered, but there are principles to be respected that are generally valid from the perspective of the current most widespread HPC architectures. From the point of view 3, however, the adaptation of the code requires, in addition to a very high commitment,

adaptability over time to the different parallel programming paradigms which can be substantially different. We distinguish four types of paradigms:

- vendor-specific: such as CUDA for NVIDIA GPUs or HIP for AMD GPUs
- standardised: such as OpenCL, OpenMP, OpenACC, SYCL
- intrinsic of the languages: C++ STL, Fortran do concurrent
- external: such as, for example, Legion, Kokkos, Raja

Each paradigm has advantages and disadvantages in terms of performance, maintainability, readability, portability and other relevant characteristics of the software that can be produced. Choosing one paradigm over another depends on the specific objectives of a certain porting activity.

2.1 CODA

During the reporting period, we achieved three major tasks: First, we evaluated the scalability improvements in CODA and FlowSimulator and compared the scalability to the baseline recorded in the previous reporting period. Second, we highlighted the benefits of the newly introduced mixed-precision mode in the sparse linear solver Spliss. Third, we extended the set of tested and supported CPU and GPU architectures.

First, we focused on evaluating the scalability of CODA on CARA with Use Case UC-1. CARA is a CPU system based on the AMD Naples architecture. The use case solves the Reynolds-averaged Navier-Stokes equations (RANS) with a Spalart-Allmaras turbulence model in its negative form (SA-neg). The use case runs on an unstructured mesh from the NASA Common Research Model (CRM) with about 5 million points and 24 million volume elements. The mesh is a rather small mesh, which has been chosen for a strong scalability analysis (fixed problem size) of CODA at currently available HPC systems. Production meshes are typically at least 10 times larger and accordingly achieve comparable efficiency on much higher scales. For the weak scalability analysis (fixed workload per core), we use different mesh sizes from the CRM mesh family ranging from 3 to 192 million elements and solve the use case with an according number of cores. The evaluation shows a significant improvement of CODA's scalability in comparison to the baseline from the start of the project. UC-1 achieves about 83% parallel efficiency (vs. 61% baseline) on the largest available partition on CARA with 512 nodes and 32,768 cores in the strong-scaling scenario. In the weak-scaling scenario, a parallel efficiency of 96% (vs. 72% baseline) was achieved on 32,768 cores.

Second, in addition to improvements in scalability, optimisations in compute speed were also considered. In particular, we evaluated the use of mixed-precision floating point calculations in the linear solver Spliss. A typical solver stack in Spliss that is used by CODA is, for example, GMRES (generalised minimal residual method) with Jacobi preconditioning. In this case, in mixed-precision mode the inner loops (Jacobi preconditioner) in the linear solver are computed with single floating-point precision (32 bits) while the outer loops (GMRES) are still computed with double floating-point precision (64 bits). The advantages are, on the one hand, the utilisation of twice the number of entries per SIMD instruction (for computed-bound sections) and the halving of the amount of data to be loaded from the memory (for memory-bound section). As a result, the calculation time of CODA could be accelerated by up to 72%. The acceleration depends on the test case and the ratio of inner to outer loops. On average, users report an acceleration of around 30% for the entire simulation.

Third, in a continuous effort to test and evaluate CODA and FlowSimulator on new CPU architectures, so far, the following systems have been studied with UC-1:

- AMD: Zen1, Zen2, Zen3, Zen4
- Intel: Saphire Rapids, Icelake
- ARM-based: Nvidia Grace, Graviton2, Graviton 3, Graviton 4

- GPU: Nvidia A100, Nvidia H100, AMD Mi210

For the evaluation, standardised benchmarks and a containerised version of CODA and FlowSimulator (see Task 3.4), including the use case UC-1, were used. These measurements allow adapting CODA to new architectures during the early-access phase and evaluate which systems offer best performance ahead of deployment to new full-scale HPC systems as well as provide valuable insight for designing DLR's own future HPC systems.

2.2 AVBP

Use case UC2 (hydrogen combustion) workflows requires two main parallel components on the road to exascale. First, an exascale-ready AVBP. This is handled in Task 3.4 with the portability of the code for AMD GPUs. Performance optimisation and efficiency of the code will be addressed in the next phases. Second, a highly parallel and efficient mesh adaptation component. With this in mind, the first period of EXCELLERAT P2 has focused on the robustness and reproducibility of the parallel mesh refinement library Treeadapt [1].
The library has been extended and is now known as KalpaTARU [2]. KalpaTARU now focuses on using PTScotch [3] as its underlying partitioning library instead of PARMetis [4] which exhibited technical and licensing issues. KalpaTARU now also supports the CGNS format for meshes and provides interoperability with AVBP's format. Specific work has been done to support periodic meshes which are a frequent occurrence in our complex simulations.

The scalability of AVBP on GPU was previously tested at large scale on V100 and A100 cards as shown in Figure 1 notably thanks to a JUWELS Booster access.

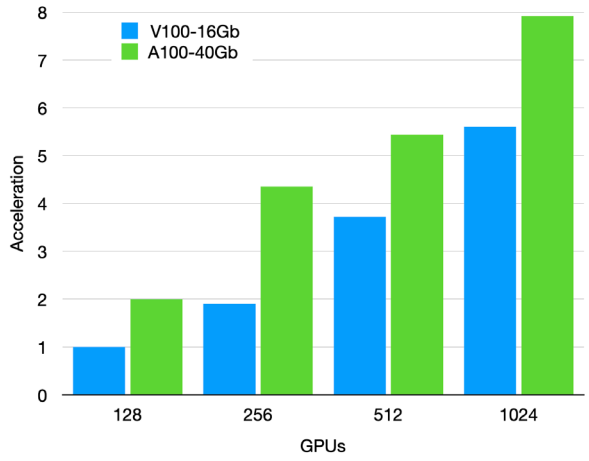


Figure 1: Scalability of the AVBP code on NVIDIA V100 and A100.

Acquisition of 4 H100 nodes at CERFACS allowed to assess the performance of AVBP on this new architecture. It required to move to more recent versions (>= 23.11) of the NVHPC compiler, which introduced new bugs. Corrections by Nvidia and workarounds on our side allowed us to successfully use AVBP with NVHPC 24.1.
The benchmarks showed consistent behaviour with previous observations: since AVBP is largely memory-bound on GPUs, the performance ratio very closely matches the memory bandwidth ratios of the various GPUs (Figure 2).

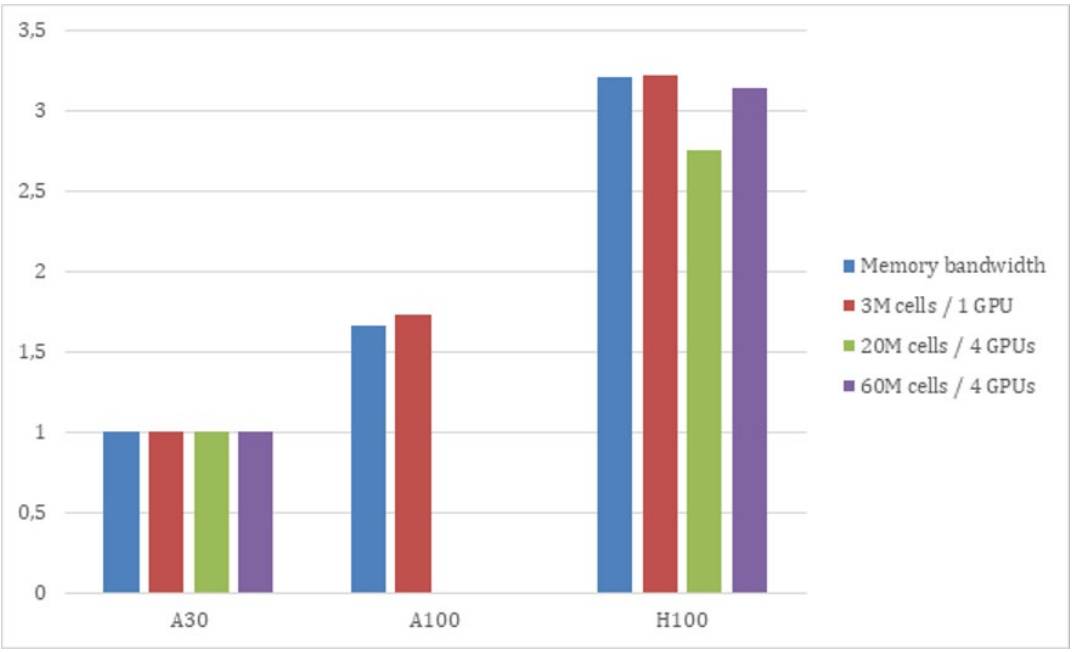


Figure 2: AVBP performance relative to A30 card on 3 reference configurations.

Codemetrics [5] tools developed in WP4 were applied in order to extend the GPU coverage inside AVBP, both in highlighting parts of the codebase that should be treated when extending the coverage, and in providing performance insights through the collection and analysis of job data points to ensure non-regression of the performance when maintaining/extending the GPU port.

The perimeter of AVBP on GPU has notably been enlarged by making the secondary lagrangian solver implementation hybrid, so that it can be used on CPU while the main Eulerian solver runs on GPU. A performance study campaign through the POP3 CoE [6] has started to assess the relevance of conducting an actual GPU port of the lagrangian solver.

General efforts have started in trying to understand the performance limitations of AVBP on GPUs. A major characteristic that has been identified is the fact that it is essentially memory-bound. This seems to come down from the fact that the code is structured in many small subroutines, leading in turn to many small GPU kernels with low arithmetic intensity. Experiments in rewriting the gradient computation part of AVBP (which represents a small amount of code but takes around 10% of execution time) as a single fused kernel could lead to 2 to 4 times faster GPU execution on this specific section. Further work will assess how we might generalise such deep transformations into the whole code base.

2.3 m-AIA

During the first project year RWTH significantly improved the performance and parallel efficiency of m-AIA for large-scale multiphysics simulations. Scaling an aeroacoustics application to the full Hawk HPC system of about 500,000 compute cores allowed the identification of performance issues which were not visible for smaller scale runs or less complex simulation setups. For example, a critical issue related to a specific inter-process communication was discovered and resolved. Efforts also targeted improvements of the dynamic load balancing approach for coupled CFD/CAA simulations and issues linked to the CPU power management as well as a workaround were identified. Strong scaling tests for benchmark cases on Hawk showed excellent parallel efficiency and superlinear speedups for m-AIA on up to 4096 Hawk nodes.

Apart from continuous improvement of the multiphysics simulation framework m-AIA, e.g., related to the dynamic load balancing approach, a key focus initiated during the second project year involved porting activities aimed at adapting the complete code to GPU/APU architectures to facilitate future use case executions and runs of the envisioned optimisation workflow on different types of HPC hardware. Apart from transforming code loops to C++ parallel-stl versions, the current improvements including code restructuring, clean-up and memory reduction are expected to enhance the execution efficiency of m-AIA not only on GPU/APU-based systems but also on traditional CPU-based HPC platforms. For the present use case, UC-3, the FV-CFD and the DG-CAA solver of m-AIA and their respective coupling need to be ported. Since the DG method is based on a polynomial solution representation in each cell or element of the grid, with a potentially high number of degrees of freedom (DOF) inside a single element, compute kernels need to be rewritten to loop over individual nodes instead of full elements. Through this approach the high computational effort required for each element can be split into small parts to be distributed among many threads. Figure 3 compares the performance in achievable time steps per second of the initially ported parallel-stl loops iterating over elements and surfaces to a first version of the rewritten kernels looping over individual nodes. The benchmark case consists of approx. 1 billion DOF and is executed on the AMD MI300A based Hunter HPC system installed at HLRS with each node equipped with 4 MI-300A APUs. As evident, the changes to the loop structures result in a speedup of a factor of 6, while for 64 nodes, i.e., 256 APUs still a parallel efficiency of about 70% is achieved.
The current main objective is to finalise the GPU/APU porting of the coupled CFD/CAA solvers. Based on an initial version supporting more complex simulations the performance will be further optimised.

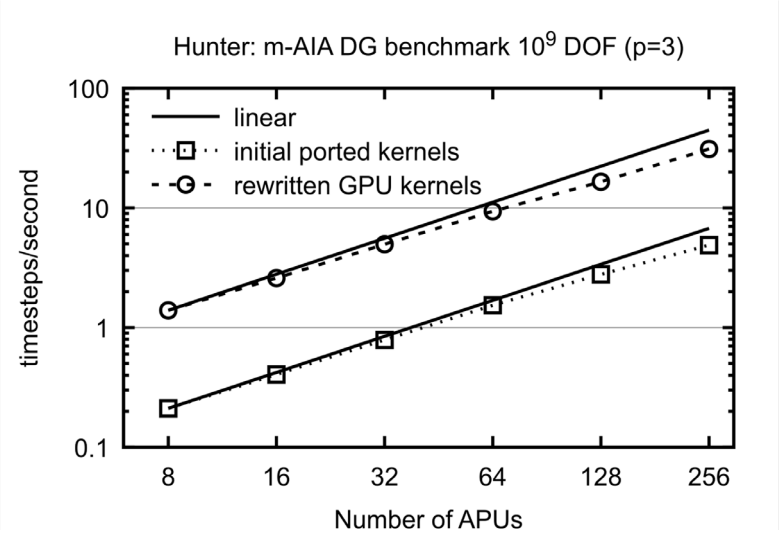


Figure 3: m-AIA DG benchmark with 1 billion DOF: comparison of initial parallel-stl ported and rewritten GPU/APU kernels on Hunter.

2.4 Alya and Sod2d

BSC is currently working with two codes: Alya, a multiphysics finite element (low order) code that has been developed since the beginning of BSC in 2006 and Sod2d, a spectral element CFD code that started as a high order alternative of Alya a couple of years ago but is now developed independently. One of the key advantages of Sod2d being a separate code is that it is completely open source, which is not the case with Alya. Sod2d can be used as a library from within Alya. Sod2d has been designed from scratch to work fully on GPUs while Alya which started as a CPU code is currently being ported to GPUs.
While in D3.1 [7] we put a strong emphasis on the work done with Sod2d, due to space limitations we will now focus more on the GPU porting and performance optimisation of Alya.

The BSC has led the GPU offloading of the Alya simulation framework using a unified code strategy to preserve both maintainability and performance portability. This effort followed a progressive approach: prototyping on miniapps, refinement within the modular Alya Library, and integration into production-grade Alya.
The adopted GPU offloading strategy focuses on using directive-based programming with OpenACC to minimise code changes while maintaining a unified codebase for both CPU and GPU targets. A key design principle was ensuring that all offloading decisions preserved existing CPU performance. Code vectorisation was adapted using a pre-processor-based abstraction that automatically adjusts loop granularity for CPU and GPU architectures. GPU memory management was aligned with CPU allocation flows to avoid costly transfers and improve correctness. The work can be divided into three stages.

Stage 1: RHS Assembly Prototype – Nastin Miniapp
The first stage explored GPU offloading through the Nastin-Miniapp, a snippet of Alya's NASTIN module focused on the right-hand side assembly for the incompressible Navier–Stokes equations. This mini-app removed complexities such as MPI, solvers, and external dependencies, offering a clean environment to learn directive-based offloading and explore optimisation strategies.
Optimisation work started by privatizing around 200 intermediate values per element, reducing memory traffic and achieving a speedup by factor 6. Memory residency was extended across timesteps to limit data movement, adding another 30% gain. Fixing loop bounds at compile time for tetrahedral elements further improved performance by factor 3, although this limited mesh flexibility.
Subsequent kernel profiling revealed memory bottlenecks and register pressure. Splitting the main kernel into smaller units reduced these limitations, doubling execution speed in non-fixed loop bounds configurations and significantly narrowing the performance gap between fixed and general cases. OpenMP Offload implementations showed equivalent performance to OpenACC.

Stage 2: Alya Library
Building on these findings, the second stage applied GPU offloading to the Alya Library, a modular reimplementation of Alya using modern Fortran and object-oriented design. Although still under development, the Alya Library offers a realistic testing ground with a structured mesh generator for easily configuring problem size and evaluating GPU performance.
The Alya-ADR miniapp, built on this library, solves scalar advection-diffusion-reaction equations using explicit and implicit time integration schemes. It simplified the physical model by assuming constant properties and focused on testing matrix assembly, CSR matrices, and GMRES solvers. Performance benchmarks showed that GPU acceleration was highly sensitive to mesh size and vector chunk size, with optimal results achieved for large models and a vector batch size of 512k elements. Solver stages benefited the most, although the pre-processing phase remained a bottleneck due to its inherent sequential nature of Krylov subspace orthogonalisation.

Further efforts focused on bottom-up integration of GPU memory management into the Alya Library's data structures, improving data lifetime handling and ensuring allocations and deallocations on device mirrored host memory management. The integration was extended to new solvers, such as CG and BiCGSTAB, and matrix formats, including COO and ELL. Performance validation using a Laplacian problem confirmed that SpMV kernels dominated execution time, and targeted optimisations such as switching to single precision, kernel configuration tuning, and Cuthill–McKee reordering provided further gains. These results laid the foundation for production-ready GPU execution.

