E-Infrastructures

H2020-INFRAEDI-2018-1

INFRAEDI-2-2018: Centres of Excellence on HPC

EoCoE-II

Energy oriented Center of Excellence : toward exascale for energy

Grant Agreement Number: INFRAEDI-824158

D2.3

Final report for WP2 programming models

Project and Deliverable Information Sheet

EoCoE-II	Project Ref:	INFRAEDI-824158
	Project Title:	Energy oriented Centre of Excellence: towards exascale for energy
	Project Web Site:	http://www.eocoe2.eu
	Deliverable ID:	D2.3
	Deliverable Nature:	Report
	Dissemination Level:	PU*
	Contractual Date of Delivery:	30/06/2022
	Actual Date of Delivery:	30/06/2022
	EC Project Officer:	Matteo MASCAGNI

* - The dissemination level are indicated as follows: PU – Public, CO – Confidential, only for members of the consortium (including the Commission Services) CL – Classified, as referred to in Commission Decision 2991/844/EC.

Document Control Sheet

Document	Title :	Final report for WP2 programming models
	ID :	D2.3
	Available at:	http://www.eocoe2.eu
	Software tool:	LaTeX
Authorship	Written by:	Mathieu Lobet (CEA)
	Contributors:	Ani Anciaux-Sedrakian (IFPEN), Bibi Naz (FZJ), Brian Wylie (FZJ), Carl Burkert (FZJ), Chantal Passeron (CEA), Dominik Ernst (FAU), Dorian Midou (CINES), Edoardo di Napoli (FZJ), Emily Bourne (CEA), Frederic Blondel (IFPEN), Garrett Good (Fraunhofer IEE), Georg Hager (FAU), Gerhard Wellein (FAU), Helen Schottenhamml (FAU, IFPEN), Herbert Owen (BSC), Jan Eitzinger (FAU), Jaro Hokkanen (FZJ), Jose Fonseca (CEA), Julien Bigot (CEA), Judit Gimenez (BSC), Marie Cathelain (IFPEN), Markus Wittmann (FAU), Michel Mehrenberger (AMU), Philipp Franke (FZJ), Sebastian Achilles (FZJ), Stefan Kollet (FZJ), Thierry Gautier (INRIA), Thomas Gruber (FAU), Tobias Kloeffel (FAU), Ulrich Ruede (FAU), Virginie Grandgirard (CEA), Yanick Sarazin (CEA)
	Reviewed by:	PEC, PBS

Document Keywords: Performance, optimisation, programming model, speed-ups, scalability, MPI, OpenMP, OpenACC, Cuda, kokkos, CPU, GPU, super-computer, ARM, task

Contents

List of Figures

List of Tables

1 Executive summary

This report presents the progress and results of the second half of the Work Package 2 of the EoCoE-II project. Work Package 2 deals with programming model aspects and also covers performance issues around parallelism, optimisations and accelerator porting. Work on linear algebra and input-output is reserved for work packages WP3 and WP4. Work Package 2 is nevertheless connected to these different work packages of the projects, notably through questions of performance, porting or implementation of new libraries. Work Package 2 is also involved in the Exascale strategy of each application in partnership with the project's Exascale Co-Design Group. Not all flagship code of the project is concerned at the same level in this Work Package. The tasks and their progress for each scientific pillar and for each code are given again in the body of the text. Changes to the original plan presented at the beginning of the project are clearly explained and justified. This report also goes into some technical details of the implementations, optimisations and porting work without going into too much of them. Most of the time, published articles, or those in the process of being published, allow this to be done. Depending on the progress of the work within EoCoE-II, the future developments needed to go Exascale are discussed.

2 Acronyms

Table 1: Acronyms for the partners and institutes therein.

Acronym	Partner and institute
AMU:	Aix-Marseille University
BSC:	Barcelona Supercomputing Center
CEA:	Commissariat à l'énergie atomique et aux énergies alternatives
CERFACS:	Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique
CIEMAT:	Centro De Investigaciones Energeticas, Medioambientales Y Tecnologicas
CoE:	Center of Excellence
EDF:	Électricité de France
ENEA:	Agenzia nazionale per le nuove tecnologie, l'energia e lo sviluppo economico sostenibile
FAU:	Friedrich-Alexander University of Erlangen-Nuremberg
FSU:	Friedrich Schiller University
FZJ:	Forschungszentrum Jülich GmbH
IEA:	International Energy Agency
IBG-3:	Institute of Bio- and Geosciences Agrosphere
IEK-8:	Institute for Energy and Climate Research 8 (troposhere)
IEE:	Fraunhofer Institute for Energy Economics and Energy System Technology
IFPEN:	IFP Énergies Nouvelles
INAC:	Institut nanosciences et cryogénie
INRIA:	Institut national de recherche en informatique et en automatique
IRFM:	Institute for Magnetic Fusion Research
NEWA:	New European Wind Atlas
MdlS:	Maison de la Simulation
MF:	Meteo France
MPG:	Max-Planck-Gesellschaft
POP:	Performance Optimization and Productivity Center of Excellence
PRACE:	Partnership for Advanced Computing in Europe
R-CCS:	RIKEN Center for Computational Science
RWTH:	Rheinisch-Westfälische Technische Hochschule Aachen, Aachen University
UBAH:	University of Bath
UNITN:	University of Trento

Table 2: Acronyms of software packages

Acronym	Software, codes and libraries
PDAF:	Parallel Data Assimilation Framework
PDI:	Parallel Data Interface
EFCOSS:	Environment For Combining Optimization and Simulation Software
ESIAS:	Ensemble for Stochastic Intergration of Atmospheric Simulations
EURAD-IM:	EURopean Air pollution Dispersion-Inverse Model
DDC:	The discrete domain computation library [1]
GISELA-X:	GYrokinetic SEmi-LAgrangian in 5D
HYPERstreamHS:	Dual-layer MPI large scale hydrological model including Human Systems
ICON:	Icosahedral Nonhydrostatic model
MDFT:	Molecular Density Functional Theory

MELISSA:	Modular External Library for In Situ Statistical Analysis
MESO-NH:	Mesoscale non-hydrostatic model
Nemo5:	NanoElectronics MOdeling Tools 5
neXGf:	non-equilibrium eXascale Green's functions
OpenFOAM:	Open Source Field Operation and Manipulation
OpenMP:	Open Multi-Processing
ParFlow:	PARallel Flow
PPMD:	Performance Portable Molecular Dynamics
ReaxFF:	Reactive Force Field
SHEMAT:	Simulator of HEat and MAss Transport
SOWFA:	Simulator fOr Wind Farm Application
SPS:	Solar Prediction System
TELEMAC:	TELEMAC-MASCARET system
TerrSysMP:	Terrestrial Systems Modeling Platform
WaLBerla:	A Widely Applicable Lattice-Boltzmann Solver
WanT:	Wannier Transport
WPMS:	Wind Power Management System
WRF:	Weather Research and Forecast model

Table 3: Acronyms for the Scientific Terms used in the report.

Acronym	Scientific Nomenclature
ABL:	Atmospheric Boundary Layer
AD:	Automatic Diffentiation
AMR:	Adaptive Mesh Refinement
AOT:	Aerosol Optical Thickness
PBE:	Perdew-Burke-Ernzerhof functional
BLYP:	Becke-Lee-Yang-Parr functional
COT:	Cloud Optical Thickness
CLM3.5:	Community Land Model version 3.5
CPU:	Central Processing Units
CSP:	Concentrated Solar Power
DA:	Data Assimilation
DFT:	Density Functional Theory
DMC:	Dynamic Monte Carlo
FSI:	Fluid-Structure Interaction
GPU:	Graphical Processing Unit
HLST:	High Level Support Team
HPC:	High Performance Computing
ITER:	International Thermonuclear Experimental Reactor
KMC:	Kinetic Monte Carlo
LES:	Large Eddy Simulations
MD:	Molecular Dynamics
MPI:	Message Passing Interface
NEGF:	Non-Equilibrium Greens functions
NREL:	National Renewable Energy Laboratory
NWP:	Numerical Weather Prediction
OED:	Optimal Experimental Design
ODE:	Ordinary Differential Equations

PBC:	Periodic Boundary Conditions
PDAF:	Parallel Data Assimilation Framework
pdf:	probability density functions
PF-CLM:	Parflow-Community Land Model
QMC:	Quantum Monte Carlo
QM:	Quantum Mechanics
SHJ:	Silicon HeteroJunction
SOL:	Scrape-Off Layer
SpMV:	Sparse matrix-vector multiplication
TDP:	Thermal Design Power
WP:	Work Package

3 Introduction

This document is the final report (M42) of the Work Package Programming Models (WP2) of the Centre of Excellence EoCoE-II. As described in the proposal, experts on Programming Models (WP2) focus on how to handle efficiently complex computing nodes having a deep memory hierarchy and possibly accelerators, how to address more operation concurrencies and to minimize development effort that maximizes performance portability. It has been decided not to pursue new research in this area but to benefit from existing developments and to stick to established standards or emerging technologies. This Work Package plays an essential role in the improvement and accompaniment of codes towards Exascale computing technologies. This work is being achieved in coordination with the Exascale Co-design Group and other Work Packages.

The tasks listed in WP2 mainly concerned flagship codes. The list of codes is given in Fig. 1. The only flagship code not directly concerned by WP2 is ESIAS-MET.

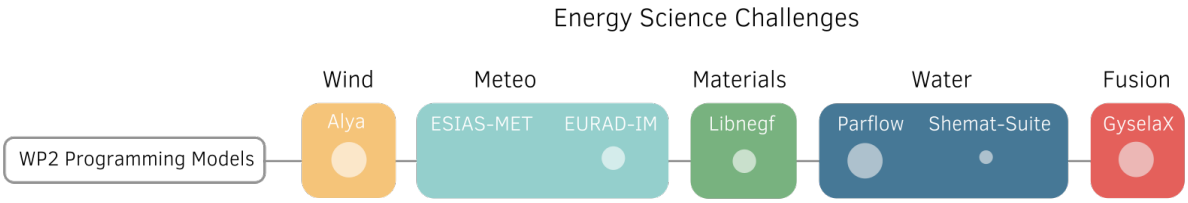


Figure 1: Drawing of the applications concerned by WP2.

In order to give an overview of the state of all codes in terms of performance models, we present the table 4 for the flagship codes and 5 for some of the satellite codes.

With the imminent arrival of Exascale, HPC technologies have undergone many changes and accelerations in recent years, particularly since the beginning of the EoCoE project. Today, a supercomputer is a cluster of a very large number of computing nodes connected to each other by a high-performance network. Each node is a sort of independent computer composed of many computing units. This structure was born in the early 1990s. Nevertheless, the technologies have evolved enormously since then, especially concerning the intra-node level. Until the 2008, computing nodes were mostly composed of a CPU (Central Processing Units) sockets. From this time, the first GPGPU (General Purpose Graphical Processing Units) accelerators appeared in some super-computers of the top500 list for numerical simulation. In the field of HPC, the choice of technologies is primarily based on a simple equation: obtain the highest performance at a low energy and financial cost. They started to provide more raw computing power than the CPU model for a similar energy envelope. For a number of scientific algorithms such as linear algebra kernels, GPUs have thus emerged as a means of complementing or outperforming CPU tasks. It was also at this time that the first programming models for GPGPU cards appeared such as CUDA. Today, GPUs are increasingly used in large-scale computers. For example, hybrid machines (super-computers equipped of both CPU and GPUs) account for half of the 50 world most powerful supercomputers [2]. Furthermore, the top 10 supercomputers in the Green500 (which lists the 500 most energy-efficient supercomputers) are mostly equipped of GPUs. The development of the GPGPU has also increased with the explosion of Artificial Intelligence (AI) applications and the convergence of AI, HPC, HPDA (High-Performance Data Analytics). The historical evolution of the hybrid super-computer top500 share is shown in Fig. 2.

On hybrid machines, GPU computing units also provide the most computing power. However, it is important to remember that the GPU only acts as an accelerator, i.e. it remains coupled to a traditional CPU. The CPU continues to handle the tasks that the GPU cannot: management, system tasks, input/output, etc. Nevertheless, CPU architectures have undergone constant development and continue to be used as the main computing unit in many systems. In 2019, Fujitsu and ARM released the ARM A64FX processor for the world's most powerful computer. ARM processors from the embedded world are beginning to conquer

Application name	main languages	x86 CPUs	ARM CPUs	NVIDIA GPU	AMD GPU	Largest run/scaling test
ALYA	FORTRAN, MPI	Optimized	Not tested	Partly ported and optimized (CUDA, OPENACC)	No	100 000 cores
EURAD-IM	FORTRAN, MPI	Optimized-x2.5 speedup	Not tested	No	No	768 threads single instance
ESIAS-MET	Python, KSH, FORTRAN, MPI	Optimized	Not tested	No	No	128 cores single instance, 49 152 cores ensemble run (1024 ensemble members)
LIBNEGF	FORTRAN, MPI	Optimized	Not tested	Optimized (OPENACC/ CUDA-based libraries) - x5 speedup	No	36 000 cores
KMC/ DMC	Python, C/C++, MPI, OPENMP	Optimized	Optimized	Underway, accessible through PPDM (CUDA)	No	up to 8192 cores (Strong scaling)
PARFLOW	C, MPI	Optimized	Not tested	Optimized (CUDA) - x28 speedup	Optimized (Kokkos)	1024 GPUs
SHEMAT-SUITE	FORTRAN, OPENMP	Optimized	Not tested	No	No	504 cores largest production run / 2016 cores largest scaling
GYSELAX	FORTRAN, MPI, OPENMP	Optimized - x3.7 speedup	Partially optimized - x1.3 speedup	No	No	729 088 AMD Epyc Milan cores

Table 4: State summary of the flagship applications at the end project. Color legend: green - working, orange - theoretically working but not tested, red - not working

the world of cloud computing and may become more common in the world of HPC as European initiatives highlight. In this context, the trendy technologies for building an exascale computer are the massive use of GPUs within the nodes. Exascale supercomputers with RISC or ARM CPUs could also be developed.

In parallel with the development of hardware technologies, programming models capable of taking advantage of the computing power of GPUs have multiplied and gained in maturity. Many important libraries for the scientific computing world have been developed allowing the creation of a more complete software ecosystem. The choice of a programming model will therefore depend on many parameters:

- Main programming language: The programming language used by a code will restrict the available parallel programming models. For example, a FORTRAN code will have much more limited choices than a C or C++ code. This explains why some teams have decided to completely rewrite their codes to go to Exascale.
- Targeted architecture: the architectures (types of CPU, types of GPU) targeted by the codes will

Application name	main languages	x86 CPUs	ARM CPUs	NVIDIA GPU	AMD GPU
waLBerla	C++	Optimized (MPI, OpenMP, SIMD)	Not tested	Optimized	No
TOKAM3X	FORTRAN	Optimized (MPI, OpenMP)	Not tested	No	No
Soledge2D	FORTRAN	Optimized (MPI, OpenMP)	Not tested	No	No
Metalwalls	Python, FORTRAN, C++	Optimized (MPI, OpenMP)	Not tested	Optimized (OpenACC)	No
MDFT	FORTRAN	Optimized (MPI, OpenMP)	Not tested	No	No
GENE	FORTRAN, C	Optimized (MPI, OpenMP)	Not tested	Yes	No

Table 5: State summary of the satellite applications at the end project. Color legend: green - working, orange - theoretically working but not tested, red - not working

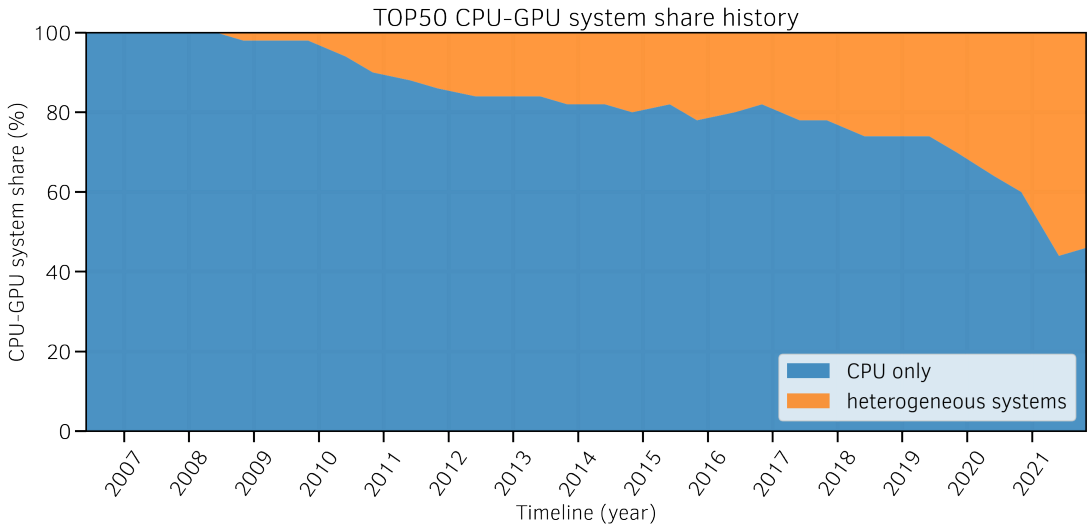


Figure 2: historical evolution of the hybrid super-computer top500 share.

also constrain the choice of a programming model. It is also important to evaluate the interest of an architecture for each code or code kernel before undertaking a headlong porting work.

- Portability: With the multiplication of manufacturers and architectures, the notion of portability is becoming more and more central to the choice of programming models. A code is considered portable from one architecture to another when it is capable of running on both without major modification. A programming model is considered portable when it enables through a unified abstracted interface to run on several hardware architectures.
- Performance: Not all models will provide the same level of performance for a given model. It de-

pends on whether the model gives the developer the ability to fine-tune its implementation for a given architecture. Generally speaking, the search for performance on a given machine requires a substantial optimisation effort and an adaptation of the implementation to the hardware specificities (instruction set, caches, etc). The result is often a loss of portability or the need to multiply implementations of the same algorithm for several architectures. The assessment depends on the needs of the developers.

- Development skills and readability: The software development and HPC skills of the development teams should be taken into account when choosing a model. In the field of scientific numerical simulation, it is not uncommon for the developers to be end users of the code themselves (e.g. physicist developer). Consequently, the porting or the optimization of a code must take into account the capacity of all the members to be able to maintain their developments. The involvement of the development teams in an HPC support activity is also essential. Code owners must retain control of the implementation and must therefore be able to read the code from the chosen programming model.
- Legacy and maintainability: The life of a code can be many years and support resources rarely cover that life. The maintainability of the code over time has to be taken into account. Mature and long term supported software technologies should be considered when codes do not have continuous HPC support. Otherwise, the use of experimental programming models can be considered.

Prior to any porting or optimisation activities, it is therefore important to identify

- the needs of the developers and users for short and long term (scientific goals, computing power, skills, etc),
- limitation of the current application (bottlenecks, functionalities, etc),
- possible solutions in term of programming models,

in order to avoid going in the wrong direction and spending unnecessary resources. In any case, there is never a magical solution. There is always a trade-off between performance, portability, maturity, readability and other criteria. One of the roles of WP2 is to guide application developers towards the best choices to meet their needs. The reader will see that several teams have chosen to port their code to GPU as shown as well in table 4 and that programing model choices are different. Most of the tools used in EoCoE-2 are mature and proven.

The three main programming languages used by EoCoE applications are FORTRAN, C and C++. Although the choice of a better language is still debated today, there is consensus that C/C++ is a good choice because most modern programming models and HPC libraries today primarily support these languages. FORTRAN, still widely used for numerical simulation, is less and less supported and is not compatible with most-advanced programming models. Nevertheless, many codes are still written in FORTRAN and the rewriting work is a significant challenge that requires skilled and up-to-date human resources on languages, time and methodology.

MPI is the standard of choice widely used in distributed computing as it is on all modern HPC machines. As processors condense more and more compute cores, it is more and more common and interesting to adopt hybrid thread parallelization at the node level. Here, only OpenMP is used for this. OpenMP uses the notion of threads to exploit the parallelism of recent processors and uses the notion of directives to simplify the development.

Programming on GPUs can be done using proprietary low-level programming language and its associated libraries. CUDA is the most widely used, but only for NVIDIA boards. This solution makes the most of the power of NVIDIA cards but is not portable. In order to be more portable, OpenACC allows GPUs to be addressed by directives like OpenMP does on the CPU. This solution has the advantage of not being tied

to a specific type of GPU card in order to remain as portable as possible. In this project, we are using both solutions.

During a GPU porting, we generally want to minimize code duplication, to have a good memory management between the host processor and the device, to have a portable implementation to avoid rewriting algorithms at each technological leap, to be able to minimize the distinction between a code intended to run on CPU and a code intended to run on GPU.

Recently, new programming models have become fashionable because they make it possible to bring all these requirements together. This is the case of Kokkos [3] and RAJA [4], both developed in the United States. In particular, they make it possible to abstract the use of memory and thus allow the development of generic CPU/GPU algorithms. The use of Kokkos will be explored in this project.

In collaboration with Work Package 4, WP2 is involved in the development and use of the PDI API. The Parallel Data Interface (PDI) is not a library itself but an interface that enables users to decouple all these I/O processes from codes through a single API ([5, 6]). As shown in Fig. 3, the API supports read- and write- operations using various I/O libraries within the same execution and allows switching and configuring the I/O strategies without modifying the source (no re-compiling). However, it does not offer any I/O functionality on its own. It delegates the request to a dedicated library plugin where the I/O strategy is interfaced. In other words, PDI offers a declarative API for simulation codes to expose information required by the implementation of I/O processes. The latter are encapsulated inside plugins that access the exposed information.

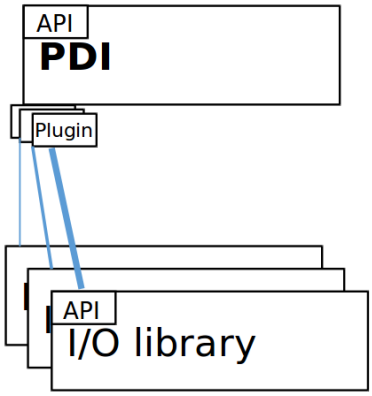


*Figure 3: Conceptual scheme of the Parallel Data Interface (*PDI*).*

The next sub-section describes the document structure.

3.1 How to read this document

Each section of this document represents one of the major task of WP2 as described in the proposal. The first task called performance evaluation and modelling is transverse to all the Scientific Challenges and concerns the missions of this Work Package. This first task is associated with the first section 4. The following tasks have been constructed to contain the work to be carried out in each Scientific Challenges respectively. As a result, the following sections are associated with each Scientific Challenges:

- section 5: Task 2.2 - Wind code optimization
- section 6: Task 2.3 - Meteorology code optimization
- section 7: Task 2.4 - Materials code optimization
- section 8: Task 2.5 - Hydrology code optimization
- section 9: Task 2.6 - Fusion code optimization

In each major task of this WP, we first remind the associated codes before describing the work carried out. This includes the members of each team and updates. We have in the proposal and then in the first deliverable divided major tasks into subtasks. Since the first deliverable, each code has had an action plan (simplified Gantt) that we update here according to the work progress, the difficulties and the encountered delays. The action plans are based on the explanatory model shown in the figure 4.

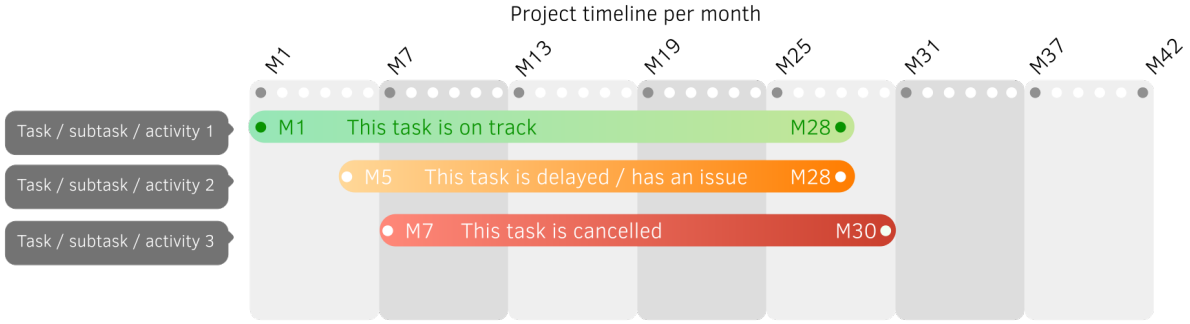


Figure 4: How to read our simplified Gantt chart.

A timeline provides approximate information on the start and end of each subtask. A green task does not present any difficulty. A task in orange has problems; it may possibly be delayed or extended. In red, the task is cancelled.

For each major task, a table of risks is shown at the end of the section.

3.2 Impact of COVID-19

Our project has been impacted by the health crisis due to COVID-19. The encountered difficulties and the impact on the project are described in the risk management sub-sections for this Work Package.

4 Task 2.1 - Performance evaluation and modelling

The goal of this task is to provide the required tools and resources to the project applications to ensure continuous and successful code optimisation and performance improvement. This task is organized around several objectives:

- Performance evaluation process of the codes
- Performance bottleneck identification
- Optimization strategies on kernels and on full applications
- Workshops and hackathons to teach tools and guide optimizations with experts
- Knowledge benefit outside the EoCoE-II community

To achieve these objectives, task 2.1 contains several actions to perform:

- Support in performance evaluation, code optimisation and code engineering through project experts
- Organization of workshop dedicated to performance evaluation and code optimisation
- Communication around external training on code optimisation (like PRACE trainings)
- Management of the computing resources

An active support is enabled thanks to the HPC experts connected to the project. Section 4.1 brings more details on support provided by our experts. The events organized by this Work Package are presented in section 4.2. The management of the PRACE computing resources is described in section 4.3.

4.1 Optimization support

Our experts are presented in Table 6.

People	Position	Role	Period
Georg Hager	FAU	Node-level optimisation, LIKWID tools	M1-M36
Gerhard Wellein	FAU	Coordinator at FAU	M1-M36
Jan Eitzinger	FAU	Node-level optimisation, Likwid tools	M1-M36
Thomas Gruber	FAU	LIKWID tools	M1-M36
Dominik Ernst	FAU	Node-level optimisation, GPU optimisation	M1-M36
Judit Gimenez	BSC	HPC expert, member of the POP COE, BSC tools	external to the project
Brian Wylie	JSC	HPC expert, member of the POP COE, JSC tools	external to the project
Thierry Gautier	CR INRIA	Expert in task-based programming model	M1-M36

Table 6: Performance and optimisation experts for support in EOCOE-II.

Gerhard Wellein coordinates the FAU's activity within EOCOE-II. Georg Hager and Jan Eitzinger are part of the HPC expert panel available within the project to help application teams optimize their code. They are responsible for organizing tutorials and hackathons with a strong node-level component. Dominik Ernst is a GPU expert with in-depth experience on code optimisation. Thomas Gruber is the main developer of the LIKWID tool suite, which is taught and used during the workshops and for most performance-centric work on application code.

Judit Gimenez is an HPC expert at BSC and a member of the POP COE team. She participates in the workshop organization on performance analysis and optimisation.

Brian Wylie is an HPC expert and a member of the HPC application support at FZJ. He is actively involved in the POP COE and is a representing FZJ tools at the workshops.

Thierry Gautier is computer scientist and an expert in HPC with a strong expertise in asynchronism and task-based methods. He has joined the project specifically to provide some support in task parallelism especially for the development of GYSELAX. Thanks to EOCOE-II resources, Thierry Gautier has improved the tools he is working on for the community (libKOMP [7], Tikki). In 2019, he leads work that results in pushing two patches of the LLVM OPENMP in the master branch to improve performance in the management of task in the runtime. It also includes the development of a performance monitoring module using the tracing method for OMPT (a first-party API for third-party performance and monitoring tools in OPENMP-5.0) called TiKKi.

The experts were allocated to the codes according to needs and issues. The table 7 shows the distribution of experts between the codes. Details of the work carried out in the codes are presented in the corresponding sections.

4.2 WP2 events

The WP2 organizes workshops dedicated to the performance analysis and the code optimizations. Our calendar of events is given in Fig. 5. At the beginning of the project (as described in the D2.1), we had in mind to organize two types of workshop:

Application	Experts	Main issues and achievements
ALYA	Dominik Ernst	GPU optimisation of the matrix assembly part (memory consumption reduction, data transfer reduction, improved code generation), CPU optimisation, performance portability, MPI Load balancing
GYSELA	Markus Wittmann, Tobias Kloeffel, Judit Gimenez, Brian Wylie	x86 CPU code optimisation, A64FX code porting and optimisation
EURAD-IM	Thomas Gruber	Optimization approach of the Rosenbrock solver, vectorization

Table 7: Distribution of experts on applications

- A performance evaluation workshop to teach the tools and helps the team to determine their application bottlenecks. This first session was to ensure that all code developers, and particularly developers involved in code refactoring and optimisation, are on the same level of knowledge.
- At least two hackathon workshops dedicated to work in the codes, developers and HPC experts together. Hackathons should therefore gather HPC experts and application developers to work on specific optimisation issues during approximately 3-day. They enable to overcome strong performance bottlenecks or complex optimisation challenges for application developers. They also help to track the optimisation progress and update performance-aware code development strategies.

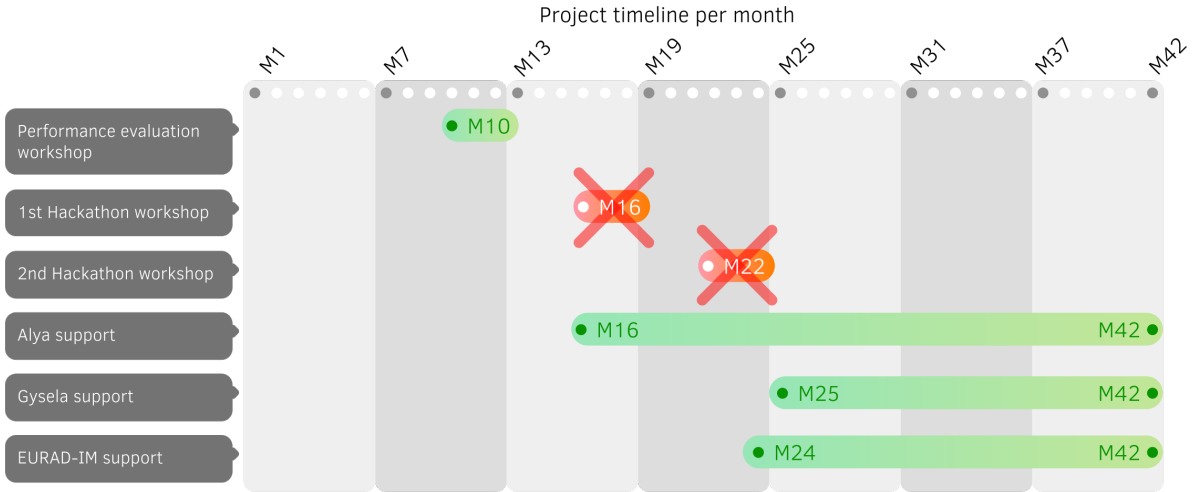


Figure 5: Events organized by the WP2.

