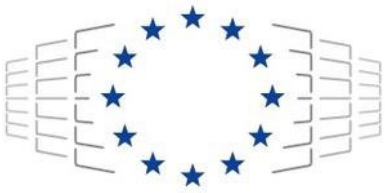
HORIZON-EUROHPC-JU-2021-COE-01

The European Centre of Excellence for Engineering Applications

Project Number: 101092621

D3.1 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
	Jonathan Vincent	KTH
	Gabriel Staffelbach	CERFACS
	Michael Wagner	DLR
	Herbert Owen	BSC
	Matic Brank	UL
	Francesco Salvadore	CINECA
	Giulio Soldati, Sergio Pirozzoli	URMLS
	Mattia Paladino	E4
	Davide Padeletti	USTUTT
Approved by	Executive Centre Management	16.11.2023
Reviewer	Florent Duchaine	CERFACS
Reviewer	Janez Povh	UL
Dissemination Level	Public	

Date	Author	Comments	Version	Status
26.10.2023	Herbert Owen	First draft	V0.1	Draft
6.11.2023	Herbert Owen	Second draft	V0.2	Draft
14.11.2023	Herbert Owen	Final version	V1.0	Final

List of abbreviations

BSC	Barcelona Supercomputing Center
CAA	Computational Aeroacoustic
CoE	Center of Excellence
CFD	Computational Fluid Dynamics
CRM	Common Research Model
DLR	German Aerospace Center
DoA	Description of the Action
DoF	Degree of Freedom
HLRS	High-Performance Computing Center Stuttgart
HPC	High Performance Computing
KTH	Kungliga Tekniska högskolan, Royal Institute of Technology
LES	Large-Eddy Simulation
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
UL	University of Ljubljana
URMLS	University of Rome LA SAPIENZA

Executive Summary

The scalability of CODA was analysed on the largest available partition of DLR's main production system. The performance of the code was tested on various upcoming CPU architectures and on Nvidia A100 GPUs.

RWTH significantly improved the performance and parallel efficiency for large-scale multiphysics simulations with the code m-AIA. The code was scaled up to utilise the full HAWK HPC system of about 500,000 compute cores.

BSC tested Alya on 3 different machines: MareNostrum IV and MN3 CTE-Power at BSC, and Polaris at Argonne Leadership Computing Facility. Moreover, it has recently received computational resources at the Karolina EuroHPC supercomputer and is currently starting tests on that machine too. Better memory management has allowed to significantly improve scalability on GPUs.

CINECA and URMLS are using a multi-paradigm approach for their code FLEW which involves a two-phase code development line. Code development, for example with new algorithms, uses a backend "master paradigm" based on CUDA Fortran. This main code is then transformed to "secondary paradigms" using automatic translations in house developed tools.

The UL team focused on the field line tracing optimisation. The code L2G has been optimised for better computational efficiency. The algorithmic improvements of the code are octree space partitioning and bounding box partitioning of large meshes to decrease time of line triangle intersection checks.

Table of Contents

1 Introduction 9
2 Task 3.1 Performance & Efficiency Engineering 10
2.1 CODA 11
2.2 AVBP 12
2.3 m-AIA 13
2.4 Alya 15
2.5 FLEW 17
2.6 OpenFoam, ElmerFEM, Raysect, Mitsuba 2 22
3 Task 3.3: Testing, Validation and Deployment 24
4 Task 3.4: Exascale Engineering 29
4.1 CODA 29
4.2 AVBP 29
4.3 m-AIA 30
4.4 Alya 31
5 Conclusion 33
6 References 34

Table of Figures

Figure 1: Scalability of the AVBP code on NVIDIA V100 and A100. 12
Figure 2: m-AIA coupled CFD/CAA chevron jet simulation: Wall time per time step for three selected MPI ranks showing a performance variability over time on HAWK with 16384 MPI ranks in total. 13
Figure 3: m-AIA coupled CFD/CAA simulation: compute time over 1000 timesteps for all 8192 MPI ranks relative to the average compute time on that MPI rank, showing process performance variability and turbo boost events on complete processors. 13
Figure 4: Strong scaling on HAWK for a coupled CFD/CAA benchmark in m-AIA. 14
Figure 5: Parallel efficiency of Alya on Marenostrum IV for different problems sizes and loads varying from 19500 to 156000 DoF per core. 15
Figure 6: Computational time per DoF, time step(ite) and GPU of on two different GPU machines Polaris (Nvidia A100) and MN4-CTE-Powe (Nvidia V100) for meshes ranging from 128^3 to 265^3 elements (57 to 508 million DoF) 16
Figure 7: Alya profiling with the Nvidia profiler 16
Figure 8: Development strategy: CUDA Fortran and translations. 18
Figure 9: STREAmS-2 object-oriented design which will be used in FLEW 19
Figure 10: Planned backend supported by FLEW: main paradigm (CUDA Fortran), translated paradigms (CPU, HIP), optional paradigms (OpenMP). 20
Figure 11: Development strategy using automatic translation. 21
Figure 12: A comparison of execution times for different mesh size (triangular mesh). The legend refers to number of threads. 23
Figure 13: Strong scaling of AVBP on the ADASTRA system using 4 Mi250 per node. Nonreactive windfarm case. 30
Figure 14: Strong scaling for a coupled CFD/CAA chevron jet application with m-AIA. 30
Figure 15: m-AIA coupled CFD/CAA simulation: baseline nozzle without chevrons showing flow structures (bottom) and the acoustic field close to the nozzle (top). 31
Figure 16: Optimizing the resources. Workflow for elastic computing of CFD simulations, involving different codes and libraries: Alya (CFD), TALP (efficiency measures) and COMPSs (elastic computing). 31

Table of Tables

Table 1: Strong scaling on HAWK for a coupled CFD/CAA benchmark in m-AIA. 14
Table 2: Comparison of Alya computational time [ns] per DoF, time step and GPU of on two different GPU machines Polaris (Nvidia A100) and MN4-CTE-Powe (Nvidia V100) before and after the Hackathon. 17
Table 3: Wall clock time [s] depending on the mesh size and number of threads. 22
Table 4: EXCELLERAT P2 codes and availability: data from CASTIEL 2 survey. 25
Table 5: EXCELLERAT P2 codes and deployment on EuroHPC systems: data from CASTIEL 2 survey. 26
Table 6: EXCELLERAT P2 deployment status and requirements: data from CASTIEL 2 survey. 28

1 Introduction

Work package three 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. Task 3.2 is omitted since it has started recently (M6) and it is not included 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. 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

Contributors: BSC, KTH, RWTH, CERFACS, DLR, CINECA, UL

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 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 three main tasks were carried out for CODA, the FlowSimulator framework, and the sparse linear systems solver Spliss that is used by CODA: First, we assessed the baseline scalability of CODA and FlowSimulator on the largest available partition of DLR's main production system CARA with the NASA common research model in a strong and weak scaling scenario. Second, we compared the performance of CODA on various upcoming CPU architectures. Third, CODA with Spliss running on GPUs was evaluated on the Nvidia A100 architecture and the performance was compared to the DLR production systems.

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. CODA achieves about 61% parallel efficiency 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 74% was achieved on 32,768 cores.
Second, in a continuous effort to test and evaluate CODA and FlowSimulator on new CPU architectures, so far, we have evaluated the AMD Zen1, Zen2, Zen3 and Zen4 architecture, the Intel Icelake architecture and the ARM-based Graviton2 and Graviton3 architecture. For the evaluation we use standardised benchmarks and a containerised version of CODA and FlowSimulator including the use case on resources in the Germany-based AWS cloud by means of a cooperation with Amazon. These measurements allow us to adapt 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 and provide valuable insight for designing DLR's own future HPC systems.
Third, we evaluated the entire workflow with Spliss running on GPUs. A significant part in computational fluid dynamics (CFD) simulations is the solving of large sparse systems of linear equations resulting from implicit time integration of the Reynolds-averaged Navier-Stokes

(RANS) equations, which are computed via the sparse linear systems solver Spliss. Next to leveraging a wide range of available HPC technologies such as hybrid CPU parallelisation, Spliss allows offloading the computationally intensive linear solver to GPU accelerators, while at the same time hiding this complexity from the CFD solver. We used Spliss to evaluate the entire workflow on a GPU system, whereas FlowSimulator and CODA are executed on the CPU part and the linear solver on GPUs. When comparing the CPU system CARO (AMD Rome) and the Nvidia A100 GPU system Juwels Booster at Jülich Supercomuting Center, the Use Case UC-1 achieves a speedup of up to 8.4 in a node-wise comparison and a speedup up to 1.9 in a power-equated comparison. The improvements made to establish multi-GPU capabilities for the Spliss solver allowing for efficient and scalable usage of large GPU systems and an evaluation of performance and scalability on CPU and GPU systems were published recently [6]. With these improvements CODA is able to support European Nvidia-based GPU systems such as LEONARDO or MareNostrum 5. The support for AMD-based GPU systems such as LUMI-G is currently evaluated.