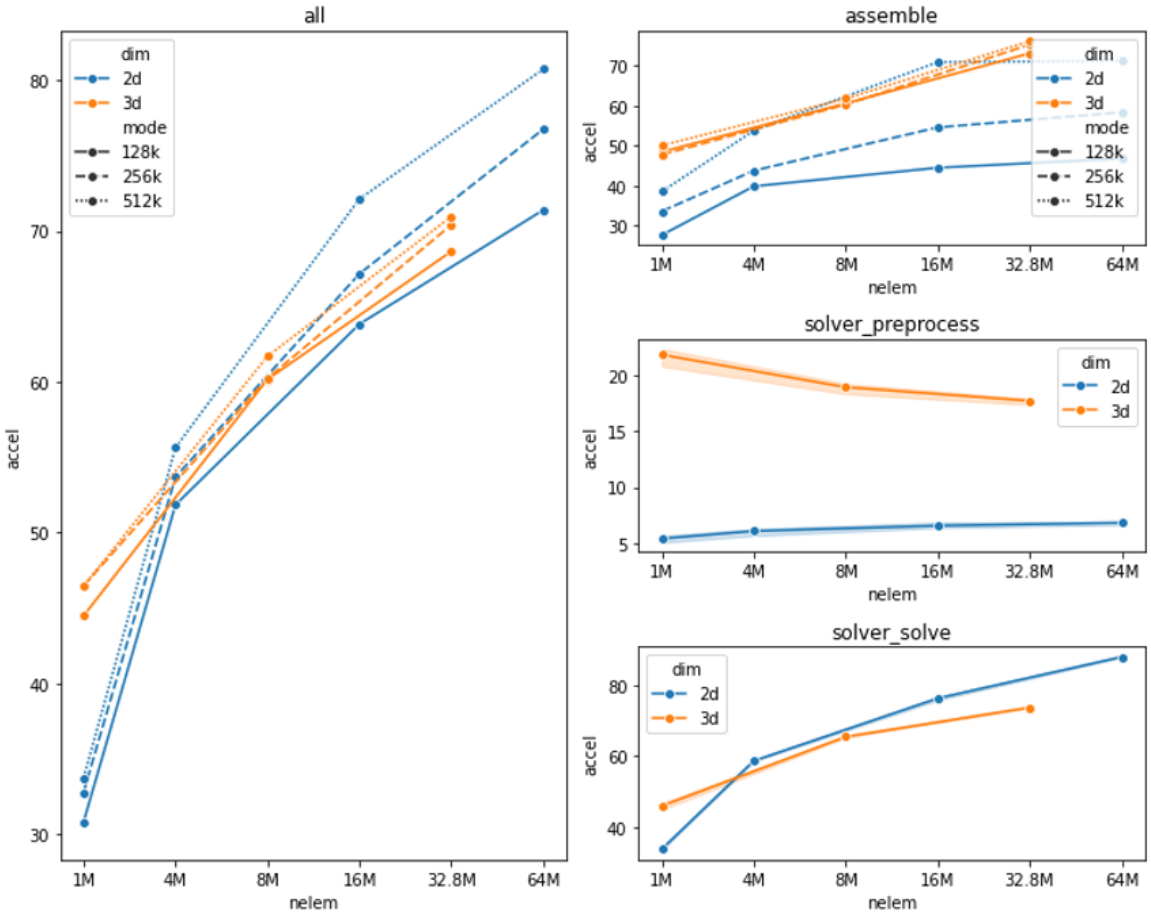


Figure 4: Speedups of key components in the Alya-ADR simulation across varying mesh sizes and VECTOR_SIZE configurations.

Stage 3: Production Alya-KERNEL and Modules

The production version of Alya, composed of the KERNEL, KERMOD, and DOMAIN modules, provides the foundation for physical models like NASTIN, CHEMIC, and EXMEDI. GPU offloading in this final stage focused on porting critical computational routines, leaving initialisation phases on the CPU. Despite its legacy design posing challenges for developers, the production code's simpler data structures facilitated compiler optimisations.

The GPU-optimized prototype of the NASTIN module, originally developed during the work published in [8], was successfully integrated with the newly offloaded data structures and solvers. This version solves the incompressible Navier–Stokes equations using a Large Eddy Simulation turbulence model, an explicit momentum formulation with third-order Runge-Kutta time integration, and a multi-step fractional step method for pressure-velocity coupling. Several simplifications were introduced for more performance.

Optimisation efforts delivered significant performance improvements. Increasing the VECTOR_SIZE parameter to 2048k reduced matrix assembly time by 40%. Cuthill–McKee reordering improved memory access patterns, cutting the cost of sparse matrix-vector multiplications by around 20%. Fine-tuning OpenACC kernel configurations and other efforts further improved performance.

Compared to a full CPU node using 80 MPI processes, the complete simulation loop on a single GPU achieved an acceleration by a factor 3. Matrix assembly showed the greatest speedup of over 10×, and solver performed 2.5 times faster. These results demonstrate that GPU acceleration already brings a substantial performance gain and ensures improved energy efficiency, making it a viable and promising solution for production Alya users.

These contributions were integrated into the main branch following continuous integration and automated testing to ensure code stability and correctness. Performance monitoring was introduced using Rooster, a tool that provides clear visual feedback on key performance indicators and tracks the impact of code evolutions to ensure that future developments deliver performance.

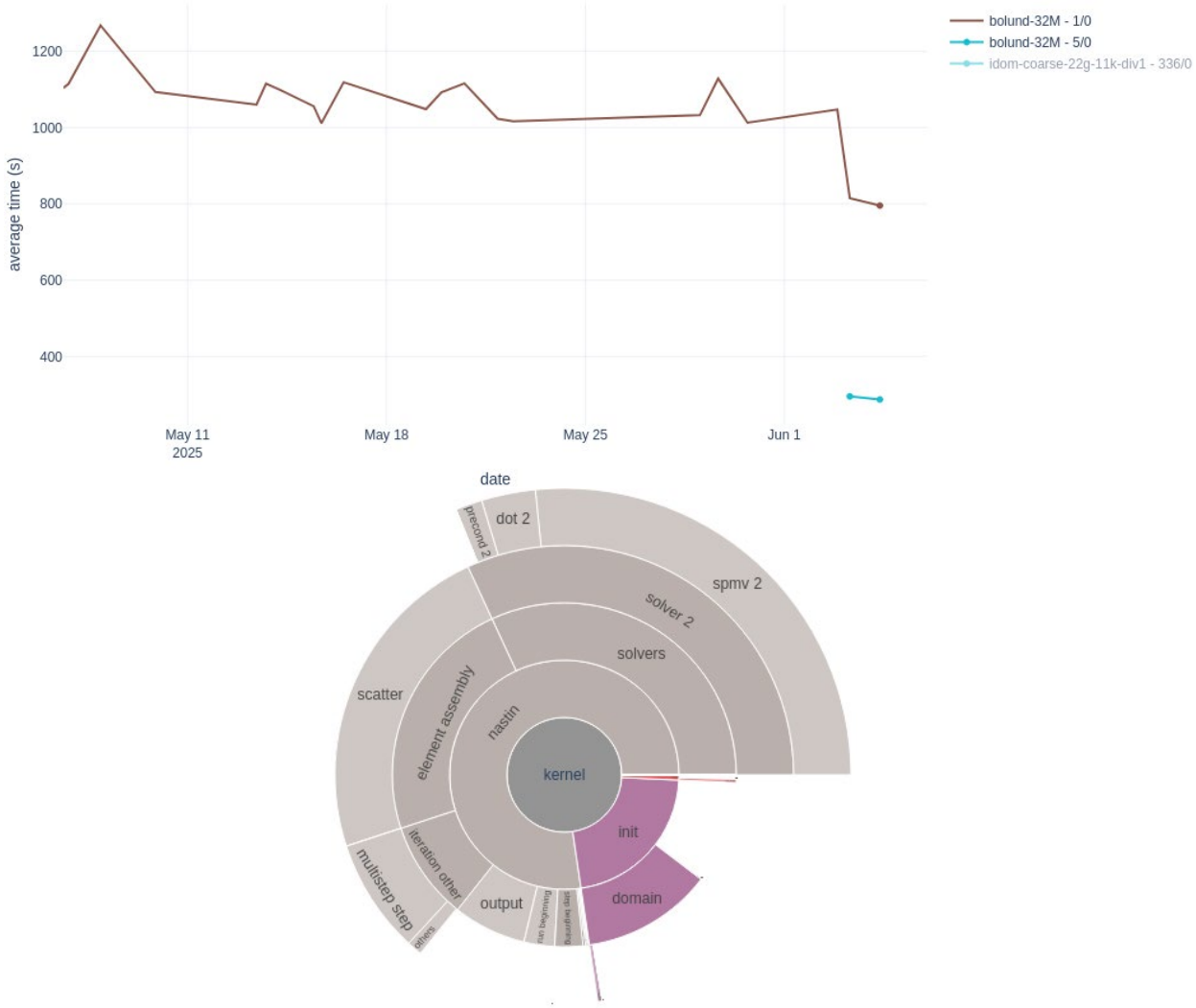


Figure 5: Performance analysis charts generated with Rooster, tracking NASTIN's GPU performance for the Bolund 32M elements case.

2.5 Neko

Work during this period focused on two areas: improving the compressible solver and improving the communication between GPU-to-GPU on NVIDIA GPU clusters.

First, strong scaling tests were performed on the compressible solver using AMD MI250X and NVIDIA GH200 GPUs. A fixed problem size was used while increasing the number of GPUs. We have used a spatial discretisation using 60x60x60 7^{th}-order elements, and a temporal discretisation using the standard 4^{th}-order explicit Runge-Kutta scheme. The test problem is the compressible Taylor-Green vortex. Results show that Neko maintains good scaling efficiency across both platforms, although the size of the benchmark is relatively small, only up to 4 nodes, each node with 8 AMD MI250X or 4 NVIDIA GH200 GPUs (See Figure 6). This is currently limited to the GH200 cluster that we are testing the code on.

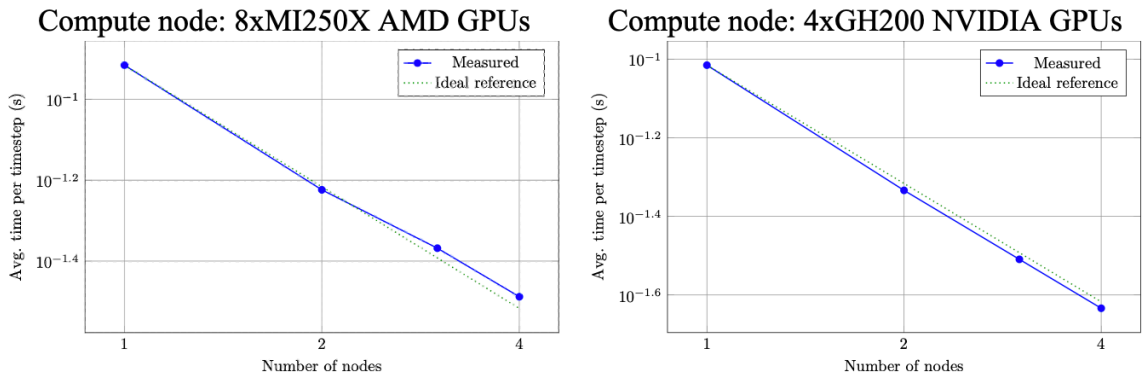


Figure 6: Strong scaling in the newly implemented compressible solver in Neko.

Separate strong scaling tests were performed on CPUs, comparing NVIDIA Grace CPUs and HPE Cray CPUs. The results allow a direct comparison between CPU types and show how the compressible solver performs on modern CPU-only systems (see Figure 7).

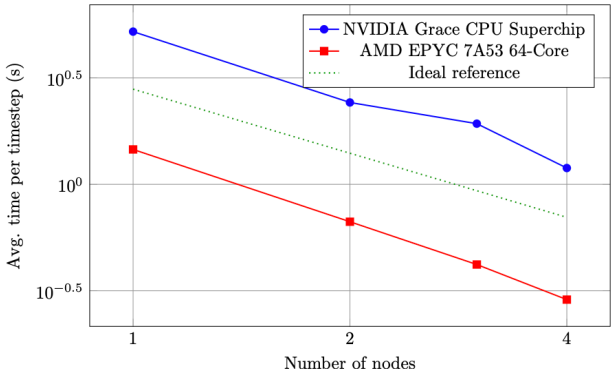


Figure 7: Strong scaling in the newly implemented compressible solver in Neko.

Second, support for the NCCL library was added to Neko. NCCL is now used as a backend for gather-scatter operations and as an alternative option for collective communication. This new feature can be enabled at compile time when targeting NVIDIA GPUs by adding the flag `-DHAVE_NCCL=1`. It improves communication performance of operations such as all-reduce or broadcast by using hardware-accelerated GPU-to-GPU transfers.

2.6 STREAmS

STREAmS has been upgraded to version 2.1, and that version has also been published online open-source at **https://github.com/STREAmS-CFD/STREAmS-2**, as it was previously decided within the EXCELLERAT project but well before the final planned timeline. The current version of the solver incorporates all the features of the FLEW [9] code for curvilinear grid support. In addition, new features have been implemented that are particularly useful for airfoil simulations. The activity of developing new code features was coupled with the activity of porting to different architectures through different programming paradigms. This was made possible by the peculiar development mode of STREAmS, which is done through two levels:

- code development in its core programming paradigm, namely CUDA Fortran
- conversion of the code to the other backends via *sutils* in-house library. Specific tuning for each backend is possible through compact *sutils* external input.

During the project, *sutils* has been updated several times to work with the latest versions of STREAmS in particular supporting curvilinear grid conversion and other new features. In addition, sutils has been extended to produce code for 4 computational backends, namely:

- CPU (pure MPI)
- OpenMP
- HIP: details on the porting have been published in [10]
- OpenMP-offload: details on the porting have been published in [11]

For CPU, OpenMP, and HIP backends, low-memory versions can be obtained that avoid unnecessary array duplication between CPUs and devices. Array duplication is clearly unnecessary for backends where the computing device is the CPU itself (CPU, OpenMP). The same may be true for the HIP backend in case it is used in the unified memory mode possible for some recent architectures (such as AMD MI300A).
The different STREAmS backends were tested and benchmarked on a 4096 x 286 x 276 airfoil grid corresponding to about 550M points. This benchmark case was tested on a single node of different systems shown in Table 1. The table also shows the main characteristics of the systems considered along with the compilers and versions used by our tests as well as the STREAmS backends adopted. The use of the diverse systems was possible through CINECA internal access (Leonardo), EuroHPC projects (LUMI, MareNostrum5), JUREAP initiative (JEDI), ALCF Director's Discretionary allocation program (Aurora) and Gauss Center for Supercomputing project (Hunter).

The results of elapsed times per iteration are displayed in Figure 8, considering both energy-preserving (central) and shock-capturing (WENO) convective schemes. The obtained performance tends to reflect in an expected way the peak performance of the considered systems with particular reference to bandwidth performance that plays a key role in a code like STREAmS. The performance of nodes with NVIDIA A100 and AMD MI250X GPUs is similar as well as that of nodes with NVIDIA H100 and AMD MI300A. The nodes with Intel 1550 GPU show similar performance as the H100 nodes, but Aurora has 6 GPUs per node. The performance of CPU nodes shows drastically worse performance without much variability moving from CPU backend to OpenMP backend.

#	Cluster	Partition	Type	PU	#PU	Backend	Compiler	Ver	MPI	Ver	BW	FLOPs	Nodes
S1	Leonardo	Booster	GPU	NVIDIA A100 (SXM4 64 GB)	4	CUDA Fortran	NVIDIA	24.3	OpenMPI	4.1.6	1635 x 4	20 x 4	1024
S2	Marenostrum5	ACC	GPU	NVIDIA H100 (64GB HBM2)	4	CUDA Fortran	NVIDIA	24.5	OpenMPI	4.1.7	2000 x 4	26 x 4	64
S3	JEDI	-	Superchip	NVIDIA GH200 (96GB, 4TB/s)	4	CUDA Fortran	NVIDIA	25.1	OpenMPI	5.0.5	4000 x 4	34 x 4	32
S4	LUMI	G	GPU	AMD MI250X	4 (8 GCDs)	HIP	GNU/ROCm	13.2.1/6.0.3	Cray-MPICH	8.1.29	3200 x 4	48 x 4	2048
S5	Hunter	GPU	APU	AMD MI300A (128GB)	4	HIP	Flang/ROCm	18.0.0/6.2.2	Cray-MPICH	8.1.30	5300 x 4	61 x 4	64
S6	Aurora	-	GPU	Intel 1550 (128GB)	6 (12 Tiles)	OpenMP offload	Intel	2024.07.30.002	MPICH	4.3.0rc3	3277 x 6	52 x 6	2048
S7	MareNostrum5	GPP	CPU	Intel Xeon Platinum 8480p	2	CPU	Intel	2023.2.0	IntelMPI	2021.10.0	600	9	128
S8	MareNostrum5	GPP	CPU	Intel Xeon Platinum 8480p	2	OpenMP	Intel	2023.2.0	IntelMPI	2021.10.0	600	9	512
S9	LUMI	C	CPU	AMD EPYC 7763	2	CPU	GNU (Flang)	13.2.1 (17.0.0)	Cray-MPICH	8.1.29	410	5	1024
S10	LUMI	C	CPU	AMD EPYC 7763	2	OpenMP	GNU (Flang)	13.2.1 (17.0.0)	Cray-MPICH	8.1.29	410	5	1024

Table 1: Systems, environments and STREAmS backends where performance was measured.

The same system is repeated if different STREAmS backends are used. The columns report acronym, cluster name, partition name, main Processing Unit (PU) type, PU model, number of PUs per node, adopted STREAmS backend, compiler, compiler version, MPI library, MPI library version, theoretical peak bandwidth per node (GB/s), peak FLOPs per node (TFLOP/s) and maximum number of nodes used for our tests.