The first performance evaluation workshop was held in Erlangen (Germany) from October 7 to the 10 2019 [8] (M10). It was co-organized and hosted by the FAU University. We have as well partnered with the POP COE [9] to propose the tools developed at BSC. The workshop was planned as follow:

- Two days were dedicated to the presentations of the CPU core architecture and the performance evaluation tools developed at FAU (LIKWID) alternating lectures and hands-on.
- The third day was dedicated to the POP COE tools (Paraver, Scalasca) alternating as well lectures and hands-on.

The workshop proved to be a great success. A picture of the training room is shown in Fig. 6. We have welcomed 14 attendees from 8 different institutions. They were representing 12 different applications

Figure 6: WP2 performance evaluation workshop held in Erlangen (Germany) from October 7th to the 10th 2019.

with 6 being EoCoE-II applications or libraries. The workshop general presentations (not including the hands-on) have been recorded and put online [10].

The second and third workshops should have been hackathon. Many application developers within EoCoE-II were waiting for it to start the close collaboration with HPC experts. Because of COVID-19, they were both cancelled. As these workshop was eagerly awaited by multiple teams, especially the first one, to start the search for blocking points and code optimisation, we invited the EoCoE-II developers to start remote point-to-point studies with the experts. We also proposed to the team to participate to the PRACE online and local training courses to meet the needs. The possibility of doing the hackathon online was not retained because the format is not appropriate. Nonetheless, specific remote working sessions between application developers and HPC experts were set up. In the end, this working method, although less user-friendly than face-to-face hackathons, allowed the WP2 HPC support objectives to be achieved.

4.3 PRACE computational resources

Every 6 months in March and September, our Centre of Excellence has received computation hours from PRACE on Tier-0 machines. The WP2 manages the computing resources allocated for the whole project. The new batch is divided between all Centres of Excellence depending on their needs. To evaluate our needs, the PRACE proposition is first scattered toward all our members. Then all members indicate what they need and a common proposition is therefore sent to PRACE for examination.

Table 8 summarizes the amount of hours granted to EoCoE-II per super-computers.

So far, we have been granted a total amount of around approximately 15 million core hours (sum over all super-computers). If we go into detail, not all machines are used at the same level. Some of them are rapidly used at the maximum of their capacity like Marenostrum. On the contrary, some machines are just requested for testing new implementation and optimisation. It happens that the amount has been overestimated and not totally used. In any case, this computational time is extremely useful for the project.

Note that this table represents a small part of the whole available resources since it does not take into account local resources at institution scale and PRACE or national access to super-computers external to the project.

Super-computer	Granted core hours
Marenostrum 4	1 424 167
SuperMUC	150 000
SuperMUC NG	454 000
Juwels	2 555 000
Juwels Booster	74 000
Joliot-Curie KNL	170 000
Joliot-Curie SKL	765 917
Joliot-Curie ROME (AMD)	1 282 000
Piz Daint	5 346 644
Marconi Broadwell	155 000
Marconi KNL	770 000
Marconi 100	2 080 000
Hawk	1 555 000
Total	17 248 712

Table 8: PRACE resources for EoCoE-II.

5 Task 2.2 - Wind code optimisation

5.1 Task overview

Task leader: BSC

Participants: BSC, FAU, IFPEN

The wind objective is to bring the Large Eddy Simulation (LES) formulation for wind farm simulation to the Exascale. In term of numerical simulation, a typical production runs should reach a resolution of 10^{10}-10^{11} grid points on unstructured grids with approximatively 1 day time-to-solution on a Exascale machine. In term of scientific purpose, the goal is to perform multiscale LES modelling of fluid-structure interactions in turbine blades and model entire wind farms with complex terrains (see WP1). For this aim, a full rotor model where the actual geometry of the wind turbine is modelled exactly should be implemented.

In the WP2, the wind challenge involves the flagship code Alya and 2 satellite codes waLBerla and Meso-NH. A brief summary of application properties and purposes is respectively given in the following sections section 5.1.1, 5.1.2 and 5.1.3.

The work to be done in these codes has been divided into 3 subtasks:

- Task 2.2.1 - Alya code refactoring and optimisation for Exascale
- Task 2.2.2 - waLBerla actuator line code extension
- Task 2.2.3 - Performance comparison between waLBerla, Alya and Meso-NH (replacing SOWFA)

The detailed content of these tasks and the progress achieved so far is described in sections 5.2, 5.3, 5.4.

5.1.1 Flagship code Alya

Alya [11] is a high-performance computational mechanics code that solves complex coupled multi-physics problems, mostly coming from the engineering realm. The code is developed at BSC (Alya website).

The main goal for ALYA is to bring the code to Exascale to tackle the simulation of full wind farm over complex terrain with up to 100 wind turbines. Within WP2, ALYA 's developers with HPC experts are refactoring and optimizing the code to be able to address heterogeneous computing nodes with maximal efficiency. They will implement a full rotor model where the actual geometry of the wind turbine is modelled.

Table 9 shows the team members of ALYA involved in EOCOE-II. Herbert Owen is a senior researcher at BSC. He has been leading the Wind Scientific Challenge since EOCOE-I. He coordinates wind energy developments of ALYA and represents this code in EOCOE-II. Guillaume Houzeaux is the manager of the Physical and Numerical Modelling group at BSC and one of the main developers of ALYA.

People	Position	Role	Period
Herbert Owen, PhD	Senior researcher at BSC	Responsible for the ALYA team within EOCOE-II and developer of the code	M1-M36
Guillaume Houzeaux, PhD	Physical and numerical group manager at BSC	Main Code developer	M1-M36

Table 9: Team Members for ALYA *within* EOCOE-II.

The work in ALYA is described in task 2.2.1 (see section 5.2) and task 2.2.3 (see section 5.4).

5.1.2 Satellite code WALBERLA

WALBERLA is a fluid simulation code that uses the lattice Boltzmann method (WALBERLA website). WALBERLA is developed at the Friedrich-Alexander University of Erlangen-Nuremberg (FAU). In WP2, WALBERLA developers will implement an actuator line model. The final goal is to be able to simulate wind turbine with the lattice Boltzmann method and to compare the results with the flagship code ALYA and the code MESO-NH (replacing SOWFA).

Table 10 shows the team members of WALBERLA involved in EOCOE-II. Ulrich Ruede is the code Coordinator at FAU. Helen Schottenhamml has been hired at M9 at FAU as a research assistant to work on WALBERLA for a duration of 8 months (until end of March 2020). She is in charge of the work in WALBERLA described in task 2.2.2 She was supposed to then move to IFPEN in France on April 1st, 2020 (M16) until M27, but the COVID-19 crisis prevented her to change her location. She worked from Erlangen for IFPEN until she could move to France in June 2020.

People	Position	Role	Period
Ulrich Ruede, PhD	FAU	Responsible for the WALBERLA code	M1-M36
Ani Anciaux-Sedrakian, PhD	IFPEN	Code optimisation and development	M1-M33
Frédéric Blondel, PhD	IFPEN	Code optimisation and development	M1-M33
Helen Schottenhamml, M.Sc.	PhD student at FAU and Engineer at IFPEN (M16-M27)	Code optimisation and development	M1-M33

Table 10: Team Members for WALBERLA *within* EOCOE-II.

WALBERLA is concerned by task 2.2.2 (see section 5.3) and task 2.2.3 (see section 5.4).

5.1.3 Satellite code MESO-NH

MESO-NH is the non-hydrostatic mesoscale atmospheric model of the French research community (MESO-NH Website) dealing with scales ranging from synoptic (1000 km scale) to large eddy scales (meter scale). It has been jointly developed by the Laboratoire d'Aérologie (UMR 5560 UPS/CNRS) and by CNRM (UMR 3589 CNRS/Météo-France).

MESO-NH has substituted SOWFA that was the code originally given in the proposal for task 2.2.3 at IFPEN. As already explained in the previous deliverables, they are several reasons that have motivated this choice. First, although MESO-NH is a LES code like SOWFA, it is more advanced from a meteorological point of view. MESO-NH can model more thermo-dynamical phenomena such as radiation, deep and shallow convection. It embarks advanced physical parameterizations for cloud and precipitation representation. It can be coupled with different modules for chemistry (aerosol...) or complex surface (vegetation, cities, ocean...) for instance. Then, MESO-NH is more advanced in term of HPC (Good scalability, vectorization) and is actively supported. The last argument to use MESO-NH is the size of the benchmarks. Simulated domains for EOCOE-II have a size of 40 km by 40 km much higher than the size usually considered in MESO-NH simulation at IFPEN.

Table 11 shows the team members of MESO-NH involved in EOCOE-II. Marie Cathelain is engineer at IFPEN in charge of coordinating the work in MESO-NH for the task 2.2.3.

People	Position	Role	Period
Marie Cathelain, PhD	Engineer at IFPEN	Responsible for the MESO-NH code	M1-M36

Table 11: Team Members for MESO-NH *within* EOCOE-II.

MESO-NH appears in task 2.2.3 (see section 5.4).

5.2 Work progress on task 2.2.1

Task 2.2.1 corresponds to the refactoring and the optimisation of the code ALYA. It aims at optimizing ALYA for Exascale to run complex terrain and full rotor with the required accuracy. It contains the following subtasks:

- PDI or Sensei integration for in-situ visualization in WP4 and WP5
- ALYA general code optimisation: Code cleaning, node-level optimisation and vectorization, Dynamic load balancing (DLB package), MPI overlapping between communication and computation, hybrid GPU implementation, coexecution on heterogeneous cluster (CPU + accelerators), Fast and scalable geometric mesh partitioning based on Space Filling Curve, Dynamic coupling between rotating meshes that following turbine blades and fixed mesh for the rest
- Scaling to Exascale: Running real cases on exascale or pre-exascale machines: complex terrain and full rotor (rely on the speedups reachable with optimizations).

Although it was not originally mentioned in the proposal, we include in this task the work performed with the code MESO-NH reserved for code comparison in subtask 2.2.3.

Fig. 7 describes the current work plan for task 2.2.1.

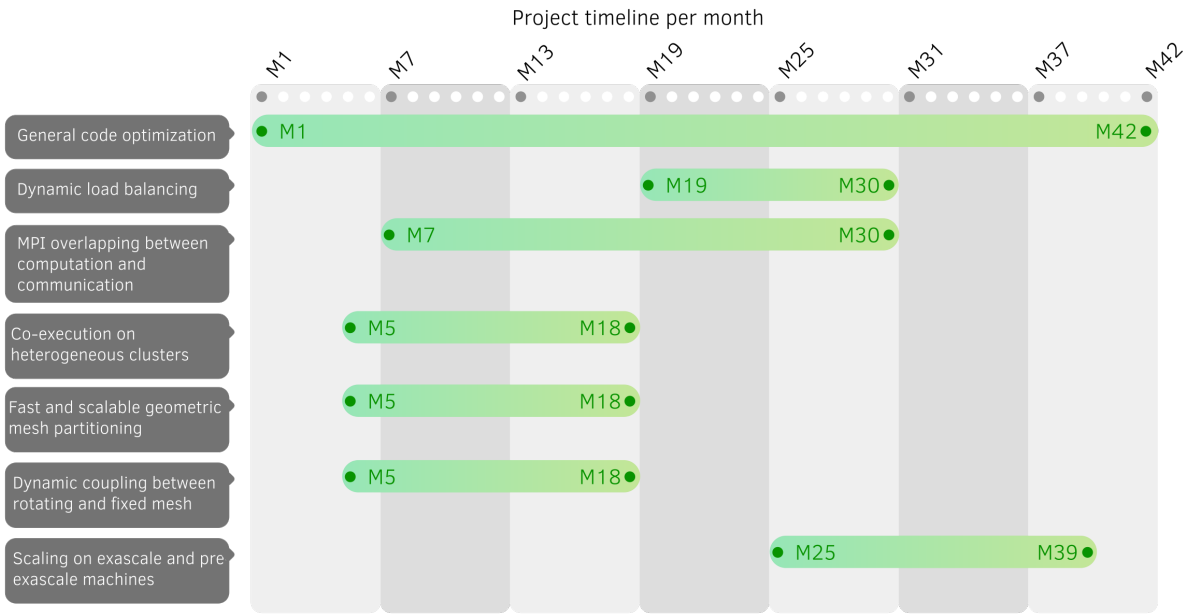


Figure 7: Breakdown (simplified Gantt chart) of the task 2.2.1 for ALYA.

5.2.1 Work progress in ALYA

The fractional step method, used to solve the Navier Stokes equations, has three basic steps [12]. In the first one, an intermediate velocity is obtained. The two main computational costs are obtaining the right-hand side vector and multiplication by the inverse of the lumped matrix. In a typical CFD with ALYA run on a CPU, obtaining the right-hand side vector is the most expensive operation and takes 70 to 80% of the total run time. The multiplication by the inverse of the lumped matrix is very cheap. The second step is the solution of a Laplacian equation for the pressure. It involves multiplication by a sparse divergence matrix, which is not too computationally demanding, and the solution of a linear system for the pressure. In typical CFD simulations with ALYA, the solution of a linear system takes around 15% or 20% of the total time. The final step, where the incompressible velocity is obtained, is cheap. It involves multiplication by a sparse Gradient matrix and by the inverse of the lumped mass matrix.

Machine	Initial	Final	Improvement
1 A100 GPU + 9 AMD cores	9.8 s	0.07 s	140X
1 Intel Skylake node - 48 cores	1.93 s	0.49 s	3.9X
Improvement	CPU 5X faster	GPU 7X faster	

Table 12: CPU and GPU optimisation - Timing for the right-hand side assembly.

The GPU optimisation has concentrated on the most time-consuming step, obtaining the right-hand side vector. In Tab. 12, we compare the initial and final CPU and GPU computational times per time step for a mesh with 5.6 million nodes and 32 million elements corresponding to a well-known wind benchmark called Bolund. The initial version, named version hh71, is suitable for both CPU and GPU. The final optimised versions are called either hh80 or hh91. Specific subroutines for the CPU and GPU are developed to obtain a much better performance on the GPU.

An enormous increase in performance has been obtained on the GPU between the original version (hh71) and the final version (hh91). The case is run in one A-100 NVIDIA GPU and 9 AMD cores. Part of the improvement is given by the fact that in the initial version, the turbulent viscosity is calculated in an external subroutine that is not yet ported to GPU. In the final version, the turbulent viscosity is calculated (on the

GPU) in the same subroutine where it is used. The cost of obtaining it in this way is negligible. In an identical laminar run, where no turbulent viscosity is calculated, the time for the original implementation on the GPU is 5.61 s. Thus, the increase in performance falls from 140 times to 80 times. That is still a considerable improvement. As usual, improvements that are suitable for the CPU were observed when optimizing for the GPU. That led to the version hh80, which is 3.9 times faster than the current implementation, on a Marenostrum IV Intel Skylake 8160 node.

From our point of view, the essential value is that the optimised GPU implementation is seven times faster on an A100 NVIDIA GPU than the optimised CPU implementation on an Intel Skylake 8160 node. While ALYA is part of several CoEs, we believe the strong interaction with the FAU node level optimisation team has allowed EOCOE-II to obtain the first efficient ALYA GPU implementation. It seems to be the most efficient low-order finite element implementation in the literature. Since energy efficiency is probably the critical enabler of exascale, we briefly estimate the energy consumption for the previous runs. To do so, we obtain the energy consumption from the Top500 list [**?**]. For Marenostrum IV, we can divide the total energy consumption (1,632 KW) by the number of nodes to obtain an energy consumption of 511W per node. For the GPU calculations, we have used a tiny cluster that is not part of the Top 500. Each node is very similar to those of the Perlmutter Supercomputer (7th in the June 2022 list), with an energy consumption of 1685 watts per node. We use only 1 GPU out of 4 in the node and less than one-quarter of the CPU cores. We thus estimate the consumption to be 421 Watts. Multiplying the energy consumption by the execution time, we obtain 250 Joules on the CPU and 29.5 Joules on the GPU. Thus, the optimised versions are 8.5 times more efficient on the GPU than on the CPU. To obtain a reference, we can compare the Power Efficiency for the two supercomputers from the Green 500 list: 27.374 GFlops/watts for Perlmutter and 3.965 GFlops/watts for MN-IV. This gives a Power Efficiency Ratio of 6.90, which is lower than the 8.5 ratio we obtain for ALYA 's assembly. We can claim that ALYA 's assembly is better suited for the GPU than for the CPU. This is an excellent result that we did not expect at the beginning of EOCOE-II. Moreover, it is interesting to remark that the optimised GPU version is 30 times more energy efficient than the current CPU implementation.

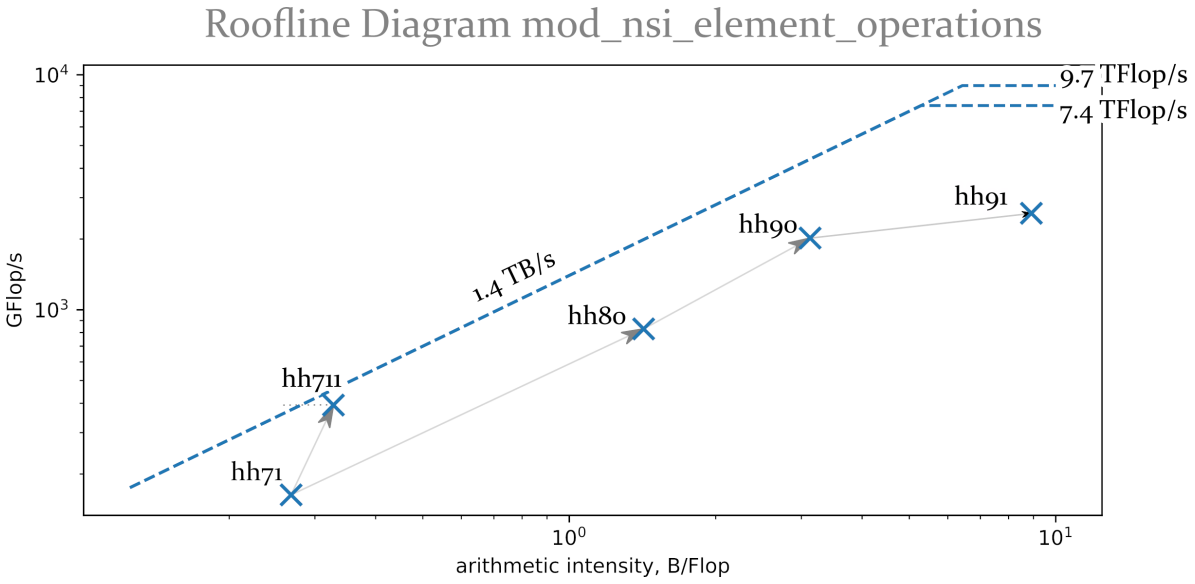


Figure 8: Roofline Diagram showing the progress of different optimisation steps of the assembly routine on a A100 40GB GPU. The optimisation steps on top of the base line (hh71) are: CPU/GPU independent specialization and restructuring (hh80), GPU specific privatization of intermediates (hh90), GPU specific restructuring (hh91). hh711 additionally shows the effects of only the privatization of intermediates.

Some optimizations are specific for the GPU, while others are valid on both the CPU and GPU. One of the optimizations for both architectures has already been mentioned; it consists of calculating the turbulent

viscosity with the Vreman model directly on the fly instead of obtaining it in a separate subroutine. A second optimisation consists in specializing only for tetrahedral linear elements whose shape functions are easy to calculate. Moreover, the gradient of the shape functions is constant within the element. This avoids the need to obtain different values for each gauss point and reduces calculations, memory, and registers. Since ALYA started with an implicit temporal treatment of the momentum equation, when switching to an explicit discretization, which is more suitable for Large Eddy Simulation, part of the original code was reused to save programming effort and obtain something suitable for both scenarios. Thus, instead of obtaining the elemental right-hand side vectors directly at the elemental level, the available elemental matrices needed for an implicit time discretization were multiplied by velocities from the previous time step to obtain the elemental right-hand side vector. In the optimised version, we have decided to abandon any retro compatibility with the previous implicit discretization and calculate the elemental right-hand side vectors directly, reducing calculations, memory, and registers. All of the previously mentioned optimizations are valid for both CPU and GPU.

As indicated in [13], a typical implementation of a finite element code consists of a loop over every single element. The following three steps are carried out for each element: Gather, Computation, and Scatter. A new data structure to enhance the efficiency of the assembly proposed in [13] consists of workings packs of elements of size PACKSIZE. The approach has a two-fold benefit. On the one hand, it improves data locality because it stores elements in dense packs. On the other hand, the code exposes the SIMD/SIMT potential to the compiler. As explained in [13], this approach has been used for both CPU and GPU leading to the unified implementation called hh71. However, during EOCOE-II, we have realized that the grouping is beneficial for the CPU implementation but not for the GPU implementation, thus leading to two separate subroutines. For the GPU implementation, we have realized that it is much better to recover the typical finite element approach where each element is treated individually with an OPENACC thread private clause (privatization of intermediates).

General Code Optimization The roofline diagram of the different versions of the assembly subroutine, in Fig. 8, shows the progression of the optimizations. The modifications reduce the amount of temporary data transferred between the DRAM and the GPU, increasing the arithmetic intensity. The measurement point labelled as hh711 is a version where only one of the GPU-specific optimizations (privatization of intermediates) has been applied. While this does not decrease the number of intermediates and therefore does not increase the arithmetic intensity as much, the utilization of the memory bandwidth is improved. The second, lower flop rate roof of 7.4 GFlop/s accounts for the specific instruction mix of fused-multiply-add, addition, and multiplication operations.

Dynamic Load Balancing Monitoring load balance. Dynamic load balance is, together with communication efficiency, one of the main performance issues in parallel programming. To monitor both the load balance and communication efficiency, we have integrated the TLP library, part of the DLB library [**?**], in ALYA.

The metrics collected by TALP are defined in the POP COE performance model, developed by the researchers of the European Centre of Excellence Performance Optimisation and Productivity, as shown in Fig. 9. The PoP metrics are a set of efficiency and scalability indicators that can be obtained for MPI applications. In our case, the efficiency metrics reported by TALP allow the user to obtain the parallel efficiency PE, which is split into communication efficiency CE and load balance LB. That is, PE = CE x LB. It is important to note that this model is multiplicative and that the parallel efficiency measures are absolute metrics while the computation scalability are relative metrics (relative to a base run).

Four of the builds of the ALYA performance suite [14] include TALP, to follow the different metrics proposed by TALP and to monitor its overhead. Fig. 10 shows the output of TALP as it appears in ALYA. It should be noted that such measures are necessary to take actions at runtime or for future executions if a high load

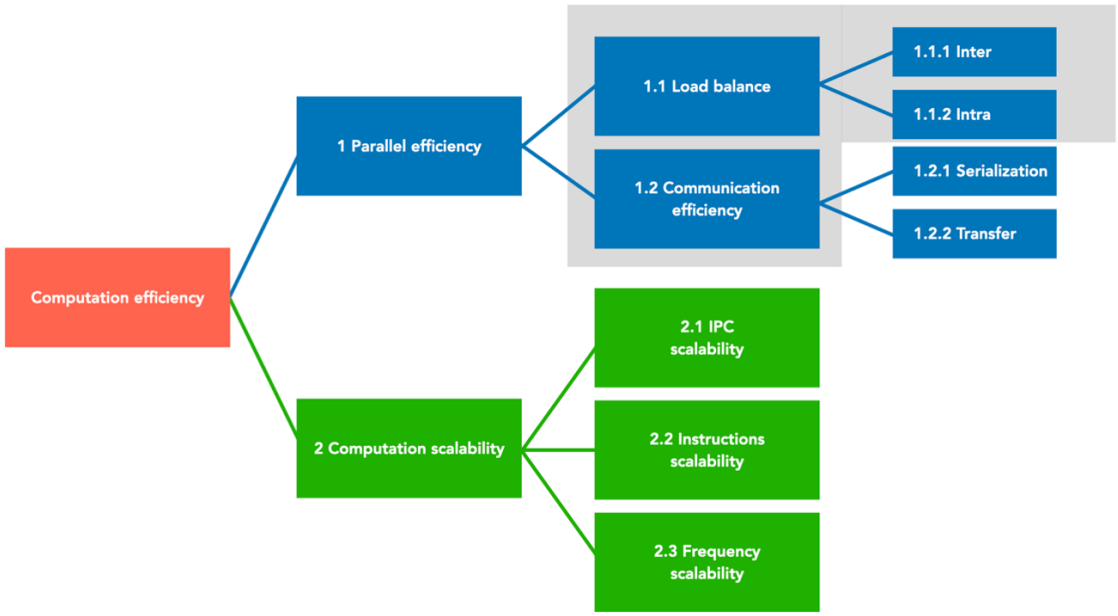


Figure 9: PoP metrics. The metrics inside the grey square are the one measured by TALP at runtime.

imbalance is detected. In fact, the library provides intra and inter-node load balance measures.

```
DLB[nvb6:28560]: dlb 3.1a
DLB[nvb6:28560]: ######## Monitoring Region App Summary ########
DLB[nvb6:28560]: ### Name:                        MPI Execution
DLB[nvb6:28560]: ### Elapsed Time :               107.85 s
DLB[nvb6:28560]: ### Parallel efficiency :        0.85
DLB[nvb6:28560]: ###    - Communication eff. :    0.88
DLB[nvb6:28560]: ###    - Load Balance :          0.97
DLB[nvb6:28560]: ###       - LB_in :              0.98
DLB[nvb6:28560]: ###       - LB_out:              0.99
DLB[nvb6:28560]: ######## Monitoring Region App Summary ########
DLB[nvb6:28560]: ### Name:                        region module 1
DLB[nvb6:28560]: ### Elapsed Time :               49.84 s
DLB[nvb6:28560]: ### Parallel efficiency :        0.90
DLB[nvb6:28560]: ###    - Communication eff. :    0.94
DLB[nvb6:28560]: ###    - Load Balance :          0.96
DLB[nvb6:28560]: ###       - LB_in :              0.97
DLB[nvb6:28560]: ###       - LB_out:              0.98
```

All the code

User-defined region

LB_in: intra-node load balance

LB_out: inter-node load balance

Figure 10: Output of TALP library and performance metrics.

Enhancing intra-node load balance. The DLB library, developed at BSC-CNS, was soon integrated in some modules of ALYA, in 2016 to enhance the intra-node load balance. The library improves the load balance of the outer level of parallelism by redistributing the computational resources at the inner level of parallelism. This readjustment of resources is done dynamically at runtime. This dynamism allows DLB to react to different sources of imbalance: Algorithm, data, hardware architecture and resource availability among others. In some publications, the efficiency of DLB was demonstrated through the solution of different use cases and published in different papers [15]. The library originally required a source-to-source translator for FORTRAN named Mercurium to be used with OmpSs. However, at the beginning of the project, the support of Mercurium to Fortran2008 was stopped. The implementation of DLB to be compatible with OPENMP, without the need for a translator, was then started. However, this version is still now in test period, so we could not carry out the integration on time. We thus decided to redirect efforts on heterogeneous architectures, as demonstrated by the work done on GPU implementations.

Enhancing inter-node load balance. Inter-node load balance appears between the different nodes of the supercomputer. The strategy chosen to solve possible imbalance issues was redistribution through MPI [13]. Such an implementation has been carried out and enables redistribution at runtime using real CPU measures or TALP. Fig. 11 summarizes the strategy.

MPI overlapping between computation and communication Regarding the overlap of communication and work, we have focused on the Pipeline Conjugate Gradient (PCG). This solver involves more computation than the classical CG but offers the possibility to overlap the global communications (consisting of the synchronization point at each iteration) with the preconditioning step. Asynchronous communication is obtained through the MPI3 MPI_IAllReduce function. Fig. 12 shows the different versions of the CG that have been proposed in the literature. The last two were implemented in ALYA.

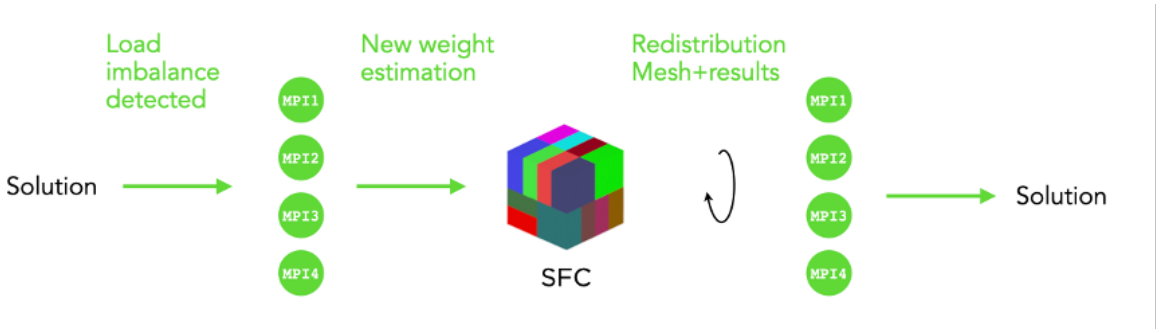


Figure 11: redistribution with MPI when a load imbalance is detected.

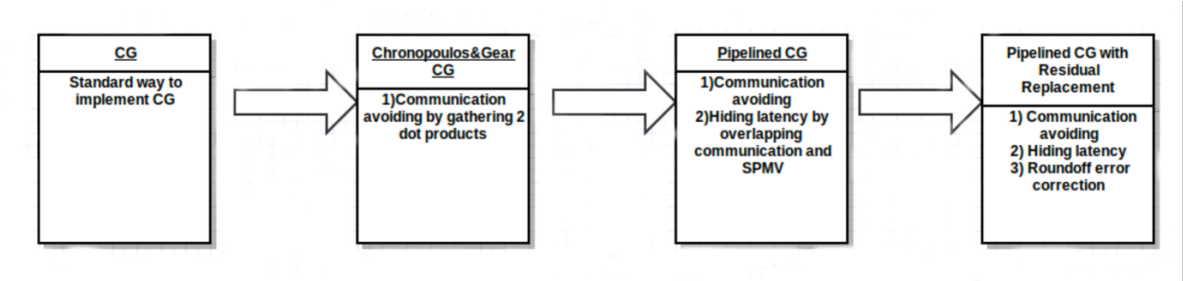


Figure 12: Different Conjugate Gradient (CG) implementations in the literature.