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 2 has focused on the robustness and reproducibility of the parallel mesh refinement library TREEADAPT. First, developed in the EXCELLERAT 1, it has already been used up to 8192 cores to generate 2B element meshes. However, it was soon discovered that results were not reproducible due to parallel effects in PARMETIS and other round-off errors. We have just recently published an alpha version of TREEADAPT (version 0.8.1) that is reproducible and usable up to 2048 cores. Further work is expected to improve the mpi core capabilities and use the hierarchical partitioning capabilities embedded in TREEADAPT.
In parallel, we have been testing AVBP on JUWELS Booster (A100-40G) and other NVIDIA GPU clusters (V100-16G) to ensure the performance of the code. Figure 1 shows the scalability of the AVBP code on NVIDIA architectures for a 1B element mesh reactive Large Eddy Simulation (LES). Scalability up to 1024 GPUs is confirmed and performance between V100 and A100 almost doubles for low GPU counts. The difference in high GPU counts is lower as the data per GPU is not sufficient for the faster A100. H100 tests were performed by NVIDIA and suggests another x2 acceleration per GPU at least, but could not be confirmed independently yet. We will test the code on Grace Hopper APU in the first quarter of 2024.

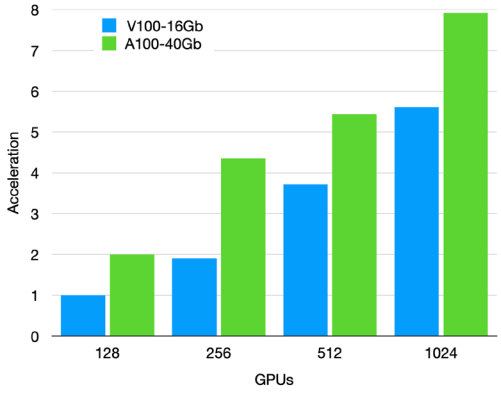


Figure 1: Scalability of the AVBP code on NVIDIA V100 and A100.

2.3 m-AIA

During the first project year RWTH significantly improved the performance and parallel efficiency for large-scale multiphysics simulations with the code m-AIA. Repeated testing of an aeroacoustics application that was scaled up to utilise 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. Appropriate changes were introduced into the critical part of the communication modules in m-AIA, which eliminated the observed performance issues at large scale.

Other work focussed on improving the dynamic load balancing for large-scale CFD/CAA simulations. After identifying load balancing issues on HAWK for coupled CFD/CAA simulations, several tests were performed to analyse the reason for apparently random CPU timers, which appeared for large-scale runs. Several tests in cooperation with HPE were performed on the full machine. The timers were obviously influenced by the power management and turbo boost features of the CPU (see Figure 2 and Figure 3). A work around was identified by not using the full core count available per CPU. Furthermore, the employed dynamic load balancing approach in m-AIA has been enhanced by introducing a mesh partitioning utilising adaptively higher levels of the hierarchical Cartesian mesh to achieve higher parallel efficiencies at large-scale.

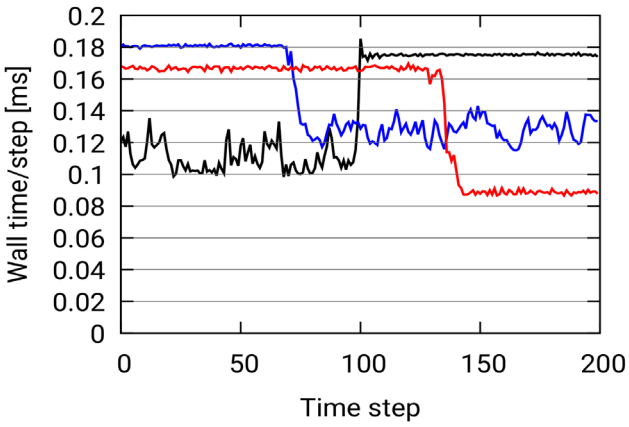


Figure 2: m-AIA coupled CFD/CAA chevron jet simulation: Wall time per time step for three selected MPI ranks showing a performance variability over time on HAWK with 16384 MPI ranks in total.

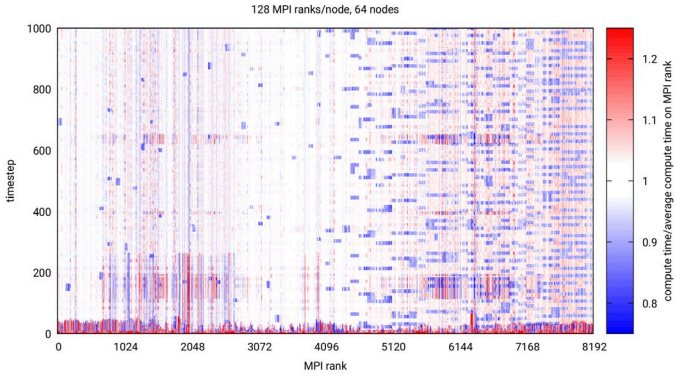


Figure 3: m-AIA coupled CFD/CAA simulation: compute time over 1000 timesteps for all 8192 MPI ranks relative to the average compute time on that MPI rank, showing process performance variability and turbo boost events on complete processors.

Strong scaling tests using the coupled CFD/CAA solvers in m-AIA for a benchmark with 1.2*10^9 CFD cells and 1.0*10^9 CAA degrees of freedom (DOF) are shown in Figure 4 and Table 1 on the HAWK system, ranging from 32 nodes up to the maximum possible allocation size of 4096 nodes, i.e., 524288 compute cores. The results show excellent parallel efficiency of the m-AIA code, where a superlinear speedup is observed for the cases with more than 1024 nodes. This is probably due to the decreasing local problem size with the increasing core number. The local problem size starts with a number of mesh cells on the order of O(100.000) which drops to O(1000) cells for the largest core number used, with higher data localities that reduce the overall memory access latencies. The conclusion that the memory access constitutes the bottleneck of the simulations is substantiated by the similar number of time steps per second that can be performed when using 128 MPI ranks with each 1 thread per compute node or just 64 MPI ranks with either 1 or 2 threads per MPI rank. The results show the efficient utilisation of the whole HAWK system using the m-AIA code, e.g., the wall time to perform 100,000 time steps can be reduced from about 80 hours on 32 nodes to less than half an hour on 4096 nodes.

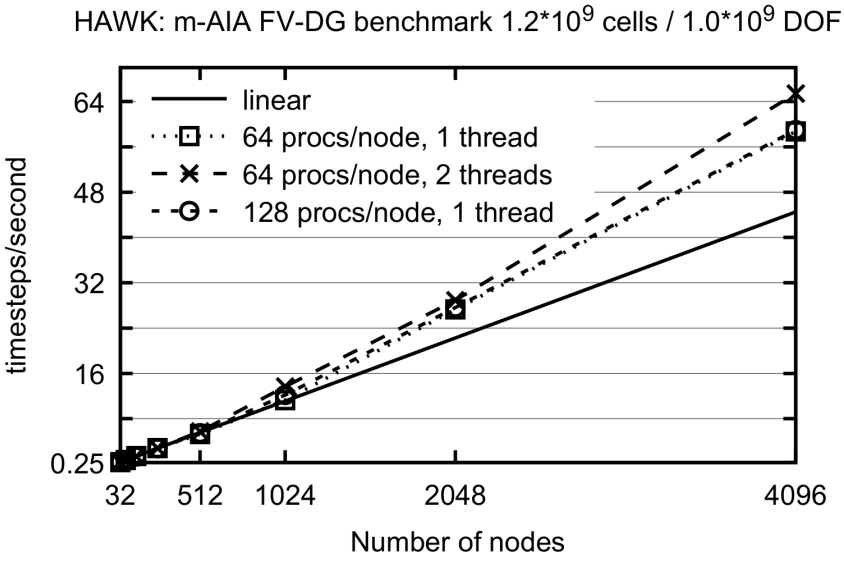


Figure 4: Strong scaling on HAWK for a coupled CFD/CAA benchmark in m-AIA.