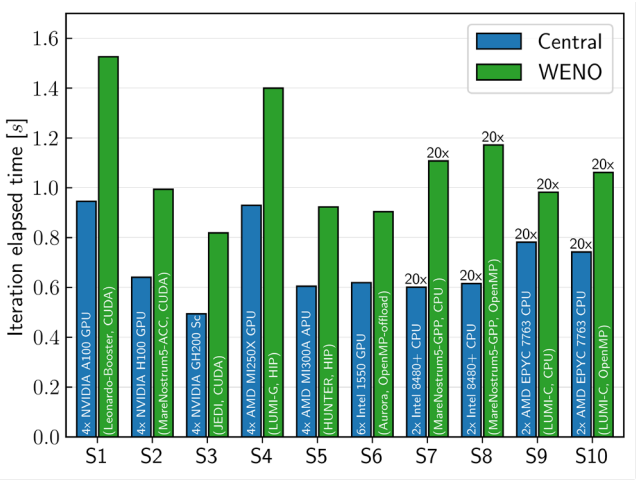


Figure 8: Elapsed time comparison of the reference case (4096 x 286 x 276 grid) using one computing node for each system considered. Both central and WENO results are reported. Note that CPU results are multiplied by a factor 20 for readability.

To gain more awareness of the absolute performance of the solver, roofline analyses of the main kernels were performed considering three reference GPUs namely NVIDIA A100, AMD MI250X (GCD) and Intel 1550 (tile). Profilers from different vendors were used, namely NVIDIA Nsight Compute, AMD Roc profiler and Intel Advisor. The results are shown in Figure 9, comparing HBM and L1 memory levels. For simpler kernels the points are near the lower region of the peak bandwidth area considering HBM level and the use of L1 does not change the situation. For more complex kernels we move toward the compute bound region and the use of L1 shows a significant decrease in arithmetic intensity and thus a higher perceived bandwidth from the programmer's point of view. Overall, the absolute performance of the code is good.

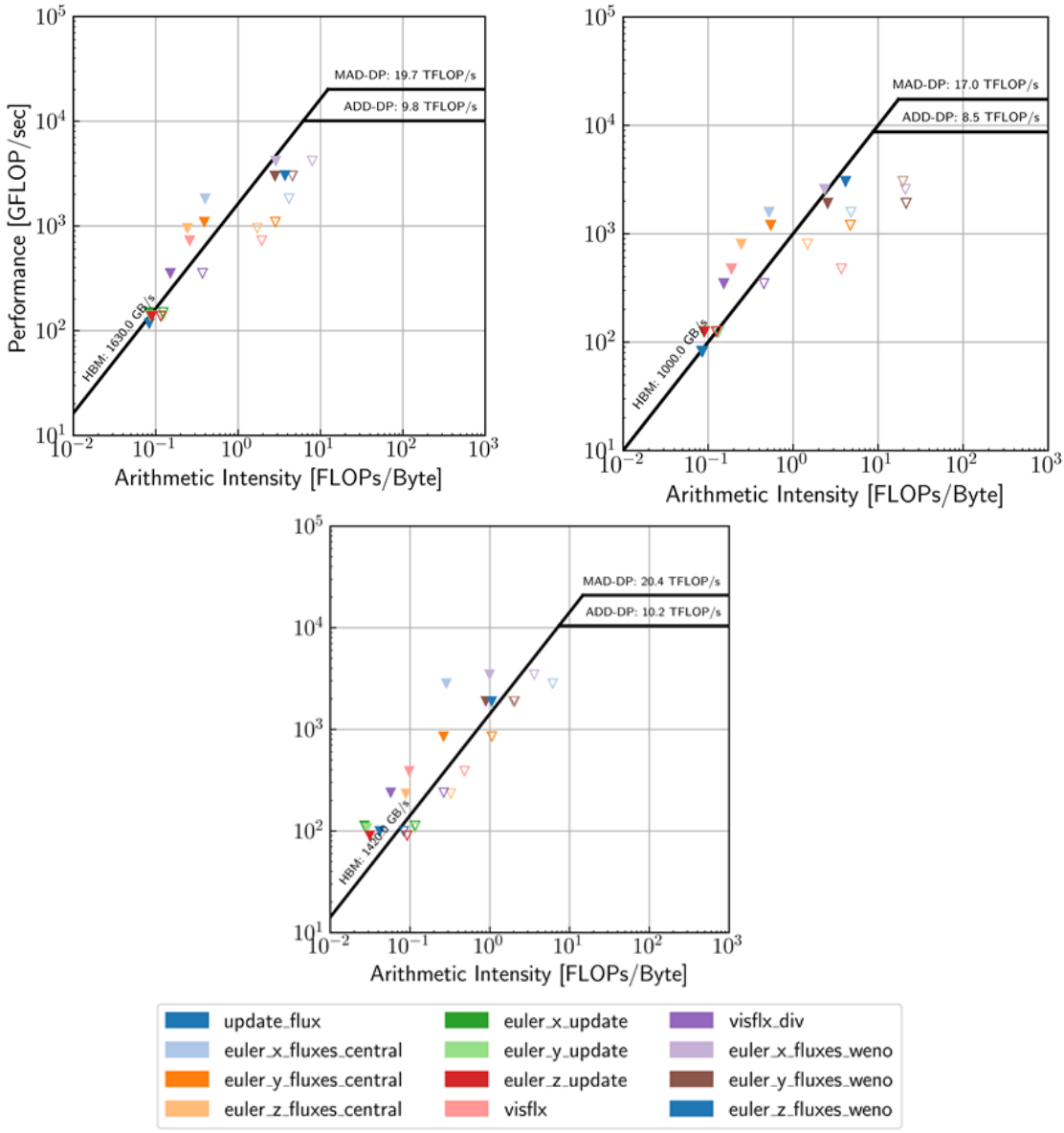


Figure 9: Hierarchical roofline analysis for HBM and L1 memory levels: FLOPs per second (vertical axis) vs Arithmetic Intensity (horizontal axis). Significant kernels are considered comparing HBM values (empty markers) vs L1 values (filled markers). Three architectures are considered, namely NVIDIA A100 GPU (left), AMD MI250X GCD (middle), and Intel 1550 Tile (right).

Weak scalability is reported for Leonardo and LUMI-G systems considering synchronous and asynchronous communication patterns. The scalability shows the times per iteration using N nodes rescaled by the time at one node corresponding to the reference case. The part with grey background shows the intra-node scalability while the remaining points show the inter-node scalability. The used computational grids correspond to physically significant cases as the Reynolds number that can be simulated with a given grid increase. It is worth noting that the memory occupation is not high (around 30% for Aurora cases for example) but this has been done in view of realistic time-to-solution conditions. The largest case (2048 nodes) corresponds to a computational grid suitable for Reynolds around 6M. The overall scaling performance is good. For Leonardo, asynchronous communication allows keeping times within 20% of the single-node case up to 1024 nodes, while for LUMI-G the asynchronous mode is not really

useful, but it is not necessary to achieve very good efficiency. We notice that the largest cases are well beyond the maximum limits available using the standard system queues.

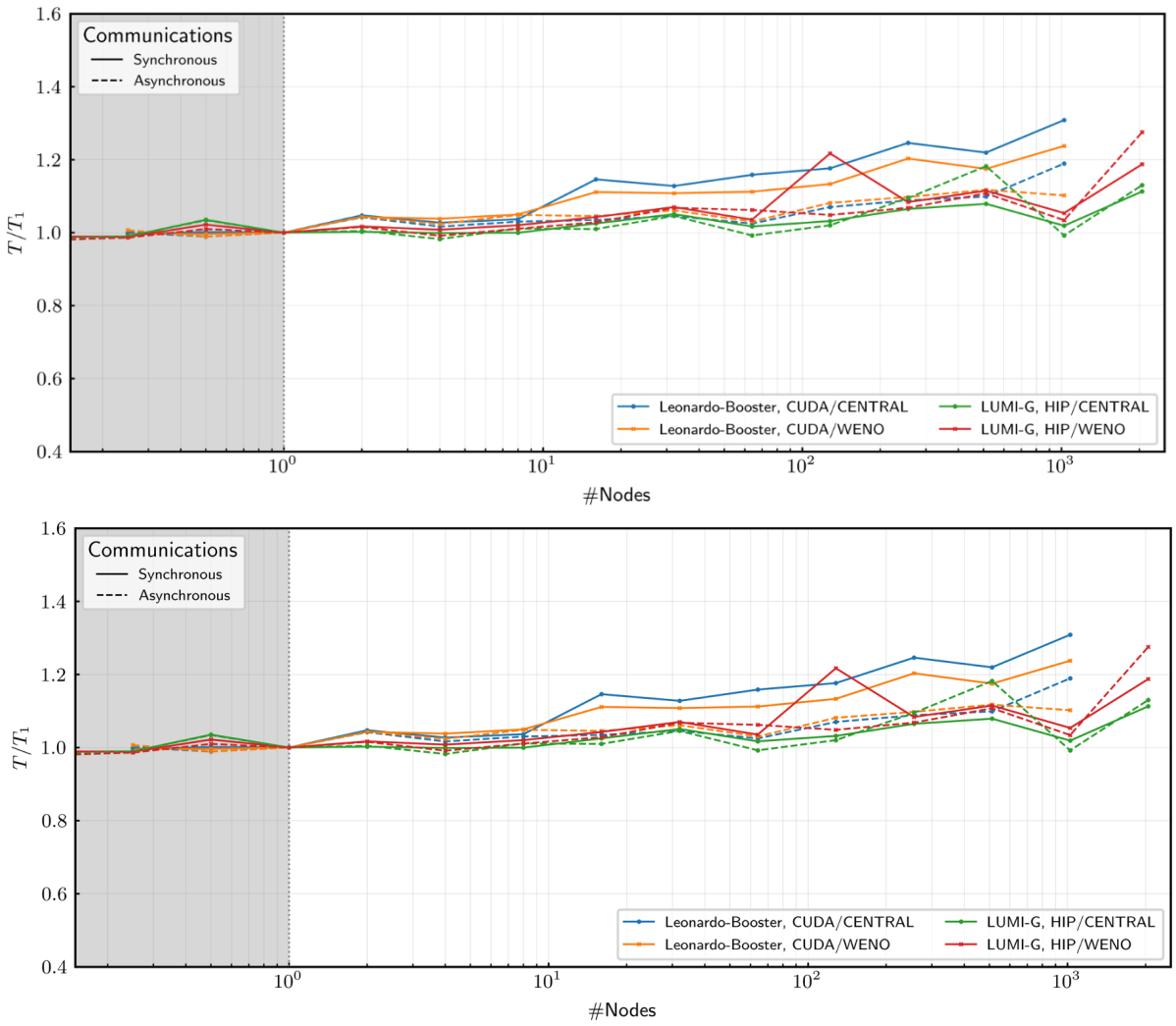


Figure 10: Weak scalability results for Leonardo-Booster and LUMI-G systems.

In Figure 10 the reference case is the single-node case. Both intra-node and inter-node scaling are reported as elapsed time T normalised with the single node value T1 versus number of nodes. Intra-node region is highlighted using a grey background. Continuous (dashed) lines are used to represent results with synchronous (asychronous) communications.

2.7 L2G, OpenFOAM, Raysect

The work during this period focused on porting the FA-case workflow to distributed memory architecture. Previously only OpenFOAM was executed across multiple computational nodes, and during this period we ported L2G and Raysect as well. Both L2G and Raysect utilise the OpenMPI library for distributed computation. L2G has a hybrid parallelisation strategy that combines OpenMPI for inter-node communication with OpenMP for threading within each node. OpenMPI was also added to Raysect for distributing rendering across multiple nodes. For intra-node parallelism we use Python's multiprocessing module. For all three code we ran strong scaling benchmarks on the Vega HPC machine. OpenFOAM simulations were executed on the `largemem` partition of Vega, which consists of 192 nodes. Each node has two AMD EPYC Rome 7H12 CPUs, with 128 physical cores per node, and has 1 TB of RAM. High-speed

HDR100 Infiniband networking is installed for inter-node communication. L2G and Raysect benchmarks were ran on the `standard` partition, where each node offers 256 GB of RAM instead of 1 TB.

In the case of L2G, strong scaling tests were performed for up to 5000 total parallel processes (50 MPI ranks with 100 OpenMP threads). The benchmarking scenario was based on the WEST reactor geometry, with a focus on two subcomponents: the baffle "target," composed of ~50000 elements, and the divertor "shadow," composed of ~2 million cells. The simulation involved launching magnetic field lines into the computational domain from the target and calculating their intersection with the neighbouring divertor shadow.

$N_{threads}$	t_{exec}[s]
1	992
2	494
1000	60
5000	21

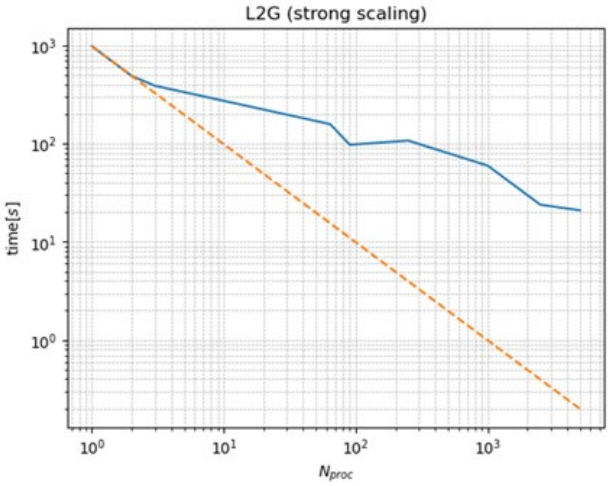


Figure 11: (left) table of thread numbers vs. time (right) Strong scaling plot for L2G (blue line) with theoretical scaling (dashed orange line).

OpenFOAM benchmarks focused on simulating the ITER reactor's first wall panel, again panel with ID 4. This geometry comprised roughly 35 million tetrahedral cells. The simulation ran for 30 timesteps and was tested for up to 1200 MPI processes. The OpenFOAM native binary format was used for mesh.

$N_{threads}$	t_{exec}[s]
1	3490
48	837
96	273
240	165
480	68.9
960	24.8
1200	16.4

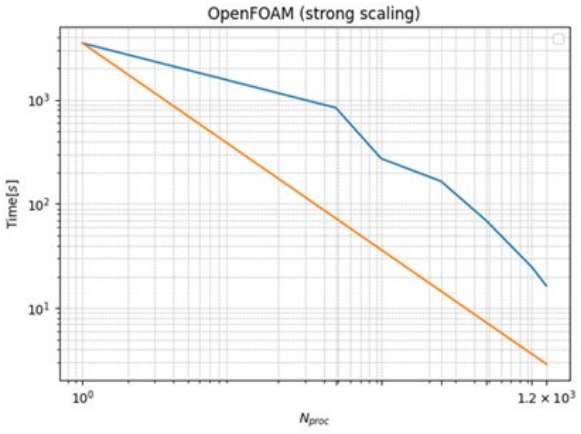


Figure 12: (left) table of thread numbers vs. time (right) Strong scaling plot for OpenFOAM (blue line) with theoretical scaling (dashed orange line).

Raysect benchmarks targeted a simplified representation of the ITER reactor, concentrating on the outer wall panels with IDs 8-18. A 1024×1024 resolution camera with an RGB adaptive

sampler was defined, launching 250 rays per pixel. The test was scaled up to 1500 MPI processes and shows distribution of computational load during rendering passes.

$N_{threads}$	t_{exec}[s]
1	13695
32	6671
96	4646
390	1990
70	1429
1150	1466
1540	1350

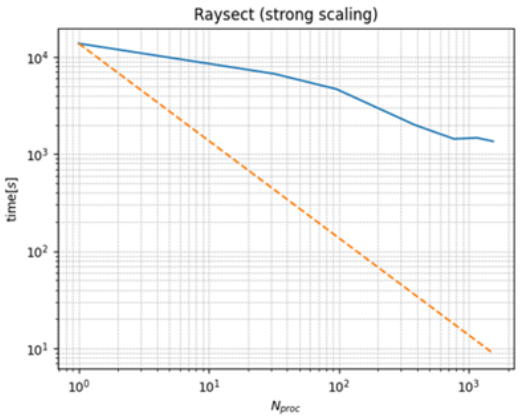


Figure 13: (left) table of thread numbers vs. time (right) Strong scaling plot for Raysect (blue line) with theoretical scaling (dashed orange line).

Strong scaling plots show some efficiency gains; however, the plots are far from ideal. For the remaining period of the project, the UL will have to put focus into improving the scalability of the proposed codes. The collaboration with another Center of Excellence (POP3) is planned to advance the scalability of L2G with the use of their libraries Extrae [12]/Paraver [13]. We also collaborate internally with SiPearl through mini-app tests for OpenFOAM application.

Regarding IO optimisation, both L2G and Raysect now include reading and writing routines for the MED file format for mesh exchange. This format is based on HDF5 and identified by the `.med` extension. It is supported via the MED file C++ library (provided by EdF R&D) and managed with the MEDCoupling API. OpenFOAM continues to use its native binary format for both mesh and simulation field data.

3 Task 3.2: Co-design lab for emerging technologies

3.1 Co-Design

Co-design in High-Performance Computing (HPC) is a collaborative, iterative process where hardware architects, software developers, domain scientists, and system designers work together to create systems tailored to specific workloads. By aligning hardware and software development, co-design maximises performance, energy efficiency, and usability. When technologies nature and system designs are fixed, co-design may shift to merely porting and optimizing applications for the dedicated hardware but can shape future hardware and software iterations.