As already noticed in the literature, the first series of tests with the pipeline implementation showed that errors are amplified, leading to divergence or residual stagnation. We have implemented the PCG with residual replacement to remedy this, noted PCG-rr. Additional operations are required with respect to the PCG. In the test cases considered, the overlap between communication and operations was not compensated by the additional number of iterations required to reach the same residual as the CG and the additional operations involved by the PCG-rr. Fig. 13 shows the convergence history of one of the use cases solved on 512 CPUs, where we observe the highest number of iterations required by the PCG-rr with respect to CG and the residual stagnation for the PCG.

Due to the relatively poor results in terms of convergence and also time to solution, obtained from this work, we have decided to focus on the interfacings of external solvers provided by the partners of the project. You can see the deliverable D3.4 for more details.

Scaling on Exascale and pre-Exascale machines ALYA is one of the two CFD codes in the UEABS (Unified European Applications Benchmark Suite). As such, its scalability has been tested widely on most European Supercomputers. It has performed production runs with up to 100 000 cores on Marenostrum-IV

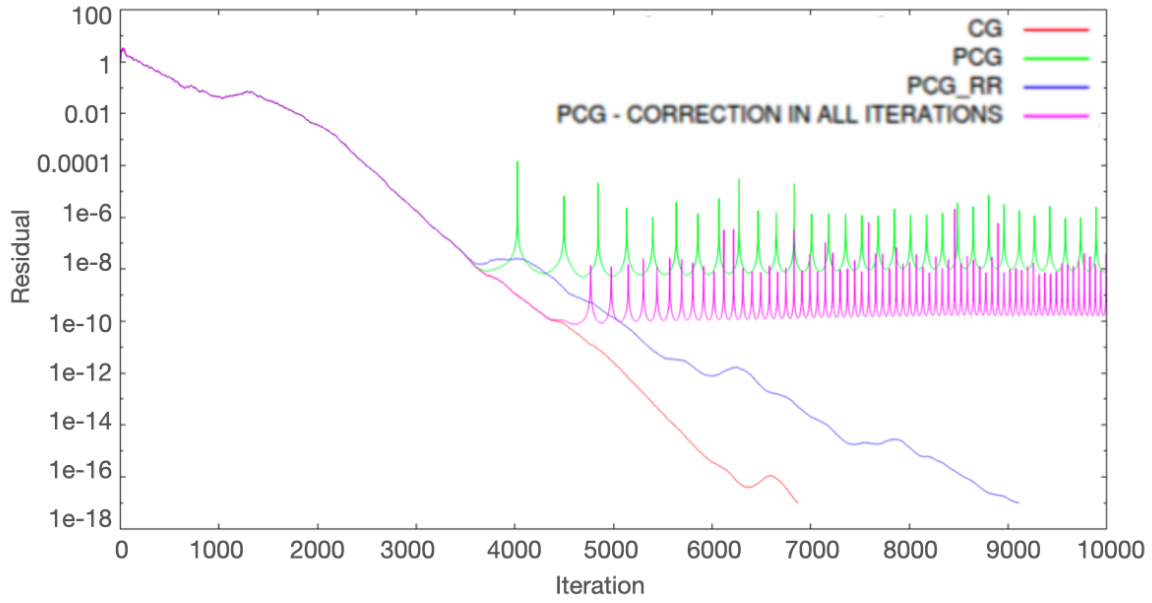


Figure 13: Convergence of the different Conjugate Gradient (CG) versions on 512 CPUs on the solution of an external flow.

for 24 hours. The solution of wind problems relies on the solution of the incompressible Navier Stokes using explicit turbulence models. The momentum equation uses an explicitly time discretization, and a fractional step scheme is used to uncouple velocity and pressure. Thus, the two most computationally expensive kernels are the assembly of the right-hand side vector for the momentum equation and the solution of a Laplacian linear system for the pressure. In a typical incompressible flow run, momentum right-hand side assembly takes close to 80% of the time and the solution of the linear system close to 20%. From the scalability point of view, the right-hand side assembly is trivially parallel, and the most challenging kernel in terms of scalability is the solution of the linear system for the pressure. For the solution of the linear system, we have interfaced with most of the linear algebra libraries available within EoCoE-II. A detailed description of their performance is presented in the deliverables of Work Package 3.

To study ALYA 's exascale scalability for wind problems, we simulate the well-known Bolund benchmark. The same mesh as in other sections of this deliverable is used; it has 5.6 million nodes and 32 million elements. Tab. 13 presents a weak scalability study starting from the previously cited mesh. To perform the weak scalability study, we use an automatic mesh subdivision algorithm [16] that splits each element into elements with half the size as shown in Fig. 14 leading to a mesh with eight times more elements and nodes for 3D problems. The mesh subdivision is applied recursively up to three times leading to a mesh of 16000 Million elements and 2800 Million nodes. The case without mesh subdivision (labelled d0) is run on one Juwels node using 46 cores. Each time a mesh subdivision is applied, both the computational resources and the mesh size are multiplied by eight. Thus, the most refined mesh (d3) is run on 512 nodes and 23552 cores. The results in Tab. 13 show the total computation time, which is also discriminated into linear solver time and 'Rest' (total – solver). It can be observed while the solver is not weakly scalable, the rest of the simulation is perfectly scalable. The solver scalability issues are solved in WP3, and the improved results using a multigrid solver are presented in deliverable D3.3.

5.2.2 Work progress in MESO-NH

Simulation work with MesoNH is done. The details concerning the comparison of scientific, numerical and physical results with WALBERLA-WIND and experimental data are in deliverable D1.2.

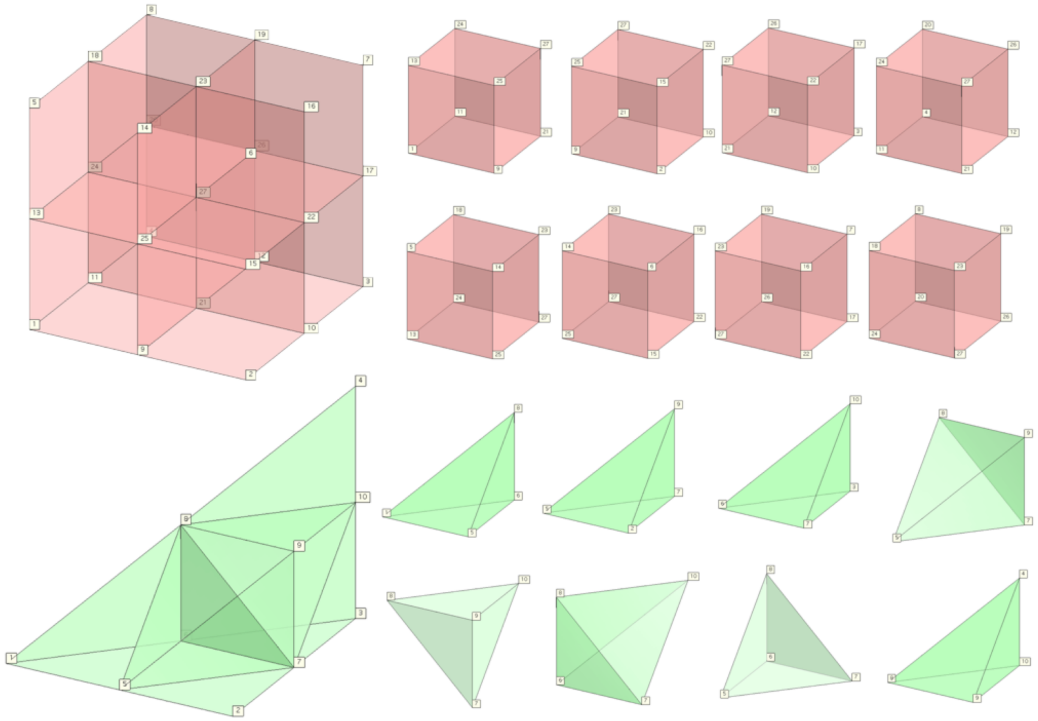


Figure 14: Mesh subdivision.

Case	cores	Million of Elements	Total time [s]	Solver [s]	Rest [s]
d0	46	32	11.11	1.57	9.54
d1	368	256	10.54	1.20	9.34
d2	2944	2048	11.38	1.94	9.44
d3	23552	16384	14.08	4.66	9.42

Table 13: Weak scalability

5.3 Work progress on task 2.2.2

The main objective of this task is to test an actuator line model in WALBERLA running on CPU. The work plan for this task was first updated in D2.1. Following the proposal and the first deliverable, this sub-task can be divided into the following points:

- WALBERLA code preparation for wind turbine
- Integration of the actuator line model
- First performance results on a single wind turbine
- Extension of the WALBERLA models from a single wind turbine to wind farms

Fig. 15 describes the current work plan for task 2.2.2.

The detailed content of these tasks are already reported in deliverable D1.2. In fact, we are developed an holistic actuator line model (ALM) approach based on WALBERLA platform. It provides a mutual code base for CPU and GPU. Not only it conserves the performance-portability of WALBERLA, but it also reduces the

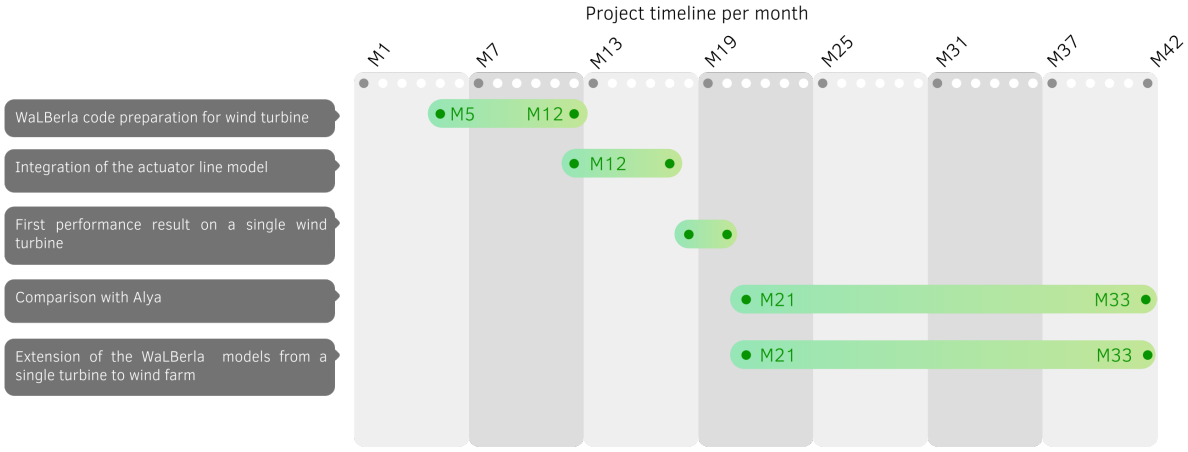


Figure 15: Breakdown (simplified Gantt chart) of the task 2.2.2 for WALBERLA.

maintenance efforts. The chosen strategy using WALBERLA allows us to supports shared and distributed memory parallelism with OPENMP and MPI, respectively, automated SIMD vectorisation, and the execution on NVIDIA graphics cards. In order to compare our new development called WALBERLA-WIND to ALYA, we improve it to handle mesh refinements on the CPU. The preliminary implementation results are illustrated in Fig. 16.

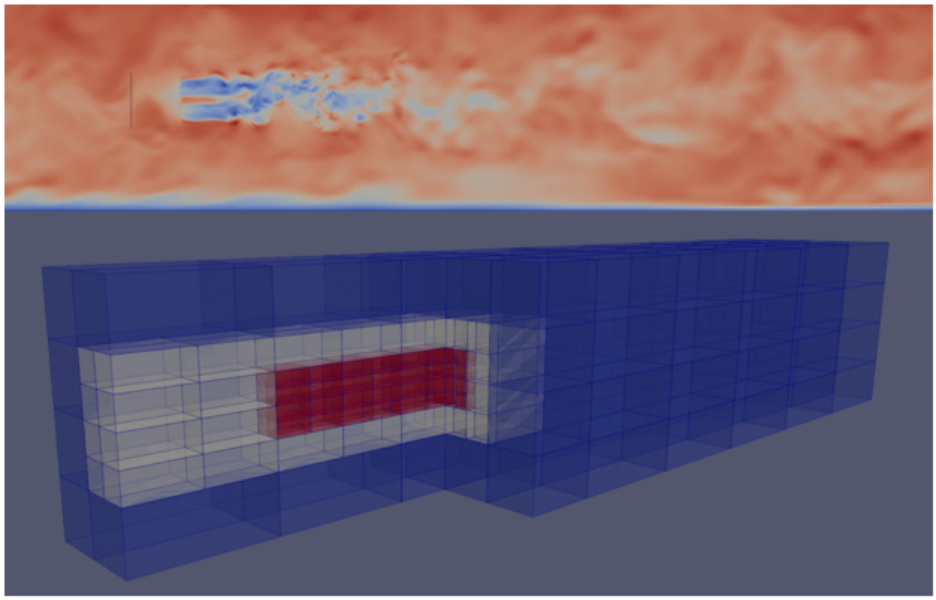


Figure 16: Illustration of WALBERLA-WIND *instantaneous velocity fields (top) and meshes (down) as employed during the simulation*

In the following section we illustrate the physical, numerical and performance results.

Scalability. In this section, we focus on the performance aspects of WALBERLA-WIND and its holistic ALM approach. All simulations run on the *Topaze* supercomputer at CCRT/CEA. Topaze has 864 compute nodes based on 2 AMD Milan@2.45GHz (AVX2) CPUs with 64 cores per CPU. Furthermore, it includes an accelerated partition with 48 compute nodes with 4 NVIDIA A100 GPUs each. The setup is described in Section5.4.1. Fig. 17(left) depicts the weak scaling of WALBERLA for CPU and GPU runs and clearly shows that the excellent behaviour of WALBERLA-WIND. Fig. 17 (right), on the other hand, compares

the strong scaling behaviour of the CPU and the GPU implementations of waLBerla in terms of mega lattice site updates per second (MLUPS). Here, we observe no perfect but still a favourable performance increase with an increasing number of compute nodes. For the GPU runs, the performance increase stalls for more than four nodes with four GPUs each. The simulation domain in these runs was too small to add sufficient workload to all GPUs, hence not further decreasing the time-to-solution. However, the weak scaling proves that we still scale when we provide a sufficient workload. These results have been published at the TORQUE conference 2022 [17].

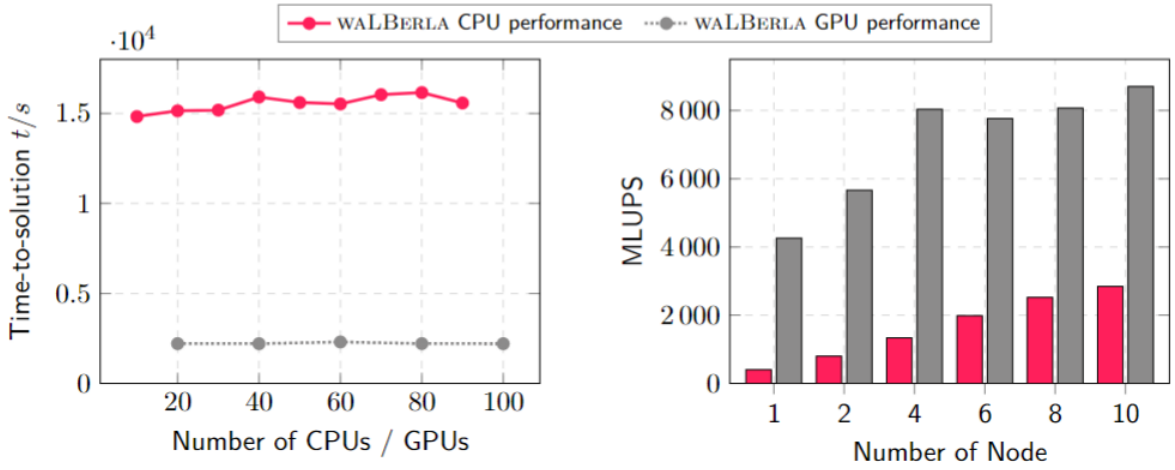


Figure 17: waLBerla *scaling experiments for 1200 s simulated physical time with* $\Delta t = 0.010054$ *s: weak scaling with 40 960 000 cells per node (left), strong scaling with 163 840 000 cells (right)*

5.4 Work progress on task 2.2.3

The main objective of this task is the performance comparison of the three codes using the flow over flat surface with wind turbine as a simulation case.

Fig. 18 describes the current work plan for task 2.2.3.

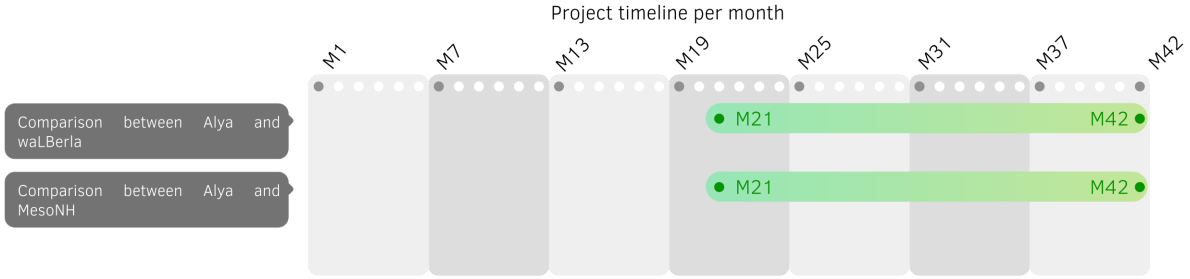


Figure 18: Breakdown (simplified Gantt chart) of the task 2.2.3 concerning the code performance comparison.

The code performance comparison has been split in two parts. At first, comparisons have been performed including SOWFA, Meso-NH and waLBerla. These comparisons have been performed using uniform meshes and a single wind turbine. This work has been published at the TORQUE conference 2022 [17]. In a second step, comparisons have been performed between waLBerla and Alya. In this test case, a single wind turbine was considered using an actuator-disk in Alya and actuator-lines in waLBerla. Mesh refinement strategies have been employed in waLBerla, while Alya employed a non-uniform mesh, more refined near the wind turbine.

5.4.1 Comparisons between waLBERLA-wind, MESO-NH and SOWFA

These comparisons (excluding MESO-NH) have been published [17]. Only the main results and additional comparisons with MESO-NH are given here.

The proposed study uses the generic DTU 10MW reference wind turbine [18] which is representative of modern large offshore wind turbines. We compare the time-averaged aerodynamic force distributions along the blade, the wake characteristics, i.e., velocity deficit and turbulent intensity profiles, predicted by the different solvers, and the code performance and computational times. To enforce a mutual boundary setup between the solvers, we use free-slip conditions at the bottom and the top of the domain, periodic lateral boundaries, and a constant uniform inflow at the inlet. The mesh resolution reaches 64 cells per rotor diameter, which is sufficiently fine to predict the wake properties correctly [19]. The simulations start with an initial period of 10 min of physical time, i.e., approximately 64 rotations of the wind turbine, during which the wake develops. Then, we evaluate the quantities of interest averaged over 10 additional minutes of physical time. The total mesh size is about 164 cells.

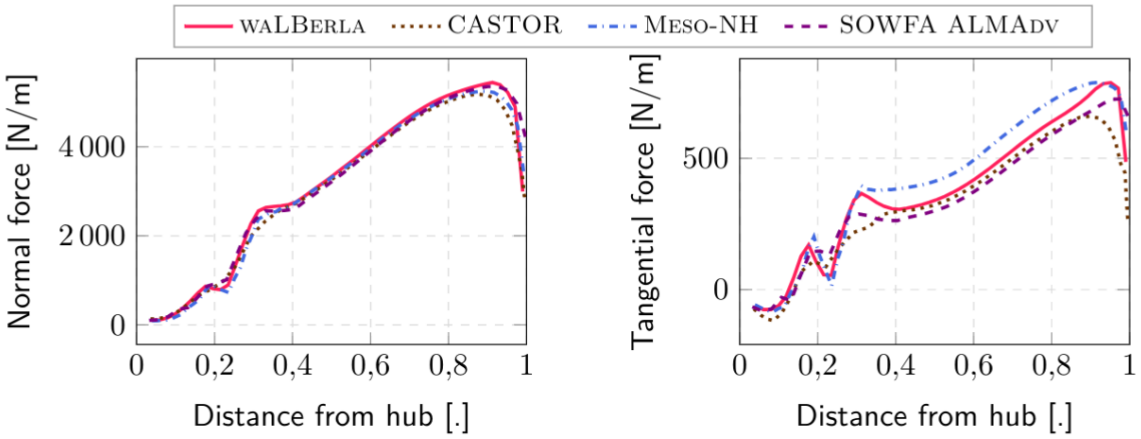


Figure 19: Normal (left) and tangential (right) force distribution along the blade, 64 cells per diameter

As shown in figure 19, blade forces compare well between the different solvers. Some slight differences are noticed and can be attributed to the differences in the blade force projection kernels and/or the velocity interpolation stencils. Some issues have been noticed near the outlet boundary for the MesoNH simulations, and the velocity interpolation method has been recently improved. Thus, the presented results should be considered has preliminary.

Fig. 20 also shows that the wake velocity profiles compare wall, in both near and far wake regions. Some slight discrepancies are observed in the intermediate wake region, i.e., near $x/d = 5$: MESO-NH velocity profiles are more smooth, indicating an earlier transition to the turbulent regime. Again, this is to be put in perspective with the issue encountered at the outlet boundary. Otherwise, waLBERLA-wind and SOWFA results match very well.

Fig. 21 shows the performance of both waLBERLA-wind and SOWFA. In terms of time-to-solution (right), waLBERLA appears to run approximately 75 times faster than SOWFA, which is a huge performance gain. Using 5 GPU nodes instead of 5 CPU nodes, the ratio increases to approximately 471. In terms of speed-up compared with 5 node simulations, both solvers show very good performances.

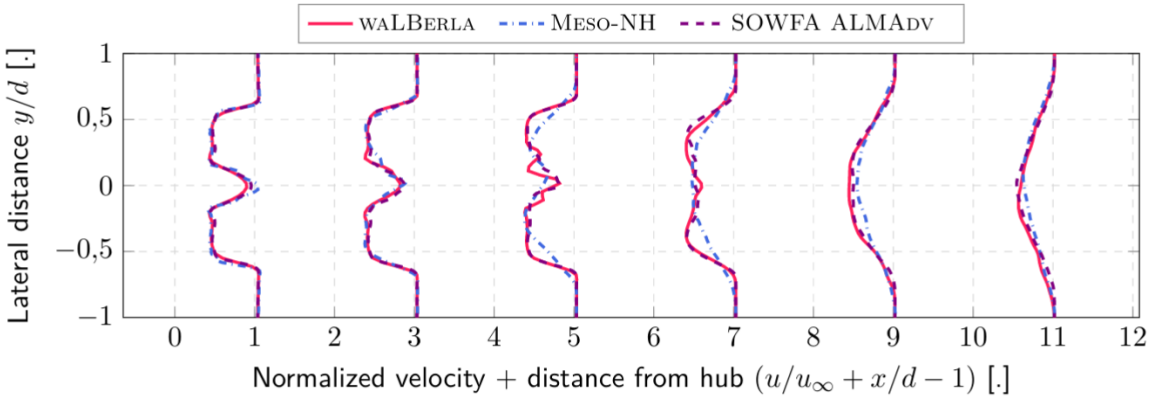


Figure 20: Wake velocity profiles, 64 cells per diameter

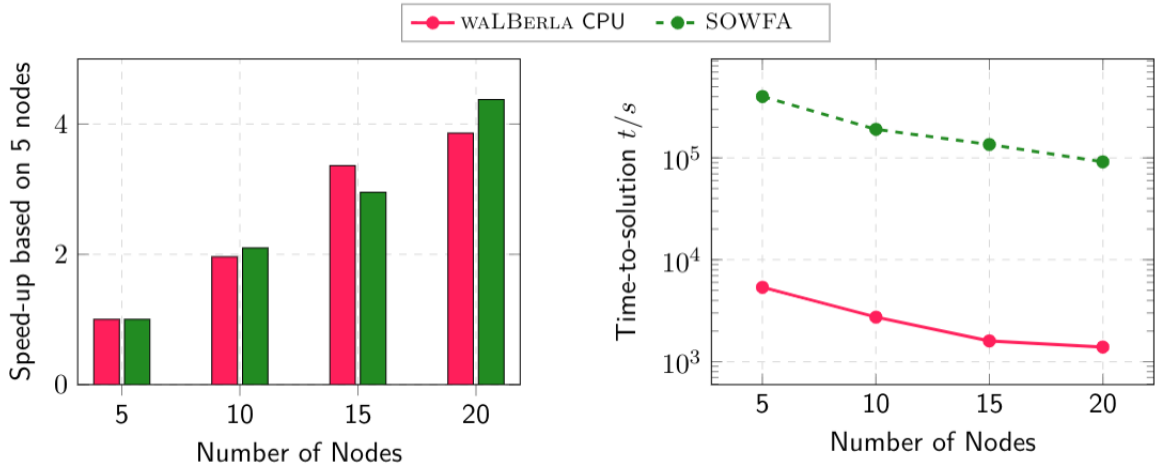


Figure 21: Strong scaling experiment with 163 840 000 cells, 1200 s simulated physical time with $\Delta t = 0.026$ s: Speed-up based on the simulation time on five nodes (left), performance in time-to-solution (right)

5.4.2 Comparisons between waLBerla-wind and Alya

The performance comparison of waLBerla-wind and Alya using the flow over flat surface with wind turbine as a simulation case are evaluated. Alya employed a constant loading actuator-disk to represent the DTU 10MW wind turbine, while waLBerla uses an actuator-line model of the same turbine. Due to the different numerical methods and mesh requirements between the two code, we could not employ the same meshes on both sides. Alya uses a non-structured grid, with vertical stretching and a finer resolutions near the wind turbine. And the other side, waLBerla-wind uses a structured mesh together with refinement boxes around the turbine and in the wake. Typical meshes can be observed in Fig. 22. The waLBerla-wind domain is periodic with a no-slip condition at the bottom, while the Alya domain consist of a constant velocity inlet free outlet, and a no-slip condition at the bottom.

Two waLBerla simulations have been run, the first one with a mesh of 4.2 and 5.6 lattice. A single Alya simulations was run, with a mesh consisting of 4.7 nodal points and 8.2 million elements. All these simulations were run on 96 cores of the AMD Milan Topaze supercomputer CPUs. A total of 2340 s of physical time was simulated, leading to a total of 136000 time steps for both solvers. Performance results are the following: the 4.2 and 5.6 million lattice simulations take respectively 7420 and 8088 s (CPU time) to run, while the Alya simulation take 160513 s. This leads to a CPU time ratio of about 20 between the two

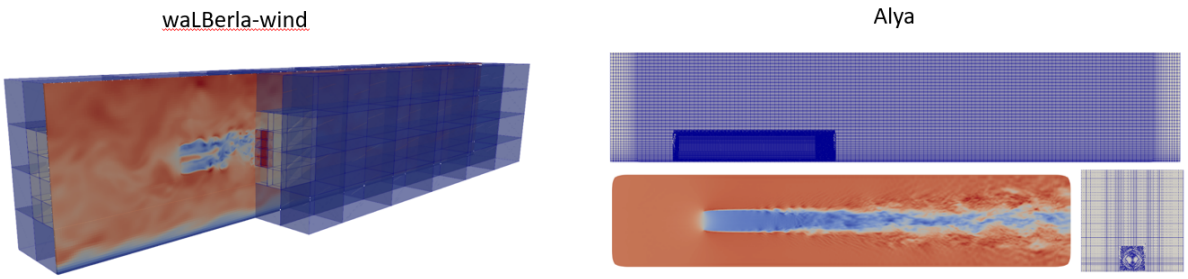


Figure 22: Illustration of WALBERLA *(left) and* ALYA *(right) meshes as employed during the simulations*

codes. Comparing an unstructured finite element code with a LBM code in terms of pure velocity is quite difficult. It is like trying to compare a structured Finite Difference code with an unstructured Finite Volume code, which according to Professor Hugo Piomelli results in an advantage of two orders of magnitude for the FD approach with the same number of unknowns. Despite the difference, both approaches are still very valid nowadays. The same happens with unstructured finite elements and the LBM approach. The former is better suited for complex geometries, such as onshore wind farms over complex terrain, while the latter is better suited for simple geometries, such as offshore wind farms. It is interesting to note that in the previous subsection it was found that waLBerla-Wind is 75 times faster than the unstructured grid finite volume code SOWFA while it is "only" 20 times faster than the unstructured finite element code ALYA. This is a clear indication of the advantages of ALYA over SOWFA. In the present study, based on simple geometries, the LBM approach performs better than the Navier-Stokes-based approaches, by more than one order of magnitude.

6 Task 2.3 - Meteorology code optimisation

6.1 Task overview

Task leader: FZJ

Participants: FZJ, FAU, CEA

The goal of the Meteorology scientific challenge is to improve weather forecasts (wind properties, cloud coverage, aerosols) for electricity production from solar and wind. Solar and Wind power prediction is performed using a framework gathering multiple codes working together. These codes, WRF (Weather Research Forecasting model [20] for meteorological analyses, and EURAD-IM (EURopean Air pollution Dispersion - Inverse Model [21]) for air quality assessments (with aerosol focus for EOCOE-II), are offline coupled and capable to perform large ensemble simulations of the order of 1000 members. The ensemble system is integrated into ESIAS. As the meteorological model WRF is a community code that is mainly maintained by NCAR (National Centre for Atmospheric Research, USA) only the code EURAD-IM, which is co-developed at FZJ, is concerned in WP2. The code and the related work is described in the following section.