128 procs/node		speedup: procs/node, #threads			
#cores	DoF/core	linear	128/1	64/1	64/2
4096	299520	1	1.00	1.00	1.00
8192	149760	2	1.93	1.96	1.96
16384	74880	4	3.76	3.84	3.83
32768	37440	8	7.47	7.51	7.57
65536	18720	16	14.68	15.26	15.42
131072	9360	32	34.15	33.87	37.56
262144	4680	64	89.54	81.53	87.70
524288	2340	128	188.94	184.55	195.64

Table 1: Strong scaling on HAWK for a coupled CFD/CAA benchmark in m-AIA.

2.4 Alya

The Alya team has concentrated its efforts during the first year on testing and improving the code scalability on both CPUs and GPUs. Moreover, the node level performance of the GPU version of the code has been improved.
The code has been tested on 3 different machines: Marenostrum IV and MN CTE-Power at BSC, and Polaris at Argonne Leadership Computing Facility. Moreover, we have recently received computational resources at the Karolina EuroHPC supercomputer and are currently starting tests on that machine too. Preliminary results are very similar to those obtained at Polaris which also uses A100 Nvidia GPUs, but slightly slower since the Maximum clock frequency is limited in Karolina to reduce energy consumption.

To test the scalability of the code we solve the Compressible Navier-Stokes equations for a Taylor-Green Vortex problem at Reynolds number, Re = 1600, and Mach number, Ma = 0.1.
For the temporal discretisation an explicit 4th order Runge-Kutta scheme is used. For the spatial discretisation third order hexahedral spectral elements are used. Meshes with N^3 elements are used, with N varying from 64 to 420. In a mesh with continuous third order hexahedral elements, each element corresponds to 64 (4^3) nodes or Degrees of Freedom (DoF). Therefore, the number of DoF vary from 7.19 to 2005 million.

Figure 5, presents the code scalability on Marenostrum IV, that uses Intel Xeon Platinum 8160 CPUs. Despite each node has a total of 48 cores, only 46 cores per node are used because it has recently been discovered that Marenostrum IV suffers important scalability issues when the full 48 cores per node are used. The reasons for this behaviour are still not clear. The figure shows that the code has an excellent weak and strong scalability on Marenostrum IV. For the strong scalability to show degradation lower number of DoF per core would have been needed. Efficiency is obtained by first calculating the Update Time (UT) for each run and then normalising with respect to the UT on 46 cores with a load of 156k DoF per core. The Update Time (UT) is defined as the computational time per time step and DoF. That is, the total computational time for one-time step divided by the average load per core (identified as r in Figure 5).

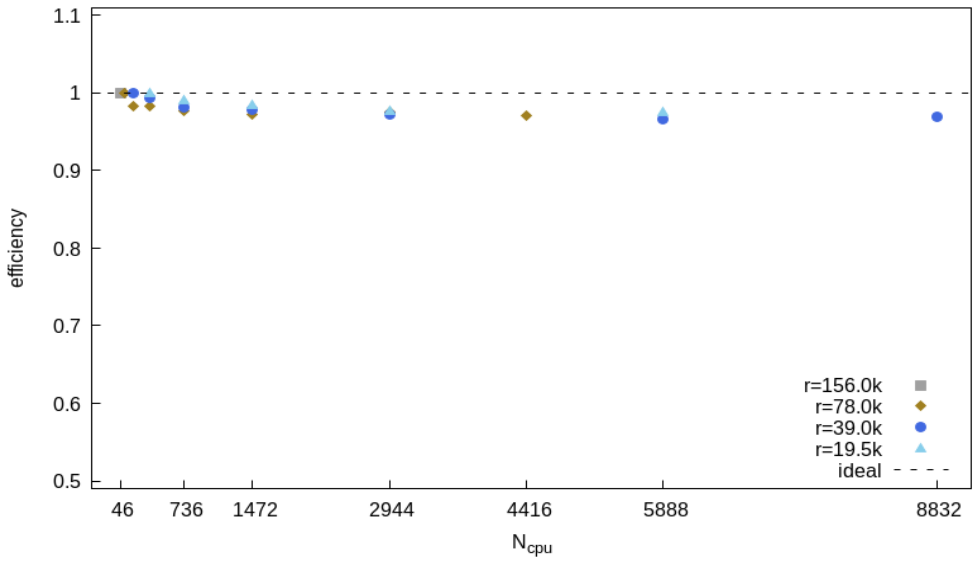


Figure 5: Parallel efficiency of Alya on Marenostrum IV for different problems sizes and loads varying from 19500 to 156000 DoF per core.

Figure 6 shows Update Time per GPU of Alya on two different supercomputers using Nvidia V100 and A100 GPUs. On the A100 GPU the Update Time remains bounded between 18 and

20 ns for loads ranging from 5 to 20 million DoF per GPU. The scalability starts to degrade for loads of less 5Million DoF per GPU. On the V100 GPUs the scalability is similar to the one on A100 GPUs but the performance also significantly degrades for high load per GPU due to the lower amount of memory available on V100 GPUs.

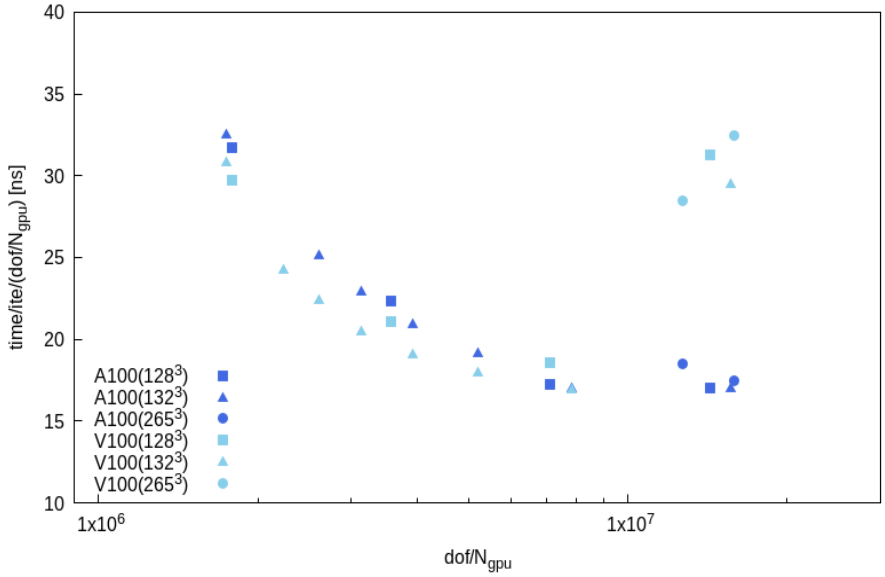


Figure 6: Computational time per DoF, time step(ite) and GPU of on two different GPU machines Polaris (Nvidia A100) and MN4-CTE-Powe (Nvidia V100) for meshes ranging from 128^3 to 265^3 elements (57 to 508 million DoF).

The Alya team participated in the ALCF (Argonne Leadership Computing Facility) INCITE Hackathon (May 2023). Motivated by the hackathon mentors, the lead decided to take advantage of **CUDA Aware MPI comms** to enhance the scalability of the code. To do so, they had to discard the use of **Unified Memory** (adding "-gpu=managed" to compile flags) that is not compatible with **CUDA Aware MPI**. The use of **Unified Memory** simplifies memory management and makes coding easier. Moreover, before the hackathon, the version of the code with **Unified Memory** was performing much better than without it.