SiPearl is a European company, issued from the efforts around the definition of the many-cores EPI processor architecture. Its first processor, Rhea, is a general-purpose ARM-based CPU equipped with both HBM and DDR memories (see Figure 14). Rhea is composed of power efficient Arm Neoverse V1 cores with the Arm Scalable Vector Extension (SVE). To address the full range of HPC workloads (including AI and Machine Learning), these SVE units support multiple precision types: double precision, single precision, BFloat16 and 8-bits integers. Incorporating in-package High Bandwidth Memory (HBM2e), Rhea also delivers extraordinary compute performance and efficiency with an unmatched Bytes/Flops ratio. Since the core count of modern processors is increasing faster than total memory capacity and bandwidth, the gap is not getting closer, so the memory subsystem is now more critical than ever, making today many applications memory-bounded, especially the codes studied in the EXCELLERAT P2 Project.

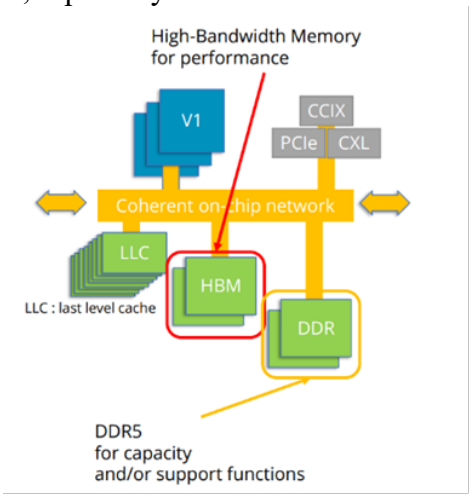


Figure 14: **ARM-based CPU equipped with both HBM and DDR memories.**

Thanks to the Mont-Blanc project, AArch64 ecosystem has reached a high level of maturity in HPC, with prototypes starting in 2014 and achieving Top500 recognition by 2018; Fugaku topped the list in 2020. Nonetheless Rhea offers several new features that may require to modify existing application and libraries by co-design efforts with the members of the EXCELLERAT Consortium:

- **SVE with 256-bit length**: requires to carefully design compilers and algorithms to benefit this technology. As others SIMD architectures, SVE allows to execute a single instruction on multiple data. However, SVE does not define a fix vector size, but vector size can be defined at hardware level, ranging from 128 to 2048 bits by 128-bit increments. Therefore, any CPU vendor can implement the extension by choosing the Vector Length (VL) size that better suits the workloads the CPU is targeting. The design of SVE guarantees that the same program can run on different implementations of the instruction set architecture without the need to recompile the code removing the barriers for auto-vectorisation. Vector Length Agnostic (VLA)

capability enables portability, scalability, and optimisations in comparison with other traditional unpredicated SIMD architectures. On such architectures, the programmer (or the compiler) needs to add an additional loop, called loop tail, that is responsible for processing those iterations at the end of the loop that do not fit in a full vector length. VLA includes instructions that allow the vector code to automatically adapt to the current vector length at runtime.

- **HBM2e and DDR5 for memory bandwidth, latency and capacity** requires to carefully place data to the best memory tiers. Indeed, DDR offers a high capacity, low latency memory while HBM offers a high latency, lower capacity (up to 64Go for Rhea). To take the most of these two memory tiers, one should carefully place data. Contrary to traditional machine, with a single memory tier, this highlights the need of having a closer look to allocations, deallocation, and memory access patterns.

These two specificities require some co-design effort for every application in EXCELLERAT P2. Nonetheless, large applications are hard to handle, profile, study and optimise. Consequently, an effort has been done to extract kernels/mini-apps from existing HPC applications. With these kernels/mini-apps it is easier to see the impact of vectorisation and memory placement. Once satisfied by performance, these changes can be backpropagated directly into the existing application to see the impact, at scale. Also, these kernels / mini-apps are well suited to be run in the early stage of the design of the processor onto existing simulators. Moreover, these snippets help to characterise code, and to see a/the common profile between CFD codes used in the project. This characterisation will help the design of future hardware.

As a consequence, for each of the code studied so far, we provide:

- **An identity card** describing the intrinsic characteristics of the application. This helps identify which parts of the software stack are utilised by the application. Thus, details are provided about the programming paradigms (MPI, OMP, Kokkos, etc.) and programming languages (C, C++, Fortran, etc.). Other information is also included, such as the datasets tested, the best compiler option to use, hot functions and the percentage of memory bound for each test case. The goal of this identity card is to provide a quick overview of the application.
- **Porting, profiling and optimisation information.** This part aims to describe both the maturity of the software stack and the level of optimisation of the (mini-) application w.r.t. a given test case. It includes the relative comparison onto various architecture (ARM, X86) performance of compilers and dependencies, information about scalability and any potential bottlenecks. A detailed analysis of vectorisation is also included, as well as any generic modifications/optimisations, through parameter activation/deactivation and the use of directives.

3.2 OpenFOAM

OpenFOAM is an open-source CFD software suite and library that includes a wide range of solvers for various types of simulations. These include, for example, complex fluid flows involving chemical reactions, turbulence, heat transfer, acoustics, solid mechanics, and more.

Code Repository	https://develop.openfoam.com/Development/openfoam.git
Version	Branch: OpenFOAM-v2312
Language(s)	C, C++
Paradigms	MPI

For OpenFOAM, two test cases 'Small' and 'Large', provided by the University of Ljubljana (UL), were used. Both test cases run the same simulation but at different resolutions, with 'Large' having the higher resolution. The 'Small' test case was too small to be parallelised, so all analyses for this case were performed on a single core.

	Small	**Large**
Compiler	GNU compilers 13.2.0	
Compilation flags	-O3 -floop-optimize -falign-loops -falign-labels -falign-functions -falign-jumps -mcpu=native -funroll-loops	
Runtime with IO	64.32s (1 cores)	1004s (1 cores) 20.52s (64 cores)
Runtime without IO	7.3s (1 cores)	159s (1 cores) 6.11s (64 cores)
Vectorisation	1.77%	
Hot Functions (without IO)	interpolateXY (13.67%) LduMatrix::Amul (11.81%) DICPreconditioner::precondition (11.44%) gaussGrad::gradf (10.77%) surfaceInterpolationScheme::dotInterpolate (4.97%)	DICPreconditioner::precondition (24.11%) gaussGrad::gradf (20.91%) LduMatrix::Amul (11.81%) surfaceInterpolationScheme::dotInterpolate (11.45%)
Memory Usage	328MB	3.25GB
SapphireRapids HBM speedup	+8%	+54%
Topdown (No-io)	Front End Bound: 17.98% Back-End Bound: 51.77% Retiring: 31.31% Bad Spedulation: 0.82%	Front End Bound: 14.69% Back-End Bound: 66.23% Retiring: 18.59% Bad Spedulation: 0.32%
Misses Per Kilo Instruction	L1D: 30.69 L2: 20.671 L3: 12.04	L1D: 9.4 L2: 2.46 L3: 10.9
Operation Mix	Load: 30.92% Store: 9.65% Integer: 28.68% NEON: 2.11% SVE: 0.52% Float: 11.46% Branch: 16.66%	Load: 35.20% Store: 10.70% Integer: 32.74% NEON: 0.87% SVE: 0.90% Float: 2.55% Branch: 17.04%

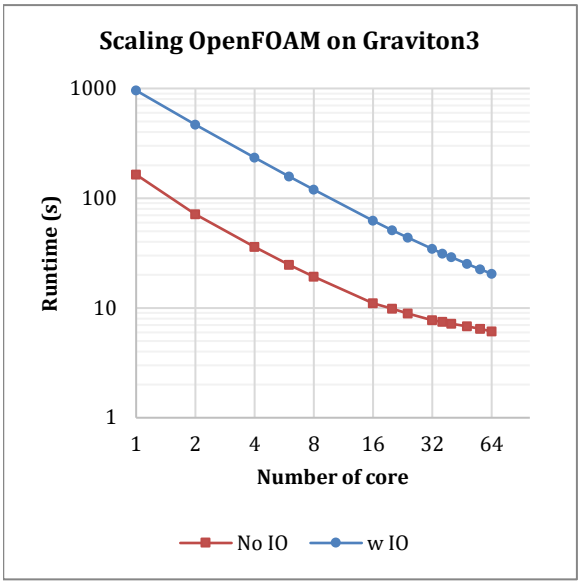
Table 2: OpenFOAM Identity card.

Extensive profiling was performed. In the top-down analysis, it can be observed that the two test cases show similar behaviour. The only significant difference lies in the levels of retiring and back-end bound instructions. Retiring instructions are those that complete successfully without encountering issues or being stalled by any component. A front-end bound state indicates that the processor pipeline cannot be fully utilised, while a back-end bound state suggests delays caused either by memory operations (memory bound) or by limitations in compute capacity (core bound). 'Bad speculation' refers to incorrect predictions or prefetches

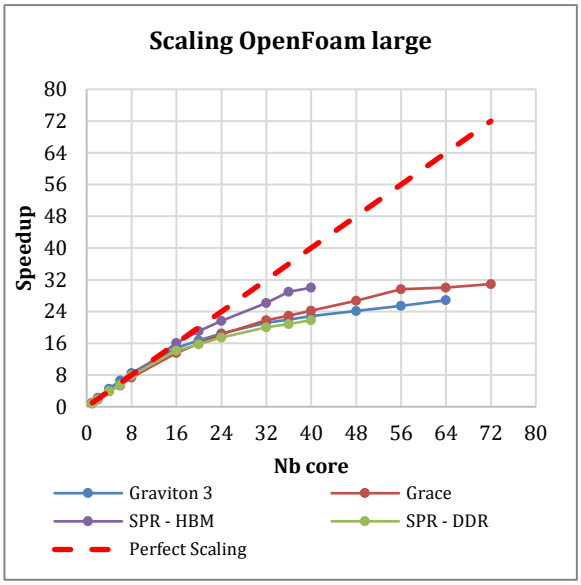
made by the out-of-order processor, which result in the need to recompute or fetch the correct data. Also, operation mix is quite similar, and the only observed difference come from the use of MPI.

When this analysis is combined with the observed speedup achieved using HBM on the 'Large Test case' (see Figure 15 right), it can be concluded that the larger test case is more memory-bound. This can be attributed to the larger mesh used, which involves more points and thus fills the cache more quickly. As a result, more frequent accesses to RAM are required. It should be also noticed that on the large test case, a correct scaling is observed. The left-hand side of Figure 15 depicts the scalability, on AArch64, with and without I/O. The trend of the plot is similar up to 32 to cores, then when no I/O are considered, some performance stall can be observed, probably due to a code that becomes more memory bounded. All these elements, tend to suggest that Rhea, with its high number of cores and HBM will perform well on these test cases.

Figure 15: **OpenFOAM runtime and scaling.**



I/O impacts performance significantly by Factor 1



Note that better speedup does not mean the runtime is better, it shows only how well the application is scaling from the sequential run

Vectorisation attempts were made and tested on both test cases, focusing mainly on Amul and the DICPreconditioner, which were identified as the two primary bottlenecks. Unfortunately, no success was achieved - at best, the performance remained unchanged, and in some cases, a performance loss was observed. It was found that the compiler was already making optimal choices regarding vectorisation and interleaving. The second loop was determined to be non-vectorizable due to data dependencies and indirect memory accesses, which could cause two consecutive iterations to access the same data. Essentially, each iteration could depend on itself or on previous iterations. An attempt was made to split this second loop to facilitate vectorisation, but none of the efforts were successful.

A domain partitioning study was conducted for the larger test case. It was found that the best performance was achieved using the partitioning scheme (1;1;X), where X represents the number of MPI processes. This approach evenly splits the domain along one dimension.

Various optimisations developed by Fujitsu for the A64FX CPU (https://github.com/fujitsu/oss-patches-for-a64fx/tree/master/OpenFOAM), which is based on the AArch64 microarchitecture derived from Neoverse-V1, were also tested. No performance improvements were observed on either the Graviton3 or the Grace Superchip. However,

considering that the A64FX also utilises HBM, these optimisations may still be of interest for Rhea.
Optimizing OpenFOAM is a complex task due to the underlying structure of the code. For the sake of genericity, OpenFOAM uses abstractions and indirection that constraints memory access and vectorisation. An in-depth optimisation is still in progress.

3.3 Neko

Neko is a library/framework for high-order spectral element flow simulations. It is written in modern Fortran and leverage on object-oriented concepts which allow for example the abstraction of the solver stack to facilitate usage on heterogeneous machines which use different kind of hardware.

Code Repository	https://github.com/ExtremeFLOW/neko
Version	N/A ; 9f03d7d41
Language(s)	Fortran
Paradigms	MPI, CUDA, HIP, OpenCL

In the context of the project, three test cases were provided in order to have an in-depth analysis. Neko_bk5 test case, can run on most mesh size and consist in a single kernel. Neko_opr test case represents a collection of seven math kernels extracted from the full application. Neko_TGV test case, a Taylor Green Vortices simulation, represent the full application. With these three test cases, in depth analysis, from the simplest to the hardest, can be conducted.

The table below depicts the identity card of Neko.

	Bk5	**Opr**	**TGV**
Compiler	GNU compilers 13.2.0		
Compilation flags	-g -mcpu=native -Ofast		-g -mcpu=native -O3** (**) some crashes were observed with -Ofast
Mesh used	8,192	8,192	32,768
lx value	10	10	
Figure of Merit used	GFLOPS	Iteration Time	WCT (sec)
FOM (all cores)	452.42	4.01E-02	514.04
Vectorisation	100%	80%	58.8%
Hot Functions (without IO)	ax_bk5_lx10 (96.2%) mca_btl_vader_component_progress* (3.04%)	cpu_dudxyz_lx10 (25%) cpu_opgrad_lx10 (12.62%) cos (12.62%) cpu_conv1_lx10 (7.94%) acos_finite (7.79%) mca_btl_vader_component_progress* (5%)	tnsr3d_cpu (17.6%) ax_helm_lx8 (17.18%) vlsc3 (6.64%) gs_gather_cpu (6.29%) cpu_opgrad_lx12 (6.05%) gs_scatter_cpu (5.79%)

		ax_helm_lx10 (4.84%) acos (4.7%) … lambda2op (1.46%)	cpu_dudxyz_lx8 (5.1%) … cpu_cdtp_lx8 (1.64%)
TopDown			
Memory Usage	2.7GB	3.1GB	39GB
SapphireRapids HBM speedup	+26.33%	+21.54%	+130%

Table 3: Neko identity card.

*: This is the function MPI use to progress every communication (P2P, Collective, Synchronisation)

First, the impact of the upcoming HBM in Rhea was evaluated. By using HBM on Sapphire Rapids, it was observed that Neko_bk5 and all Neko_opr kernels were no longer memory-bound, indicating that the full compute power of the CPU could be utilised. During the profiling of Neko_opr, opportunities for optimisation related to mathematical functions were identified. In the lambda2 function, a significant amount of time was found to be spent in calls to cos, acos, and acos_finite. These math calls were also found to hinder vectorisation, as no vectorised versions of these functions exist in the standard library. However, vectorised math libraries are available: by using SLEEF or ARMPL, making small modifications to the function, and applying a patched version of GCC, full vectorisation of the function was achieved. This significantly reduced the time spent in mathematical operations and led to a speedup of factor 2 for the function, and a speedup of factor 1.21 for the overall mini-app. Figure 16 depicts the gain on the various kernels.

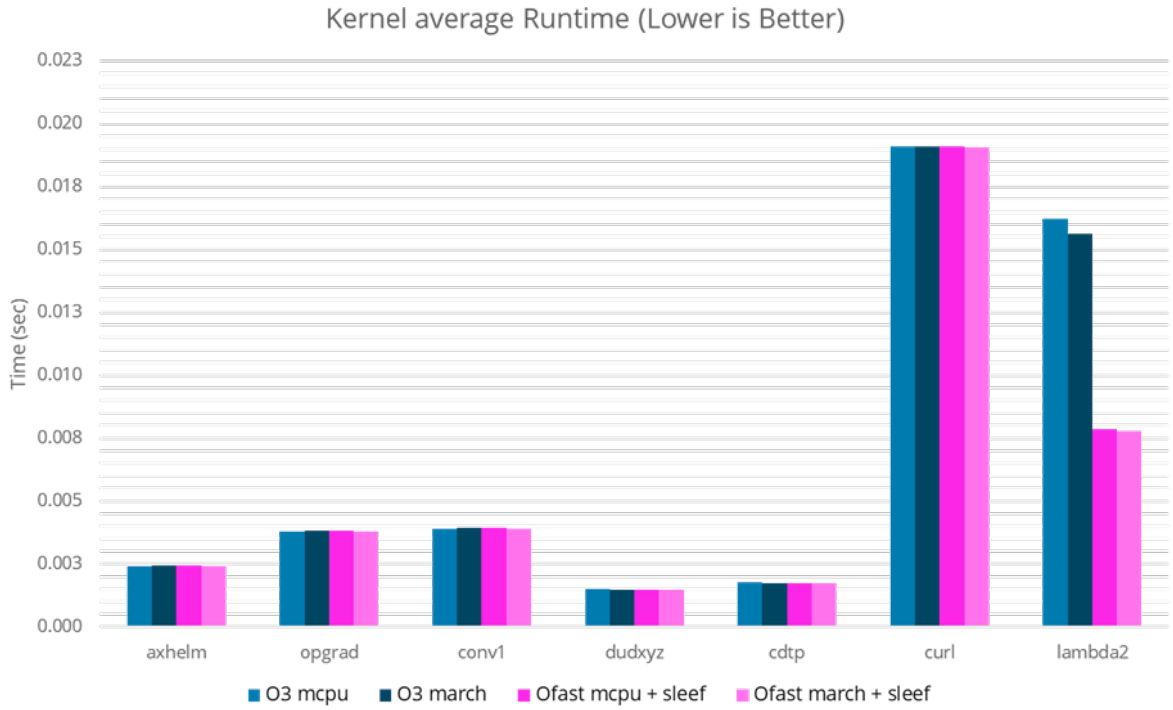


Figure 16: Neko_opr kernels runtime.