6.1.1 Flagship code EURAD-IM

EURAD-IM simulates the formation and transportation of atmospheric chemical species and particles (aerosols) on the regional to continental scale with up to 1 km^2 horizontal resolution. It is offline coupled with the regional meteorological model WRF. An advection-diffusion-reaction equation, with multiple solvers for chemistry and aerosols, is used. As has been observed in previous studies, the stiff solver for gas phase chemistry is one of the main performance bottlenecks and most time consuming part in EURAD-IM.

People	Position	Role	Period
Hendrik Elbern, PhD	Senior scientist at University of Cologne (RIU)	Former Scientific coordinator for Meteorology	Retired
Garrett Good, PhD	Scientist at Fraunhofer Institute for Wind Energy Systems (Fraunhofer IEE)	Scientific coordinator for Meteorology	M1-M42
Philipp Franke, PhD	Postdoctoral fellow at FZJ	EURAD-IM code expert	M1-M42
Carl Burkert	software engineer at FZJ	Performance analysis and GPU porting of EURAD-IM	in-kind contribution, not funded by EoCoE-II, M1 - M42
Thomas Gruber	Software engineer at NHR@FAU	Performance analysis of ADCHEM solver	Funding unclear

Table 14: Team Members and contributors to the optimisation work in EURAD-IM *within* EoCoE-II.

The EURAD-IM is used in data assimilation applications, which besides particle filtering covers three and four dimensional data assimilation. The four dimensional data assimilation method requires the use of the adjoint model enabling to project model - observation discrepancies onto the initial state and emission data. Within EoCoE-II, the main focus was on the ensemble generation and efficient execution of the EURAD-IM, which do not require the adjoint model. However, code optimizations also impacted the adjoint model and needed to be tested as well.
The objective of WP2 is to improve the codes efficiency to address the Meteorology simulation challenges with main items:

- PDI integration (with CEA PDI experts) for IO optimisation in WP4 and ensemble runs in WP5,
- Code refactoring (with FAU) including change of data structure for vectorization and memory management,
- Node level optimisation (with FAU) and vectorization of the stiff gas phase ODE solver,
- Hybrid parallelization MPI + OpenMP/OpenACC to improve the parallelization on large-scale CPU machines first and leverage the possibility of GPU usage.

Table 14 shows the core team members and contributors to the optimisation work in EURAD-IM involved in EoCoE-II. Hendrik Elbern was the Meteorology Scientific Leader at the beginning of the project. He has retired at the end of 2019. Garrett Good is the new leader of the Meteorology SC. Philipp Franke, postdoctoral fellow at FZJ, is now coordinating activities around EURAD-IM in WP2. Carl Burkert is a software engineer and computer scientist at FZJ.

6.2 Goal and work summary of task 2.3

Fig. 23 describes the current and updated work plan for task 2.3. The optimisation work has been divided into subtasks in deliverable D2.1. Compared to the provisional dates provided in the first deliverable, we

have postponed certain tasks by a few months, partly due to the health crisis, partly due to the identification of model errors that needed to be removed. This was a time consuming effort and details are given below. Discussions with the experts on node level optimisation made the need for node level optimisation obvious, before porting routines to GPUs. Thus, focus was laid on the optimisation of the routines on node level. Due to the health crisis, the identified model errors, and the limited resources available for the optimisation of the EURAD-IM, most parts of this task could not be finalized until the end of the but will be finished in the upcoming month.

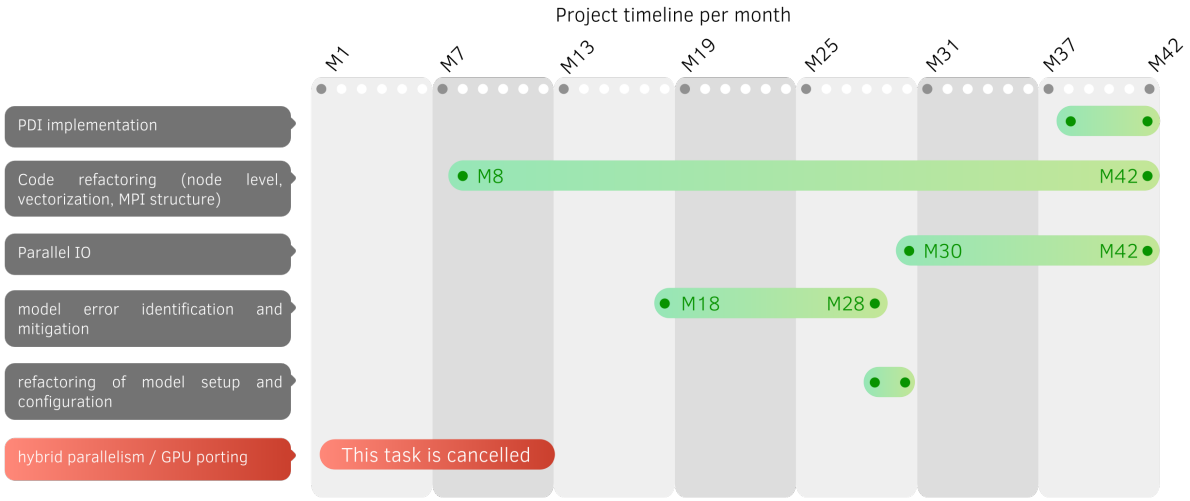


Figure 23: Breakdown (simplified Gantt chart) of the task 2.3 for EURAD-IM.

6.3 Work description

6.3.1 Detailed performance analysis

The performance analysis has been conducted in a simplified setup to evaluate the code's performance under real conditions. The simplifications comprise a reduced number of time steps and iterations per simulation. These simplifications were necessary to limit the memory and compute time required for the analysis. However, the results of this analysis can be extrapolated to full simulations with more time steps and iterations.

The strength of EURAD-IM is the ability to performed four dimensional variational data assimilation (4D-var) analysis for atmospheric constituents. Using 4D-var, the observation-model discrepancy within a time window (assimilation window) can be projected onto initial values and emission rates of chemical species, which are two of the key drivers of forecast uncertainty. This data assimilation method includes the use of the adjoint code of the forecast model. In EURAD-IM, the adjoint code is designed for each routine separately to ensure a modular code setup. The adjoint code includes the forecast model to calculate the model state at which to linearize the model, essentially at each line of the code. In total, the 4D-var model calculation takes about 3-4 times longer per iteration than the pure forecast model.

By definition, there are certain similarities in the code structure and layout of the forecast model and the adjoint code. Thus, optimisation of the forecast model can be adopted in the adjoint model and will consequently be amplified. Although the focus of the simulations within EoCoE-II is on ensemble analysis that only use the forecast model, the performance of the adjoint code is analysed as well in order to maintain a consistent model environment.

The performance analysis was performed for the forecast code and its adjoint separately using 239 cores on the JUWELS super-computer. The simulation included two iterations and three simulation hours (54 time steps) for the European model grid (15 km horizontal resolution, 348 $\times$ 289 grid boxes, 30 vertical

layers) on January, 01, 2016. Real analyses comprise 24 simulation hours and 15-20 iterations. The tools used to measure the time needed by each module are Score-P and Vampir. Score-P can instrument source code to let it generate a performance report while executing. Vampir is a tool for visualizing the reports generated by the instrumented program. Besides of showing the total runtime of each module, it can plot the active functions at any time for each process and thread. This helps to find modules which have significant load imbalances.

Besides the stiff solver for gas phase chemistry, further performance bottlenecks have been identified (see also Tab.15). These main performance bottlenecks were the

- adjoint code of the stiff gas phase chemistry solver (ADCHEM in Tab.15);
- adjoint of the aerosol module for secondary inorganic aerosols (AD_EQL5);
- adjoint implicit solver for vertical diffusion (ADVDIFFIM);
- writing and reading of intermediate model states to/from file for later use in the adjoint code (TRAJ_IO);
- MPI parallelization, mainly the separation of the master (IO operations) from the workers (model calculation);
- serial netCDF IO from the master process leading to idled worker processes;
- load imbalances in multiple modules, especially the transport modules for advection and diffusion

The load imbalances in the transport modules are the result of different wind speeds across the modelled area. According to the Courant-Friedrichs-Lewy criterion, the faster the wind, the more iterations are needed for the transport. Therefore, some processes are many times slower than others. As local domain boundaries are exchanged by the worker processes, this leads to idle times of processes. If the work would be perfectly shared between all processes, the adjoint advection (ADCWADVEC) would be 1.9 times faster.

It is emphasized that the relative shares of CPU-time may differ between simulated days because of the differences in the simulated chemical regime. Nonetheless, the key bottlenecks of the codes performance stay the same.

6.3.2 Code refactoring

During the optimisation of the EURAD-IM two major code errors have been identified. The first error concerned the advection routine, which also affected the optimisation strategies described in the previous deliverable. The second error became active in the joint assimilation of trace gases and aerosol species. In the following, the two errors and mitigation strategies are described.

Advection is the main process for the dispersion of trace gases and aerosols. Within EURAD-IM, four advection schemes are implemented. However, the need for a monotone, positive definite advection scheme for the transport of the adjoint signal is essential. This is provided by the advection scheme of [22]. During the code optimisation, unrealistically large aerosol concentrations have been identified. The source of these large aerosol concentrations was an improper scaling with the pressure variable $p*$ (pressure minus pressure at model top), which lead under specific wind conditions to an increase of aerosol concentrations in regions with high topography and became obvious only after a long model integration of about one month simulation time. The scaling with $p*$ is required for the terrain following vertical coordinate used within EURAD-IM to omit topographic effects in the transport of trace gases and aerosols. In contrast to the other advection schemes implemented in EURAD-IM, the advection scheme by [22] makes explicit use of $p*$, which was falsely implemented in the erroneous model version. This issue was fixed by changing the conversion of variables prior to the advection for the use of the advection scheme of [22]. Further, the air density used as scaling factor within the advection scheme needed to be replaced by $p*$.

Routine	forecast	adjoint
MPI_BCAST	554 s / 10.9 %	16,797 s / 49.3 %
MPI_GATHER	—	2,310 s / 6.7 %
MPI_ALLGATHER	365 s / 7.2 %	584 s / 1.7 %
CHEM	1,102 s / 21.7 %	789 s / 2.3 %
ADCHEM	—	6,527 s / 19.2 %
WADVEC	1,211 s / 23.9 %	1,672 s / 4.9 %
ADCWADVEC	—	1,678 s / 4.9 %
EQL5	237 s / 4.7 %	237 s / 0.7 %
AD_EQL5	—	988 s / 2.9 %
VDIFFIM	126 s / 2.5 %	242 s / 0.7 %
ADVDIFFIM	—	537 s / 1.6 %
TRAJ_IO	634 s / 12.5 %	464 s / 1.4 %
MEGAN_GAMMA_VALUES	153 s / 3.0 %	—

Table 15: Accumulated exclusive time for selected modules and its relative contribution to the total accumulated run time of the performance analysis of EURAD-IM *in Task 2.3. Large accumulated exclusive times for MPI modules indicate load imbalances between the MPI threads in other modules. The* EURAD-IM *modules listed are: CHEM: stiff ODE solver for gas phase chemistry; ADCHEM: adjoint of CHEM; WADVEC: advection scheme (horizontal and vertical); ADCWADVEC: adjoint of WADVEC; EQL5: solver for secondary inorganic aerosols; AD_EQL5: adjoint of EQL5; VDIFFIM: implicit solver for diffusion; ADVDIFFIM: adjoint of VDIFFIM; TRAJ_IO: IO of intermediate model states for use in the adjoint code. MEGAN_GAMMA_VALUES: calculator for biogenic emissions; For the advection modules the accumulated exclusive time after the code refactoring is given in parenthesis.*

This affected also the approximate parallel implementation that was used and updated as described in the previous deliverable. In the approximate parallel implementation of the advection scheme by [22], only the local boundaries values are exchanged between the worker processes. In fact, for the exact calculation of the advection, the transport within on direction needs to be performed consecutive from grid box $i = 0$ to $i = I_{max}$. After correcting for the errors, the approximated parallel implementation did not provide reliable results anymore, especially for long model integration times. Thus, the approximate parallelization could not be used and needed to be replaced by an exact parallelization, which is less scalable and required much more computing time as processes are executed consecutively.

Further, the standard unit of the aerosol variables in EURAD-IM had to be converted from mass concentrations to mass mixing ratios. This was necessary to ensure a consistent transformation between mass mixing ratios, which is required for the transport, and mass concentrations, which is the standard output unit of aerosols. This conversion was realized by applying unit conversions prior to every use of the aerosol variables in EURAD-IM. The alternative was to update all routines dealing with the aerosols in order to use mass mixing ratios instead of mass concentrations. For efficiency reasons, this was postponed to save time for the proposed tasks within EoCoE-II.

In addition to the issues in the advection scheme, the conversion of the adjoint aerosol signal in the adjoint aerosol module was not correct, which lead to unrealistically large gradients of the aerosol species. As for the advection scheme, the investigation of this issue and the identification of the error source was highly time consuming as the root cause of the issue became only obvious using a certain configuration in the 4D-var setup. This configuration required observations of gaseous ammonia (NH3) and an optimisation of aerosol species, which is a very special and unrealistic model configuration as in situ NH3 observations are not available for the assimilation due to its measurement technique that provides only weekly or monthly averaged values.

Further, due to the time consuming process finding and solving the code errors, mitigation strategies

have been initialized and implemented in the EURAD-IM in order to avoid severe model errors in future. The mitigation strategies are the implementation of a configuration check, in which the execution of non validated model configurations is rejected. In addition, some basic model tests are implemented, which test main features of the EURAD-IM, including

- check sums for testing the copying of model states between the forward and adjoint run.
- validation of the handling of IO files, especially between the various model setups (e. g. proper read of restart files between the forward run and data assimilation analyses).
- definition of a test environment that allows to add tests in a plugin-manner.
- user interface and documentation.

In addition to these global test strategies, comprehensive unit tests have been performed and partially implemented for all modified subroutines of the EURAD-IM. Unfortunately, the identification and mitigation of the model errors largely affected the code optimisation, which is why not all tasks could be finalized until the end of the project.

As described above, the advection scheme suffered from an error that also affected the performance improvements described in the previous deliverable. Thus, the approximate parallelization in which only the neighbouring MPI processes interact and which was described and optimised in the previous deliverable, did not provide reliable results anymore, especially for long model integrations. Hence, in order to ensure monotone, positive definite transport, the advection needs to be calculated consecutively from grid cell $i = 0$ to grid cell $i = I_{max}$ and back, which largely reduces the scalability. Further, the implementation of the vertical advection is not independent on the horizontal advection after removing the model error. Thus, the CFL criterion is global and violations of the CFL criterion for the vertical advection, which is more likely to happen (especially in regions with large topography gradients) than for the horizontal advection, reduces the time step in the whole domain, and not only locally for the affected vertical model column.
In order to improve the performance and scalability of the advection scheme. Two major updates to the model have been implemented:

1. Implementation of a pre-processing routine that calculates the maximum possible time step given the restriction posed by the CFL criterion, thus, avoiding CFL violations;
2. Exchange of sub domain boundaries between neighbouring MPI worker processes after the integration of any single row of grid boxes (listing 1) instead of an exchange after the calculation of the full advection in one direction of each worker (listing 2).

In addition, the initialization of the advection routine (e. g. the pre calculation of parameters) was analysed and refactored where possible. Thus, the implemented DO-loops have been reduced to a minimum.

Listing 1: advection optimised

```
DO I = 1, IMAX
   DO K = 1, KMAX
     IF (IAM .GT. 0) &
          CALL RECV_BOUNDARY()

       CALL ADVECTION_1D()

     IF (IAM < NWORKER) &
          CALL SEND_BOUNDARY()
   END DO
END DO
```

Listing 2: advection reference

```
IF (IAM .GT. 0) &
     CALL RECV_BOUNDARY()

DO I = 1, IMAX
   DO K = 1, KMAX
      CALL ADVECTION_1D()
   END DO
END DO

IF (IAM < NWORKER) &
     CALL SEND_BOUNDARY()
```

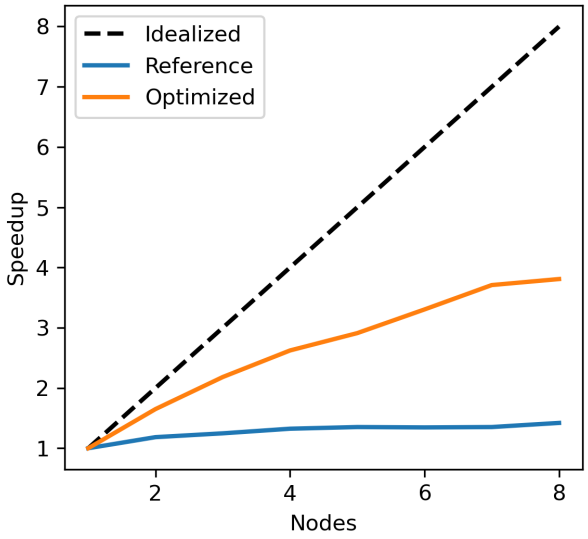


Figure 24: Scaling of the advection scheme within EURAD-IM*. The ideal speedup is shown by the dashed line.*

The improvements in runtime and speedup obtained by this code refactoring are shown in Fig. 24 and Tab. 16. While the advection still suffers from a weak scaling behaviour due to the characteristics of the monotone, positive definiteness, it could be improved from almost no scalability of a factor of 1.42 for the reference run to a factor of 3.81 for the optimised run (both values for the use of 8 nodes). Besides increasing the scalability, Tab. 16 shows a large decrease in runtime for the optimised advection scheme. This comes to a cost of additional MPI communications of small arrays between worker processes instead of fewer communications of large arrays. However, the runtime decreases between 53$\%$ (1 node) and 83 $\%$ (8 nodes) in the optimised run.

# Nodes	Runtime reference	Runtime optimised	Scaling factor reference	Scaling factor optimised
1	2718 s	1272 s	1	1
2	2296 s	771 s	1.18	1.65
3	2179 s	583 s	1.25	2.18
4	2053 s	485 s	1.32	2.62
5	2011 s	437 s	1.35	2.91
6	2020 s	385 s	1.35	3.30
7	2011 s	343 s	1.35	3.71
8	1914 s	334 s	1.42	3.81

Table 16: Runtime and scalability of the walcek advection scheme [22].

As the EURAD-IM originates from around 1990, it is older than most of the popular version control software available today. Hence, the version control of the model code was outdated and lacking features that ease the shared working on the same code. So far, the EURAD-IM versions have been maintained via dedicated colleagues and version control was mainly done by numbering of different model versions, which have been frozen in the archive. The identification of the model error described above enforced the need for a proper version controlling. To provide sophisticated version controls, a Git repository was initialized and backfilled with all available archived versions of EURAD-IM. Within the transition to Git, a

larger effort was initialised to refactor the configuration and building of the EURAD-IM to a modernised model setup. Currently, the EURAD-IM is run via a Kornshell-script that initiates the configuration and build of the model via various scripts and Makefile structures. The various scripts scan the model paths and detect and set dependencies for the code compilation. Luckily, the 2019 founded FORTRAN-lang community tackled the need of a FORTRAN-specific build system and developed the FORTRAN Package Manager `fpm`[1]. With `fpm`, the build process can be simplified effortlessly because it allows to replace all Makefiles and scripts for detecting the dependencies without further configuration. `fpm` has successfully been integrated in the build process for the EURAD-IM. In addition to the refactoring of the EURAD-IM configuration and build process, the FORTRAN Documenter (`ford`[2]) was implemented, which suits best for inline documentation of FORTRAN source codes, since it addresses the issue of Doxygen's poor handling of FORTRAN projects.

Since the source code of EURAD-IM is several hundred thousand lines long and execution of the whole model takes many hours, even on multiple compute nodes, optimizing the code is a very time-consuming process. To facilitate modifications of single modules, a self-developed snippet tool comes into use, which can generate FORTRAN functions for loading and storing subroutine arguments.

The snippet tool consists of two parts: Firstly, a FORTRAN library for accessing netCDF files in parallel with a plain interface. Secondly, a parser, which reads FORTRAN source files and generates a FORTRAN module with functions for loading and storing the arguments of the target subroutine. For example, attributable to the nested data type structure of the EURAD-IM, the 14 arguments of the adjoint vertical advection subroutine (`advdiffim`) expand to 333 intrinsic data types. Thus, the development of the snippet tool enables to optimize efficiently selected modules with realistic model configurations, enables to develop and apply unit tests, and reduces the computational costs updating the model code.

Performance analysis of ADCHEM subroutine The FAU team analysed the ADCHEM subroutine using the EURAD-IM snippet tool on the Meggie cluster[3]) and proposed several optimizations that could lead to substational performance improvements if applied to all stages of the subroutine. In the following we give a brief overview over these.

Stage	L1 ↔ L2 data volume (data)	L1 ↔ L2 data volume (total)	Ratio
`shuffle`	0.2 TByte	1.2 TByte	17%
`rosenbrock`	0.75 TByte	11.0 TByte	7%
`unshuffle`	0.17 TByte	0.77 TByte	22%
`limit-values`	0.05 TByte	0.38 TByte	13%

Table 17: Ratio of data and instruction traffic between L1 and L2 cache for each stage of the ADCHEM routine.

By measuring hardware performance events with the LIKWID tools in the computational part of ADCHEM, very high instruction traffic within the cache hierarchy and lack of vectorization were identified as two major performance issues. Only up to 22% of the data transfers of the 4 stages (`shuffle`, `rosenbrock`, `unshuffle` and `limit-values`) were related to workload data, while the dominant part were instruction transfers (see Tab. 17). The instruction traffic is caused by the complex initialization of arrays with chemical and physical coefficients required for computation. These initialization functions like `Update_Rconst.f90` contain computationally expensive operations with low throughput like exponentials and square roots. The traffic could be reduced by applying these functions not only to a single 3D grid entry but on the whole innermost dimension. The array initialization code is then loaded only once instead of for every element. In

[1] https://fpm.fortran-lang.org
[2] https://github.com/Fortran-FOSS-Programmers/ford
[3] `https://hpc.fau.de/systems-services/systems-documentation-instructions/clusters/meggie-cluster/`

addition, this enables vectorization inside these functions. Various similarly structured functions in different stages are amenable to the same optimizations as listed in Tab. 18.

File/Function	Scalar execution	Vectorized execution	Reduction
`Update_Rconst.f90`	$9.69 \cdot 10^{-5}$ s	$2.97 \cdot 10^{-5}$ s	69%
`Fun.f90`	$3.08 \cdot 10^{-5}$ s	$2.13 \cdot 10^{-5}$ s	30%
`Jac_SP.f90`	$8.31 \cdot 10^{-5}$ s	$3.67 \cdot 10^{-5}$ s	56%
`JacTR_SP_Vec.f90`	$2.26 \cdot 10^{-5}$ s	$1.49 \cdot 10^{-5}$ s	34%
`HessTR_Vec.f90`	$2.69 \cdot 10^{-5}$ s	$2.35 \cdot 10^{-5}$ s	13%
`KppSolve.f90`	$3.12 \cdot 10^{-5}$ s	$1.71 \cdot 10^{-5}$ s	45%

Table 18: Runtime improvements by vectorization of ADCHEM subroutines from all stages measured with proxy applications containing only the function and synthetic data to show the benefit of the refactoring for improved vectorization.

The most time-consuming `rosenbrock` performs forward- and backward-integration and stores intermediate results on a stack. The stack depth might be different for each grid element and therefore is allocated fresh in each solver call. This was already identified as performance issue early in the EoCoE-II project and the allocation was moved out of the solver with a configurable maximum stack depth. With respect to the proposed optimizations to the ADCHEM subroutines, the FAU team advised to move the loop over the innermost dimension into the `rosenbrock` solver to enable vectorization. The function should traverse the innermost dimension in chunks with a fixed size, preferably a multiple of the SIMD vector size of the CPU. The stack entries need to be enlarged to hold all data for a chunk of elements with an additional boolean array to mask out already done elements. Depending on the compiler and hardware features, the chunks can be efficiently traversed using gather-scatter or vector masking operations.

The benchmarks of the EURAD-IM snippet tool showed reasonable performance improvements by refactoring the code to enable vectorization. The recommendations derived from the snippet tool are given back to the project team to apply them to the EURAD-IM production code.

6.3.3 EURAD-IM on GPUs

To test the EURAD-IM porting on GPUs, unit tests need to be implemented to avoid implementing unseen errors. Further, the porting to GPUs require a compiler which supports GPU programming as well as a well-structured and node-level optimised codebase, which was highlighted by the programming experts within EoCoE-II.

Unit tests are important to verify the correctness after modifying the model's code. By using a different compiler (version), reordering instructions, and utilizing different computation units, the porting to GPUs is likely to produce slightly different simulation results. The tests calculate the relative error

$$\frac{x_{new} - x_{orig}}{x_{orig}}$$

of the modified code's output compared to the original output. These unit tests target each unit of the model, e.g., the horizontal diffusion. Since the output array contains mainly zeros, the arrays have to be masked before calculating the relative error. Otherwise, the test code would run into divisions by zero, which results in an infinite relative error. To detect errors at positions where the original value is zero, the divisor of the relative error formula is replaced with `tiny(1.)` (=1.17549435E-38).

Listing 3: Code of the relative error function

```
elemental function rel_err(v, v_actual) result(relative_error)
    !! if the actual value is 0., use tiny(1.) to avoid '/ 0.'
```

```
    real, intent(in) :: v
    real, intent(in) :: v_actual
    real :: relative_error

    relative_error =   (v - v_actual) &
                     / max(abs(v_actual), tiny(1.))
end function rel_err
```

For saving the original output, the snippet tool mentioned above was utilized. Additionally, it loads the original output for the error calculation. For a quick overview, the test program prints the maximum and the Euclidean norm of the relative error.

Listing 4: Unit test for the adjoint vertical diffusion scheme

```
! [...]
call advdiffim(spec, vda, ! [...]
    ! call refactored adjoint vertical diffusion scheme
pn_file = pn_open("out-orig.nc", MPI_COMM_WORLD)
advs = pn_file%load_variable(advs, "ADVS", ["vda"])
    ! load original output data
call pn_file%save_file()

associate(err => rel_err(vda%advs, advs))
    write(*, "('Error max : ', g0)") maxval(abs(rel_err))
    write(*, "('Error norm: ', g0)") sqrt(sum(err**2))
end associate
```

Currently, the EURAD-IM uses only MPI for parallelization. Utilizing GPUs requires a hybrid parallelization: MPI handles the communication between compute nodes and CUDA or OpenACC drive the GPU calculations. For this, a compiler enabling the use of CUDA or OpenACC is required. Due to the changes of the model configuration described above, the EURAD-IM utilize the FORTRAN standard library[4] which requires the FORTRAN 2008 standard. However, the few compilers capable of building FORTRAN GPU code, which are the NVIDIA and the Intel compilers, do not implement all required features, or are not available on the JSC high-performance computers on which the core applications of the EURAD-IM are run. Thus, and because of the extra time required for the identification and mitigation of the model errors, porting the model code to GPUs is postponed until the compilers fulfil all requirements.

As was indicated by the experts on node-level optimisation, having a highly optimised code at node level is fundamental for efficient porting to GPUs. Hence, instead of GPU porting, the focus was shifted to the optimisation of the model on node-level computing using modern FORTRAN features. Since the EURAD-IM codebase grew over decades, many routines are outdated and do not use the latest FORTRAN features. For example, the vertical diffusion routine mentioned above is still written in fixed source form FORTRAN, which originates in punched cards. Main focus was on cleaning the code and implement vectorization to avoid hard coded loops. These changes lead to significant speed-ups depending on the compiler options. While the original code took about 0.409 s for one execution of the horizontal diffusion routine, the refactored code only needs 0.177 s (both without compiler optimisation). This is a speed-up of 2.3. With compiler optimisation turned on, the original code performs much better and only takes 0.045 s. The refactored code is still faster with 0.040 s, a speed-up of 1.13.

[4] https://stdlib.fortran-lang.org/

Table 19: The time in seconds for a single execution of the adjoint vertical diffusion scheme is always faster with the refactored code, which achieves speed-ups ranging from 1.11 to 2.31 depending on compiler optimisation

	no flags	`-O3 -funroll-loops`	`-O3 -funroll-loops -march=native`
original	0.409	0.050	0.045
refactored	0.177	0.045	0.040
speed-up	2.31	1.11	1.13

6.3.4 parallel IO implementation

As was previously described, the EURAD-IM suffered from two major performance bottlenecks. Firstly, the serial IO handling was identified as highly time consuming. Secondly, the strict separation between the MPI master process, which was solely attributed to the model initialization and data IO, and the MPI worker processes, which were attributed to the model integration, caused large idle times of processes during the model run. Thus, the core of the EURAD-IM was refactored in order to enable the MPI master process to take part in the model integration. This required a reconfiguration of the domain splitting between the MPI processes. While in the setup of the domain splitting, the only change was a shift in the processes to include process 0 (master process) in the calculations, the refactoring of the EURAD-IM was more complex. The MPI master process and the MPI worker processes were assigned different routines. These routines needed to be merged to enable the master process to contribute to the calculation. The distribution of the input data via MPI exchange routines needed special treatment after including the master in the model integration. In a first step, the IO was transformed to parallel netCDF where possible. However, certain input data are still read from binary or ascii files (e. g. horizontally distributed emissions data). Thus it was decided to leave these inputs to be handled by the master process and copied to the worker processes afterwards. With this, the IO was parallelised to more than 95%. Only some of the IO during the model initialization still remain serial.
The performance improvements of the EURAD-IM are shown in Fig. 25. Instead of using the full number of cores provided by the nodes (i. e. 96 cores using hyperthreading) the number of cores have been optimised in order to fit with the domain size. This ensures an almost equal size of the subdomains for each MPI process. The number of unused cores is indicated by the black line in Fig. 25. It is clearly seen that after enabling the master process for the model integration the number of unused cores is largely reduced. Only for four nodes (4 unused cores) and 8 nodes (12 unused cores). Fig. 25 shows the runtime for different model parts before and after the code refactoring of the EURAD-IM. The improvement in terms of runtime is clearly visible throughout the scaling test. However, with increasing number of nodes the model IO increases as well. This is currently under investigation but is likely because the applied parallel IO strategy, which is realized through netCDF-4 and not through pNetCDF, requires collective access in large parts of the IO. netCDF-4 was chosen over pNetCDF in order to keep the changes of the EURAD-IM code minimal. Thus, large parts of the IO routine could be used with only moderate modifications (e. g. addition of start and count arguments) instead of changing the whole function calls, which is required for pNetCDF. Although the model output scales weakly, even for 8 nodes it is still faster than the serial IO. Further, the improvements of the advection scheme lead to large reductions in runtime of the main integration loop. Additional reductions of runtime were obtained in the initialization of the model, which reduces from around 400 s to approximately 10 s. In general, the weak scaling in the new IO strategy prohibits further runtime reductions. Thus, the use of PDI as output interface will be tested in a next step (see also next section).