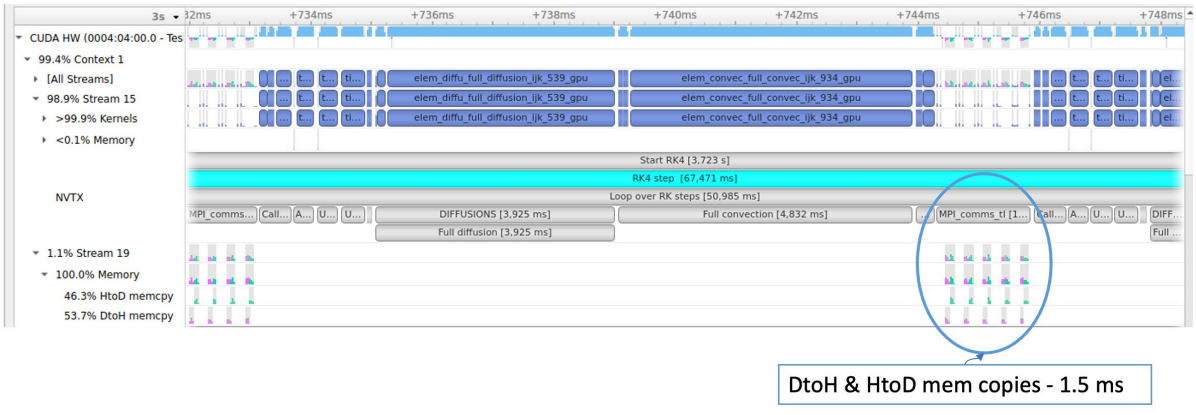


Figure 7: Alya profiling with the Nvidia profiler.

Figure 7 shows results obtained with the Nvidia profiler for the **Unified Memory** version of the code where one can see that the device to host and host to device memory copies take approximately 1.5 milliseconds. With the optimised **non-Unified Memory** version, the time for the memory copies reduces to less than one millisecond.

GPUs	DoF/GPU	A100 old	A100 new	A100 ratio	V100 old	V100 new	V100 ratio
4	14266656	17.02	10.25	**1.66**	31.30	17.15	**1.82**
8	7133328	17.22	10.02	**1.72**	18.56	17.72	**1.05**
16	3566664	22.35	10.60	**2.11**	21.11	19.46	**1.08**
32	1783332	31.69	14.80	**2.14**			

Table 2: Comparison of Alya computational time [ns] per DoF, time step and GPU of on two different GPU machines Polaris (Nvidia A100) and MN4-CTE-Powe (Nvidia V100) before and after the Hackathon.

Table 2 presents a Comparison of Alya computational time [ns] per DoF, time step and GPU of on two different GPU machines Polaris (Nvidia A100) and MN4-CTE-Powe (Nvidia V100) before and after the Hackathon. On the Nvidia A100 GPU there has been a significant (>1.66) gain in performance for all four analysed loads. The scaling is nearly perfect up to 3.5 million DoF per GPU. On the V100 GPU the gains are more modest except on the case with a high load per GPU. Not using **Unified Memory,** a much better memory management is obtained which eliminate de degradation of performance for high load per GPU which had been observed earlier.

2.5 FLEW

As already mentioned in the introduction of this section software developers deal with three main performance-oriented objectives:

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

FLEW uses explicit Runge-Kutta schemes for the temporal evolution and explicit finite-difference schemes for the spatial discretisation, both very suitable from a parallel programming perspective.

Within the scope of EXCELLERAT P2, we refer first of all to architectures available in EuroHPC resources during the project timeline. CPU-wise, these are mostly x86-64 machines but also some ARM platforms. From the point of view of accelerators, these are NVIDIA GPUs of different generations (Ampere, Hopper) and AMD GPUs (Instinct). In particular as regards FLEW, according to the DoA of the project, the development of a code capable of efficiently exploiting both NVIDIA GPUs and AMD GPUs is envisaged. From the point of view of the choice of paradigms, the most important characteristics to favour in the choice are:

- possibility of achieving high level performance, in reference to the peak performance of the hardware but also to the performance actually achievable in similar cases according to existing literature;
- create a code that remains maintainable over time, i.e., susceptible to future algorithmic evolutions, limiting the corresponding development effort on the part of the developer community;
- create a code that is portable and/or has generality with respect to future expansions of the calculation backends.

Unfortunately, at the current moment, there is no paradigm that allows you to achieve all three of these objectives. More precisely, there is no paradigm implemented in a sufficiently advanced way to achieve the objectives. To have portability in the strict sense, a standardised paradigm is necessary. The choice of OpenCL would require a restructuring of the total code, a significant burden that would make future algorithmic evolution difficult, as well as

essentially abandoning the Fortran language. The choice of other standard paradigms such as OpenMP and OpenACC is potentially valid, but clashes with the low/incomplete implementation of paradigms for different accelerators. Unfortunately, the ideal compromise between the three objectives requires interpreting the objective of code portability in an extended way, i.e., adopting a multi-paradigm approach (instead of the portability in the strict sense that would be obtained using a single paradigm). The multi-paradigm approach allows you to achieve the optimal performance that vendor-specific paradigms allow you to achieve. Secondly, the software can potentially support new paradigms by adding support for new backends and thus achieving portability on different existing and non-existing architectures. The challenge of creating a code that is maintainable and susceptible to future algorithmic evolutions with a reasonable effort remains open.

The approach that we have decided to undertake in FLEW is therefore the multi-paradigm approach which however involves a two-phase code development line (see Figure 8):

1. code development, for example with new algorithms, using a backend "master paradigm".
2. transformation of the calculation part using different "secondary paradigms" starting from the code of the main "paradigm".

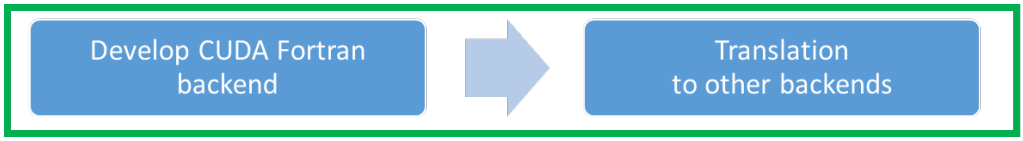


Figure 8: Development strategy: CUDA Fortran and translations.

CUDA Fortran was chosen as the main paradigm, capable of optimally exploiting NVIDIA GPUs. The choice is due to three main reasons:

1. NVIDIA GPUs represent the computational core of most EuroHPC computing architectures and are therefore a reasonable hardware reference for the development of CoE software;
2. good readability of the CUDA Fortran code, thanks also to the cuf automatic kernel directives, and in particular good ability to "expose parallelism"
3. adequate quality of the compiler update that the vendor guarantees to make adequate use of its devices.

The first sub-paradigm is actually the traditional paradigm of code designed to run on a traditional CPU. The choice not to have the CPU as the main paradigm derives from the fact of preferring a parallelisation-oriented paradigm such as CUDA Fortran, compared to a generic CPU-type paradigm. The calculation code for the CPU is then generated starting from CUDA Fortran in order to guarantee that development always takes place in a parallel oriented manner.

To support AMD GPUs, the secondary paradigm developed is HIP/HIPFort. The difficulty of supporting HIP for a Fortran code is notable because, unlike CUDA Fortran, HIPFort is not able to compile calculation kernels in Fortran, but only to provide a connection infrastructure (via interfaces and more) between code Fortran and kernels written in C. It is therefore a question of managing, among other things, a massive transformation of the computing kernels from CUDA Fortran to C.

Having a master paradigm means that all code developers can easily develop using CUDA Fortran and then delegate the task of translating the code to a different phase and/or group of people. For this process to work, a clear structural separation of the code is needed so that different backends can be programmed separately and at different times. In this regard, software architecture is crucial. The STREAmS community code (https://github.com/STREAmS-

CFD/STREAmS-2) is developed by the same partners that develop FLEW (Sapienza/CINECA) and has important algorithmic similarities with FLEW. The architectural development of the two codes therefore takes place conceptually simultaneously, but the development is taking place at this stage mainly on STREAmS which is therefore currently in a more advanced stage. From a structural point of view, STREAmS can be considered as a simplified platform compared to FLEW for code design testing. The object structure of STREAmS is represented in Figure 9.