Unfortunately, this patch will not be upstreamed as it has been refused by the GCC maintainer (more details here) but this is available natively with the LLVM compiler. For now, llvm-flang cannot compile Neko, but the compilation is going further each month as flang is progressing fast.

In order to see the potential impact of the HBM in Rhea, some memory consumption analysis was made, on both Neko_bk5 & Neko_opr. Since results are similar, only the BK5 example is depicted here. Figure 17 shows the memory consumption w.r.t. mesh size and LX parameter. This analysis is essential to know, **a priori** (in advance), the size that the test case will occupy, since the HBM in Rhea is limited to 64 GB. Similar analysis will be done later for the TGV test case.

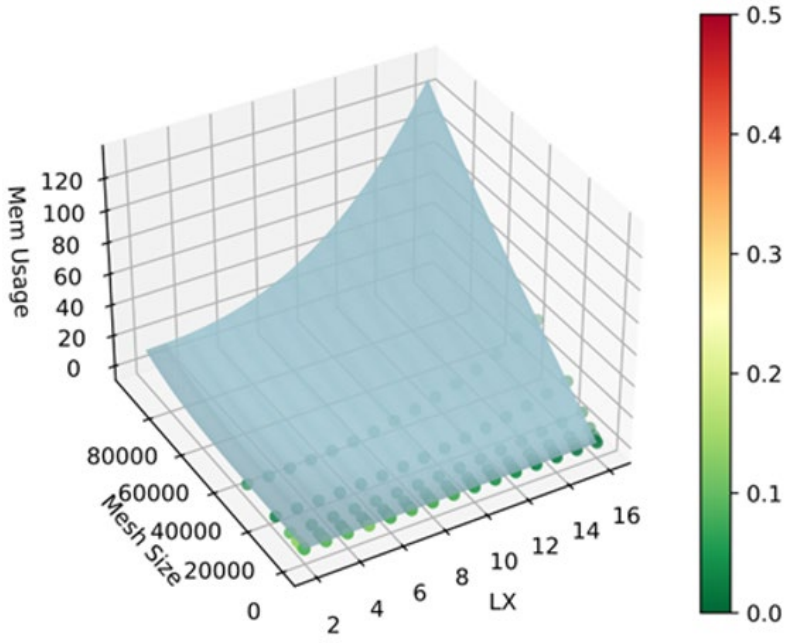


Figure 17: Neko_bk5 Memory Consumption projection.

Also, the memory access patterns have been studied for Neko_bk5. Most of these accesses are regular, and the prefetcher should be able to predict them to reduce the cache miss as much as possible.
Finally, some scalability analyses were made. Figure 18 shows the scalability, for the various test cases and for Neoverse-V1, below. It can be observed that as the complexity of the test case increases, its scalability decreases. This is due to the combination of multiple kernels, each exhibiting different scaling behaviours.

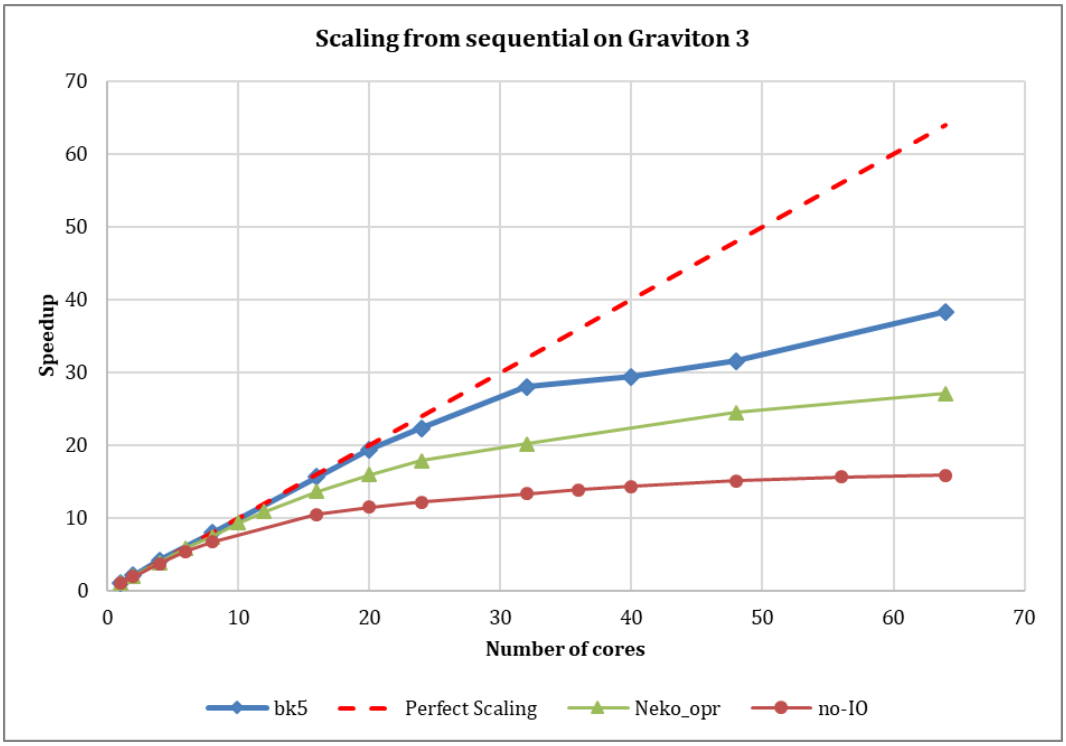


Figure 18: Neko scaling of all miniapp/cases on Graviton 3.

3.4 m-AIA

m-AIA is a multiphysics simulation tool that includes multiple solvers for, among others, Navier-Stokes equations and acoustic perturbation equations.

Code Repository	Open Source
Version	N/A
Language(s)	C++
Paradigms	None

An important part of the application is the lbcum solver that was extracted by the code owners and profiled by SiPearl. The extracted solver was originally sequential and invoked within MPI processes in the full application. A small addition of OpenMP and MPI support was made to the mini-app. Equivalent performance and results were obtained with both parallelisation paradigms.
Based on profiling feedback, the initial version was enhanced by the code owners. Consequently, several versions of the mini-app were made available: 'OLD', that represent the initial version provided by developers, 'SOA', a version using structure of arrays to pack memory accesses, SOA-VEC, that builds upon SOA but with some focus on vectorisation, and SOA-VEC-Template that generalises SOA-VEC. For the sake of clarity, in the following we reference by VEC (respectively SOA-VEC), the VEC-Template version (respectively SOA-

VEC-Template). It should be noted that VEC is handmade outer loop vectorisation that supports SSE, AVX/AVX2 & AVX512. VEC-Template benefits from the C++ template concepts. Since this template versions did not support AArch64, some efforts have been made to provide a specialised version for Neoverse architecture. Nonetheless, this vectorisation cannot be vector length agnostic due the fact that sve intrinsics/type cannot be used inside of union because their size is unknown at compile time. Consequently, the flag -msve-vector-bits is now required to compile on ARM. This flag is used to define the size of SVE register at compile time. The table below depicts the profile of all the aforementioned versions.

	Old	**Soa**	**Vec**	**Vec Template**
Compiler	GNU compilers 13.2.0			
Compilation flags	-march=native -g -O3 -fopenmp		-march=native -g -O3 -fopenmp -msve-vector-bits=256	
Number of cells	10000000			
Runtime (ms)	5705.94	4928.54	2805.53	2817.4
Vectorisation	0%	72%	100%	100%
Memory Usage	4.4GB	4.4GB	4.4GB	4.4GB
SapphireRapids HBM speedup	0%	11%	69%	54%

Table 4: m-AIA identity card.

It should be noted that SiPearl has a patch for automatic OuterLoop Vectorisation inside of LLVM (see pull request here). When using this patch on the SOA version, similar performance than the VEC version is obtained, but with the advantage of being more flexible and target independent.

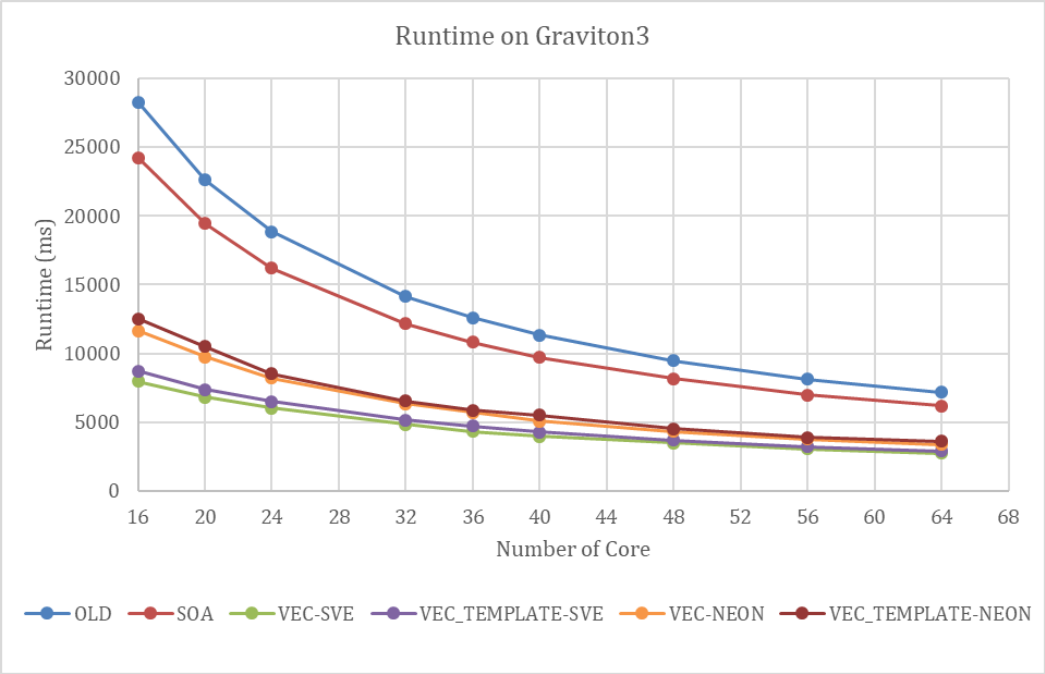


Figure 19: Runtime of all implementations on Graviton3.

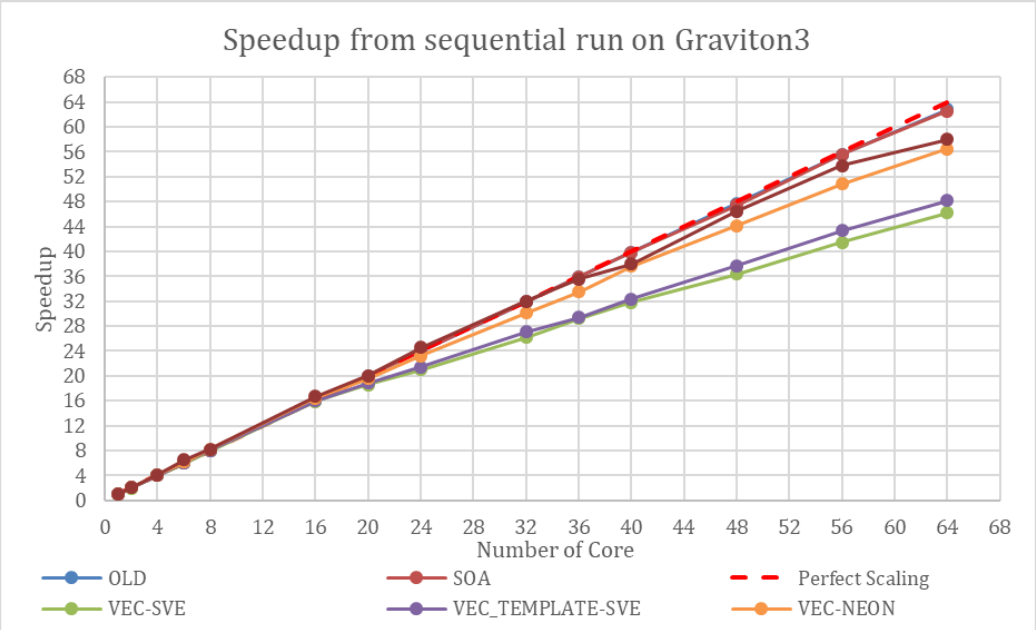


Figure 20: Speedup from Sequential on SapphireRapids with HBM or DDR.

Some experiments have been also carried out to analyse the impact of HBM. Figure 20 depicts the impact of such memory on SapphireRapids. As shown, when HBM is used, the versions that make intensive use of compute capabilities (i.e., the most vectorised versions) are found to be significantly less memory-bound. This explains the substantial speedup observed with HBM and this is promising for Rhea.

3.5 STREAmS

STREAmS solves the unsteady Navier-Stokes equations for perfect gases, to perform numerical simulation of compressible turbulent flows. STREAmS shares algorithmic similarities with FLEW and uses the same development concepts. The current version of STREAmS includes all the FLEW functionalities.

Code Repository	N/A
Version	July 2024
Language(s)	Fortran
Paradigms	MPI/OpenMP

Two versions of the code are considered in this section: ‘CPU’ that used only MPI, and ‘OMP’ that uses only OpenMP.

Version	**CPU**	**OMP**
Compiler	GNU compilers 13.2.0	
Compilation flags	-O3 -fopenmp -march=native -g	
Domain Size	320x240x320	
Average Iteration time in second	0.92	0.83
Vectorisation	21.71%	

Hot Functions	visflx_nosensor_subroutine 23.28% euler_z_hybrid_kernel 14.17% euler_y_hybrid_kernel 13.16% euler_x_fluxes_hybrid_kernel 12.51% exp 7.17% pow 6.92% visflx_subroutine 2.71% visflx_div_subroutine 2.25% init_flux_subroutine 2.00%	
Memory Usage	10.6GB	
SapphireRapids HBM speedup	+43.82%	+20.79*%

Table 5: STREAmS identity card.

**: The speedup is lower for the OpenMP version since Sapphire Rapids have 4 numa nodes per socket and allocation is done outside of an OpenMP parallel region. This creates a numa effect that could be removed by using at least one MPI per numa node. This combination of both versions (MPI and OMP) is called HYB in the following.*

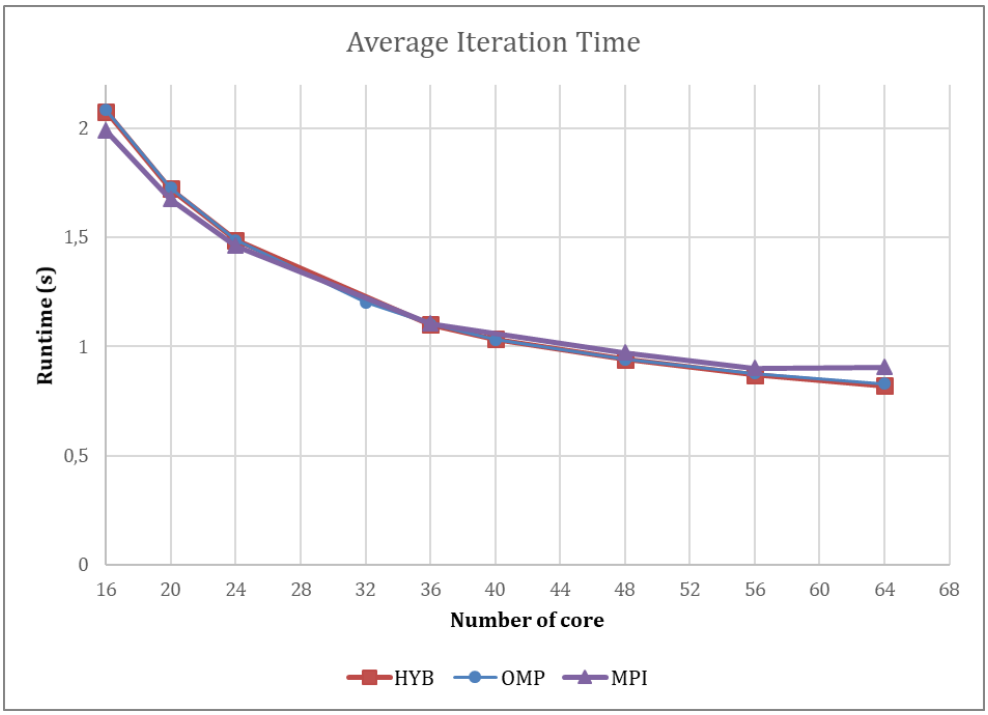


Figure 21: STREAmS average iteration time.