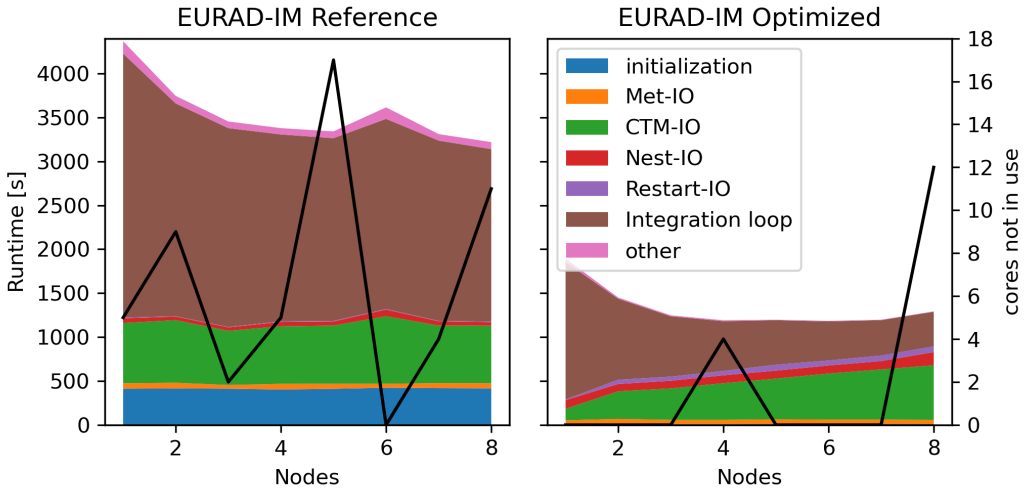


Figure 25: Runtime improvements of EURAD-IM *after code refactoring. As the focus of the application of the* EURAD-IM *within* EoCoE-II *are ensemble runs, the scaling tests have been performed for a 24 hour model forecast with realistic model output. All model configurations are adapted from the most recent production runs. The scaling test was performed on the JUWLES super computer at JSC using hyperthreading. Left: Runtime of the reference model scaling tests. Right: Runtime of the* EURAD-IM *after model improvements described in the text. In addition to the runtime, the black line indicate the number of cpus which could not be used for the model integration.*

6.3.5 PDI integration

The purpose of integrating PDI into EURAD-IM was to ease and improve the exchange of data between ensemble members for improving the performance of large ensembles. However, while discussing with EoCoE-II partners, the use of MELISSA-DA in order to efficiently execute ensembles was suggested. MELISSA-DA provides a large flexibility in terms of fault tolerance handling, efficient use of the requested nodes and limited changes to the EURAD-IM. Further, it enables the online coupling of ESIAS-MET and ESIAS-CHEM, which is aimed for in near future for investigations of the effect of aerosol emissions in the formation of clouds.
Thus, the EURAD-IM was successfully coupled with MELISSA-DA, which is called ESIAS-CHEM in the remainder of this report. In ESIAS-CHEM, PDI can be used to efficiently expose data to the MELISSA server without the need of conversions of the data structure, which is required using solely MELISSA-DA. Fig. 26 shows the scaling behaviour of ESIAS-CHEM to perform a 256 member ensemble analysis simulating a sensitivity run. For this, the ensemble members are run fully independent (i. e. assigning each ensemble member a fixed node). As expected, the scaling of ESIAS-CHEM is nearly ideal. This is a consequence of the ensemble setup, in which the communication between the runners and the server within MELISSA-DA is suppressed. For the optimal exchange of data between the runners and the server PDI was implemented in EURAD-IM. However, the PDI-plugin for MELISSA is under preparation but not finally released, thus, it could not been tested so far. In order to test the PDI implementation the HDF5-plugin as well as the trace-plugin were successfully utilized. Besides time indices, a couple of four dimensional arrays need to be exchanged between runners and the server, which are the meteorological fields, the aerosol and gas phase species, and emission data.

7 Task 2.4 - Materials code optimisation

Task leader: FZJ

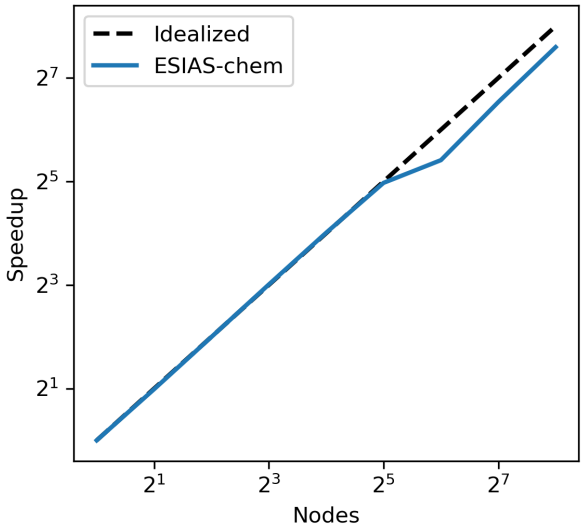


Figure 26: Scaling plot for ESIAS-CHEM*, which is the* MELISSA-DA *implementation of* EURAD-IM*. The scaling is normalised to 1 node. The scaling is based on 256 independent ensemble member, which mimics a large sensitivity run.*

Task participants: FZJ, CNR, FAU

In the Materials scientific challenge, one of the goal focuses on improving the modelling of solar cell device at atomic scale and use our high-end numerical tools to determine the properties of new materials for photovoltaic. Table 20 shows the team members and the external contributors to the task 2.4 involved in the WP2 of EOCOE-II. Additional members of the team are involved in the WP1 and are not listed in the table.

Task leader: FZJ

Task participants: FZJ, CNR, FAU

The flagship code LIBNEGF LIBNEGF is a LGPL project seeded in 2008 at the University of 'Tor Vergata' and CNR, hosted on github (`https://github.com/libnegf`). It is a general purpose non-equilibrium Green's function library to compute the density matrix and transport in open quantum systems such as nano and molecular devices. The library is developed as a general-purpose tool that can handle any input Hamiltonian, from most diverse problem formulations. Indeed it has been interfaced to several different codes such as,

- Density-Functional Tight-Binding (DFTB) code (`https://gihub.com/dftbplus`)
- Finite element code (TiberCAD) for both k.p and effective mass Hamiltonians (proprietary code)
- Empirical Tight-Binding Hamiltonians (within TiberCAD)
- Hessian matrices for phonon transport (development branch of dftb+)

Besides being integrated in other academic codes, LIBNEGF is embedded in the proprietary package suite "Materials Studio", formerly developed by Accelrys, acquired by Dassault Systems and renamed as Biovia. Biovia has extended the interface of LIBNEGF also to the ab-initio software DMOL3.

Code developments within EoCoE-II The functionalities of LIBNEGF needed to be extended for solar-cells and device simulations. The code was equipped with routines to compute the density matrix under

People	Position	Role	Period
Edoardo Di Napoli	Senior scientist at the Jülich Research Center (Forschungszentrum Jülich - FZJ)	Supervises and coordinates the libNEGF activity	M1-M36
Sebastian Achilles	Research Scientist at FZJ and PhD student at RWTH	HPC expert: In charge of the refactoring and the parallelization	M1-M30
Alessandro Pecchia	Senior scientist at CNR	Main Developer of libNEGF functionalities	M12-M36
Gabriele Penazzi	Research Scientist	libNEGF developer	M12-M20
Daniele Soccodato	PhD Student	GPU porting in collaboration with Dr. Pecchia	M20-M36
Georg Hager	Senior Scientist at FAU	Expertise and advisor node-level code optimisation	M25-M28
Markus Hrywniak, Kaveh Haghighi-Mood, Andreas Herten	NVIDIA Application Lab Julich	Advisors for GPU developments	M20-M30

Table 20: Team Members and external contributors for task 2.4 within the WP2.

bias conditions but without interactions. Transport was limited to coherent, ballistic transmission and elastic dephasing models. A large development effort has been devoted within EoCoE-II to the development of interacting self-energies and routines to compute non-coherent transport. This is an important development in order to study charge carrier dynamics in bangap defects of aSi of Task 1.3.1. We have developed as much as possible over existing routines and kept the main underline data structures. In particular, all Green's functions are computed and stored internally as block tri-diagonal dense matrices. This representation is motivated by the fact that Green's functions (GF) are usually dense matrices which are stored in main memory and therefore could impose restrictions on the simulation of large systems. The block tri-diagonal form decreases memory consumption but requires that intermediate calculations should keep this form and is an approximation that in some case requires careful validation.

There are other numerical approaches that make severe approximations and implement algorithms that restrict from the start the Green's functions and self-energies to the non-zero pattern of H. The matrix elements needed for further calculations, such as current or charge density, are in fact just those on the pattern of the non-zeros of the Hamiltonian (H), which for large systems can be considered a sparse matrix. Within these approaches, the Green's functions are computed as solutions of large sparse linear systems or exploit the 'selected inversions' algorithm. The interaction self-energies are therefore computed consistently on the sparsity pattern of H. The problem with such an approach is that it neglects wave interferences originating from the complete off-diagonal structure of the retarded Green's function G^r. On the other hand, this approach allows to save considerable chunks of memory and is suitable to deal with large systems. Other code implementations, such as OMEN, works with mixed-precision numerical kernels and store intermediate matrices even in half-precision.

An advantage of the block-dense representation is that all algorithms rely on dense matrix algebra (GEMM or direct linear solver operations) that scale very well on multicore and GPU accelerators. Furthermore, in our formulation, the sparsity pattern of H is a subset of the 3-diagonal block-dense pattern shown in figure 27 and it is easy to convert the block-dense matrices to sparse, especially for storage and computation of the self-energies.

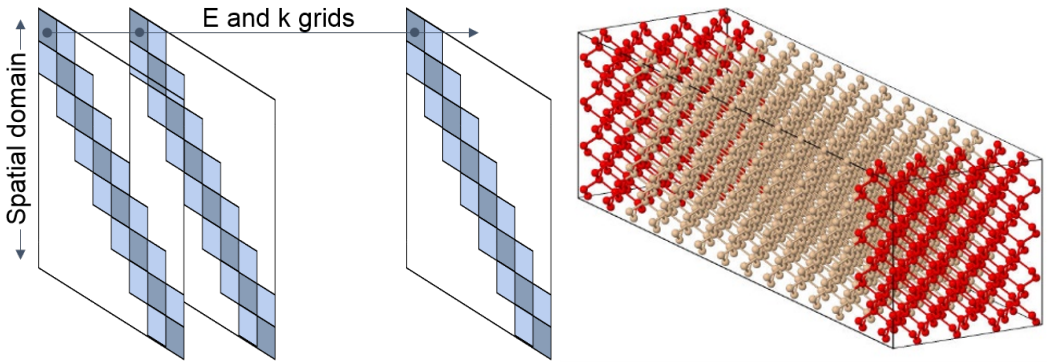


Figure 27: right *Graphical representation of block tri-diagonal matrices corresponding to structure layers and energy or k distribution dimensions.* left *Atomic structure of the 4x4 Si supercell.*

The developments of LIBNEGF proceeded in two directions.

- Node-level optimizations in order to speedup the calculation of contact self-energies and system Green's functions.
- Implementation and optimisation of the non-local interaction self-energies.

At node level, we have considerably improved multi-threading scalability of the block-matrix inverse. This was already reported in the previous deliverable, D2.2 so here we only give a brief account of this work. The non-threaded *zgetri* routine has been substituted with multi r.h.s. solve step using *zgetrs* with an explicit OPENMP parallelization. As a result the OMP scalability on multicore CPUs has greatly improved.

Input	tasks p. n.	threads p. t.	time origin [sec]	time new [sec]	Speedup
2x2	4	12	38.61	25.02	1.54
3x3	4	12	399.62	149.81	2.67
4x4	4	12	2190.27	615.98	3.56
5x5	4	12	9949.08	1836.85	5.42
6x6	4	12	33125.51	5015.48	6.60

Table 21: Comparison of the different supercell silicon test inputs for the original version and the optimised version on 10 JUWELS node with 4 MPI ranks per node and 12 OPENMP *threads per rank.*

Table 21 shows the runtime comparison of the different input cases for the original and the optimised versions of the code. This experiment used 10 nodes and distributed the k points across these. 4 MPI ranks per node have been used to distribute the energy points and 12 openMP threads per task are used to compute each Green's function. Since the multithreading scaling has been much improved, the speedup is increasing with matrix sizes. Improving parallelism on the single node enables the extraction of an overall performance that is closer to the theoretical peak. In doing so, we are now capable of 1) solving bigger problems efficiently, and 2) attenuating the memory constrains, which again allows us to tackle larger system sizes.

7.1 Parallelization extensions

Within EOCOE-II we have extended the original parallelization level of LIBNEGF which only explicitly included energy points distribution. The k-point distribution was driven externally by the calling code with a split communicator strategy. Since inelastic self-energies require also communication between k-points, the code was extended to a 2-dimensional MPI cartesian grid. The data relative to the momentum and

energy levels are distributed in parallel using MPI specifications. These levels form a hierarchy in which each node is performing the calculation of the subset of energy and momentum points that are stored locally in memory. The data is distributed across the nodes in a 2D grid fashion. An example of the distribution is visualized in Fig. 28 for $N_K = 10$ and $N_E = 12$.

$K = 0, 1$ $E = 0, 1, 2$	$K = 2, 3$ $E = 0, 1, 2$	$K = 4, 5$ $E = 0, 1, 2$	$K = 6, 7$ $E = 0, 1, 2$	$K = 8, 9$ $E = 0, 1, 2$
$K = 0, 1$ $E = 3, 4, 5$	$K = 2, 3$ $E = 3, 4, 5$	$K = 4, 5$ $E = 3, 4, 5$	$K = 6, 7$ $E = 3, 4, 5$	$K = 8, 9$ $E = 3, 4, 5$
$K = 0, 1$ $E = 6, 7, 8$	$K = 2, 3$ $E = 6, 7, 8$	$K = 4, 5$ $E = 6, 7, 8$	$K = 6, 7$ $E = 6, 7, 8$	$K = 8, 9$ $E = 6, 7, 8$
$K = 0, 1$ $E = 9, 10, 11$	$K = 2, 3$ $E = 9, 10, 11$	$K = 4, 5$ $E = 9, 10, 11$	$K = 6, 7$ $E = 9, 10, 11$	$K = 8, 9$ $E = 9, 10, 11$

Figure 28: Exemplary momentum and energy distribution of $N_K = 10$ and $N_E = 12$ on 20 nodes (black squares). In this example every node stores $N_{K,\text{local}} = 2$ local momentum data points and $N_{E,\text{local}} = 3$ local energy points.

7.2 GPU porting of recursive solvers

All kernel algorithms of LIBNEGF have been ported to NVIDIA GPUs and benchmarked on A100 cards on JUWELS Booster at JSC, comprising 936 compute nodes equipped with 2x AMD EPYC 7402 24 cores and 4 GPUs A100 each with Infiniband dragonfly+ interconnect. We have used both OPENACC pragmas within the FORTRAN code and low level CUDA instructions. The OPENACC/FORTRAN combination leads to an easier code development but has some drawbacks. For instance not all C/CUDA interfaces to kernel solvers within the CUBLAS, CUTENSOR, and CUSOLVER libraries are available in FORTRAN. Some low level CUDA instructions are also needed for the developments of fine-tuned tensor-core kernels and for customized mixed-precision operations. Additionally, the compilation of FORTRAN modules containing OPENACC requires NVIDIA FORTRAN compiler which brings dependencies and require, as in the case of the dftb+/LIBNEGF interfacing, to compile the whole project with NVFORTRAN, which frequently leads to compiling issues. The latter was the main motivation to also develop a CUDA version of the code. Data movements have been explicitly handled by wrapping either OPENACC or CUDA calls by higher level routines. Similarly, all calls to CUBLAS and CUSOLVER kernels have been wrapped, resulting in a slimmer code that is easier to develop and maintain.

A benchmark has been performed on Silicon supercells with different sizes, labelled 2x2 to 6x6 and 10 nm in length. Fig. 29 shows a benchmark for the largest supercell, 6x6. Both GPU and CPU runs with 4 MPI tasks per node, corresponding to the number of NVIDIA devices per node, and increasing number of nodes from 4 to 64. For comparison, the CPU runs have been performed on Intel Xeon Platinum 8168 CPU with 2×24 cores, 2.7 GHz. Henceforth, each MPI task use 12 OMP threads on the CPU. The CPU code has been compiled with Intel FORTRAN 19.1 and use the threaded MKL 2021.4 library. The GPU version has been compiled with NVIDIA suite NVHPC 22.3. The right panel shows the GPU speedup for different parts of the computation: the surface Green's functions (SGF) and the device Green's functions. The largest speedup of x15 is observed for the SGF algorithm which does not involve memory transfers during the computation. The device Green's functions, on the other hand, require device/hosts transfers, especially needed because of memory limitations on the device, leading to a lower speedup of about

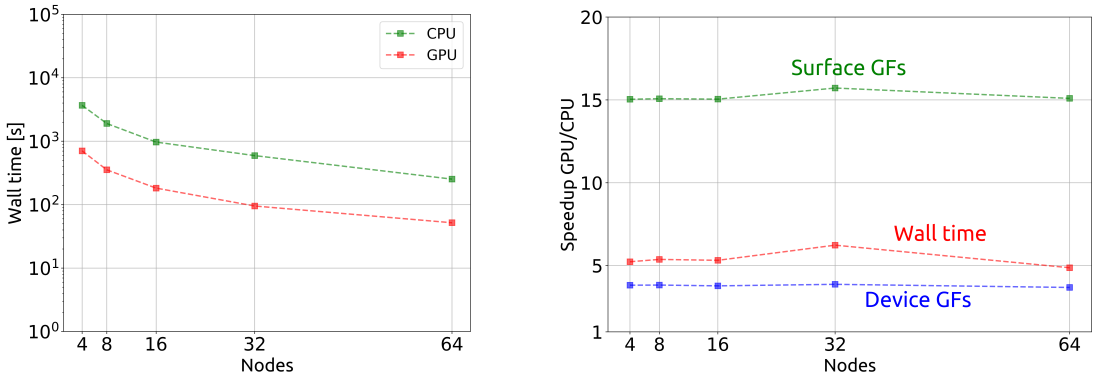


Figure 29: Speedup of GPU run compared to CPU for a 6x6 silicon supercell test. GPU calculations are run on A100 GPUs compared to Intel Xeon Platinum 8168 CPU. Panel (right) shows the scaling with number of nodes and (left) the speedup of different portions of the code.

x3. The overall wall clock speedup is found to be slightly above x5. Much larger speedups have been obtained in single precision or enabling the tensor core acceleration for extracted mini apps. For instance the decimation algorithm for the surface Green's functions can reach speed ups of x100 in single precision, reaching 90 Tflops/ of peak performance on a single device, as shown in Tab 24. This motivated to try a single-precision compilation of the whole dftb+/libnegf project but several energy point calculations turned out unstable. We conclude that a more complex mixed precision strategy is needed.

7.3 Code restructuring and developments

While developing the new functionalities we have made an effort in streamlining the existing code and restructuring some of the key algorithms. In order to reduce memory consumption, we carried out several optimisation on the allocation of temporary matrices. An exemplary development concerned the implementation of a new algorithm to compute the correlation Green's function $G^n = -iG^<$. Inspired by the algorithm implementing the computation of $G^<$ in PVNEGF, we designed and implemented a recursive algorithm for the solution of the linear system

$$[ES - H - \Sigma^r]G^n = \Sigma^n G^{r\dagger}, \tag{1}$$

quite similar in spirit to the computation of G^r. This implementation is much better both in terms of memory savings and speed than the original algorithm that computed directly the triple product, $G^n = G^r\Sigma^n G^{r\dagger}$, which becomes particularly cumbersome when interactions make Σ^n itself a block tri-diagonal matrix.

7.4 Computation of inelastic self-energies

The computation of the self-energies is the most time-consuming step of the whole calculation because of MPI communications overheads. Thanks to the Einstein oscillator approximation typically used for photons or optical phonons, the communications across the energy grid are restricted between the energy point E and $E + \hbar\omega$ and between E and $E - \hbar\omega$. This is handled with non-blocking send/recv communications into buffer arrays. For systems with lateral periodicity the Green's functions and Self-energies depends on a k-vector. Therefore, the calculation of the self-energies for a given energy point also require global communications across the k-grid.

The kernel calculation require evaluations of the form,

$$A_{ij}(k) = \sum_q F(k, q, \|z_j - z_i\|)B_{ij}(q), \tag{2}$$

where the indices i and j run over dense matrix blocks, z_i is the coordinate of an atom corresponding to matrix index i and the summation over q extends over the whole k-grid sampling the Brillouin zone. In order to make this computation more efficient the functional form F has been pre-computed and stored on a 3-dimensional tensor. The atom pair distances, $\|z_j - z_i\|$, have been sampled on a discrete grid. Calculation of the convolution, which is mimicked as a tensor product $F \times B$, was overlapped with non-blocking communications between MPI processes. Rather than an all-to-all communication, partial sums are accumulated by each task and communicated to the next in a ring scheme. The computation for the self-energy is done in two steps: First, the necessary data for the energy shift is communicated followed by the communication relative to data for the momentum summation. For example, the communication across the momentum index is carried out along the nodes of the horizontal direction of the MPI grid by a loop over all nodes. The sender and receiver are determined with `MPI_Cart_shift`. Each node gets a *source* and a *destination* for the communication. Each node sends the matrix to the destination and receives the matrix from source.

The loop over the matrix indices i and j has been parallelised with OpenMP. This scheme was benchmarked JURECA Booster whose configurations is Intel Xeon Phi 7250-F with 96 GB memory per node. The network is a Intel Omni-Path Architecture with non-blocking fat tree topology. The simulation has been carried out with 64 k-points and 1792 energy points for an artificial physical system. The corresponding scaling plot is shown in Fig. 30.

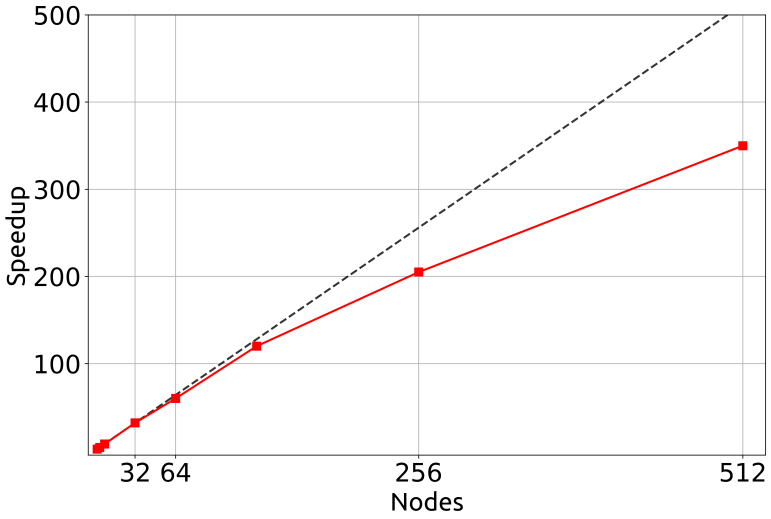


Figure 30: Scaling of the self-energy routine on the PVNegf code.

The code, originally developed for PVNegf, has been adapted to deal with the general block-matrices of libNEGF and the new extended code has been tested on Si supercells with different sizes, 2x2, 4x4 and 5x5 on JUWELS cluster. The corresponding block-matrix sizes are reported on table 22.

These test calculations include 32 energy points and 16 k-points distributed over 128 computing nodes and 4 tasks per node. The timings shown in Table 22 demonstrate that the calculation of the inelastic self-energies represent the most time-consuming portion of the runs. The interconnects of JUWELS cluster is based on InfiniBand EDR (Connect-X4) with 100 GB/s of nominal speed, hence further analysis is needed in order to make significant improvements here.

Concerning parallel scaling of the calculations, we report the tests on the 2x2 and 4x4 supercells, distributing 64 k-points and 32 energy points in total. The results are shown in Fig. 31. Due to memory limitations on the regular Juwels cluster nodes (96GB), the number of points that can be handled by each node is limited, especially for the largest 4x4 system. For this reason, we could not run calculations with less than

Structure	Block Size	Code Section	Time [sec]	Fraction
2x2	288 1.3 MB	Surface GF Device GF Self-Energies	0.16 0.32 1.6	7.5% 14.9% 77.6%
4x4	1188 22.6 MB	Surface GF Device GF Self-Energies	6.7 6.6 25.0	17.5% 17.3% 65.2%
5x5	1800 51.8 MB	Surface GF Device GF Self-Energies	20.5 21.0 61.2	20.0% 20.4% 59.6%

Table 22: Computation of inelastic scattering Si supercells. Timings of different code sections and relative fractions of the total computation time are reported.

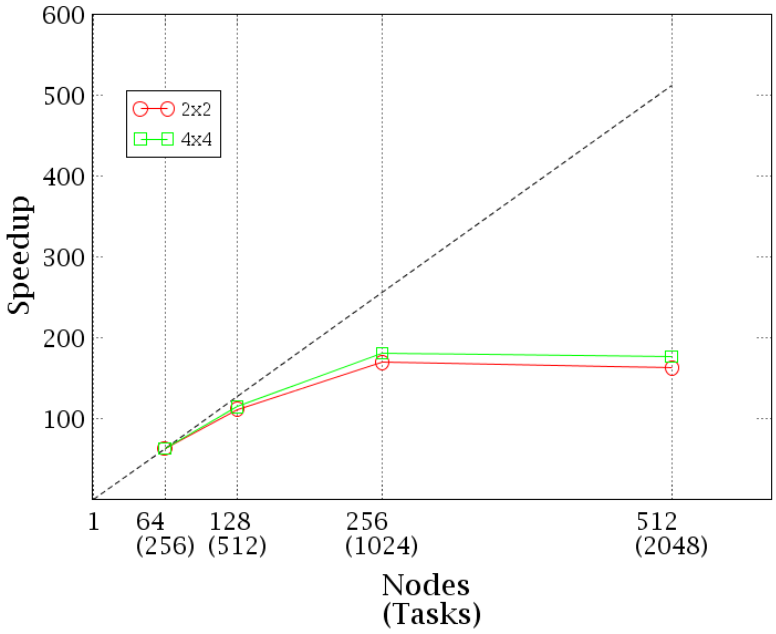


Figure 31: Scaling of the inelastic code for the dftb+ Hamiltonians for two different supercell sizes.

64 computing nodes and for the smallest run we also had to resort to the large memory nodes (192GB). These calculations have a quite different memory fingerprint compared to the benchmarks shown in Fig. 30 that involved many more energy points, therefore the two cannot be immediately compared. We remind that for the way the code is currently structured the Green's functions for different energies and k-points are computed in a first phase and stored in RAM for a fast retrieval when computing the Self-energies. The peak memory required by each task is approximately given by $M = M_{PL} * (3N_{PL} - 2) * N_E * N_K * 4$, where M_{PL} is the memory needed by a single matrix block, $M_{PL} = N^2_{atoms} * N^2_{basis} * 16$ Bytes for double precision complex numbers, $(3N_{PL} - 2)$ are the number of tri-diagonal blocks, $N_E * N_K$ are the number of energy and k-points processed by each task and the factor of 4 is for the four matrices, $\Sigma^{r,n}$ and $G^{r,n}$. Additionally, 3 tri-diagonal block matrices are currently allocated as work matrices in order to store $ES - H - Sigma^r$, G^r and $G^<$ for a given k and E. The symmetry property of G^n and Σ^n is exploited. As an example, the memory required per energy and k-points for the system 4x4 exceeds 5 GB, setting a clear limitation on the processable number of points by each task. The strong scaling found is quite similar for the two systems, showing a communication-limited computation. This fact can also be appreciated on Tab. 23, showing the actual computation timings. The column "Time 1" refers to the tensor-product

implementation without OPENMP parallelization, which has been switched on in the case of "Time 2". The result is that in the case of the slower routines there is still a 30% gain in going from 256 to 512 nodes, whereas, when using the faster parallelised routines the wall timing is obviously better, but saturate at about 256 nodes, indicating that the communication bound has been reached. In order to further improve the scalability the communication overhead has to be hidden with more calculations. One possibility is to overlap the calculations of the interaction self-energies while computing the Green's functions, as soon as corresponding block-matrices are ready. However, this task is highly non trivial and probably require a deep re-thinking of the implementation.

Structure	Nodes	Tasks	Time 1 [sec]	Time 2 [sec]
2x2	64	256	68.72	43.47
2x2	128	512	36.28	28.80
2x2	256	1024	29.31	16.38
2x2	512	2048	22.17	17.07
4x4	64	256	1181.0	790.4
4x4	128	512	600.0	440.0
4x4	256	1024	512.0	279.4
4x4	512	2048	341.0	284.9

Table 23: Code timings for different structure size and node/tasks parallel distributions. The code timings "Time2" refers to the OPENMP *parallelised version of the inelastic self-energy*

7.4.1 The exascale potential of LIBNEGF

The main motivation behind the joint effort within EOCOE-II is to fill the gap of available tools for quantum transport simulations on large supercomputing facilities, especially in the perspective of exascale computing facilities. The NEGF formalism is a highly computing intensive method that provides an excellent example of exascale application. Scaling of the method up to 100 000 cores have been demonstrated, at least within OMEN, thanks to 3 levels of parallelism obtained by distributing **k**-points, energy-points and a domain decomposition.

Further developments of LIBNEGF has to follow a similar pathway. Domain decomposition is necessary in order to reduce the memory load on each node. Storage of computed Green's functions and Self-energies should be done in reduced precision. An example of performance and potential of the computation is shown in Tab. 24, demonstrating the performance of the miniapp created in single precision for the calculation of the surface Green's function of Si supercells with increasing cross-section. Performances of up 90 Tflops/s can be reached on a single GPU exploiting tensor cores. Even using normal cores and performance of 15 Tflops/s, considering runs parallelised on 2048 GPU (512 Nodes), there is a potential of reaching 20 Eflops/s of peak performance.