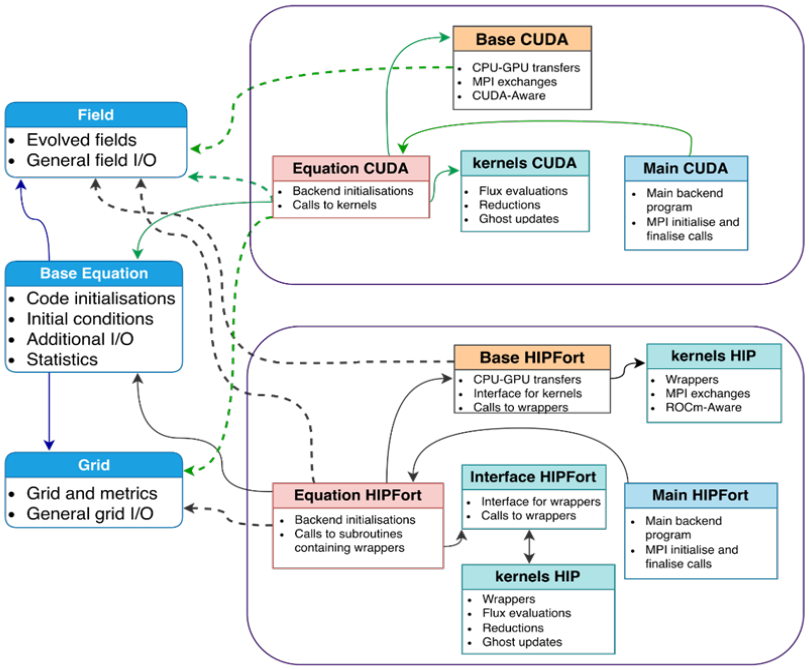


Figure 9: STREAmS-2 object-oriented design which will be used in FLEW.

The sketched object architecture allows the FLEW code to be separated into four sections with respect to the implemented equation and the supported computational backend:

1. Sections equation-independent and backend-independent
   - object (Fortran derived type) : parameters
   - object (Fortran derived type) : field
2. Sections equation-independent and backend-dependent
   - object (Fortran derived type) : base_<backend>
3. Sections equation-dependent and backend-independent
   - object (Fortran derived type) : <equation>/equation_base
4. Sections equation-dependent and backend-dependent
   - object (Fortran derived type) : <equation>/equation_<backend>
   - pure functions (Fortran module) : <equation>/kernel_<backend>

Excluding computational kernels, the components are implemented via Fortran derived types in an object-oriented fashion where each derived type includes procedures as well as data attributes. Kernels are instead implemented as pure functions which therefore take all the variables they work on as arguments. This separation allows great versatility in the implementation of kernels which can be implemented with relative ease for example in C. Adding a new computational backend in this final FLEW design corresponds to creating objects and functions of points 2) and 4) whereas 1) and 3) sections can be unchanged.

Given the algorithmic similarity, albeit partial, between STREAmS and FLEW, FLEW could not only inherit the object structure of STREAmS but even be unified with STREAmS. The unification could be partial, in the sense of sharing some parts of code or related tools, or more complete in the sense that the result could be a single code. The possibility of such unification is still being studied. The unification between the codes can be helpful, also in terms of development effort, but the risk of creating excessive code complexity as well as limiting the optimisation of the calculation kernels must be carefully weighed.

The proposed approach is substantially valid if the transformation procedure of the main paradigm into the secondary paradigms occurs with a reasonable effort. To make this easier, the choice that has been made is to limit ourselves to a CUDA Fortran code that is programmed efficiently but without particularly strong optimisations that would make the maintainability of the code as well as the conversion to other paradigms difficult. This choice proved to be adequate in STREAmS as the performances obtained are realistically close to those of the GPUs used.

Figure 10 shows the planned backend which will be supported by FLEW, where CUDA Fortran is the primary developed paradigm, CPU and HIP are the derived ones, and OpenMP is the optional paradigm which will be attempted.

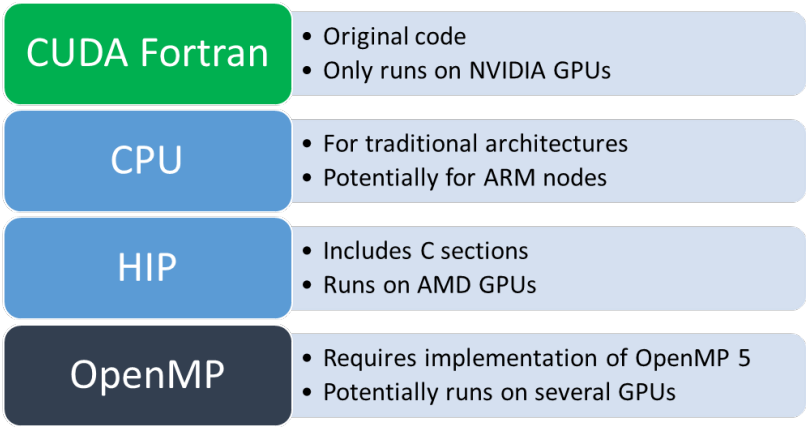


Figure 10: Planned backend supported by FLEW: main paradigm (CUDA Fortran), translated paradigms (CPU, HIP), optional paradigms (OpenMP).

For paradigms such as CPU, the conversion from CUDA Fortran is conceptually quite easy and includes the following main steps:

- removal of device attribute for memory used for computation
- removal of cuf directives and GPU synchronisations
- transformation of explicit (global) kernels into traditional loops
- cache oriented reordering of all loops
- removal of all asynchronous communication components, which are not considered necessary to replicate in the CPU context as there are no CPU-GPU transfers involved

For paradigms such as HIP, the conversion requires significantly greater efforts. Among the main steps we consider:

- transformation of device memory into pointers and allocations using HIPFort interfaces
- contextual creation of indexing maps to access device memory from C in a readable way

- explicit kernel transformation with multi-layer structure: Fortran procedure, Fortran interface for the wrapper, wrapper in C, kernel in C,
- transformation cuf kernel into explicit kernels and transformation as in the previous point
- replacement of asynchronous parts and in CUDA Fortran library calls in HIP equivalents

Further paradigms are currently being studied and in particular the OpenMP paradigm which, being standard, could potentially provide considerable portability. This is a part of the development not strictly required in this project as CUDA Fortran and HIP are able to run optimally on NVIDIA and AMD GPUs, the reference architectures foreseen for FLEW in the EXCELLERAT DoA. OpenMP is one of the paradigms favoured by Intel GPU compiler implementations, which we believe are an important development to consider when developing HPC code. The OpenMP paradigm can potentially be also tested as an alternative for other GPUs, especially AMD GPUs.

As seen, the transformation of the code from the main paradigm requires several steps which, for a code of a certain size, can be considerable. For this reason, an automatic (or semi-automatic) conversion tool capable of generating the code for the other backends from the CUDA Fortran backend is being developed. The transition from manual conversion to automatic conversion (even if possibly with some manual input) makes the ordinary code development path significantly more agile (see Figure 11). The tool is in Python language and is designed to be extensible itself for the production of new backends in subsequent phases.

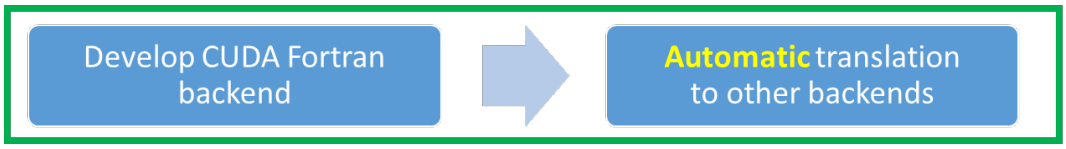


Figure 11: Development strategy using automatic translation.

All the activities described in this document are currently under development. At present, there are therefore three main lines of development:

1. development of STREAmS (version 2) which, given the algorithmic similarity with FLEW, can provide decisive indications for the choices to be made on FLEW.
2. development of FLEW by evolving the original code for algorithmic and physical validation in test cases. A first version of the CPU code has been completed alongside a preliminary (partial) version that supports NVIDIA GPU: these versions can be already used for production runs.
3. development of the new FLEW rewritten in a modular, object-oriented and multiback-end way according to a technological infrastructure similar to that of STREAmS, as described in this document. In this context, a possible unification of STREAmS and FLEW, also given that the developer community is the same, is being studied.