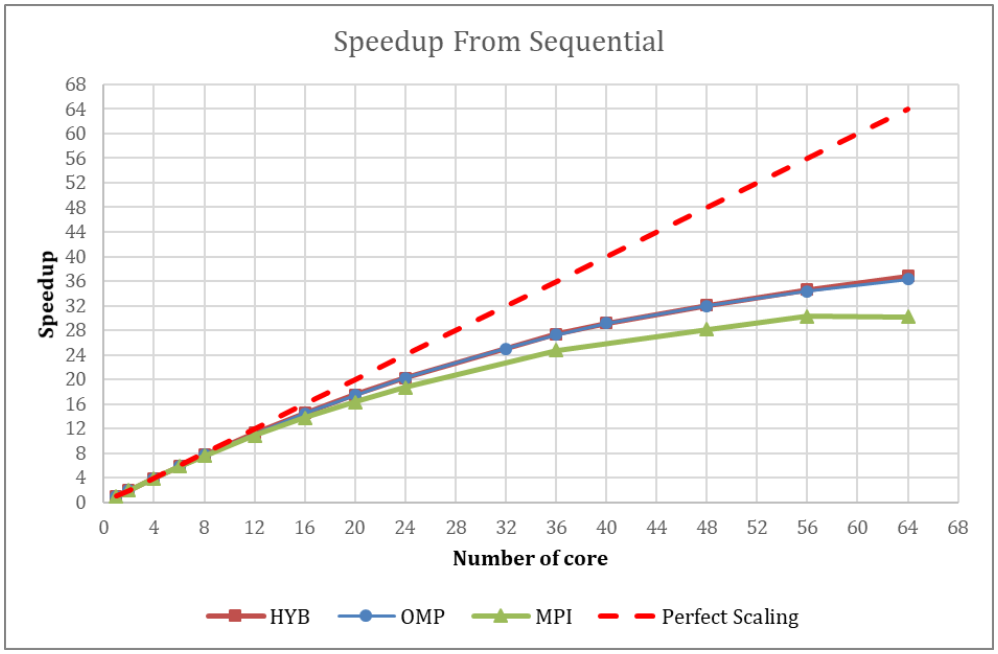


Figure 22: STREAmS Speedup from sequential on Graviton 3.

Figure 21 and Figure 22 depict the scalability of the three versions in both Average Iteration Time and speedup based on the sequential version. It should be noted that the HYB is built upon the best combination of OpenMP(OMP) and MPI for each point. Also, small code modifications were made to improve slightly the performances and increase the amount of vectorisation. For example, the addition on OpenMP Single Intruction Multiple Data (SIMD) directives and the reordering of loop to ensure consecutive memory access. This helps the compiler to vectorise innermost loops. The use of Automatic Outer Loop vectorisation was also used to vectorise the visflx_nosensor_subroutine. These two modifications slightly increase the code vectorisation, bringing it to 23.43%.

The impact on performance is still being evaluated: for the moment the performance is the same with or without the outer loop vectorisation. The line:

```
cploc(i) = cploc(i)+cp_coeff_cpu(ll)*(tt(i)/t0)**ll
```

contains an exponential but AArch64 micro architecture does not have a vectorised exponential instruction. Some approximation could be made but at the cost of precision loss. Thus, the SLEEF library was used, as it provides implementations for vectorised mathematical functions. However, this introduced another issue: in this case, the left-hand side of the exponential operation is a floating-point value, while the right-hand side is an integer. Even within SLEEF, no such exponential function exists for this combination. A temporary workaround was applied by casting the integer to a floating-point type. However, this made the exponential computation more costly. An internal implementation of such an exponential function is currently being developed, which is expected to be faster than the full floating-point version. Performance improvements are anticipated as a result.

3.6 Co-Design Service for exaSim project

The task was also opened for service to the exaSim project, funded by the BMBF in Germany. The applications NeoFOAM and NeoN were target of the co-design services for benchmarking, profiling, and performance engineering on Nvidia's Grace Hopper 200 (GH200) CPU-GPU and AMD's MI300 APU. The former Grace CPU is also ARM-based and was initially proposed by SiPearl as a good proxy system to their Rhea architecture. The latter is of particular interest at HLRS because of the new system Hunter that utilises the AMD APUs for which data

management is unified on the hardware level. Hence, applications can and should omit additional data copies to and from a GPU as device.

NeoFOAM/NeoN from the exaSim project is an effort to port core PDE solver functionality to the GPU while relying on the well-established and industry-relevant DSL of OpenFOAM. Hence, NeoFOAM is used as an adapter in which user-coding of OpenFOAM cases ties OpenFOAM and NeoN together where NeoN can be expanded to gradually replace the compute intensive parts by accelerated kernels.

Hence, the first test case for co-design services used the scalarAdvection case of NeoFOAM. We used LLVM/clang compiler flags to check for vectorisation potential. It was found that some core device code showed missing vectorisation in loops that are often reused for acceleration through threading when used on the Grace CPU only. Resulting optimisation to enable compiler-based vectorisation yielded a performance gain of 3% on the Grace CPU.

Secondly, the scalarAdvection case was used for GPU profiling with Nvidia's Nsight Systems. Here, highly abundant allocations on the Hopper GPU could be identified and removed after fully using NeoN's DSL (PR #38). Moreover, the hierarchy of GPU kernel utilisation was identified as 38% setField, 31% computeDiv, 18% interpolate, 7% fieldBinaryOp, and 5% scalarMul. The major bottleneck setField is filling arrays with given values like 0 for initialisation. This leaves room for optimisation regarding the memory management.

Profiling the test case on CPU and GPU together revealed that the calculation of the Courant number was still using CPU kernels, majorly contributed to the overall runtime (30% and more), and could be easily ported to GPU using NeoN's accelerated loops. We measured the average Finite Volume Operations per Second (FVOPS) with varying case size in terms of grid elements using the GPU and CPU versions of the CFL number calculations on the GH200 and MI300A.

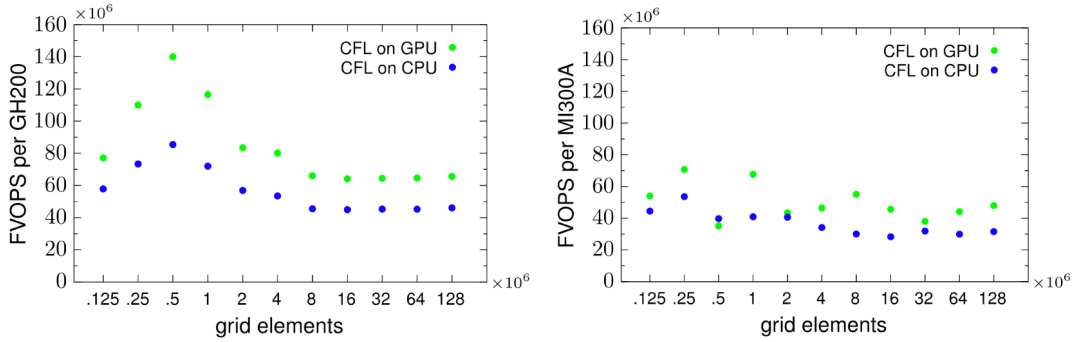


Figure 23: FVOPS for different case sizes on GH200 (left) and MI300A (right).

The graphics in Figure 23 show that the GPU ported CFL calculation improves FVOPS by up to 75% at the performance peak on the GH200 at 500k grid elements. On the MI300A we could not observe a similar peak and the GPU implementation of the CFL number only shows performance gains for particular case sizes. Here, further investigation regarding the accelerated loops on the MI300A is required.

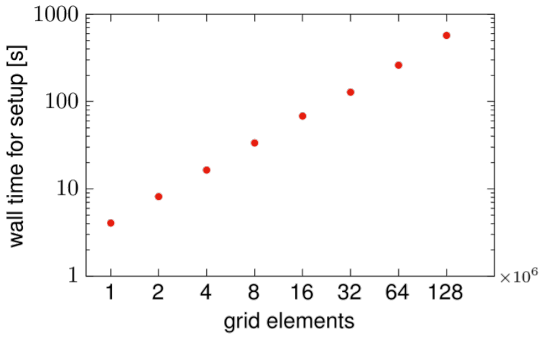


Figure 24: Wall time for initial case setup for different case sizes.

Finally, the wall time for the initial case setup was investigated (see Figure 24) because it was quickly found, that it would noticeably increase with larger cases. Here, NeoFOAM relies on the legacy I/O of OpenFOAM and then copies the data to NeoFOAM and NeoN data containers. OpenFOAM's I/O is particularly not suited for high performance because the files are typically loaded on line-by-line basis or even by individual characters. Thus, a NeoFOAM native I/O functionality is recommended to cut the I/O costs by reading full arrays reducing I/O operations and to avoid data duplication overheads.

4 Task 3.3: Testing, Validation and Deployment

As the workflows comprise multiple components that may run on potentially diverse hardware, the validation and benchmarking results will span from individual component assessments to full-scale simulations and complete workflow rounds.

In summary, this task comprises three core aspects, revolving around a unified testing platform serving the purposes of validation, deployment, and benchmarking. The specific definition and execution of this testing platform, as well as the overall approach of Task 3.3, has been dependent on the input and direction provided by the CASTIEL 2 project managing the coordination and support for National Competence Centres (NCC) and Centres of Excellence (CoE) on a European level.

The evolutions of the CASTIEL 2 project defined a standardised way to perform testing, validation and deployment on most hosting entities part of the EuroHPC JU alliance. In particular, applications should adopt a Gitlab Runners solution for testing and validation duties and rely on EESSI for deployment. Following these directives, we started adapting the procedures already in place, also overviewed in deliverable D3.1. Our first candidate for the adoption of the tools for testing, validation and deployment is the Alya application, part of UC4. Alya already employs Gitlab Runners for testing and validation, but it does not rely on EESSI for deployment. Our efforts to switch to EESSI found a limitation in the closeness of the source of the application, which requires an explicit collaboration agreement. While there are many possibilities to handle this issue (like treating the closed source as an external dependency), they take time to implement, as they also involve legal disputes over the software licenses. Alongside the work on the Alya application, we are considering the AVBP one part of UC2 since it also uses Gitlab Runners for automated testing and validation. For this application, the adoption of EESSI can follow a different pattern, as we can rely on the containerised solution already present. In the upcoming month, we envision the adoption of Gitlab Runners and EESSI for both applications, and we may extend the interaction considering an additional application.

Alongside these efforts aimed at adopting the tools defined by CASTIEL 2, we focused on the introduction of an ad-hoc pipeline for automatic testing, validation and benchmarking for the

STREAmS application, which was lacking any mechanism for it. The following subsection covers the developments in that direction.

4.1 STREAmS automatic testing, validation and benchmarking

Testing and deployment

At present, the STREAmS application has been deployed and tested on ten HPC partitions, both CPU- and GPU-based: 8 of them are EuroHPC JU partitions, while 2 are external, one (Hunter Supercomputer) in Germany and one (Aurora) in the United States. The benchmarking campaign is still in progress. As for the deployment of the in-situ part, given the effort related to compilation, it has been done on the systems for which concrete use is expected. In particular, the deployment on LUMI-G is of relevance given the availability of hours provided within the last EuroHPC extreme project (EHPC-EXT-2024E02-130) awarded for EXCELLERAT simulations. Table 6 summarises the deployment status of STREAmS on the targeted systems.

	Architecture	Backend	Tested	Benchmarked	In situ	Production
Leonardo-Booster	A100	CUDA Fortran	y	y	y	ongoing
Leonardo-DCGP	Sapphire	CPU & OpenMP	y	ongoing		
MareNostrum5-ACC	H100	CUDA Fortran	y	y	y	ongoing
MareNostrum5-GPP	Sapphire	CPU & OpenMP	y	y		ongoing
LUMI-G	MI250X	HIP	y	y	y	**ongoing, EuroHPC extreme: 1M node-hours for EXCELLERAT runs**
LUMI-C	Zen3	CPU & OpenMP	y	y	y	
Hunter	MI300A	HIP	y	ongoing		
Aurora	1550	OpenMP-offload	y	ongoing		
MeluXina	A100	CUDA Fortran	y	y	y	completed
JEDI (JUPITER)	GH200	CUDA Fortran	y	ongoing		

Table 6: Deployment summary of STREAmS on the different HPC resources tested.

The columns report system name, type, architecture, adopted STREAmS backends, and the statuses of test, benchmark, in situ and production activities.

Benchmarking automation

To facilitate performance testing of STREAmS, we have significantly enhanced our TEBE (TEsting Benchmarking Engineering code) tool. Somewhat inspired by the JUBE Benchmarking Environment [14], TEBE allows us to prepare, manage, submit, and analyse benchmarking cases of STREAmS and potentially other codes as well. TEBE's architecture has two layers: the first layer implements generic functionality while the second layer contains the connection layers between TEBE and different codes. This second layer allows TEBE to properly interact with the code to be benchmarked, thus defining the set of test simulations in a compact way using a simple syntax based on algebraic operators. As an example, in Figure 25 we show a part of TEBE input where we define weak scaling cases (params-1 dictionary), strong scaling cases (*params-2* dictionary), and cases at varying threshold of shock-capturing

patterns activation (*params-3* dictionary). The initial *params_expression* formula combines the three dictionaries so that all cases of strong and weak scaling for different convective patterns are generated.

```
param_expression  = params_1*params_3|params_2*params_3

[params_1]
flow-Reynolds=157213.3,208707.53,276903.84,367190.36,486531.04
mpi-x_split=1,1,2,4,4
mpi-z_split=1,2,2,2,4
grid-nxmax=2688,3200,4096,5632,7168
grid-nymax=328,398,486,582,710
grid-nzmax=156,216,276,336,432

[params_2]
flow-Reynolds=486531.04,486531.04,486531.04,486531.04,486531.04
mpi-x_split=4,8,16,32,64
mpi-z_split=4,4,4,4,4
grid-nxmax=7168,7168,7168,7168,7168
grid-nymax=710,710,710,710,710
grid-nzmax=432,432,432,432,432

[params_3]
numerics-sensor_threshold = 2.0,-1.0
```

Figure 25: Example input section defining strong and weak scaling using TEBE benchmarking tool.

After creating cases, TEBE manages the submission to the queuing system (supporting SLURM and PBS schedulers) via templates of submission scripts that can be adapted to the system under consideration. TEBE also allows results to be extracted and saved in tables that can then be used for analysis or visualisation. TEBE is implemented in Python and has been installed and used on all the systems tested, drastically reducing the effort of managing the many test cases on the different machines.

Continuous Integration

The past EuroHPC project EHPC-EXT-2023E01-034, based on the use of the STREAmS solver, has been selected among the 15 projects of the highly ranked Extreme scale, AI and AI Boost projects and the strategic Destination Earth project to join JUREAP, the JUPITER Research and Early Access Program. This allowed us to test STREAmS on JEDI, the preparatory system for JUPITER. In this context, we also worked to ensure that STREAmS was connected to the Continuous Integration system developed for JUREAP. Since STREAmS did not have any form of CI, this allowed us to start this development in a way that was also appropriate for the context of a EuroHPC system. The used technology is based on four basic elements:

- *GitLab runner*: the agent that picks up a CI job, runs it as defined in the pipeline configuration file and sends the result to the GitLab instance.
- *JACAMAR CI*: the CI/CD driver for HPC that uses GitLab's custom executor model.
- *exaCB*: a framework containing a set of tools and configurations to enable Continuous Benchmarking (CB) through GitLab on JSC systems
- *JUBE*: a benchmarking environment that provides a script-based framework to easily create benchmark sets. Additionally, a component is under development to use our TEBE tool instead of JUBE, which allows for more agile management of STREAmS configuration sets.

A preliminary Continuous Benchmarking (CB)-oriented pipeline is currently working on JEDI, and it is certainly a useful proof-of-concept. Further development is needed to make it more useful in the real context of STREAmS development and testing. In particular, the evolution of

the pipeline may depend on what is established within the framework of EXCELLERAT and CASTIEL 2.

5 Task 3.4: Exascale Engineering

In the previous deliverable (WP3-D3.1) some progress has been reported towards reaching large-scale readiness required in exascale simulations. This early work requires the direct interaction of the application's user/developer, the HPC centre's operation and software support staff as well as the HPC centre's on-site staff to ensure an efficient execution of the run. Since achieving such a task is challenging, , Task 3.4 continues the work that has been done so far and has been reported WP3-D3.1.

5.1 CODA

During the reporting period, two main tasks were carried out: First, we evaluated the performance and scalability of CODA, FlowSimulator and Spliss delivered in container images on the DLR HPC systems and compared them to the installations using the native software stack of the system. Second, we evaluated the newly developed hierarchical mesh partition method in FlowSimulator.

First, CODA, FlowSimulator, Spliss and all the workflow dependencies can now be delivered as a single container image. Thus, CODA can be executed in a container on any HPC system independently of the installed software stack. The main benefits are easier delivery to users and customers, easier deployment on different systems and significantly improved portability. An evaluation of the container images with UC-1 on the DLR HPC systems showed comparable performance and scalability to the installation using the native software stack of the system. Thus, the delivery and deployment of the containerised workflow will be become the default for the next internal releases.

Second, after investigating several improvements to mesh partitioning in the previous reporting period, we have now evaluated a newly developed hierarchical partitioning method. The hierarchical partitioning method distributes the mesh at three levels: first, it distributes the mesh across all involved compute nodes, then within each node across all MPI processes, and finally for each MPI process across all threads. This method is flexible to use different graph partitioners such as Parmetis or Zoltan for each level. This method significantly speeds up mesh partitioning for large meshes and large numbers of cores (up to one order of magnitude), while not degrading the resulting load balance for the CFD solver. The improved mesh partitioning also allows simulations of larger meshes with more than one billion elements and we are now able to successfully run simulations on 131,072 cores (the full DLR CARO system).

5.2 AVBP

Efforts to bring AVBP to exascale-ready performance have focused on the portability of the code for AMD GPUs. Indeed, at the moment the largest clusters of the EuroHPC JU and in the world are equipped with AMD Mi250 GPUs and MI300A APUs.

An effort was done to try using OpenMP instead of OpenACC through automatic translation tools, since it was hinted by AMD that this should lead to better performance on their hardware and would also allow usage of Intel GPUs. However, OpenMP does not possess equivalence for all OpenACC constructs and handles management of complex data structure differently, notably derived types cannot be partially uploaded on the GPU memory.