Size	Mat Size	CPU [s]	GPU [s]	Tens-C [s]	Speedup	TFlops/s
4.8 nm	5832	119	1.4	0.42	x30-x120	60
6.5 nm	10368	450	4.0	1.19	x100-x400	91
9.2 nm	20736	3600	30.0	8.2	x120-x440	72

Table 24: Code profiling of contact self-energy calculations for complex single-precision operations on NVIDIA *A100 GPUs.*

8 Task 2.5 - Hydrology code optimisation

8.1 Task overview

Task leader : FZJ

Participants: FZJ, FAU, CEA, RWTH

The goal of the hydrology scientific challenge is to enable high-resolution (down to 100m) continental-scale hydrological simulations with mixture of active and inactive regions to make prediction of hydropower supply more accurate.

Two flagship codes are concerned by the WP2 technical challenge: ParFlow and SHEMAT-Suite. They are respectively and briefly describe in the following sections 8.1.1 and 8.1.2.

The work in this code is divided into 3 different subtasks:

- Task 2.5.1 - ParFlow code optimisation
- Task 2.5.2 - SHEMAT-Suite code optimisation
- Task 2.5.3 - Unified platform to unify the physics of both code

The detailed content of these tasks and the progress achieved so far is described in sections 8.2, 8.3, 8.4.

8.1.1 Flagship code ParFlow

ParFlow is a parallel, integrated hydrologic model, which simulates surface and subsurface flow (ParFlow website). It is based on the shallow water equations coupled with the three dimensional Richard's equation. The code provides a solver for the latter based on a cell-centered finite difference scheme on regular Cartesian meshes. Time integration is performed with an implicit Euler method. The resulting system of nonlinear algebraic equations is solved by a multigrid-preconditioned Newton-Krylov method.

Table 25 shows the team members of ParFlow involved in EoCoE-II. Stefan Kollet is the Scientific Leader of the hydrology scientific challenge. He is as well the scientific coordinator of the ParFlow code. Jose A. Fonseca is a postdoctoral fellow in computer science and mathematics modelling at MdlS, CEA. He is working on AMR aspects in ParFlow. Jaro Hokkanen is a postdoctoral fellow in computer science at FZJ. He is focusing on porting and optimizing ParFlow on GPU. Mathieu Lobet is coordinating the ParFlow at MdlS and is working on PDI aspects in this code.

People	Position	Role	Period
Stefan Kollet, PhD	FZJ	Scientific coordinator	M1-M42
Bibi Naz, PhD	FZJ	PDI aspects	M1-M16
Jose A. Fonseca, PhD	Postdoc at MdlS, CEA	HPC expert for AMR aspects of ParFlow	M5-M36
Jaro Hokkanen, PhD	Postdoc at FZJ	Computer Scientist on code optimisation and GPU aspects	M9-M33
Mathieu Lobet, PhD	Research-engineer at MdlS, CEA	coordinator and PDI aspects	M1-M42

Table 25: Team Members for ParFlow within EoCoE-II.

8.1.2 Flagship code SHEMAT-SUITE

The SHEMAT-SUITE is a code for simulating single- or multi-phase heat and mass transport in porous media (SHEMAT-SUITE code repository). It solves coupled problems including heat transfer, fluid flow, and species transport. SHEMAT-SUITE can be applied to a range of hydrothermal or hydrogeological problems, be it forward or inverse problems.

Table 26 shows the team members of SHEMAT-SUITE involved in EOCOE-II. Johanna Bruckmann, Research Associate at RWTH Aachen University, is coordinating the SHEMAT-SUITE activities in the different WPs of EOCOE-II and is responsible for the scientific challenge related to SHEMAT-SUITE. Berenice Vallier has been recruited as a postdoctoral fellow to work on SHEMAT-SUITE related tasks in WP2, WP3 and WP4.

People	Position	Role	Period
Johanna Bruckmann, M Sc	Research Associate at RWTH Aachen University	SHEMAT-SUITE coordinator; scientist for WP1	M1-M36
Berenice Vallier, PhD	postdoctoral fellow at RWTH Aachen University	PDI implementation and PDAF	M10-M23

Table 26: Team Members for SHEMAT-SUITE *within* EOCOE-II.

8.2 Work progress on task 2.5.1 - PARFLOW optimisation

This section presents all the work carried out in subtask 2.5.1 on the optimisation and porting of the PARFLOW code. At the beginning of the project, PARFLOW was only working on CPU architectures. Within the framework of EOCOE-II, we focused on 3 distinct development and optimisation activities:

- Improvement of the code efficiency on CPU using Adaptive Mesh Refinement (AMR) capability: The main interest in using AMR is the possibility to use a wide range of different spatial resolutions at a reduced computational cost. With the upstream version of PARFLOW, the minimum required spatial resolution is used to solve the whole domain with the same accuracy even where a lower resolution would be sufficient. AMR enables multiple scales in the same simulation saving computational resources for high-resolution regions. Furthermore, the spatial discretization can be refined dynamically to adapt in time to the physical parameter evolution. A former project leaded by Carsten Burstedde at the University of Bonn and Stefan Kollet at FZJ, funded by the German Research Foundation (DFG) has led to the integration of PARFLOW and the AMR library p4est without explicitly exploiting the AMR capabilities provided by this library. The first development work extends the existing PARFLOW 's integration with p4est by using the AMR routines of the latter and allowing the use of locally refined meshes. This work was done at Maison de la Simulation by Jose Alberto Fonseca Castillo. He left the project at M36. The work summary and conclusion are described in section 8.2.3.
- Porting of PARFLOW on GPU architectures: This work was handled at FZJ by Jaro Hokkanen. He left the project at M33. The work summary and conclusion are presented in section 8.2.2.
- PDI implementation: The role of the PDI implementation is to allow a better management of the code outputs (addition of new output formats and capability) and to allow coupling with other libraries in connection with WP4 and WP5. The work related to PDI is presented in Sec. 8.2.1.

Since the beginning of the code, PARFLOW general implementation has been historically based on an internal lightweight domain-specific interface (also referred to as PARFLOW eDSL) for the memory manage-

ment, data structure access, message passing and looping. The scientific computational kernels for solving the relevant physics are directly built upon this interface based on C pre-processor macros. The structure between the domain-specific interface and other components is schematically described in Fig. 32. Initially, this interface was only used to abstract, simplify and separate the purely computational aspects from the numerical and scientific implementations (scientific kernels) making the programming easier for physicists for instance. Before the EoCoE-II project, this domain-specific interface was only used for a single backend allowing the code to run exclusively on the CPU. However, this approach appeared completely suitable for the implementation of GPU and AMR backends. It can in principle be leveraged to enable easy accommodation of any kind of low-level programming models (CUDA, HIP, OpenACC, OpenCL, OpenMP, etc) and flexible third-party libraries (Alpaka, Kokkos, or RAJA). Thus, insourcing development to support all required accelerator architectures on the local backend can be avoided, and the interface for memory management and looping is independent of the used accelerator programming model allowing full customization in the underlying scientific domain. Furthermore, the cost of choosing a wrong accelerator programming model is minimized and adding support for new programming models including libraries is straightforward. This approach has many pros:

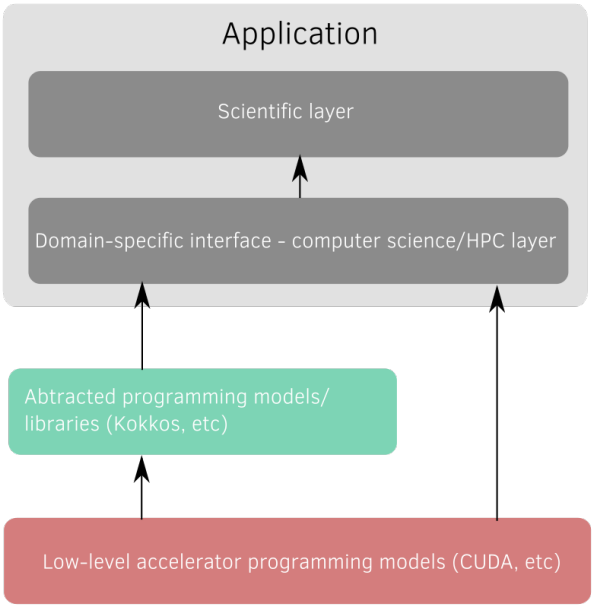


Figure 32: Description of the ParFlow domain-specific interface.

- Separation of concerns
- Incremental adoption
- Flexibility with algorithms and data structures
- Fully customizable interface for compute kernels and memory management Task
- Codebase remains well maintainable
- Easy adoption of one or more accelerator programming models or libraries
- Cost of choosing a “wrong” programming model or library is minimized

And few cons:

- Development and implementation of a lightweight adaptor layer
- Compatibility of the adaptor layer with all future backends is not guaranteed

In the framework of EoCoE-II, a new CUDA and Kokkos backends were implemented to target GPU accelerator. Fig. 33 is a detailed description of the current code structure after the work done in EoCoE-II. The CUDA implementation offers the best performance on NVIDIA accelerator. The Kokkos backend ensures performance portability across major GPU accelerator vendors (NVIDIA, AMD, Intel).

8.2.1 PDI implementation

A first PDI implementation has been done in ParFlow that will be certainly improved after more intensive tests. ParFlow currently has three solvers implemented. Each solver use its own physical parameters

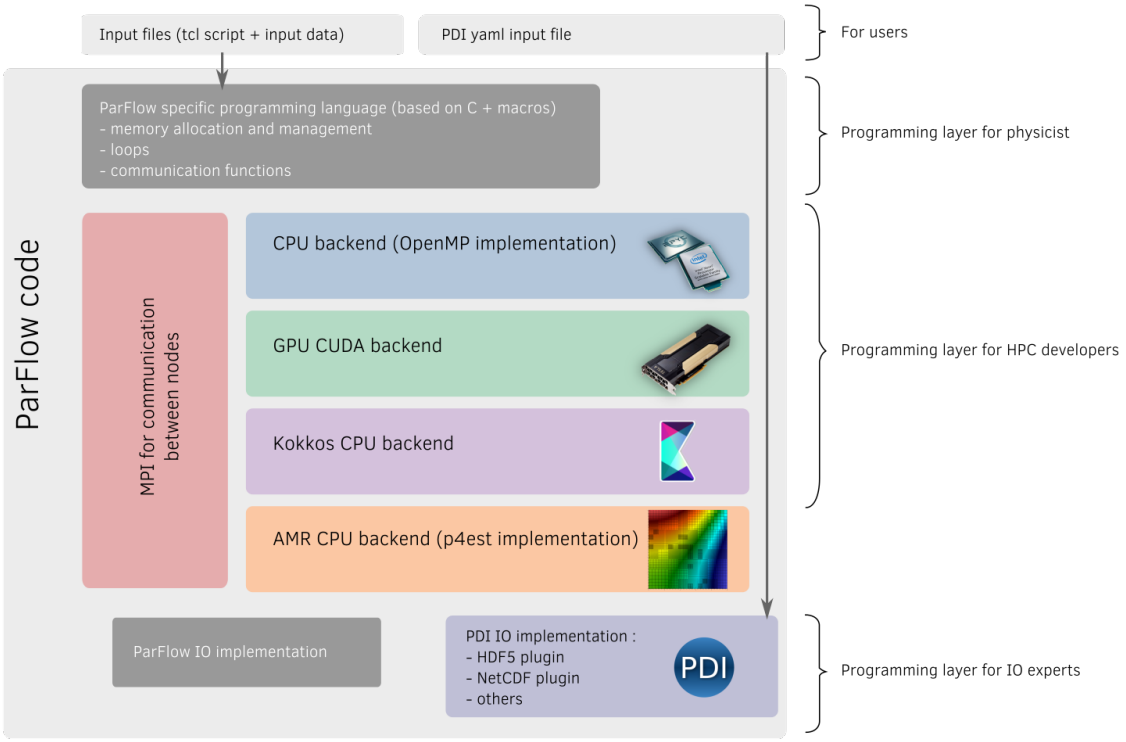


Figure 33: Description of the PARFLOW *programming model.*

and therefore has a specific output process. However, each solver calls the same generic output functions. All of them have been updated for PDI.

The PARFLOW output code structure is shown in Fig. 34. It includes how PDI is plugged. They are three kinds of output format that can be used: PARFLOW binary files (PFB), SILO, NetCDF. The PFB format is a home-made binary type of output. Each format has its own generic functions used by the solvers. Output parameters and period can be controlled in a similar way via the tcl input script. PDI has been implemented as a 4th possible output format for each solver. It therefore respects the same formalism and the same implementation structure. Similarly, PDI options in the tcl input script are similar to what has been done for other output formats. Currently, parameters available via PDI are the same as for the PFB format.

An advantage of PDI in the future is the possibility to make all output options in PARFLOW uniform. Indeed, all physical quantities cannot be written on disk in all formats. Some of them are only available with a specific format. Thanks to PDI, this limitation would be solved easily. Once a physical quantity can be exposed to PDI, it can then be written using any PDI plugin.

Contrary to the format currently used by PARFLOW that requires a pre-process of the data (for instance, for the PFB files, the content of the physical data vectors is treated), PDI uses the full data structures. The treatment of the data is let to PDI and depends both on the YAML configuration file and the plugin. The YAML is now provided with the code.

The PDI implementation has been validated using the `default_single` and the `default_richards` test cases. In our tests, we have used the HDF5 plugin. We have developed a PYTHON script that compares the data on disk in `.pfb` files and `.h5` files.

The PARFLOW version with PDI is not officially released and is only accessible within the project. When this version will be validated with WP4 and PDI experts, this new feature will be proposed to PARFLOW's developers for integration in the master version of GITHUB.

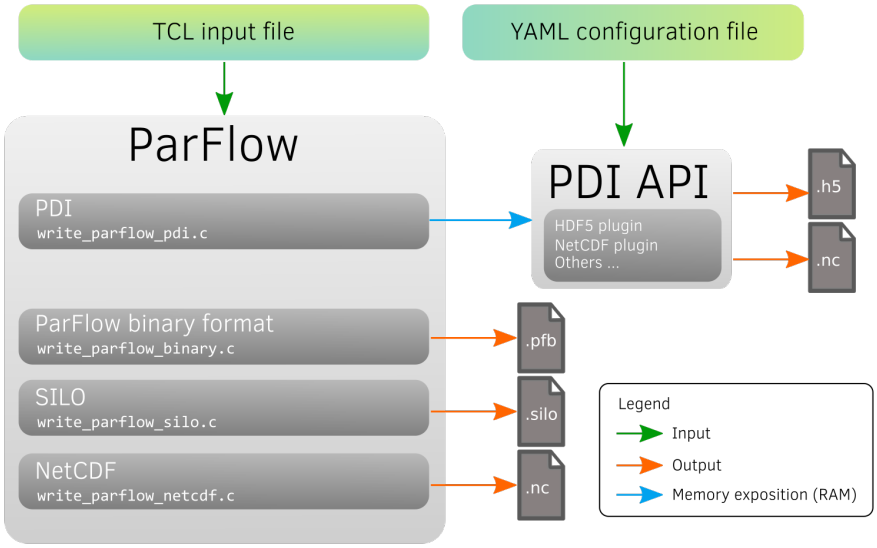


Figure 34: PDI *integration in* PARFLOW.

8.2.2 GPU porting

The work done on the GPU porting of PARFLOW is described in this section. The developments started at M8 when Jaro Hokkanen was hired at FZJ. As shown in Fig. 35, we have subdivided the work into micro-tasks. PARFLOW is now fully functional on GPU. The new version is now integrated to the master one. The following describes the integration details and performance results.

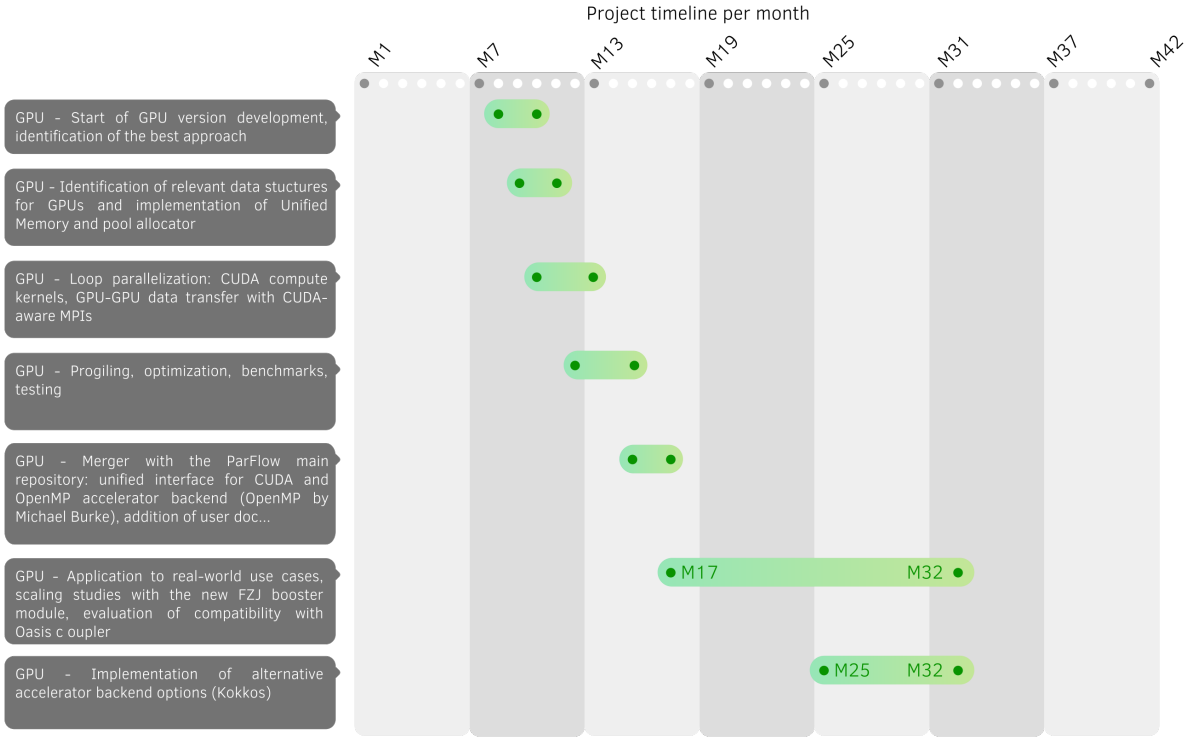


Figure 35: Breakdown (simplified Gantt chart) of the task 2.5.1 concerning GPU in PARFLOW.

The GPU acceleration is built directly into the PARFLOW embedded domain-specific language (PARFLOW eDSL) headers such that, ideally, parallelizing all loops in a single source file requires only a new header file. This is possible because the PARFLOW eDSL provides an interface for looping, allocating memory,

and accessing data structures. As already mentioned, the decision to embed GPU acceleration directly into the eDSL layer resulted in a highly productive and minimally invasive implementation.

Adding CUDA GPU-support into ParFlow eDSL The first accelerator backend supporting GPUs is based on CUDA. Features provided by CUDA C++ such as Unified Memory (with a pool allocator) and host-device lambdas were extensively leveraged in the ParFlow implementation in order to maximize productivity and codebase maintainability in the long-term. Efficient intra- and inter-node data transfer between GPUs rests on a CUDA-aware MPI library and newly developed application side GPU-based data packing routines. Modern supercomputers typically consist of large numbers of nodes which may have multiple GPUs available. For a program such as ParFlow which originally relies solely on a message passing library for parallelism, the best option often is to launch the same number of processes as there are GPUs intended to be used. Each process then uses only one GPU and the communication between GPUs relies on a CUDA-awareMPI library that supports direct GPU-GPU data transfers.

Memory management As the physical memory spaces between the host and the devices are separated, it is essential to make sure that the relevant stored data is accessible for each device. With the current CUDA version, this means replacing the standard host memory allocations and deallocations by the functions provided by the CUDA toolkit. While the CUDA API provides means to allocate memory on the host-side such that the device can access this data directly through the PCI Express bus or NVLink, storing the data in the device memory is usually more efficient. This is achieved by the CUDA specific functions for device-pinned or Unified Memory allocations. Furthermore, cudaFree must be used for deallocations. One drawback of simply replacing the required standard host memory allocations by a call to cudaMallocManaged is the significantly increased memory allocation overhead. This may cause a problem in case of recurring allocations and deallocations. For this reason, ParFlow supports using Rapids Memory Manager (RMM) for Unified Memory allocations. Instead of calling cudaMallocManaged directly, a function rmmAlloc provided by the RMM API is called which then calls cudaMallocManaged internally. RMM provides a pool allocation mode in which the memory pool is prefetched to the device and the memory is never deallocated while the pool is in use and allowed to grow. A call to rmmFree makes room for new allocations without decreasing the pool size. This removes the overhead of recurring Unified Memory allocations without a considerable increase in peak memory usage (although the average memory consumption is increased). In ParFlow, dynamic memory allocation and deallocation at the scientific implementation layer (see Fig. 32) is handled by specific eDSL preprocessor macros. Depending on the targeted architecture (CPU or GPU), the domain-specific layer decides which allocator to choose (default CPU allocator, CUDA unified memory allocator, CUDA RMM allocator, etc)

Loop parallelization In ParFlow, loops over the discretized domain are always accessed through the eDSL API. Similarly to memory management, the loop execution is defined by preprocessor macros. However, only a few general loop macros are provided for which the loop body is given as a macro argument. In ParFlow, the loop body is typically provided as the last argument to the macro (the loop body refers to the contents within the curly brackets). This approach allows using the same loop macro for a large number of loops with different loop logic and a varying number of variables required by the loop. In fact, there are over one hundred loops with often very different loop bodies that use a single loop macro in ParFlow. The definitions for the loop macros depend on whether they are executed on the host or the device. The sequential definition is straightforward as the loop body is just placed inside the innermost loop in the macro definition. However, in the parallel version, a GPU kernel must be launched for which the loop bodymacro argument cannot be directly passed. Instead, a relatively new CUDA feature known as extended host-device lambda is used to pass the loop body to the GPU kernel. The loop body macro argument is placed inside the lambda function such that the lambda function contains all required information about the loop logic; the variables found inside the loop body are captured by their value. Now

all required information about the loop body can be passed to the GPU kernel as a single argument, i.e., the lambda function. This approach allows incremental development and easy parallelization of a large number of compute kernels, while minimizing the amount of new code. However, it is important to note that the parallel loop macros pose some additional restrictions to the loop body; the most common restrictions are listed below:

- Host variables defined outside the loop body cannot be changed
- Pointers must point to Unified Memory allocations
- Functions called inside the loop body must have host and device identifier
- Operations causing race conditions (e.g. increment) must use atomic functions

GPU-GPU direct communication Most of the recurring intra-node and inter-node communications between the processes such as the halo exchange involve data that is stored on a GPU and needed by another GPU. Therefore, efficient data transfer between GPUs on a node and also across nodes is important. The data could be copied from a GPU to a staging buffer on the host, then transferred to the host staging buffer of another process using a message passing library, and finally copied back to a GPU device. In this case, the choice of the accelerator architecture would not pose any requirements for the message passing library, but the resulting performance would be bad due to many unnecessary operations that are not properly pipelined. Better performance can be obtained by leveraging direct GPU-GPU communication such as NVIDIA GPUDirect or AMD DirectGMA. For example, GPUDirect Peer-to-Peer (P2P) and Remote Direct Memory Access (RDMA) enable direct data transfers between two GPUs (intra-node) and a GPU and a network adapter (inter-node), respectively. However, usage of these technologies requires additional support from the message passing library. For example, at the Jülich Supercomputing Centre, Germany, ParFlow is frequently run with MVAPICH2-GDR and Parastation MPI which both support GPUDirect P2P and RDMA, and are often referred to as CUDA-aware MPI libraries. The default message passing option in ParFlow relies on derived MPI datatypes and MPI library-side data packing and unpacking. When this message passing option is used with GPU acceleration, the pointers passed to the MPI library point to Unified Memory allocations. However, the authors found no CUDA-aware MPI library which would pack and unpack the data for the underlying MPI data type on the GPU, and leverage the fast GPUDirect data transfers. An optimised GPU-aware application-side data packing and unpacking has therefore been implemented using the standard MPI byte type. Efficient GPUDirect data transfers have leverageable with both aforementioned CUDA-aware MPI libraries, MVAPICH2-GDR and Parastation MPI. When using GPU acceleration in ParFlow, the application side data packing and unpacking for each process is performed in multiple streams on a GPU using a pinned GPU staging buffer; a pointer to this staging buffer is then passed to the MPI library. Using pinned GPU memory instead of Unified Memory for the staging buffers typically results in better performance because the MPI library must internally use a pinned buffer anyway (GPUDirect data transfers do not support Unified Memory). The changes required to leverage GPU-GPU message passing have been implemented into the message passing layer of ParFlow which is not solely based on preprocessor macros but is instead compiled as a separate library.

Optimization for the CUDA backend No architecture-specific optimizations were introduced into the loop body macro arguments which are shared between the host and device compilation paths.

- Minimizing data transfers between host and device (large impact): Little performance improvement was realized until most of the frequently executed loops were offloaded to the GPUs. This is explained by page faults and recurring data migrations between the host and device along the PCI Express bus or NVLink. When a virtual page is accessed that is not mapped to a physical page on the memory of the underlying processing unit, a page fault is generated. The issue is resolved during the runtime by locating and copying the data and remapping the virtual page to the corresponding physical page such that it is now accessible to the processing unit in question. This is referred to as

on-demand paging and is supported on the GPUside since the NVIDIA Pascal architecture; for older NVIDIA architectures, all Unified Memory is always migrated to the device memory prior to launching a GPU kernel. The single most important part of the optimisation was to minimize the page faults and avoid recurring memory transfers between host and device. This was mostly achieved by offloading all loops accessing the same data to the GPUs therefore minimizing the need to migrate pages to the host memory.

- Adding efficient parallel reduction kernels (large impact): Instead of using *atomics*, Efficient parallel reduction kernels that leverage the CUB (CUDA Unbound) header-only library were added to the ParFlow eDSL.
- Using a pool allocator for Unified Memory (large impact)
- Coalescing global device memory accesses (large impact): Another high priority memory optimisation is coalescing the global device memory accesses. Considering the architectures starting from Pascal, the global memory is accessed in transactions of 32 bytes in size. If a warp of 32 threads executing the same instruction accesses a 32-byte aligned array of 128 bytes, only a maximum of 4 global memory transactions are required. This is the case for example when k-th thread within a warp accesses the k-th 4-byte integer of a 32-byte aligned array. In case the array is not aligned, 5 transactions are required. On the other hand, for a strided array with the stride size larger than 32 bytes, each thread requires a separate transaction resulting in a total of 32 transactions. In ParFlow, the data along the x-dimension of the domain is typically stored in consecutive memory locations. Therefore, mapping this dimension to the x-dimension of a three-dimensional grid of a GPU kernel results in an ideal memory access pattern for non-strided arrays.
- Avoiding unnecessary synchronizations (small impact): With CUDA, multiple GPU kernels can run concurrently while the CPU is performing other tasks. Therefore, better performance is achieved when the number of synchronizations between host and device are minimized. Generally, the CPU does not need to wait for a GPU kernel to finish immediately after launching a kernel and can instead continue the program execution also queuing more GPU kernels for execution. However, a synchronization is required before the CPU accesses data that is relevant for the device kernels. Unfortunately, in ParFlow, the adaptor layer is not aware of the program control flow of the scientific code and does not know when synchronizations are needed. Therefore, as a default option, a specific CUDA function is called after each kernel launch guaranteeing that the CPU does not continue before the kernel is finished. However, an option to prevent synchronization after a kernel launch is provided by the API for advanced users, and is used for the most benefiting code regions.
- Tweaking with kernel launch configurations (small impact): For optimal computing efficiency and memory coalescing, the number of threads per block should be a multiple of warp size (32 threads for the currently available NVIDIA architectures). In ParFlow, the block size for the x-dimension is currently set to 32 for best memory coalescing, while the block sizes for y- and z-dimensions are dynamically adjusted based on the problem size.

Kokkos backend The NVIDIA CUDA programming language is mostly adapted to NVIDIA GPUS. This programming model is now partially portable to AMD recent GPUs through AMD HIP CUDA translating and compiling tools. Nonetheless, more general performance portability can only be achieved by adding new proprietary programming models or leveraging existing abstracted performance portable libraries such as Kokkos, Alpaka, RAJA or equivalent. It is important to remind that (see Fig. 32) these libraries use vendor-specific programming models such as CUDA, HIP, OpenCL and others. Following the work already done for the CUDA backend, adding backend support for performance portable libraries to target more architectures is straightforward and does not require any major changes. This is because, in ParFlow, the approach of passing loop contents form a lambda function to a general CUDA kernel can also be used with the aforementioned libraries as demonstrated for the ParFlow Kokkos backend in this study. Thanks

to the macro-based abstraction layer, the Kokkos backend is not a compulsory dependency for ParFlow. This is important to hedge the risk of introducing third-party dependencies of unknown sustainability. Also, the Kokkos API is accessed through only a small number of bundled ParFlow eDSL macros, such that replacing Kokkos with another similar backend is easy, e.g., in case Kokkos development stagnates in the future. In our experience, the CUDA backend option required several months of development time from a single developer. Adding Kokkos as an alternative to CUDA required just few weeks of additional full-time work with no prior knowledge of Kokkos. Compared to CUDA, Kokkos lacks a C interface, thus the Kokkos API calls must be placed into a separate C++ compilation unit that provides wrapper functions callable from C code in case of ParFlow. For memory management, Kokkos allocator and deallocator are used behind the ParFlow interface instead of more advanced Kokkos Views object for instance. The implementation of compute kernels in ParFlow follows an approach similar to the memory management. As for the CUDA backend, the loop execution is defined by preprocessor macros which take the loop body as a macro argument (typically provided as the last argument to the macro). This approach is key in allowing to use the same macros for a large number of compute kernels regardless of the loop logic and the number of variables involved in the calculations. Similarly to the memory management, the definitions for the loop macros depend on whether they are executed on the host or the device. The macro used with sequential execution just places the loop body macro argument inside the innermost loop in the macro definition. In case of the Kokkos backend, the loop body forms a lambda function that is passed to the Kokkos kernel, where the lambda function simply captures all required variables by their value (i.e., with pointers the captured value is just the address the pointer is pointing to, and the data is accessed by denoting the offset with the conventional square bracket syntax).

Large-scale performance results on AMD Rome + NVIDIA A100 A representative benchmark problem was run on the booster module1 of the JUWELS supercomputer where each utilized node is equipped with dual AMD EPYC Rome 7402 processors (2 × 24 cores at 2.8 GHz) and 4 NVIDIA A100 40 GB GPUs. The nodes are connected through 4HDR200-InfiniBand devices. This design is now becoming common in more and more HPC systems.