At the moment, intermediate results are only available for the first two points. Since these are not results related to the final code (objective of the third point), all optimisation, porting and performance results will be presented when the code in the final design is available, at the end of the activity of point 3.

The activities described follow without deviations what was foreseen in the global GANTT of Use Case 6. The completion and validation phase of the starting version of the FLEW code was completed at M9 (see D2.14), and we are now proceeding (M9-M15) to rewriting/refactoring

the code according to the design described in this document. The actual porting will be the subject of activities M14-M27, and the deployment/validation/benchmarking activities on different EuroHPC clusters will be performed in a similar time window (M18-M32).

2.6 OpenFoam, ElmerFEM, Raysect, Mitsuba 2

The UL team focused this year on the field line tracing optimisation, since this is the first simulation step of further-application case. The code L2G [1] developed at UL and ITER has been optimised for better computational efficiency. The algorithmic improvements of the code are:

- Octree space partitioning
- Bounding box partitioning of large meshes to decrease time of line triangle intersection checks.

The parallelisation of field line tracing is also achieved by dividing the trace into multiple smaller traces. The cases in the L2G module are run with 1 OpenMP thread by default.

A benchmark case, named *Inres1* case was run for 1, 2, 4, 8 and 16 OpenMP threads. For performance tests on huge meshes the target and shadow geometries from the Inres1 case were remeshed (using the SMESH module in SALOME) to element sizes of approximately 10, 3, 2 and 1 mm. The last size resulted in a target mesh of 3046416 triangles and a shadow mesh of 25309694 triangles. This means that the algorithm launched 3046416 traces and in each step each trace was checked for intersection with 25309694 triangles, yielding a number 25309694*3046416/1e9 = 77103.856756704 billion of total intersection checks (see Fig. 12 and Table 3) Of course, the usage of algorithmic improvements described earlier reduced the number of checks. In table below a total execution time is given depending on the mesh size (the multiplication of both target and shadow triangles gives number of intersection checks) and number of threads.

No. of triangles		No. of threads				
Target geometry	Shadow geometry	1	2	4	8	16
3350	72260	3.621	3.299	3.309	3.214	2.887
22002	177694	7.499	6.260	5.531	4.982	4.612
352580	2826308	79.002	55.481	47.170	40.154	34.612
782438	6421866	172.536	121.317	104.671	85.653	74.232
3046416	25309694	665.733	457.932	395.761	333.765	283.704

Table 3: Wall clock time [s] depending on the mesh size and number of threads.

This specific test was conducted on HPC at Faculty of Mechanical engineering, University of Ljubljana. Similar checks are underway on Vega HPC.

Figure 12 shows total execution times depending on the mesh size and number of threads used. The results show that many threads execution speeds up field-line tracing on million size meshes for a factor more than 2 with the use of 16 threads.

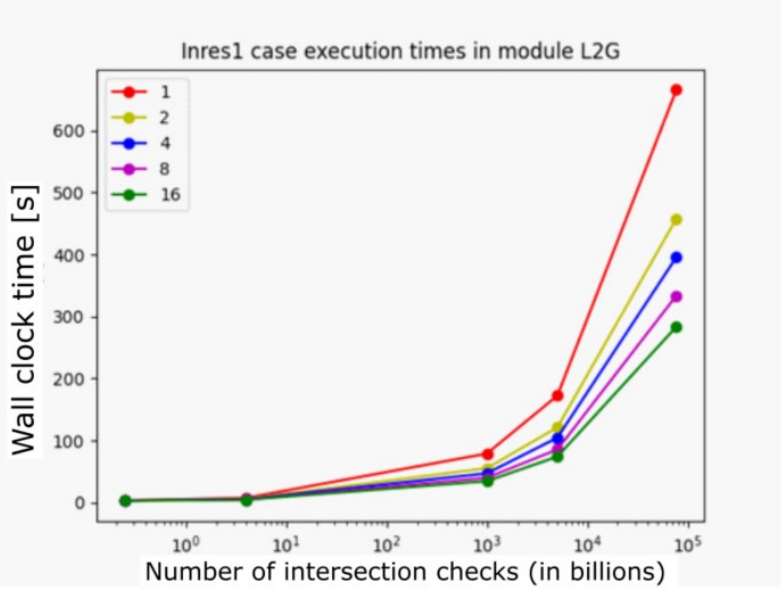


Figure 12: A comparison of execution times for different mesh size (triangular mesh). The legend refers to number of threads.

For L2G code the output meshes and results are stored in Med file format (based on HDF5 library), the standard binary file format for storing results in SALOME open source simulation environment and used at UL extensively for pre and post processing. Next year, the UL work in Task 3.1 will focus on optimisation of thermal modelling (OpenFOAM) and ray-tracing (Raysect) codes.

3 Task 3.3: Testing, Validation and Deployment

Contributors: E4, BSC, KTH, RWTH, CERFACS, DLR, CINECA, UL

In this section, we will provide an overview of the work conducted within the framework of Task 3.3. It is essential to mention that due to project First Amendment (July, 2023), the start of this task was rescheduled from Month 1 to Month 9. Therefore, in the present D3.1, only activities relating to M9-M12 are reported.

The aim of this task is to focus on the testing and validation activities of various methodologies and workflows applied to execute the use cases. This includes the development of automated testing strategies and deployment processes, ensuring that the results can be replicated and extended for broader applications. These tests will be consolidated into a testing platform capable of assessing performance and accuracy across diverse architectural environments. The successful execution of this task relies on close collaboration with application and computational scientists to guarantee the effectiveness and efficiency of these advancements.

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, will largely depend on the input and direction provided by CASTIEL 2 project managing the Coordination and Support for National Competence Centres (NCC) and Centres of Excellence (CoE) on a European Level Phase 2. Since Task 3.3 is inherently coupled with our collaboration with CASTIEL 2, the activity is carried out in close coordination and, in particular, timelines and main decisions should be agreed in the CASTIEL 2 context.

Over the past few months, CASTIEL 2 has conducted two surveys addressing the needs of different CoEs to collect information useful to guide its coordination role. In particular, we report and comment here the main results coming from the surveys which can be useful for what concerns the present T3.3 context. Table of results are provided in Table 4, Table 5, and Table 6.

Code	History, Versions	Owner	Code & Licence	URL	Terms of Use
CODA	More than 20 years development (TAU, Flucs, CODA)	CODA is jointly owned by ONERA, DLR and Airbus.	proprietary, based on individual agreement	No public access	
AVBP	AVBP has been developped since the end of the 90's at CERFACS.	CERFACS	proprietary, based on individual agreement	No anonymous access.	
m-AIA	Development since more than 15 years	RWTH	Not decided yet.	The code will go open source in 2023.	Not decided yet.
Alya	Developed since 2004.	BSC	Available Source or Software as a Service licenses	https://gitlab.com/bsc-alya/alya, https://bsc.es/research-development/research-areas/engineering-simulations/alya-high-performance-computational	Free under collaboration agreement
Neko	Developed since 2018.	KTH	Open Source (BSD)	https://gitlab.com/ExtremeFLOW/neko , https://neko.cfd/	No Restrictions
FLEW	Developed since 2011.	Sapienza University of Rome	Proprietary, based on individual agreement (until it will be open source).	The code will go open source before the end of the project.	Not decided yet
OpenFoam/ Raysect	Developed since 2014	OpenCFD Ltd (OpenFOAM), Alex Meakins (Raysect)	Open Source	https://github.com/raysect https://develop.openfoam.com/Development/openfoam/-/wikis/precompiled	No restrictions

Table 4: EXCELLERAT P2 codes and availability: data from CASTIEL 2 survey.

We selected the most relevant information to highlight the most significant challenges in the creation of the unified platform foreseen in T3.3. In particular, we discuss two main challenges:

1. Code availability, and in particular open-source issue
2. Platform architecture deployments

The first challenge is well described in Table 4 where we show the main information of EXCELLERAT application codes highlighting three main points:

- Owner of code
- Code availability and license
- Terms of usage

Designing and implementing a unified platform is a clear challenge considering the availability of the codes is very limited. Indeed, only two of the 7 scenarios (6 Use cases and 1 FA) are based on publicly available codes. Two additional scenarios work with codes which will become open-source during the project (one during 2023 and one at the end of the project). The remaining scenarios deal with closed-source code, in some cases available only after individual agreements.