Experiments with AMD's compiler were unsuccessful as it suffers from multiple bugs when trying to generate OpenMP GPU kernels from AVBP's code.

Surprisingly, compilation and execution on Nvidia hardware using NVHPC compiler were successful, although with a strong performance penalty which mainly comes from implicit memory copies between the host and the device.

The code supports GPU acceleration using the OpenACC framework. Therefore, compatibility for now remains limited to HPE systems equipped with the CRAY compiler suite. Fortunately, we have access to two of those systems in Europe, the ADASTRA Tier 1 system from GENCI/CINES in Montpellier France and LUMI G at CSC Finland.

A previous prototype implementation using OpenACC on AMD was able to run large non-reactive simulations (Figure 26). Benefitting from an access to ADASTRA new MI300A partition, we resumed work on this prototype. We managed to identify three major issues from the Cray compiler in its OpenACC implementation

- The !$ACC LOOP SEQ directive does not work
- The !$ACC KERNELS construct is broken in some cases
- The !$ACC DECLARE directive is also broken

We were able to implement workarounds for those issues and managed to run H2 burner reactive simulations at large scale using up to 96 of the 112 MI300A APUs on ADASTRA. Figure 27 shows the performance using various mesh sizes. Overall AVBP exhibits similar behaviour than on Nvidia hardware, with good strong scalability and excellent weak scalability. However, the overall performance of AMD cards is still underwhelming in this case, a single MI300A providing less than twice the performance of an A30 card, while having six times the memory bandwidth. Further work will investigate this performance gap, but overall, we are essentially limited by the compilers at the moment.

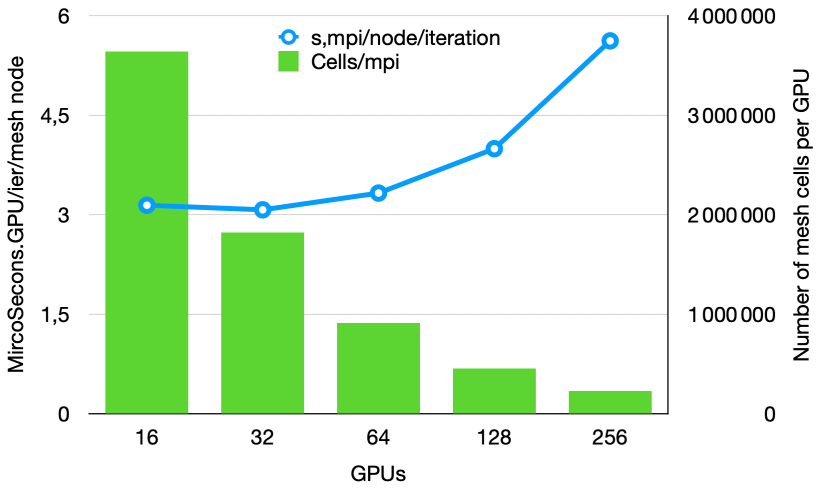


Figure 26: Strong scaling of AVBP on the ADASTRA system using 4 Mi250 per node. Nonreactive windfarm case.

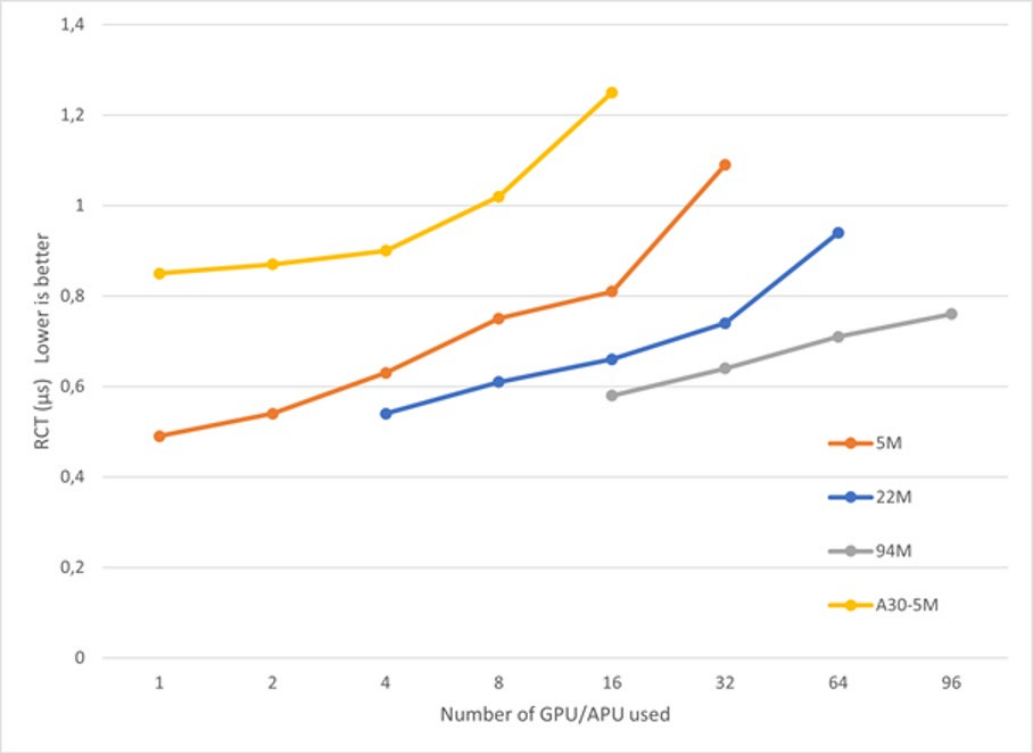


Figure 27: Scaling of AVBP on the ADASTRA system using 4 MI300A per node + comparison with CERFACS' A30 nodes. H2 burner reactive case.

5.3 m-AIA

The m-AIA code has been thoroughly tested on the HAWK supercomputer at HLRS, and deployments on the EuroHPC systems Vega, MeluXina, Discoverer and Karolina were successfully carried out. Strong and weak scalings for the m-AIA code using coupled FV-DG benchmarks have been performed. Three different configurations concerning the number of MPI ranks used per node and usage of OpenMP threads have been tested to compare the achievable performance. That is, 64 MPI ranks are used with each 1 or 2 OpenMP threads, and 128 MPI ranks with each 1 thread are placed on a compute node in different runs. All runs have been repeated at least two times in different job allocations and the best runs in terms of performance were selected. For the strong scaling a benchmark with about 150 million FV cells and 130 million degrees of freedom for the DG solver was used. As a baseline for the scaling a minimum of 4 compute nodes are used due to memory constraints. As evident by the curves in Figure 28 showing the total time to complete a timestep the m-AIA code was able to achieve excellent scalability up to the maximum of 256 or 512 compute nodes used during testing, while for each system the best performing configuration is shown. The weak scalability for m-AIA was tested using a coupled FV-DG benchmark with around 2.1 million cells per node. The time required per timestep is shown in Figure 28, starting at one node the scaling is performed up to 256 nodes, which corresponds to a problem size of 500 million cells in total. Overall, a good weak scalability is achieved when using 128 MPI ranks per node with the communication overhead increasing only slowly with higher node numbers. As evident, the tested HPC systems all achieve a comparable performance, while pronounced differences are only visible for 256 nodes. In summary, the scaling results show a very good strong scalability and a good weak scalability for the m-AIA code on the four tested EuroHPC systems.

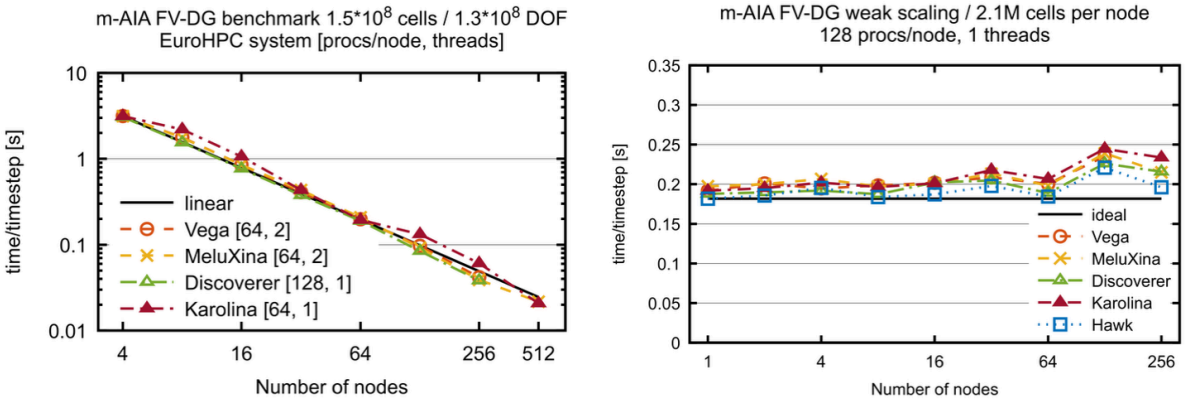


Figure 28: Strong (left) and weak (right) scalings for coupled FV-DG m-AIA benchmarks on different EuroHPC CPU based systems.

In the first project year, the strong scalability of a realistic coupled CFD/CAA chevron jet application with m-AIA on HAWK has been demonstrated. The setup included about 300 million CFD cells and 1 billion CAA DoF, representing a smaller scale run according to the exascale execution profile defined in WP2 for UC-3. The code showed excellent scalability when going from 2048 up to 262144 MPI processes, i.e., the maximum allocation size on HAWK, achieving about 86 simulation timesteps per second compared to 0.68 for the baseline. Building upon that, since the second year of the project and in collaboration with HLRS a large-scale chevron nozzle CFD simulation of 3.7 billion cells was initially tested and then run on 2048 nodes of the HAWK system. Furthermore, the corresponding coupled CFD-CAA simulation, adding 4.9 billion DOF for the CAA part, was run in an XXL session on the full HAWK system using 4096 nodes totalling to 524k CPUs. The workload distribution for the coupled case is compared in Figure 29 on 256 and 4096 nodes, using 64 MPI processes per node. The dynamic load balancing approach in m-AIA is used to distribute the workload among processes. At high node counts it becomes increasingly difficult to eliminate all imbalances, due to very small partition sizes, while simultaneously the per-process performance benefits from a reduced memory footprint resulting in improved cache usage. Figure 30 shows the scalability of the large-scale simulations, compared to baseline runs on 256 nodes a nearly linear speedup is obtained on 2048 and 4096 nodes of Hawk, i.e., a full HPC system can be used effectively for aeroacoustic predictions with m-AIA. The results show that the numerical approach is highly scalable, and it enables the execution of large-scale use cases on a pre-exascale level.

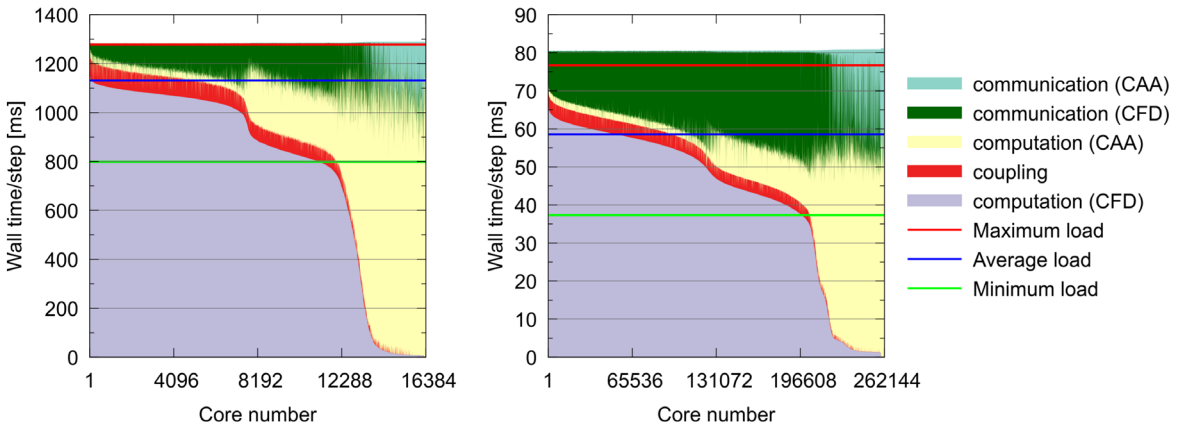


Figure 29: Comparison of workload distribution of large-scale coupled CFD-CAA simulation on 256 and 4096 Hawk nodes.

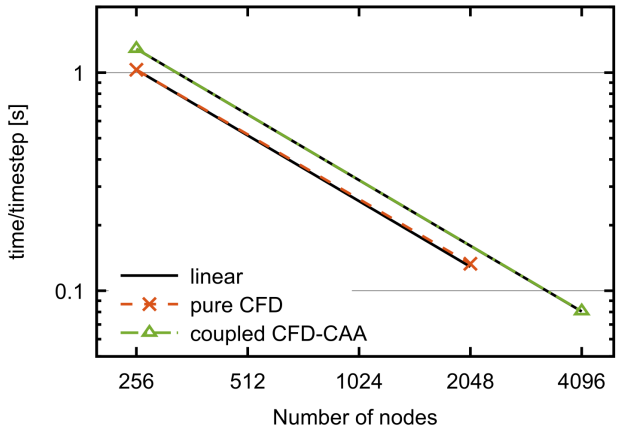


Figure 30: Scalability of large-scale CFD and coupled CFD-CAA simulations on Hawk.

5.4 Alya

Most HPC applications operate inside a fixed resource allocation which cannot be adjusted at runtime. This work revolves around integrating the computational mechanics simulator Alya with the malleability framework DMR to enable physics simulations which can resize at runtime to operate inside a desired efficiency range. In a previous work [15], we have developed a workflow to ensure a target communication efficiency, as shown in Figure 31.

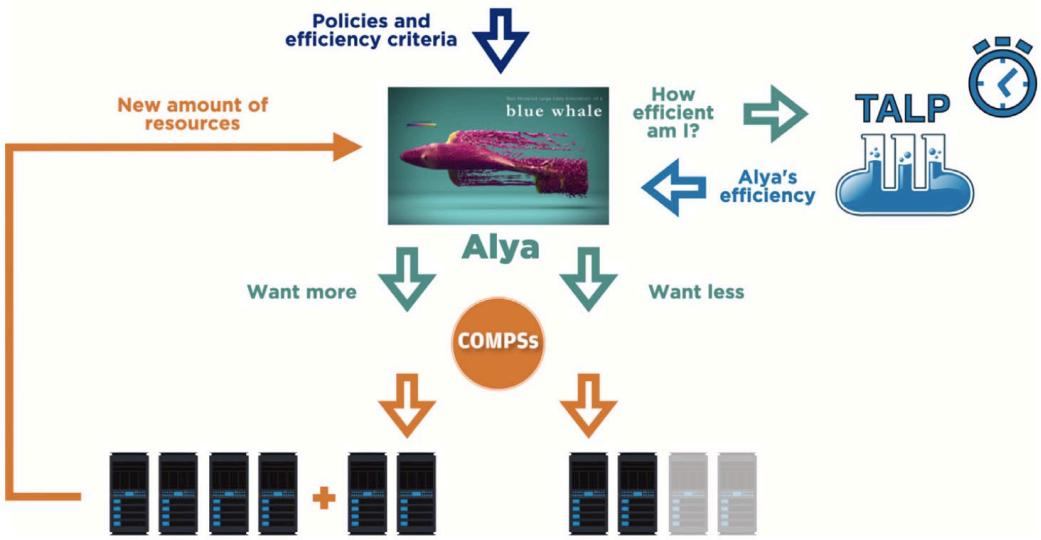


Figure 31: Optimizing the resources. Workflow for elastic computing of CFD simulations, involving different codes and libraries: Alya (CFD), TALP (efficiency measures) and COMPSs (elastic computing).

The workflow ensures an elastic computing methodology that adapts at runtime the resources allocated to a simulation automatically. The criterion to control the required resources is based on a runtime measure of the communication efficiency of the execution. According to some analytical estimates, the resources are then expanded or reduced to fulfil this criterion and eventually execute an efficient simulation. The methodology was based on the CFD code Alya together with a runtime library TALP [16] to measure performance metrics, and finally COMPSs to orchestrate the workflow and interact with SLURM workload manager.

The work proposed here follows a different strategy, although the main objectives are maintained, that is to ensure a parallel efficient simulation. The strategy is now based on DMR runtime, which handles the MPI communicator and oversees expanding or reducing the resources. In this new approach, TALP is now integrated in DMR library [17,18], thus simplifying the interactions of the CFD code Alya and the computing environment. The workflow is illustrated in Figure 32.

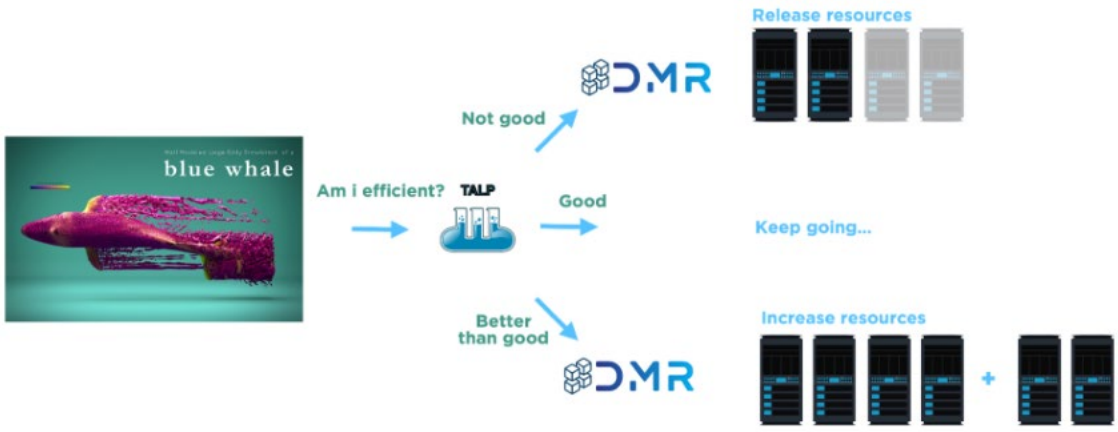


Figure 32: Workflow using Alya and DMR to control the communication efficiency.