The benchmark consists of a variably saturated infiltration problem into a homogeneous soil with a fixed water table at a 175 depth of 6m, and a constant infiltration rate of 8×10^{-4} m/hour. The vertical and lateral spatial discretization was 0.025 and 1 m, respectively. The time step size was 1 h. The profile was initialized with a hydrostatic profile based on a matric potential of -9 m at the top resulting in a considerable initial hydrodynamic disequilibrium with respect to the water table at the bottom boundary. The number of grid cells in the lateral directions was varied to change the total number of degrees of freedom in performance testing (weak scaling).

Fig. 36 shows the performance gain from GPUs for the first performance study on a single node. The horizontal and left vertical axes represent respectively the problem size and the performance in number of cells per second. The performance with GPUs is plotted using CUDA directly with pool allocation (no Kokkos), and using CUDA through Kokkos with and without pool allocation.

For this study, the number of grid cells in the lateral directions was varied to change the total number of degrees of freedom in performance testing (between 1442 $\times$ 240 and 10082 $\times$ 240 cells). The upper value for the number of cells was limited by the available GPU memory (4 × 40 GB). The system of equations formed from the nonlinear problem is solved for each iteration at each time step using the GMRES method along with the PARFLOW internal multigrid preconditioner. The input file for the benchmark problem is available in the PARFLOW repository [23]. The reference results on CPU (red line) were obtained without accelerator devices by launching an MPI process for each CPU core. In the case of the accelerated runs, 4 MPI processes per node were launched such that each process uses one GPU and the halo exchange leverages GPU-based application-side data packing (unpacking) before (after) the MPI communication takes place. The simulation results between accelerated and non-accelerated simulations are not bit-identical due to floating-point operations which are conducted in a different order. However, the difference is

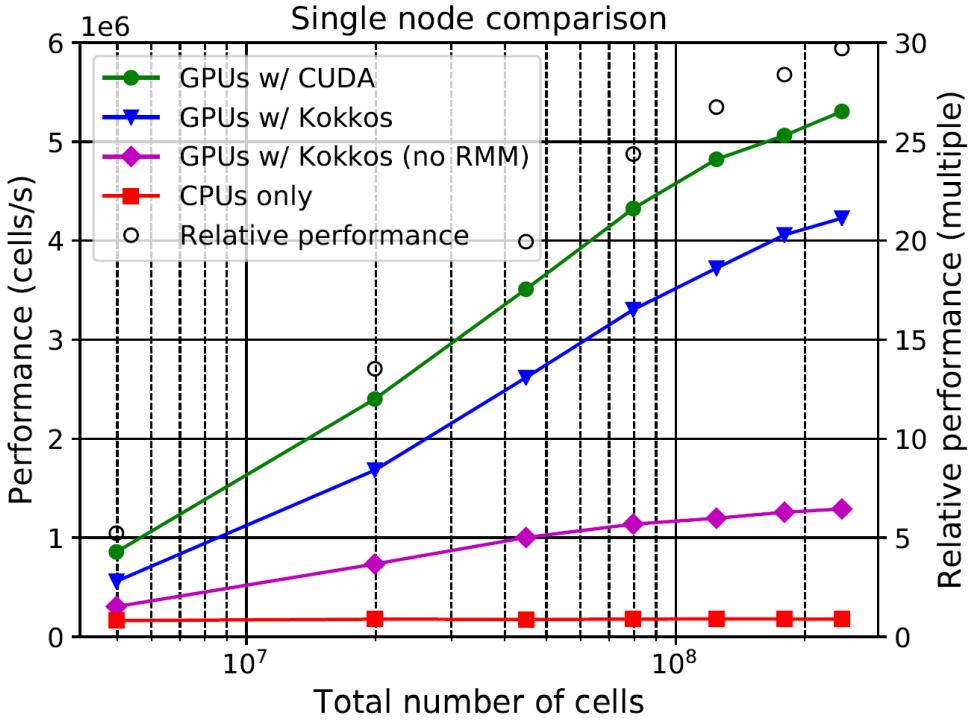


Figure 36: PARFLOW *single node performance comparison. The horizontal axis is the problem size. The left vertical axis is the performance in computed cells per second. On the right axis, relative performance refers to the performance ratio between CUDA and CPU implementations of the code. RMM denotes Rapid Memory Manager.*

negligible not only for the presented benchmark problem and several other real world applications but also for more than one hundred automated tests cases that are run frequently to validate the implementation.

The impact of using the pool allocator (comparing the purple and blue lines) increases with the increasing number of cells and more than triples the performance for the largest problem sizes. The Unified Memory pool allocation introduces architecture-specificity, as the chosen memory manager library only supports CUDA. The plot further suggests that even better performance could be achieved with GPUs providing more memory capacity, although with diminishing returns. Relative performance, which is the ratio between the accelerated and non-accelerated simulation when using CUDA directly, is given as circles in Fig. 36. The relative performance increases from ~ 5 to ~ 30 with increasing problem size.

Using Kokkos without CUDA-specific code results in a 20% performance reduction for the largest problem size when compared with the CUDA implementation. This is mainly caused by parallel reductions, array initializations, and usage of pinned host/device memory for MPI staging buffers to enable GPU-direct P2P communication with Remote Direct Memory Access (RDMA) when using Kokkos. The latter two overheads can be easily resolved by using CUDA-specific function calls or template arguments with the Kokkos library, which, however, leads to an undesired non-architecture-agnostic implementation. The performance boost of the pool allocator is similar to the one resulting from using directly the CUDA backend. This is explained by the recurring Unified Memory allocations and deallocations during the simulation in PARFLOW.

Fig. 37 represents weak scaling for 1, 4, 16, 64, and 256 nodes using the largest problem size from Fig. 36. The relative performance when directly using CUDA saturates at ~ 28 when increasing the number of nodes which suggests good weak scaling behaviour and performance. In comparison, the performance achieved with the generic Kokkos backend is about 20-25% worse.

Finally, it is noted that the relative performance of ~ 26 achieved in the weak scaling study on multi-

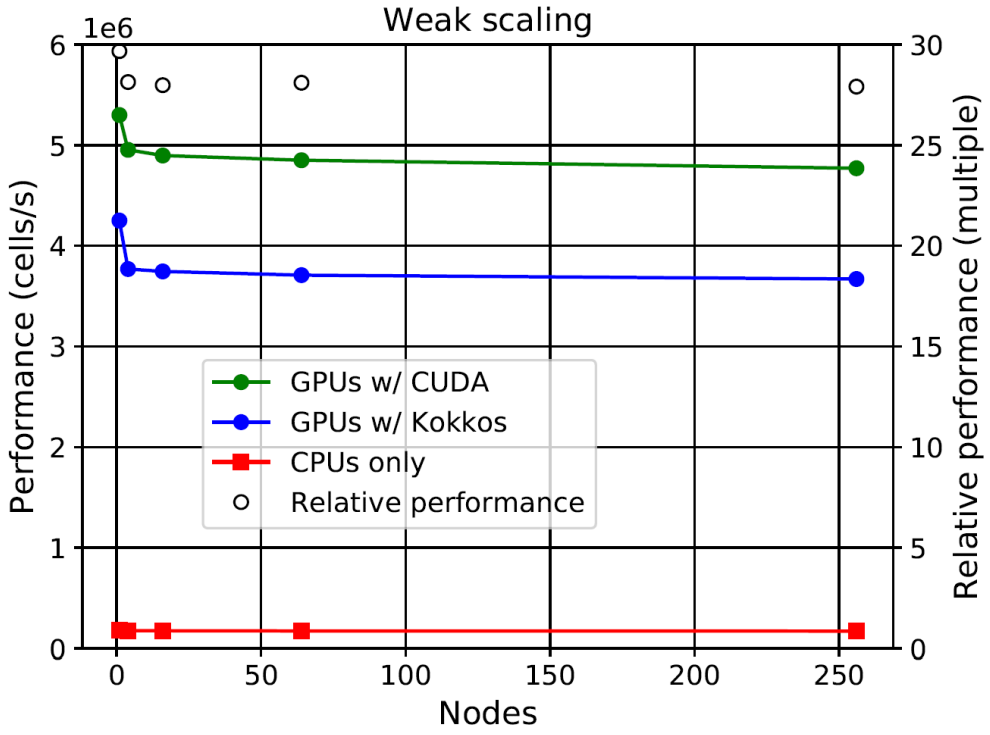


Figure 37: Weak scaling comparison between the CPU, the CUDA and the Kokkos GPU backends.

ple nodes may represent a more meaningful metric of the performance gain from using GPUs, because the proportionally higher communication overhead in the non-accelerated single-node simulation vanishes when the number of nodes is increased. It is also emphasized that the achievable speedup is highly dependent on the underlying problem, numerical methods, and implementation.

Conclusion for the GPU porting The original design of Parflow based on a domain-specific interface capable of leveraging different parallel programming models transparently for the scientific developers enabled to speed-up the GPU porting with minimal impact on the code and without touching the scientific compute kernel. EoCoE-II has enabled the implementation of two distinct programming models:

- CUDA backend: maximize performance on NVIDIA GPU architectures and enable performance portability on AMD GPU architectures (via AMD CUDA to HIP compilation) probably at the cost of a small performance loss (to be tested).
- Kokkos backend: enable performance portability on major GPU architectures (NVIDIA, AMD, INTEL) that will be used in Exascale systems. However, the performance is slightly worse than for a proprietary backend like CUDA.

This work has lead to a first publication focused on the CUDA backend implementation in 2021 (see [24]). A second paper will be finalized soon about the Kokkos backend and performance results.

The work carried out in ParFlow thanks to EoCoE-II has been successfully transposed to another geophysical code called MPDATA. The same DSL-based approach was adopted. This work has demonstrated that many geophysical research applications, which for various reasons can not rely on a large team of software engineers, nevertheless may gain access to rapid computational benefits applying the proposed eDSL concept. At the same time, code readability has been maintained and established coding paradigms has remained accessible to the domain scientists. This approach could therefore be taken as an example and generalized to many codes.

8.2.3 AMR implementation

Converting a mature and a complex code like PARFLOW to AMR is a challenging task because two fundamental parts of the code are constructed under the assumption of a uniform mesh: first, the way in which information is communicated between processes, i.e., the parallel partition and second, the mathematical operators that represent the underlying PDEs the code aims to solve. We have identified several tasks to follow in order to solve these challenges and summarized them in Fig. 38. In the rest of this subsection, we present a summary of the work done in each of these steps and we conclude about what is left for the future.

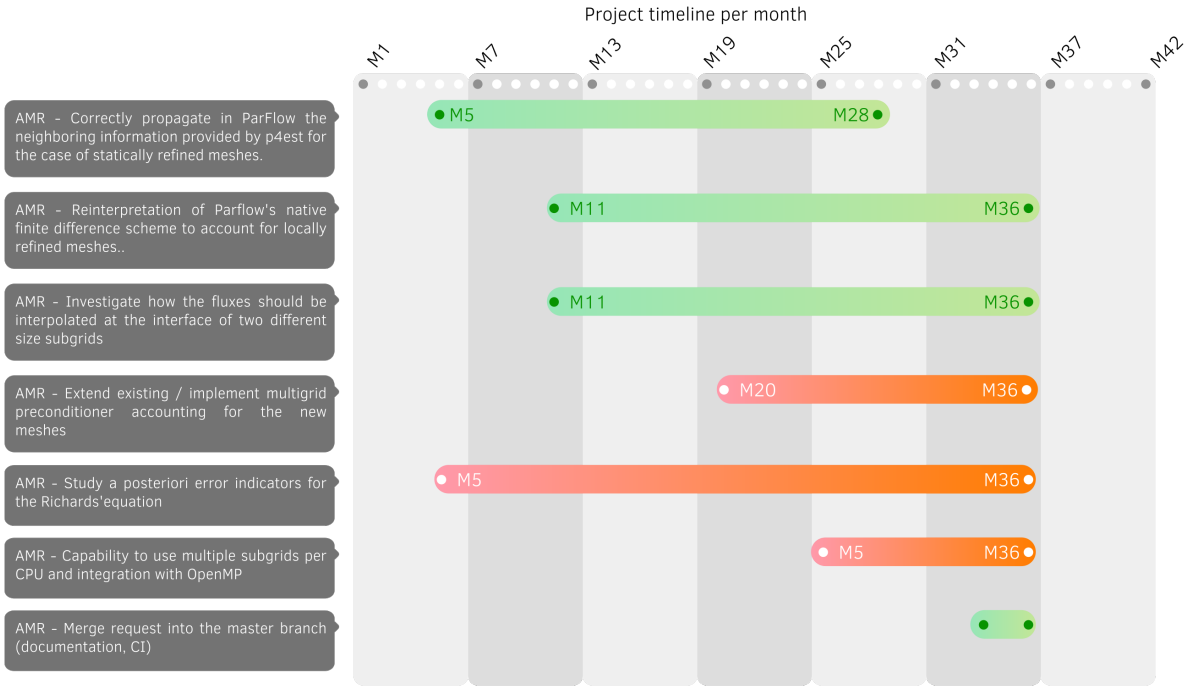


Figure 38: Breakdown (simplified Gantt chart) of the task 2.5.1 concerning AMR in PARFLOW.

Extensions for locally refined meshes In the publication [25] we exposed the integration of PARFLOW with the parallel AMR library `p4est`. We demonstrated with numerical examples how this enlarged the range of process counts that PARFLOW may be executed with and improved parallel scalability. These results were obtained for uniform meshes and hence without explicitly exploiting the AMR capabilities of the `p4est` library. In this section we present the our algorithmic approach to extend the code for local mesh refinement.

The `p4est` library builds upon the concepts of an octree which is naturally associated to a mesh covering a cubic (square in 2D) domain [**?**, **?**]. Furthermore, the class of domains that may be represented can be enlarged by considering unions of octrees, conveniently named forest of octrees. A core functionality of tree based AMR libraries like `p4est` is to dynamically change the mesh elements by traversing the quadrants of the corresponding forest and either

- refine the mesh by replacing a quadrant by its eight (four in 2D) children,
- coarsen the mesh, replacing a family of eight quadrants (four in 2D) by its common parent.

The `p4est` implementation of the previous routines takes as argument a user defined callback function that marks the quadrants to be considered for refinement or coarsening. A key feature of the SFC approach

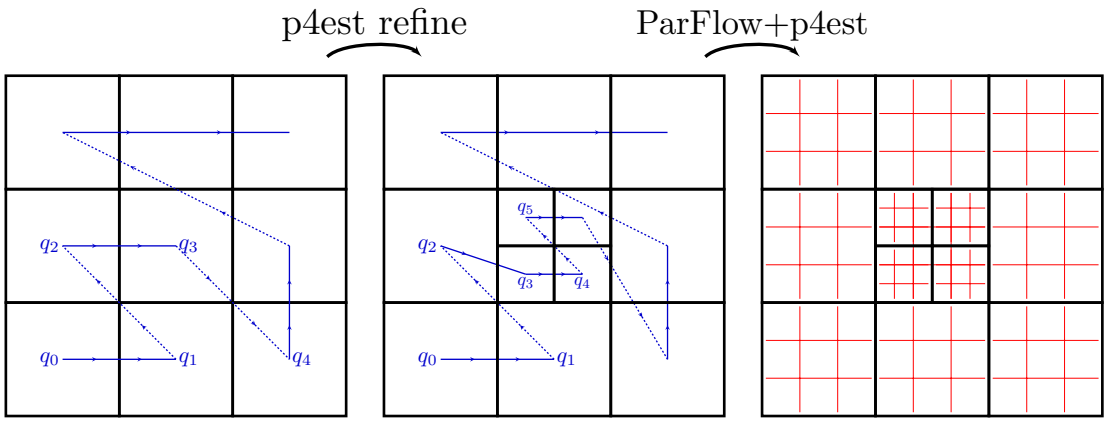


Figure 39: PARFLOW

- [Work flow to enable AMR]Work flow to enable locally refined meshes in PARFLOW. We create a `p4est` object representing the mesh displayed in the left picture. The corresponding space filling curve is shown in blue. With the `p4est` refine routine, this `p4est` object is modified to represent the mesh displayed in the middle. Following the work [25], we keep the idea of attaching a subgrid to each of the quadrants in the later `p4est` object. The number of points per subgrid remains fixed but we adjust the mesh spacing when attaching a subgrid to a refined quadrant.

used in `p4est` is that the ordering of the quadrants is maintained after refine or coarsen are executed. Hence, manipulation of the mesh resolution can be achieved with small movements of data. Furthermore, quadrants of different refinement level are allowed to be neighbours of each other. This translates into meshes where elements of different sizes share parts of a mesh face or edge. Optionally, the `p4est` library guarantees that the size of the difference is at most a factor of two. This is known as 2:1 balance condition which will be used in our work. The main idea of how a locally refined mesh is introduced into PARFLOW is displayed in Fig. 39.

Evidently, such changes in a fundamental part of the code like the mesh management introduce several challenges in other parts of the code. Other PARFLOW structures have to be correctly informed about the new mesh layout, for example:

- Routines taking care of executing loops over the local degrees of freedom.
- Parallel vector/matrix updates.
- Discretization and numerical solvers.

We discuss the approach taken to solve these points in the following subsections.´

Loops construction with AMR PARFLOW implements an octree-space partitioning algorithm to depict domains with more heterogeneous structures. For example, multiple subdomains with different physical parameters, e.g., conductivity, topography or boundary conditions. The approach taken is classical: The object of interest is enclosed in the smallest possible box that contains it, the box is recursively refined and the resulting box children are flagged accordingly if they intersect, contain or are contained in the target object. These translates into an octree structure, whose leaves may represent a portion or the whole object, and hence, be used to implement loops. See 40. The level of refinement of this octree is automatically determined by the code, we will call it (as in the actual code) `MaxReflevel + bg_octree_level` for reference.

The PARFLOW's internal octree structures and loops are fundamental parts of the code at the core of almost all numerical computations. It was important to minimize the code modification in these sections. In order

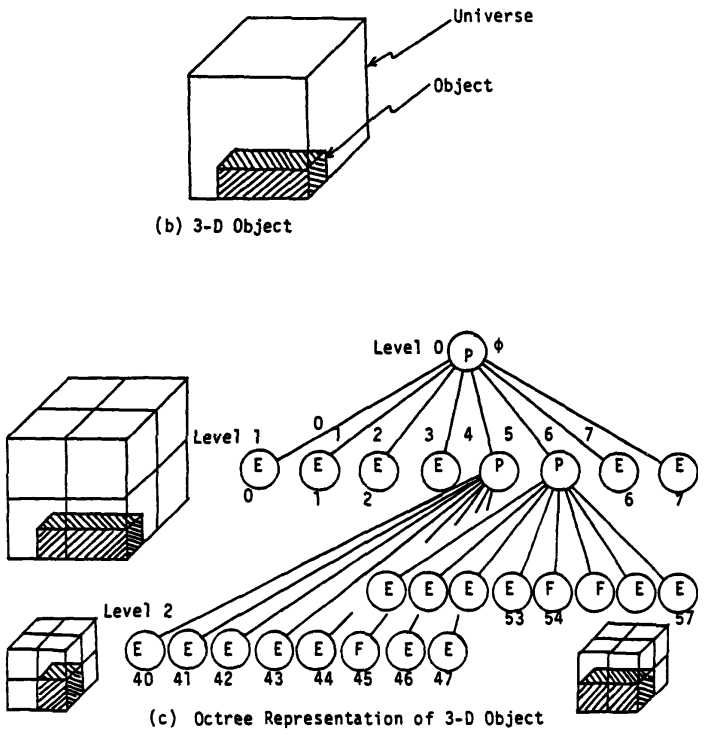
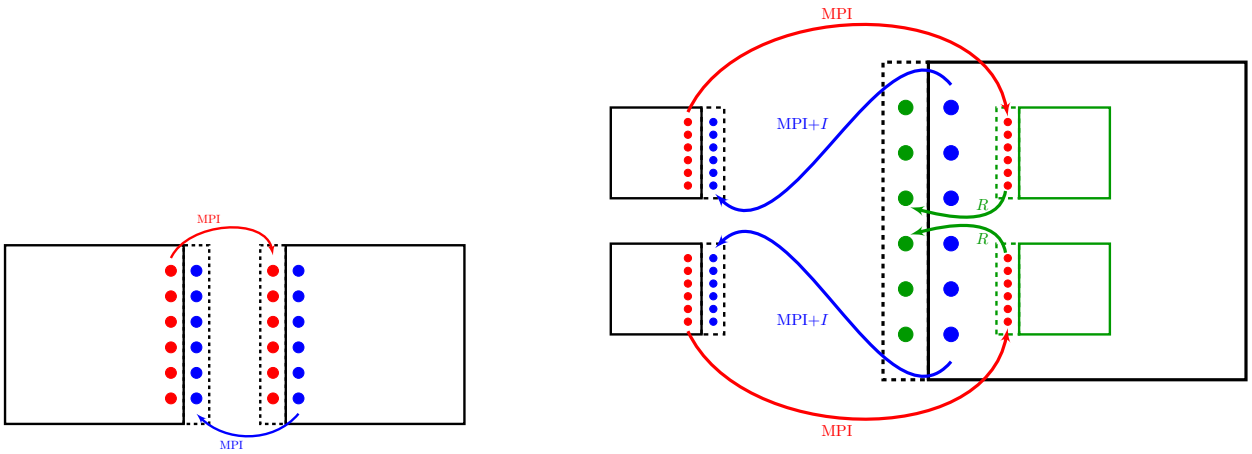


Figure 40: Octree structure implemented in PARFLOW. *Original figure from: Donald Meagher, Octree Generation, Analysis and Manipulation 1982.*

to avoid too disruptive changes in the code, our approach was to find a relation between the PARFLOW's internal octree and the `p4est`'s octree structure. We have decided to exploit an existing parameter called `MaxRefLevel`, which controls the maximum refinement of such PARFLOW internal octree structure. For practical applications this parameter was always set to zero in the classical version of PARFLOW. Our idea was then, use this parameter to control the maximum level of refinement of the `p4est` structure instead, and assume then that the `p4est` tree structure represents a domain that is always contained in a big box of sides two to the power of `MaxReflevel + bg_octree_level`. All (integer) coordinates computations to locate a particular subgrid are then performed relative to this big box. With these reinterpretation, we are able to continue using the same loop structures as in the classical version with almost no changes.

Parallel updates (Most of) PARFLOW's stencils require information from adjacent face neighbours only. Whenever this data lies on a foreign process, the code must provide additional storage so that data transfers could take place cleanly. As in many codes based on finite difference discretizations, PARFLOW meets this requirement by storing an additional strip of degrees of freedom at the boundary of each process, see Figure 41(a). If a locally (2:1 balanced) refined mesh is enforced, we need to provide additional storage for the situation in which a parallel update of information occurs between two or more different size subgrids, see Figure 41(b).

Implementation details The `Grid` dimensions are read from the input file, a suitable `p4est` object is created and carefully chosen subgrids are attached to its leaves. In order to generalize the code to the situation pictured in Figure 41(b), we exploit the information contained in a dedicated `p4est` structure (`p4est_mesh.h`). Such structure encodes neighbouring information for a 2:1 balanced mesh that we can query to decide when a inner ghost cell should be allocated. Those inner ghost subgrids are always local to the current `MPI` rank.



(a) (b)

Figure 41: Left (a), default communication pattern for a stencil propagating information in the x–coordinate direction. The values that need to be exchanged are displayed in red and blue dots and the ghost layer where these are written to, is enclosed in dotted lines. Right (b), schematic representation of the approach taken to propagate numerical information for a locally refined mesh in ParFlow*. We impose a 2:1 balance condition on the mesh such that this is the only relevant case to treat. A coarse subgrid requires to share information with two neighbouring finer subgrids. We create additional ghost subgrids internal to the coarse one, in the figure displayed in green, which in view of the 2:1 balance condition match the size of the neighbouring ones. We conveniently call them "inner ghost subgrids". The arrows clarify the flow of information, MPI denotes communication, I interpolation and R restriction.*

Reinterpretation of ParFlow's native finite difference scheme to account for local refinement In [**?**] it was observed that in order to obtain a globally second order discretization of the Laplacian, one may use discretizations that are only first order accurate at locally refined points but reduce to second order accurate when applied at locally uniform points Such approach has been employed for example in the works [**?**, **?**] to develop solvers for the incompressible Euler and variable Poisson equations on locally refined grids respectively.

Employing finite differences on a (2:1 balanced) locally refined mesh will require values of the function to differentiate at locations where such values are not available. Using linear interpolation to derive the required missing values will add new terms in the corresponding Taylor analysis that may degrade the accuracy of the approximation compared to the uniform case. Nevertheless, as shown in [**?**], its is possible to weight tweak the approximations to regain the accuracy one should obtain if no interpolation was employed.

For example, in a two dimensional mesh and in view of the 2:1 balance condition, one of the cases to handle is displayed in Fig. 42. Here, a first attempt will be to approximate the Laplacian $P_{xx} + P_{yy}$ at the location P_3 by $L_x + L_y$ where as defined as

$$L_x := \left(\frac{\hat{P} - P_3}{3h/4} + \frac{P_4 - P_3}{h/2} \right) \frac{2}{3h/4 + h/2}, \tag{3a}$$

$$L_y := \left(\frac{\tilde{P} - P_3}{3h/4} + \frac{P_5 - P_3}{h/2} \right) \frac{2}{3h/4 + h/2}, \tag{3b}$$

where $\hat{P} := \frac{3}{4}P_2 + \frac{1}{4}P_0$ and $\tilde{P} := \frac{1}{4}P_0 + \frac{3}{4}P_1$. The corresponding Taylor analysis shows that

$$L_x \approx P_{xx} + \frac{1}{5}P_{yy} + O(h) \qquad L_y \approx P_{yy} + \frac{1}{5}P_{xx} + O(h) \tag{4}$$

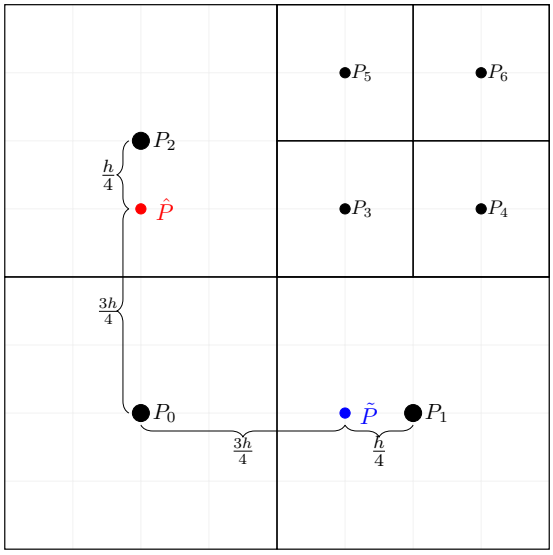


Figure 42: We wish to approximate the Laplacian at P_3 using a standard 5 point stencil using the values at $P_3, \hat{P}, P_4, \tilde{P}$ and P_5. The function values at $\hat{P}$ and $\tilde{P}$ are obtained by linear interpolation using the available data.

Hence, correcting our ansatz to $\frac{5}{6}L_x + \frac{5}{6}L_y$ gives a first order accurate approximation of the Laplacian at the point P_3. Other cases can be handled in a similar way, yielding different constants to ensure the first order accuracy of the approximations. We are currently identifying all relevant cases in a three dimensional setting and adapting them to the form of Richards equation implemented in PARFLOW.

Conclusion for PARFLOW AMR As shown in Fig. 38, we were not able to complete all the sub-tasks required to fully port the code to AMR. We are able to correctly manage the different grid levels as well as the data exchange between the levels takin advantage of `p4est` and PARFLOW structures. On the other hand, the solvers and in particular the preconditioners are not fully adapted to the AMR grid. This task turned out to be much more complex than expected.

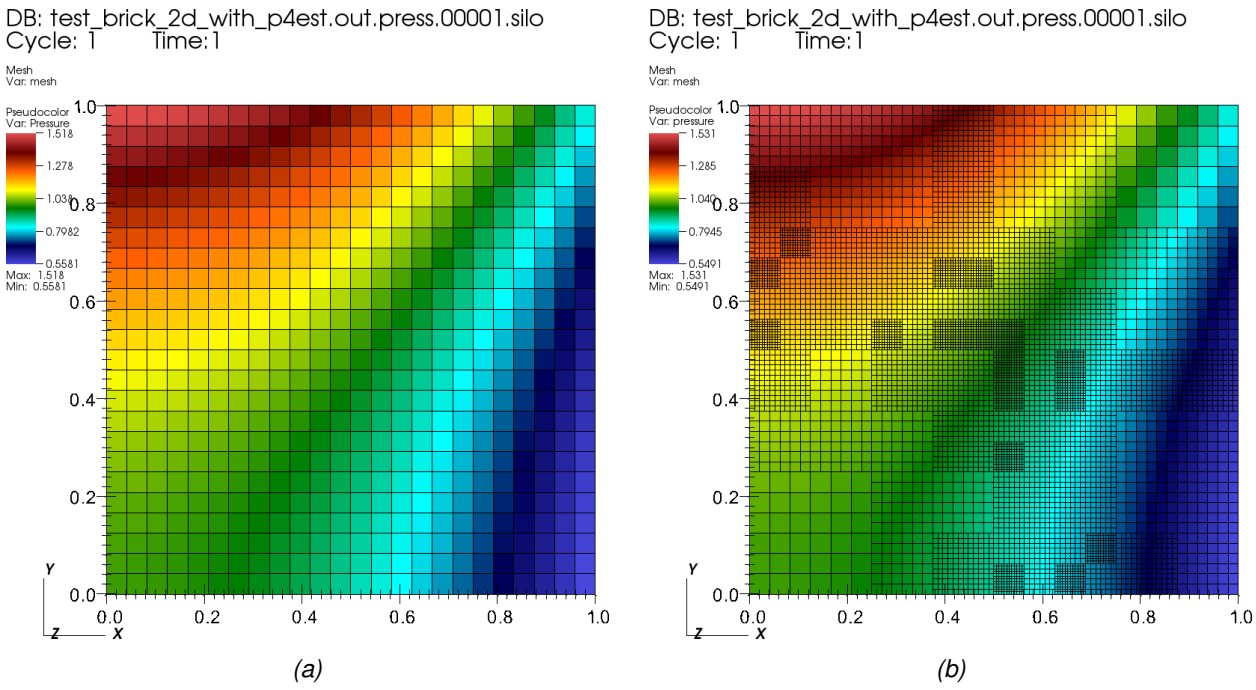
With the current approach we are able to run simple test cases, for example, by appropriately choosing the parameters appearing in the Richard's equation we can reduce it to a Poisson equation and thus, use PARFLOW to solve the later. In Fig. 43 we display an example of a numerical solution obtained in this way. Large-scale simulations capable of taking advantage of the AMR could not therefore be carried out.