	Lumi		Vega		Karolina		Discoverer	MeluXina			Leonardo		Deucalion	MareNostrum 5			
State (**P**roductive, **W**ork in Progress, **N**ot (yet) done **Planned** (**Y**es, **N**o, **M**aybe)	LUMI-C	LUMI-G	V. CPU	V. GPU	K. CPU	K.GPU		M. CPU	M. GPU	M. FPGA	L. Booster*	L. Data Centric		M5 GPP	M5 ACC	NGT ACC	NGT GPP
CODA																	
State	N	N									N	N		N	N	N	N
Planned	Y	Y									Y	Y		Y	Y	Y	Y
AVBP																	
State	P				P	P	P										
Planned		Y						N	Y	N	Y	Y	N	Y	Y		
m-AIA																	
State	N	N	W	N	W	N	W	W	N	N	N	N	N	N	N	N	N
Planned																	
Alya																	
State				N		P					N						
Planned		M															
Neko																	
State	P	P	N	N	N	N	N	N	N	N	P	N	N	N	N	N	N
Planned			Y	Y	Y	Y	Y	Y	Y	Y		Y	Y	Y	Y	Y	Y
FLEW																	
State	N	W	N	N	N	P	N	N	N	N	P+W	N	N	N			
Planned	Y	Y	N	Y	N	Y	N	N	Y	N	Y	Y	N	N	Y	M	M
OpenFoam/Raysect																	
State	W	W	P		W	W		W	W								
Planned				Y										Y	Y		

Table 5: EXCELLERAT P2 codes and deployment on EuroHPC systems: data from CASTIEL 2 survey.

The platform diversity against EXCELLERAT application software is summarised in Table 5. In particular, we consider 17 different computing architectures corresponding to different partitions of EuroHPC machines. For each computing unit and each code, we collected two main information related to deployment:

- **Status**: which may be Productive, Work-in-progress or Not yet done
- **Planned**: which may be Yes, No, Maybe

Despite at this stage, the results are still not completely clean (further collection will be performed in the next months) it is already possible to extract a general information. Only one code declares the plan to be able to deploy the code to each EuroHPC partition. As for the other codes, there is a significant variability in the number and type of target partitions. As such, a possible unified platform mandatorily inherits this kind of limitations. However, even

supporting the planned architectures is a significant challenge even considering different types of involved heterogeneous accelerations.

Regarding the timeline of the planned activities, Table 6 shows that some applications already started deployment and production to different architectures. The need for resources is clearly an additional issue which is, however, not directly related to the present T3.3 task. The second part of the survey (Questions 5-8) specifically deal with feedback on the first indications coming from CASTIEL 2 meetings. In particular, CASTIEL 2 is focusing on the possibility to establish a centralised EuroHPC-driven source code repository. The repository could be based on GitLab technology and the GitLab runner might be an adequate platform which can be used for CI/CD. In this context, GitLab runner could be part or represent itself, as the unified platform used in T3.3. However, as discussed from Table 4 data, there is the issue of managing not open-source codes which can be contrasting with respect to centralised source code repository. It's worth noting that, at present, only one of the use cases is open source, and some partners intend to keep their code proprietary. CASTIEL 2 leans toward an open-source approach, but we need to find a consensus on this matter.

In our roadmap planning, our approach is to track CASTIEL’s lead. Should CASTIEL offer a comprehensive and inclusive framework, we will harmonise our efforts with it. However, if CASTIEL’s framework is not all-encompassing, our task will involve a detailed evaluation of all projects and the establishment of guidelines for the documentation required for validation, deployment, and benchmarking.

ID	Question	Answer
1	**Which access calls have you used? (Benchmarking, Development, Regular, Extreme)**	Benchmarking (5); Development (1); Regular (2).
2	**How many different codes were already executed successfully on at least one EuroHPC system?**	8 codes successfully executed.
3	**Are the resources requested and assigned are sufficient to reach the objectives of your CoE? Likely the resources will be sufficient for the first year. Can you already do a projection for the following years?**	Mostly all code owners state that resources are sufficient for the first year (2023), where the code is tested and benchmarked. But more resources will be needed from 2024 till the end of the project. BSC (Alya) estimates for next years: 2024 3M Core Hours, 2025 5MCore Hours, 2026 9M Core Hours. CINECA (Flew) estimates that about 1M GPU hours/year will be sufficient. 3 out of 7 code owners could not make any predictions.
4	**Please report briefly about the experiences made on the EuroHPC systems? How was the application process? Did you face any difficulties getting your codes to run? Did you get (technical) support by EuroHPC hosting sites if needed?**	The experience is overall positive. Application is smooth but bit too long to be granted access (over a month). Some technical support from Hosting Sites was given. One code owner reported different access procedures and conditions to each system though soi t would be great to have a unique profile for EuroHPC users.
5	**Can you please provide your feedback regarding the need of having a common GitLab and give some preliminary information about what you would push to such repository? E.g. share the actual code, deployable software artefacts, binaries, deployment recipes, ...**	All have GitLab or Github repositories, one code owner has already GitLab runners to run the CICD pipelines. If required by Castiel2 we are available to share the source code and deployment recipes, as docker/singularity images of the protected software (2 or 3 application) and opensource packages (all).
6	**Can you start deploying codes on a EuroHPC system in the next weeks using GitLab Runner?**	yes (4), possibly testing the software stack beforehand; not (2)
7	**What will be your first code to share with the common GitLab server? Can you provide a timeline? (describe some technical limitations that you might face)**	The non-opensource code cannot be uploaded as it is to the GitLab server without some information on the access policy. The opensource ones can be upload anytime. One code owner still needs to complete the development of the code (M18-M24), then will be ready to share code.
8	**Do you have further feedback to us?**	The GITLAB/CI requirement in this document seems to imply its coming from us whereas it was not on the original proposal. We can adapt to this situation but a clarification seems in order. Also looking at the different EuroHPC hosting sites, support for GitLab CI and/or containers seems very heterogeneous. A panorama of support per system would be appreciated. It would be good to have the contact details to the technical support of the common GitLab server or the EuroHPC system.

Table 6: EXCELLERAT P2 deployment status and requirements: data from CASTIEL 2 survey.

4 Task 3.4: Exascale Engineering

Contributors: CERFACS, KTH, RWTH, BSC, DLR, CINECA.

This task 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. Achieving exascale simulations is challenging not only from the software point of view, but also from the hardware operation, as large number of resources need to be allocated to a single application. In fact, ensuring high efficiency and performance in large-scale workflows requires specific directives on the bash scripts and use of resources, so the applications can be distributed over large number of computing nodes. 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 to make ensure an efficient execution of the run.

4.1 CODA

During the reporting period one main tasks was carried out for the FlowSimulator framework used by CODA: Based on an initial analysis, we improved FlowSimulator's support for very large meshes and very large core counts. We identified and solved multiple issues in the mesh partitioning stage, which consists of a fast pre-partitioner based on recursive coordinate bisection (RCB) and a following graph-partitioner such as ParMETIS. In the FlowSimulator implementation of an RCB we solved an integer overflow and an out-of-memory error for very large meshes (more than 1 billion elements). In the graph-partitioner we solved an out-of-memory error in the graph-extraction phase for very large core counts. In combination with other improvements, we were able to overcome a previous scalability limit at about 8,192 MPI processes that limited the execution of simulations with more than 32,768 cores on the DLR production systems CARA and CARO. We are now able to run successfully on 131,072 cores (the largest partition of the CARO system). In addition to increasing the scalability of FlowSimulator, the pre-processing stage was also drastically accelerated.

4.2 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 in EuroHPC and in the world are equipped with AMD Mi250 GPUs.

The code supports GPU acceleration using the OpenACC framework. Therefore, compatibility for now remains limited to HPE systems equipped with the CRAY compiler suite. Thankfully, 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.