In a first stage, we have worked on the interfacing of DMR with Fortran language, as DMR which was originally written in C. A mini-app reproducing the workflow of Alya has been finalised and tested. The library together with the mini-app have been containerised and can be found here [19].

Task 3.4 has several objectives, one of which is to focus on achieving optimal efficiency and performance in large-scale workflows. The proposed flexible workflow is designed to dynamically manage parallel efficiency during runtime by selecting the appropriate resources based on metrics such as communication efficiency and load balance. Predicting the efficiency of a simulation beforehand is challenging, primarily because strong scalabilities are typically evaluated relative to a baseline. If the baseline is already in an unfavourable state, this approach can yield highly inaccurate results. Furthermore, a priori strong scalability tests fail to provide insights into how parallel efficiency deteriorates. Parallel efficiency is affected by two key factors: load balance and communication efficiency, each of which can be addressed using distinct methodologies. Load balance issues can be rectified through redistribution or the utilisation of runtime mechanisms at the node level, such as DLB. On the other hand, communication efficiency can be enhanced through the strategy proposed here, which involves resource control, improved scheduling, or better repartitioning strategies, among other techniques.

The task description specifies that *"These simulations require the direct interaction of the application's user/developer, the HPC centre's operation and software support staff as well as the HPC centre's on-site staff"*. Specifically, the work on malleability requires active cooperation between the CFD code developer (Alya) and the runtime developer (DMR), and their interactions with the support team, especially since DMR interfaces closely with SLURM. To this end, DMR has been fully integrated in Alya, through 3 branches, to:

- Create the Fortran wrapper and adapt the build of Alya.
  - Commit hash: c35f9796d303c5328ffe9a0f38764402ecd8f427
- Clean the interface and debugging.
  - Commit hash: 9c9c06d00e8d7e3e2ce9d3d728c2e24c3eb6516c
- Add tests and include automatic prediction of the target nodes.
  - Commit hash: 2b12387fa7d6e78a90b69685541df0b1574056d0

On the DMR side, the library was refactored to integrate new OpenMPI implementations and change the spawn strategy.

Figure 33 shows an instance of Alya running with DMR when applying the reconfiguration policy described. The target communication efficiency area is shaded in grey. Process count starts around 1000 and is increased until CE drops, then process count is reduced, and CE remains within the target region. Results have been presented at the 17th Joint Laboratory for Extreme Scale Computing (JLESC) Workshop, 13 - 15 May 2025, that took place at the Argonne National Laboratory (ANL) in Argonne (USA).

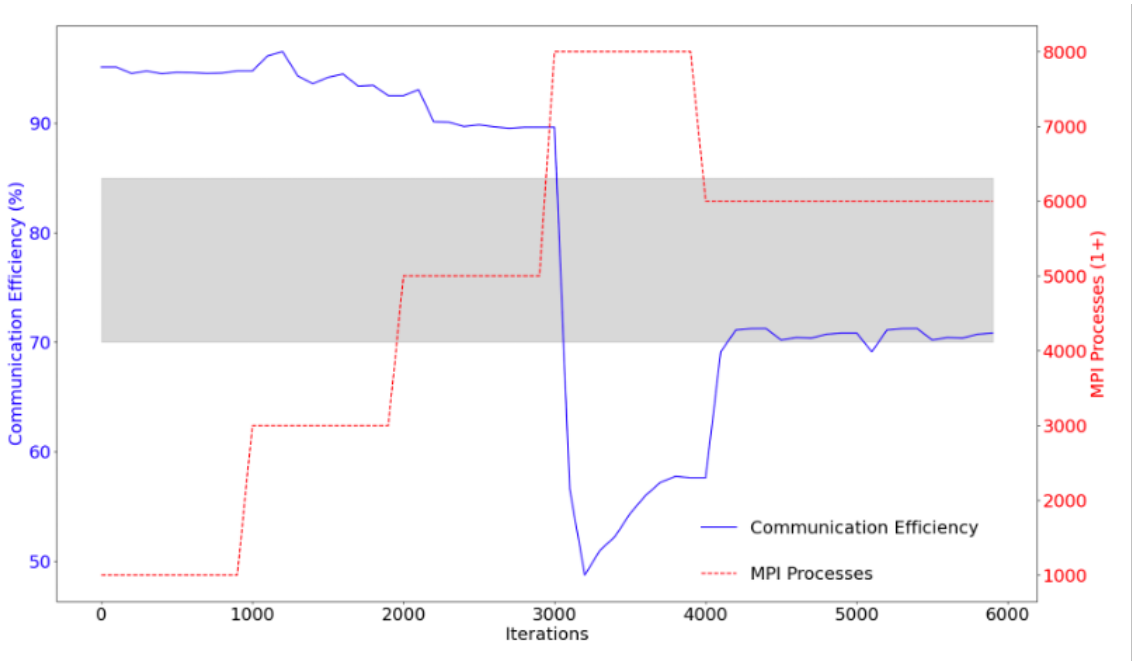


Figure 33: Dynamic resizing of Alya using DMR.

Next, the DMR library was tested using the same case and same target efficiency as before but allowing six jobs to run concurrently. The methodology thus consists of a self-adapting batch system. Figure 34 shows the evolution of the distribution of cores allocated to the different jobs. The discontinuous dark line shows the total amount of resources used while maintaining the CE in the target range.

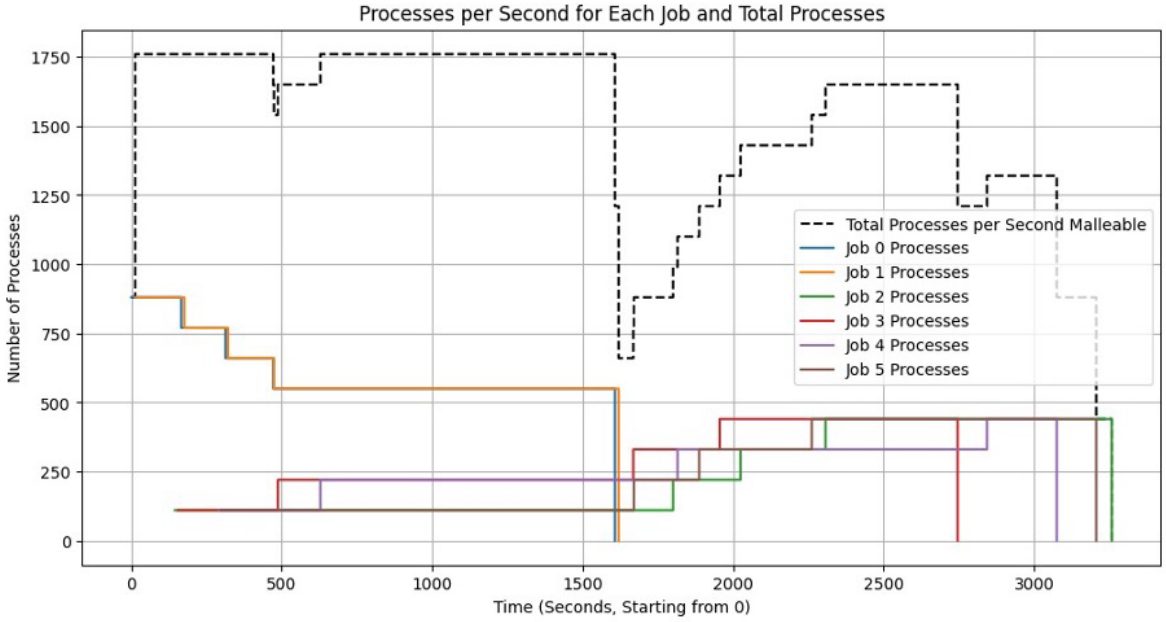


Figure 34: Evolution of the distribution of cores to run 6 concurrent jobs. The discontinuous dark line shows the total amount of resources used while maintaining the CE in the target range.

5.5 STREAmS

The use of STREAmS involves a number of operations and a workflow that can become challenging especially for the most computationally demanding cases. We summarise significant points in this regard with particular emphasis on the development activity of the past 18 months.

Workflow: the preparation of the mesh and of the field used for the initial and boundary conditions involves a preliminary RANS run, alongside some interpolation and processing. This preparatory workflow is explained in more detail in deliverable D2.15 [20] and has been almost fully automated to minimise user effort and improve the reproducibility of the simulations performed.

Grid generation: Construct2D has been significantly improved in a number of respects, such as the shape of the C-mesh to improve the quality of the grid around the trailing edge and the ability to more explicitly define grid spacing along both wall-normal and wall-tangent directions. Construct2D has also been equipped with the ability to make restarts useful in cases of particularly large grids. In addition, for even more challenging cases, an alternative workflow has been developed in which Construct2D produces only a sparse grid and then an ad-hoc C-mesh refinement application produces the final grid. This alternative workflow is not necessary for the grids used for EXCELLERAT simulations but is prospectively useful for simulations at even larger scales.

Visualisation: visualisation of STREAmS results can be done by reading the 3D fields, which, however, can become very challenging for large-scale simulations. Alternatively, slice extraction along coordinated planes and saving using the plot3D format has been implemented. Implementation of in situ visualisation functionality using Catalyst2 technology has also been completed as part of WP4 activity, allowing images to be generated directly during simulation execution.

Statistics: statistics involve averages along the z-direction of periodicity that are performed run-time. Time averages can then be performed at post-processing time or also run-time. Fully run-time statistics can be particularly useful for very large simulations.

Time-spectra: spectra are also normally performed as post-processing. However, run-time spectra computation has also been implemented. Spectra are performed according to the Welch strategy with two partially overlapping windows.

Input-output: for demanding calculations, the output of 3D fields is normally disabled, but it is still necessary to save whole fields for checkpointing purposes. For this reason, two modes are provided for check-pointing: the first is MPI-I/O based and allows a single file to do simulation restarts, while in the second mode each process saves its own restart file. For particularly large cases the serial mode is usually preferable. On the input side, several modes of read grid management have been implemented. In particular, it is possible to have a completely parallel management of the 2D grid (including metric generation). For particularly large cases, it is possible to decompose the grid before starting the simulation so that each process can read its part of the grid more efficiently.

Post-processing: the main post-processing applications, namely the statistics analysis application and the application for spectra, have been completed. The functionality of these tools depends on the case under consideration, and specific work was devoted to the airfoil cases considered in this project.

6 Conclusion

Progress has been made across the codes to meet the specific requirements of the use cases. Scalability has been evaluated on both CPU and GPU-based supercomputers. Several enhancements have been obtained. New developments have also been achieved in the porting and optimisation of the codes for GPU architectures. CINECA and URMLS are adopting a multi-paradigm strategy in the development of their STREAmS code, thus enabling compatibility with a broad spectrum of hardware platforms. All teams are making steady and satisfactory progress within WP3. The algorithmic and computational strategies for implementing their respective use cases are becoming increasingly well-defined, and efforts dedicated to WP3 have intensified in recent months.

Task 3.1 is dedicated to enhancing the computational performance of the simulation methods used in the use cases, targeting both inter-node and intra-node optimisation. As previously noted, several teams have concentrated on assessing and refining their parallelisation strategies. Meanwhile, others have focused on boosting performance on GPU accelerators or exploring solutions capable of supporting multiple programming paradigms.

Task 3.2 shows that HBM seems to be profitable for codes in EXCELLERAT. This HBM combined with the large capacities of DDR (also available in Rhea) triggers challenging memory allocations to benefit both the bandwidth of HBM and the capacity of DDR. Future work aims at extracting the Roofline profiles to see if some common classification of codes can be extracted. More mini-apps still need to be provided by code owner to consolidation this classification. Also, with such classification, the aim is to be able to suggest optimisation per profiles. Some first investigation has already been done with outer loop vectorisation, but some other optimisation can be considered: impact of vectorisation, interleaving, bandwidth saturation, memory allocation layout, and mixed precision. Finally, some work on LLVM-flang is required to handle efficiently the various code.

Task 3.3 comprises three core aspects, revolving around a unified testing platform serving the purposes of validation, deployment, and benchmarking. The efforts have focused on adopting the tools defined by the CASTIEL 2 project managing the coordination and support for National Competence Centres and Centres of Excellence on a European level, for two of the codes, Alya and AVBP. Moreover, significant work has been done on the introduction of an ad-hoc pipeline for automatic testing, validation and benchmarking for the STREAmS application, which was lacking any mechanism for it.

Task 3.4 is focused on the specific developments required to extend the simulations workflows from Task 3.1 to achieve the large-scale readiness required in exascale simulations. DLR compared the performance of their codes using container images to the installations using the native software stack of the system. They also evaluated the newly developed hierarchical mesh partition method in FlowSimulator. AVBP have focused on the portability of the code for AMD GPUs. They tried using OpenMP instead of OpenACC through automatic translation tools. The m-AIA code has thoroughly tested their code on the HAWK supercomputer at HLRS. Also, deployments on the EuroHPC systems Vega, MeluXina, Discoverer and Karolina were successfully carried out. The Alya team has done significant progress with an elastic computing methodology that adapts the resources allocated to a simulation automatically at runtime using the DMR library.

7 References

[1] EXCELLERAT_P2: Success Story: Enabling parallel mesh adaptation with Treeadapt. https://www.excellerat.eu/success-story-enabling-parallel-mesh-adaptation-with-treeadapt/
[2] KalpaTARU - Toolkit for Topology-aware Load-balancing and Adaptation of Unstructured Meshes. Library repository: https://gitlab.com/cerfacs/kalpataru
[3] Cédric Chevalier, François Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. Parallel Computing, 2008, 34 (6-8), pp.318-331. https://inria.hal.science/hal-00402893
[4] ParMETIS - Parallel Graph Partitioning and Fill-reducing Matrix Ordering. Library repository: https://github.com/KarypisLab/ParMETIS
[5] Anubis, a codemetrics tool. Library repository: https://gitlab.com/cerfacs/anubisgit
[6] POP: Performance Optimisation and Productivity, a centre of Excellence in HPC. Website: https://pop-coe.eu/
[7] EXCELLERAT_P2 D3.1: Report-on-Exa-Enabling-Methodologies. https://www.excellerat.eu/wp-content/uploads/2024/02/EXCELLERAT_P2_WP3_D3.1_Report-on-Exa-Enabling-Methodologies.pdf
[8] H. Owen et al., "Alya towards Exascale: Optimal OpenACC Performance of the Navier-Stokes Finite Element Assembly on GPUs," 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS), San Francisco, CA, USA, 2024, pp. 408-416, doi: 10.1109/IPDPS57955.2024.00043.
[9] Soldati, G., Ceci, A. & Pirozzoli, S. FLEW: A DNS Solver for Compressible Flows in Generalized Curvilinear Coordinates. Aerotec. Missili Spaz. 103, 413–425 (2024). https://doi.org/10.1007/s42496-024-00199-4
[10] Sathyanarayana, S., Bernardini, M., Modesti, D., Pirozzoli, S., & Salvadore, F. (2025). High-speed turbulent flows towards the exascale: STREAmS-2 porting and performance. In Journal of Parallel and Distributed Computing (Vol. 196, p. 104993). Elsevier BV. https://doi.org/10.1016/j.jpdc.2024.104993
[11] Salvadore, F., Rossi, G., Sathyanarayana, S. et al. OpenMP offload toward the exascale using Intel® GPU Max 1550: evaluation of STREAmS compressible solver. J Supercomput 80, 21094–21127 (2024). https://doi.org/10.1007/s11227-024-06254-y
[12] Extrae, generate Paraver trace-files for post mortem analysis. https://tools.bsc.es/extrae
[13] Paraver: a flexible performance analysis tool. https://tools.bsc.es/paraver
[14] Lührs Sebastian, Rohe Daniel, Schnurpfeil Alexander, Thust Kay, & Frings Wolfgang. (2016). Flexible and Generic Workflow Management. In Advances in Parallel Computing. IOS Press. https://doi.org/10.3233/978-1-61499-621-7-431
[15] Houzeaux, G. et al. Dynamic resource allocation for efficient parallel CFD simulations. "Computers and fluids", 2022, vol. 245, article 105577, p. 1-13.
[16] TALP for monitoring the Programming Model efficiencies. https://pm.bsc.es/ftp/dlb/doc/user-guide/how_to_run_talp.html
[17] DMR library: S. Iserte, R. Mayo, E. S. Quintana-Ortí and A. J. Peña, "DMRlib: Easy-Coding and Efficient Resource Management for Job Malleability," in IEEE Transactions on Computers, vol. 70, no. 9, pp. 1443-1457, 1 Sept. 2021, doi: 10.1109/TC.2020.3022933.
[18] DMR library repository: https://gitlab.bsc.es/siserte/dmr/-/tree/alya?ref_type=heads
[19] Fortran miniapp using DMR: https://gitlab.bsc.es/siserte/sleepmalleablefortran.
[20] Sergio Pirozzoli, Giulio Soldati & Francesco Salvadore (2025) - EXCELLERAT_P2 D2.15: Updated Report on the STREAmS Application Use Case