Portability of the current code build is on-going and an AMD GPU capable code prototype for non-reactive flows exists. Performance on a small test case shows that good scaling is possible with 1M elements per GPU (see Figure 13 below).

The extension to reactive flows, requires some code refactoring being undertaken by the CFD team at CERFACS outside of the EXCELLERAT project and is expected to be completed by

February 2024. Tests could not be performed in LUMI G as the CRAY compiler version was too old at the time of access but has been updated since and will be tried in Q1 2024.

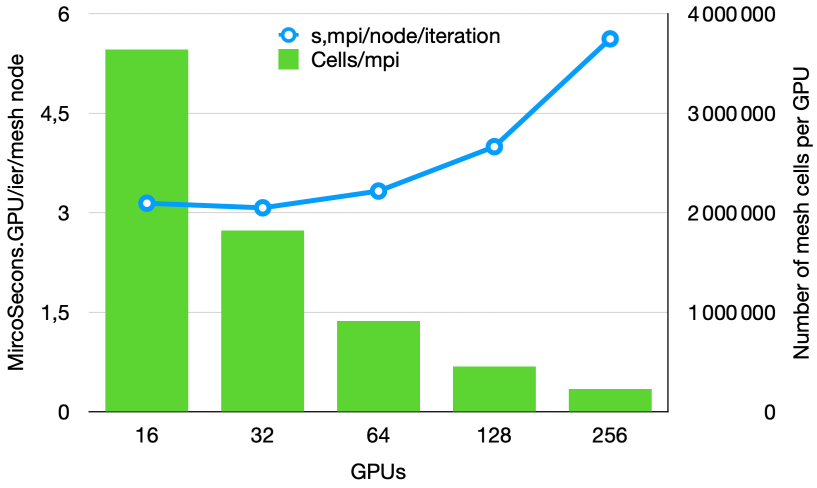


Figure 13: Strong scaling of AVBP on the ADASTRA system using 4 Mi250 per node. Nonreactive windfarm case.

Additionally, a benchmark access has been requested to test AVBP on LEONARDO Booster, ensuring that the refactoring of the code does not impact NVIDIA GPU performance.

4.3 m-AIA

The m-AIA code has been thoroughly tested on the HAWK supercomputer at HLRS, deployment on the Vega EuroHPC system took place and benchmarking is currently under progress. Computational resources on the Karolina and MeluXina EuroHPC supercomputers have been granted as well and code deployment and testing will start once the access to the systems is provided.

The strong scalability of a realistic coupled CFD/CAA chevron jet application with m-AIA on HAWK is shown in Figure 14. The predicted flow field and the acoustic field for a baseline nozzle without chevrons is visualised in Figure 15. With about 300 million CFD cells and 1*10^9 CAA DoF this setup corresponds to a smaller scale run according to the exascale execution profile defined in WP2 for UC-3. As evident the code shows 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.

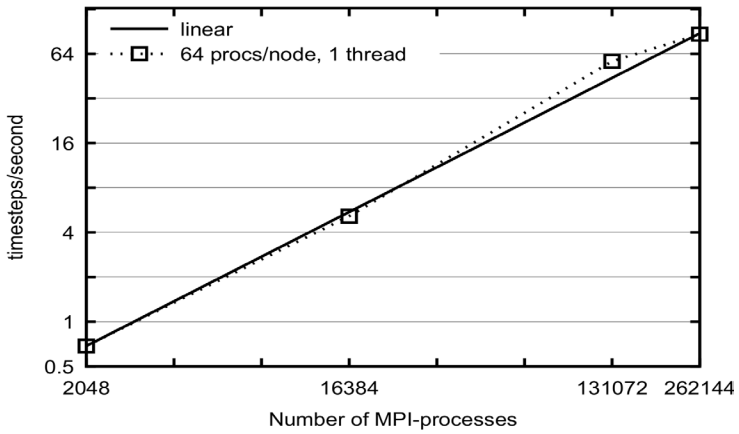


Figure 14: Strong scaling for a coupled CFD/CAA chevron jet application with m-AIA.

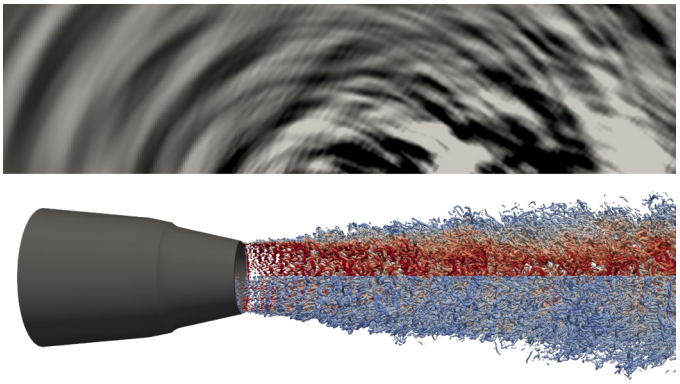


Figure 15: m-AIA coupled CFD/CAA simulation: baseline nozzle without chevrons showing flow structures (bottom) and the acoustic field close to the nozzle (top).

4.4 Alya

CFD users of supercomputers usually resort to rule-of-thumb methods to select the number of subdomains (partitions) when relying on MPI-based parallelisation. One common approach is to set a minimum number of elements or cells per subdomain, under which the parallel efficiency of the code is "known" to fall below a subjective level, say 80%. The situation is even worse when the user is not aware of the best practice for a given code and a huge number of resources can thus be wasted. In a previous work [2], we have developed a workflow to ensure a target communication efficiency, as shown in Figure 16.

Figure 16: 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 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 [3,4], thus simplifying the interactions of the CFD code Alya and the computing environment.

During this period, 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 miniapp have been containerised and can be found here [5].

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.

5 Conclusion

Advances in the different codes to satisfy the needs of the use cases have been presented. Scalability has been tested on CPU and GPU supercomputers and improvement have been implemented. Advances in the porting and optimisation for GPUs has been presented. CINECA and URMLS are using a multi-paradigm approach for their code FLEW that will allow them to run on a wide range of hardware. Most of the teams are advancing satisfactorily in WP3. The NEKO team has focused more on their use case (WP2) and on the activities in WP4 during the first year of the project. The algorithmic and computational developments of the different methodologies defined to execute their use case are now clearer and they will increase their dedication to WP3 during the following months. Since we still have 3 more years, we believe that there is currently no risk for the correct completion of the proposed activities for NEKO.

Task 3.1 focuses on optimising the computational efficiency of the simulation methodologies employed in the use case at inter- and intra-node levels. As mentioned above, some teams have worked on evaluating and improving the parallelisation strategy. Others have worked on improving the processing on GPU accelerators or developing approaches that can handle multiple paradigms.

Task 3.3 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, will largely depend on the input and direction provided by CASTIEL 2 project managing the Coordination and Support for National Competence Centres and Centres of Excellence on a European Level Phase 2.

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. The CODA team has focused on mesh partitioning within their FlowSimulator framework. The AVBP team has worked on the scalability for AMD GPUs. The m-AIA team has analysed the strong scalability of a realistic coupled CFD/CAA chevron jet application. The Alya team has worked with an elastic computing methodology that adapts the resources allocated to a simulation automatically at runtime.

6 References

[1] G. Simic, S. Carpentier, R.A. Pitts, L. Kos, F. Fernandez, M. Brank “Enhancements and applications of the SMITER magnetic field line tracing and heat load mapping code package,” presented at the 30th IEEE Symposium on Fusion Engineering, Oxford, UK, 7 2023.
[2] Houzeaux, G. [et al.]. Dynamic resource allocation for efficient parallel CFD simulations. "Computers and fluids", 2022, vol. 245, article 105577, p. 1-13.
[3] 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.
[4] DMR library repository: https://gitlab.bsc.es/siserte/dmr/-/tree/alya?ref_type=heads
[5] Fortran miniapp using DMR: https://gitlab.bsc.es/siserte/sleepmalleablefortran.
[6] J. Mohnke and M. Wagner, “A Look at Performance and Scalability of the GPU Accelerated Sparse Linear System Solver Spliss”, In Euro-Par 2023: Parallel Processing. LNCS, vol 14100, 2023. https://doi.org/10.1007/978-3-031-39698-4_43