The development version of PARFLOW + `p4est` is up to date with the official version but cannot be brought back into the official version as it is not fully operational. As mentioned in D2.2, the developments carried out for AMR can be exploited to allow advanced MPI + OPENMP hybridisation. As the work on AMR was not completed, we did not work on this second aspect. It is nevertheless a potential outcome and extension of the work carried out here in EOCOE-II.

A detailed technical report was written to enable the work to be resumed and the various issues raised by the AMR to be understood and resolved. The latest developments for AMR are available as open-source on the PARFLOW GitHub project [23] via the dedicated branch called `adaptive`.

8.3 Work progress on task 2.5.2

Subtask 2.5.2 is dedicated to the SHEMAT-SUITE application. The task goal was to improve code performance for ensemble runs (see WP5) by integrating PDI and the Parallel Data Assimilation Framework (PDAF). Ensemble runs will be used for stochastic parameter estimation and uncertainty quantification



(a) *(b)*

Figure 43: Approximate solution of the Poisson equation on the unit square. The right hand side and boundary conditions are selected such that $p(x, y) = \cos(x)\cosh(y)$ is the analytical solution. Left, we show the solution computed by PARFLOW *on uniform mesh with 24 cells peer coordinate direction. Right, we display* PARFLOW*'s computed solution on a randomly refined mesh, allowing up to three levels of refinement with respect to the uniform case.*

within geothermal reservoirs. Berenice Vallier was hired for 12 months to complete the PDI and PDAF activities in WP2 and WP4.

The work plan summary is presented in Fig. 44. The work on the PDI integration began in November 2019 (M11). Only PDAF has not been integrated due to lack of time and resources.

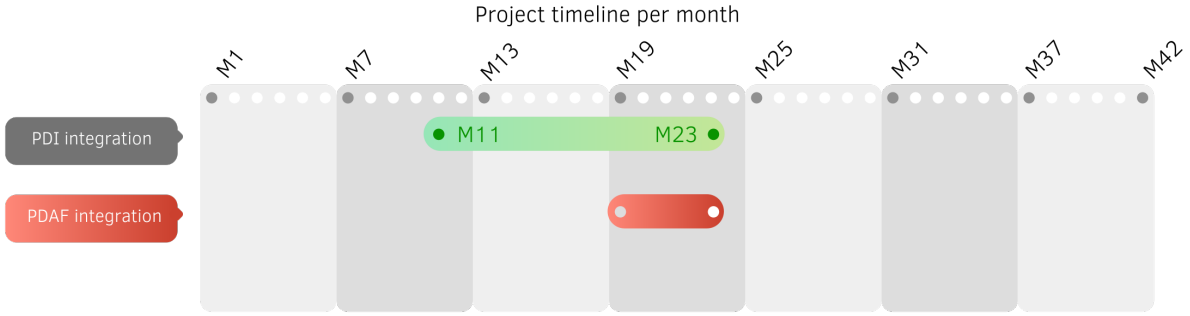


Figure 44: Breakdown (simplified Gantt chart) of the task 2.5.2 for SHEMAT-SUITE*.*

Many I/O processes are associated to scientific simulations such as in the SHEMAT-SUITE software. Some are enhanced before the actual simulation, such as the initialization of data. Others occur during the simulation, such as the intermediate or final checkpoints to account for execution failures. Finally, after simulation, post-processing, diagnostics, storage to the disk and visualization of the results are additional I/O processes associated with SHEMAT-SUITE. All these I/O processes can be managed individually thanks to libraries like HDF5, MPI I/O.

By integrating PDI into SHEMAT-SUITE, we aim to minimize the changes required in SHEMAT-SUITE along the I/O processes. The main goals are to: (i) increase efficiency of the I/O processes; (ii) decouple I/O from the simulation; (iii) facilitate the usage of different I/O libraries; (iv) integrate PDI into SHEMAT-SUITE will enable to make use of functionalities like in-situ visualization or big data and ensemble handling in the future.

First of all, the preliminary work of the PDI integration has been the installation of PDI on the RWTH cluster CLAIX-18 and the realization of the tutorial explained on the official website of PDI [6]. Thanks to the help of the developers of PDI, Julien Bigot and Karol Sierocinski, the installation and the tutorial have been conducted successfully. A documentation of the preliminary steps as well as the integration is written in parallel of the task 2.5.2 as guidelines for the future users of PDI in SHEMAT-SUITE.

The implementation of PDI mainly focuses on a declarative API, the few changes required in SHEMAT-SUITE code itself. Indeed, we define a unique YAML file called specification tree supporting the calling of libraries. Each library call described in the specification tree relies on: (i) Data storages for data transfer referring to the list of parameters allowing this transfer. (ii) Event subsystem for control transfer called by example when a new parameter is made available in the store.

The typical structure of the specification tree is described in Fig. 45. The `data` and `metadata` sections specify the type of the data in buffers exposed by the application. For `metadata`, PDI keeps a copy while it only keeps references for `data`. The `plugin` section specifies the list of plugins to load and their configuration.

```
pdi:
  metadata:
  rank: int
  width: int
  height: int
  data:
  main_field: {type:, size:[$width, $height]}
  plugins:
  decl_hdf5:
        - file: output_${rank}.h5
        write: [rank, main_field ]
```

Figure 45: Structure of a typical specification tree in YAML *format.*

The code annotation API is the main interface to use in the SHEMAT-SUITE source code. The initialization of PDI is called by the `PDI_init` function, the configuration file is parsed and the `decl'H5` plugin is loaded. This plugin initialization function is called and analyses its part of the configuration to identify the events to which it should react. For the finalization, the `PDI_finalize()` function is releasing all resources at the end of the simulation. Exposing and reclaiming data to PDI are called by the `PDI_share()`, and `PDI_reclaim()` or `PDI_expose()`. The `PDI_event()` function is a PDI notification in a specific location in SHEMAT-SUITE, the plugins are then reacting to the event. For integrating PDI into SHEMAT-SUITE, the subroutines dealing with I/O processes have been identified in SHEMAT-SUITE. More details about the SHEMAT-SUITE can be obtained in the references works of [26] and [27]. Fig. 46 lists the SHEMAT subroutines related to I/O processes. By example, the main SHEMAT-SUITE subroutine containing the functions for reading the input files is called `read_model`. All the keywords referencing the readable parameters are included as metadata and the data are the arrays such as the temperature, pressure, head and concentration. The reading of the keywords should be done in the right order because some parameters depend on previous ones.

Different I/O processes	SHEMAT subroutines related to I/O processes
reading the input files	- read_array - read_bc - read_check - read_control - read_data - read_model - read_property - read_split - read_time
writing the output files	- write_data - write_dense_3d - write_logs - write_monitor - write_outt - write_status_log - write_tecdiff - write_tecplot - write_tecplots - write_text - write_vtk
reading/writing/opening/closing/modify files of hdf5 format	- close_hdf5 - closeopen_hdf5 - mod_hdf5_vars - mod_input_file_parser_hdf5 - open_hdf5 - read_hdf5 - read_hdf5_int - test_hdf5 - write_all_hdf5

Figure 46: List of I/O related SHEMAT-SUITE *f90-subroutines for the forward code.*

The same process of identifying data, metadata and writing the configuration tree is done for other files covering the reading or opening of input or output. Fig. 47 shows an example of configuration tree written in a YAML file for a SHEMAT subroutines related to I/O process. The reading or writing is triggered in the SHEMAT-SUITE source by events, the plugins are then reacting to the events. Subsequently, the output routines of SHEMAT-SUITE will be replaced by PDI calls.

This part of the PDI-integration has been completed and test models have been defined. We begin with a simple test case and will increase the test model complexity, i.e. the amount and complexity of I/O data, successively. First, it will be a simple steady-state 2D case with only one active model state, i.e. variable input array. The tested output format will be HDF5. If this first test is successful, we will add other types of input arrays to the test model and add other input parameters, e.g. by switching from a stationary to a transient simulation. In a next step, the PDI- integration needs to be tested with a 3D model. Finally, the PDI integration will be extended to I/O processes related to advanced SHEMAT functionalities, such as inversion.

8.4 Work progress on task 2.5.3

Task 2.5.3 originally concerned the development of a common base for the PARFLOW and SHEMAT-SUITE codes, bringing together CPU/GPU multi-platform parallelization and AMR capabilities as partly described in the proposal and the D2.1.

The platform called EXATERR was to be based on Kokkos. This idea, although ideal for questions of stability and pooling of resources, proves to be far too ambitious in comparison to the needs solicited by tasks 2.5.1 and 2.5.2. The integration of AMR into PARFLOW and GPU porting will take all the resources allocated to these activities.

```
pdi:
  metadata: # small values for which PDI keeps a copy
    i:    int
    j:    int
    filename:    character
    line:    character
    filename_data:    character
  data: # values for which PDI does not keep a copy
    isplit: { type: array , subtype: int, size: ['$nsplit'] }

  plugins:
    test:
    decl_hdf5:
      - file: ${filename}.h5
        on_event: event_1
        read:
          nsplit:
            dataset: '$filename_data'
      - file: ${line(${i}:${j})}.h5
        on_event: event_2
        read:
          nsplit:
            dataset: '$filename_data'
```

Figure 47: Example of configuration tree in yaml file for a SHEMAT-SUITE *subroutine.*

Some of the work done for the GPU porting is getting closer to the goals of this task. Indeed, the Kokkos backend implementation to the PARFLOW domain specific interface (eDSL) is the preliminary work for this subtask.

The needs of such a platform are also questionable and will not call into question the optimisation of PARFLOW for Exascale. The SHEMAT-SUITE code could have benefited from this for its optimisation but these are not the objectives of EOCOE-II for this code.

9 Task 2.6 - Fusion code optimisation

9.1 Task overview

Task leader : CEA-IRFM

Participants: CEA-IRFM, FAU, INRIA, MPG

The goal of the Fusion Scientific Challenge is to bridge the gap between gyrokinetic core transport modelling and edge plasma physics for reliable predictions of fusion performance, which will require a number of numerical and physics bottlenecks to be overcome. The Fusion Scientific Challenge is composed of a single flahship code GYSELA and satellite codes TOKAM3X and SOLEDGE2D. Only GYSELA [28] is concerned by the WP2. The objective is to develop a new numerical tool to address the core-edge issue, which will consist of refactoring and rewriting the flagship gyrokinetic code GYSELA [29], targeting the disruptive use of billions of computing cores expected in exascale-class supercomputers. The new code after refactoring will be named GYSELAX. As already mentioned in previous reports, the CEA-IRFM team has lost one of the pillar developers of the GYSELA code, with an expertise on computational science and high-level parallelism which cannot be replaced by nonpermanent staff. As a consequence, the rewriting of the code has been abandoned and replaced by its complete refactoring with enhanced modularity targeting exascale supercomputers.

The challenges have again shown that this work would not have been possible without close collaboration between physicists, mathematicians and computer scientists. The work in GYSELA in the context of the

WP2 aims at modernizing and adapting the code for forthcoming super-computers including first pre-exascale prototypes and demonstrators (see simplified Gantt chart Fig. 48) while the associated physical advances are detailed in WP1.

The work concerning Task 2.6 is divided into 2 subtasks :

- Subtask 2.6.1 - Prototype of GYSELAX
- Subtask 2.6.2 - Advanced GYSELAX

Subtask 2.6.1 was dedicated to the prospection of the best solutions in terms of performance, readability and longevity for the GYSELAX prototype. The work on the C++ prototypes had shown that it was not viable to consider rewriting the GYSELA code in C++ within the timeframe of EoCoE-II. Task 2.6.1 was presented as completed in the mid-term report. However, feedback on subtask 2.6.2 during the second part of the project and its extension by 6 months has encouraged us to pursue investigations. We present in section 9.3 the new Gyselalib++ library based on the new library DDC (Discret Data and Computation) which should be two essential building blocks for a complete rewriting in modern C++ of GYSELAX in the future (blue area in Fig. 48).

Subtask 2.6.2 is entirely devoted to the computational and numerical improvement of the GYSELA code with the leitmotiv of preparing the exascale simulations of the future. The addition of more modularity via the integration of PDI is the subject of task 2.6.2(i) and is detailed in section 9.4.1. The improvement of the numerical schemes aiming at more and more realistic plasma turbulence simulations was the subject of tasks (ii) and (iii) which are respectively detailed in sections 9.4.2 and 9.4.3. The feedback from porting GYSELA to pre-exascale architectures and more precisely to Fujitsu-A64FX architectures (task (iv)) is detailed in section 9.4.4.

All the results detailed in this section have been presented by Virginie Grandgirard at the Platform for Advanced Scientific Computing (PASC) Conference remotely in June 2021 [30] and at Bâle/Switzerland in June 2022 [31].

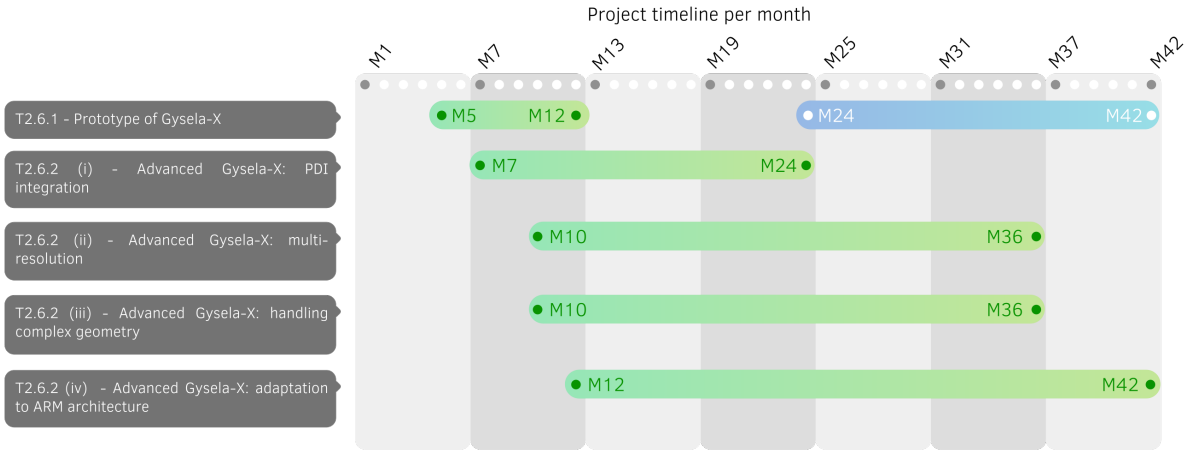


Figure 48: Breakdown (simplified Gantt chart) of the task 2.6 for GYSELAX*. The blue area in task 2.6.1 corresponds to the rewriting of* GYSELA *using modern C++ started after the preliminary phase at the beginning of the project. This extra work has been started thanks to additional resources outside the* EOCOE-II *project but is clearly related to the initial task of* EOCOE-II*.*

9.2 Flagship code GYSELA

GYSELA is a 5D global full-f gyrokinetic code developed at the IRFM/CEA for 20 years to simulate electrostatic plasma turbulence and transport in the core of Tokamak devices (GYSELA website). The code is

written in FORTRAN 90 with some I/O routines in C. The time evolution of the full distribution function of each ion species (major species as e.g. Deuterium + one minor impurity) and electron is governed by a 5D non-linear gyrokinetic Vlasov equation self-consistently coupled to a 3D Poisson equation.

Fig. 49 gives a schematic view.

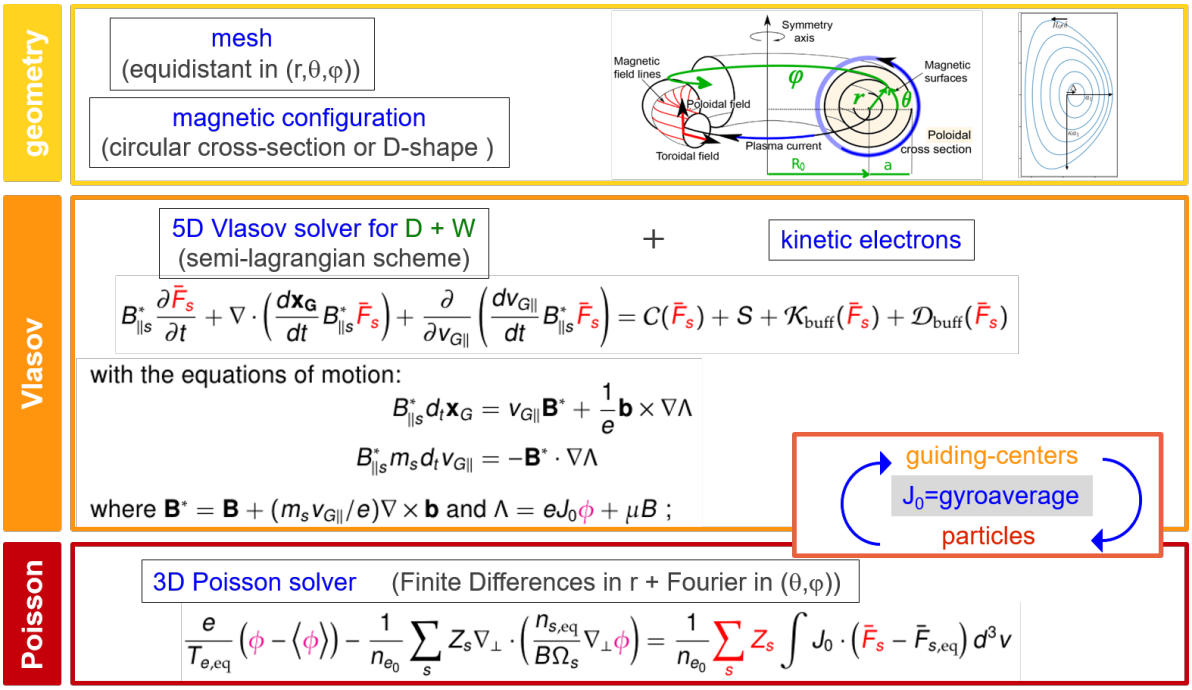


Figure 49: Schematic view of the equations handle in the GYSELAX *code.*

One peculiarity of the GYSELA code is to be based on a semi-Lagrangian method which is a mix between Particle-In-Cell (PIC) and Eulerian approaches. In this approach, the phase-space mesh grid is kept fixed in time (Eulerian method) and the Vlasov equation is integrated along the trajectories (Lagrangian method) using the invariance of the distribution function along the trajectories. From a numerical point of view, its strength is to take advantages of both methods, namely limited numerical dissipation with limited numerical noise, leading to good properties of local conservation laws for charge density, energy and toroidal momentum. From a parallelisation point of view, its Eulerian character is an advantage because there is no problem of load-balancing. On the other hand, the need to use an interpolation which is non-local is clearly a disadvantage leading to a code much more difficult to parallelize than a PIC code. However, extensive efforts of parallelization over the past 10 years have enabled GYSELA to run efficiently on more than 100 000 cores on current standard architectures (Intel Broadwell, Intel Haswell, Intel Skylake (SKL), Intel Knights Landing (KNL) and AMD EPYC (Rome and Milan) architectures). Our CPU time consumption, which has been growing exponentially since 2001, has been saturated for a few years due to the limitation of available resources and is currently reaching 150 million hours per year. We already know that simulations of the turbulence of ITER plasmas, which are much larger than the plasmas we currently simulate, will require exascale HPC capacities.

During EOCOE-II, the GYSELA code progressively evolves towards an upgraded version, targeting exascale supercomputers and solving electromagnetic turbulence from the core to the far edge region in ITER-relevant magnetic geometry.

During this project, we intensively used several Petascale European supercomputers for which the GYSELA team succeed in obtaining CPU hours every year via GENCI and Eurofusion projects:

- Occigen supercomputer at CINES, the National Computer Centre for Higher Education, in Montpel-

lier/France where we used both Broadwell and Haswell partitions.

- Joliot-Curie supercomputer at TGCC (Très Grand Centre de calcul du CEA) in Bruyère-le-Chatel/France where we used the three partitions Irene-SKL, Irene-KNL and Irene-Rome.
- Marconi supercomputer at CINECA italian HPC center where we used the SKL partition dedicated to fusion European community.

In addition to that we had access to two pre-exascale machines in top 20 (June 2022 ranking `https://top500.org/lists/top500/2022/06/`) of the world supercomputers.

- Supercomputer Fugaku (A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu) - 7 630 848 cores - installed in 2020 at RIKEN Centre for Computational Science (RCCS) in Japan.
- CEA-HF - (BullSequana XH2000, AMD EPYC Milan 7763 64C 2.45GHz, Atos BXI V2, Atos) - 810 240 cores - installed in 2022 at CEA TERA center in France.

An important member has left at the beginning of the project, Guillaume Latu. He was HPC engineer coordinating and actively working on the refactoring of GYSELA. He was as well one of the main architect of the code structure with a long experience dealing with GYSELA performance issue. Therefore, the leave of Guillaume Latu has significantly impacted the project and particularly the WP2 tasks which he was initially in charge of supervising. Virginie Grandgirard was in charge of supervising Task 2.6 following his departure. Table 27 shows the other team members of GYSELA involved in this workpackage.

People	Position	Role	Period
Virginie Grandgirard	Researcher, CEA-IRFM	Numerical Analyst, GYSELA main developer	M1-M42
Chantal Passeron	Developer, CEA-IRFM	Support to GYSELA development	M1-M42
Julien Bigot	Researcher, CEA-MdlS	Computer Scientist, expert in HPC and I/O	M1-M42
Michel Mehrenberger	Researcher, AMU	Applied Maths., GYSELA developer	M1-M42
Emily Bourne	PhD student at CEA-IRFM and AMU	Computer Scientist, handling complex geometry and mesh refinement	M10-M42
Kevin Obrejan	Post-doc, CEA-IRFM	Physicist handling complex geometry	M12-M42
Dorian Midou	HPC Research-Engineer at CEA-IRFM	external expert, optimisation for ARM architecture	M12-M28
Yanick Sarazin	Researcher, CEA-IRFM	Physicist, coordination and reporting	M1-M42

Table 27: Team Members for GYSELA *within* EOCOE-II.

9.3 Work progress on task 2.6.1

As already discussed the rewriting of the code in C++, which was the initial objective for the GYSELA code in EOCOE-II, was first abandoned after M6. The strategy to improve the existing FORTRAN version of the code that was applied throughout the project paid off as it allowed us to optimise the code on over 500k AMD cores. Historically, the strategy adopted by the Gysela developers was to focus on specific architectures (mostly Intel processors) and accepting to be less efficient on the few others. However,

learning from the feedback of porting the code to ARM-based architectures over the last two years of EoCoE-II and more recently to GPU, this strategy seems no longer viable in the long term given the increasing heterogeneity of emerging computing technologies. In both cases it is clearly identified that obtaining performance will require a dedicated rewriting of most of the computing kernels. The fact that the code is written in FORTRAN clearly appeared as an additional difficulty. In addition, the studies of semi-Lagrangian schemes on non-equidistant meshes, performed on the 2D application VOICE (see Task 2.6.2 and Task 1.5.2-2), shown that this will be a very good option for a more accurate treatment of realistic core-edge gyrokinetic simulations but its implementation in GYSELAX will require a major rewriting of the code.

Following this feedback, the GYSELA team decided to start defining a strategy for rewriting the code (See schematic view in Fig. 50). This rewriting will be done in C++ and will rely on modern programming models. The C++ version of GYSELAX will be based on the DDC [1] mesh management library, the development of which was recently initiated at the Maison de la Simulation (CEA-Saclay). This library provides tools to facilitate both the writing and the parallelization of algorithms based on different types of discretizations (meshes, splines, etc.), in particular basic meshes, iterators or even arrays associated with these meshes. The writing of a collection of C++ components required for semi-Lagrangian codes has started recently. A first proof of principle of the using of this new library GYSELALIBXX (`https://github.com/gyselax/gyselalibxx`) is under development on a 2D prototype VOICE++. DDC is de-

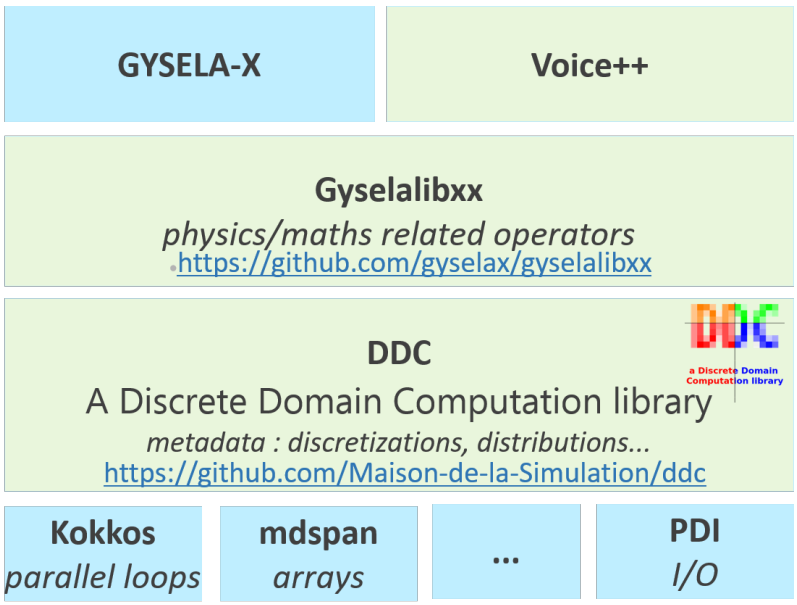


Figure 50: Schematic view of the rewriting strategy of the code GYSELAX.

signed to be coupled with modern perfromance-portable backends sucha as Kokkos. This will enable GYSELA to run efficiently on many architectures (including GPU accelerators), and provide a good balance between performance, portability and maintainability. The collaboration with Yuuichi Asahi continued throughout the EoCoE-II project, to evaluate the performance portable implementation of a kinetic plasma simulation code with C++ parallel algorithm to run across multiple CPUs and GPUs. In a paper recently submitted [32], Y. Asahi evaluate the capability of C++ parallel algorithm ("stdpar") as a performance portable framework in comparison with an approach based on libraries with higher level abstraction like Kokkos and an approach based on directives such as OPENMP. Relying on the language standard parallelism "stdpar" and language standard high dimensional array support "mdspan", we demonstrate that a performance portable implementation is possible without harming the readability and productivity. We obtain a good entire performance of mini-applications in the range of 20 % to the Kokkos version on Icelake, NVIDIA V100 and NVIDIA A100. The language standard parallelism can be a good candidate to

develop a performance portable and productive code targeting the exascale era platform, assuming this approach will be available on AMD and/or Intel GPUs in the future.

9.4 Work progress on task 2.6.2

Four main activities constitute the backbone of the GYSELAX development, some being backed by the outcomes of task T2.6.1. As described in the previous section, it was decided that the GYSELA code would stay the pillar of the new code GYSELAX by focusing our efforts on refactoring the code to add more modularity.

9.4.1 PDI integration and enhanced modularity developments

All the work done work in this task has been done in strong collaboration with WP3 under the supervising of Julien Bigot (MdlS/France) and Bruno Rafin (INRIA-Grenoble/France). All the PDI implementation was done by Yacine Ould-Ruis (engineer hired for 2 years in EOCOE-II at INRIA-Grenoble/France). The prospective studies on in-situ diagnostics have been done with the help of Antoine Lavandier (6 month EOCOE-II internship at MdlS/France) and in the framework of Amal Gueroudji's PhD (MdlS/France).

PDI for handling huge amount of 5D data to be saved The GYSELA code produces very large amounts of data. A typical 5D mesh contains several hundreds of billions of points, which leads to 5D distribution functions of the order of 2 TB to be followed at each time iteration. Knowing that a simulation can represent several tens of chained runs of more than 10 000 iterations, it is not conceivable to store the temporal evolution of these distribution functions. They are only saved at the end of each run in temporary files to allow the restart (checkpoint-restart). Currently, each MPI process in GYSELA saves the part of the domain it processes in a different file. This approach has recently shown its limits for large-scale tests: file systems do not support the simultaneous writing of more than 10000 files. This problem of efficient writing to disk on exascale machines should be a research topic in its own right. This reinforced our idea that the coupling to PDI (Parallel-data Interface) [6], to deport the Input/Output (I/O) management to specialists of the topic, was a very good strategy.

At M18, the restart files were saved in HDF5 format via the PDI API.

Since M18, the sequential HDF5 writing commonly used has been compared to a parallel HDF5 writing with the objective to reduce both the number of inodes and the number of files written simultaneously, both becoming already a bottleneck for the largest simulations performed with GYSELA. One of the big advantages of coupling GYSELA to PDI is that switching from one to the other is now very easy: you only need to modify 1 line of a YAML configuration file without having to modify or recompile the code. One of the difficulties encountered during performance tests is that the results are highly dependent on the use of the file system by others, which leads to large error bars. This explains the large error bars that can be seen in the Fig. 51 for a weak scaling performed on Irene-SKL partition. All presented cases in Fig. 51 correspond to an average on 10 simulations. On this figure, the cases of writing with HDF5 sequential without PDI (in blue) and with PDI (in green) correspond to a writing of 2 GB of data per MPI process. In the case of the use of HDF5 in sequential mode, the fact that even taking into account the error bars, the writing is 2 to 3 times faster when using the PDI API remains a mystery and investigations are continuing. These sequential HDF5 writing are compared to parallel HDF5 writing (orange and red results) where one file per μ values is saved, reducing the number of files by 8 in the present case (because $N_\mu = 8$). As expected HDF5 parallel writing is slower than sequential writing. This still need to be tested but this trend could be reversed for a larger number of MPI processes. Anyway, the writing via parallel HDF5 should become the standard writing mode in GYSELA for exascale simulations. Indeed, the largest GYSELA simulations are already at the limit of the number of files that can be written simultaneously on

the file system without crash. Therefore, work will pursue to try to optimize the parallel writing. First tests are promising. As shown in the Fig. 51, taking care of the data contiguity (red results) can reduce the CPU time by a factor 3. Again, the advantage of using the PDI API is that the choice of the data storage structure only needs to be described in the YAML configuration file.

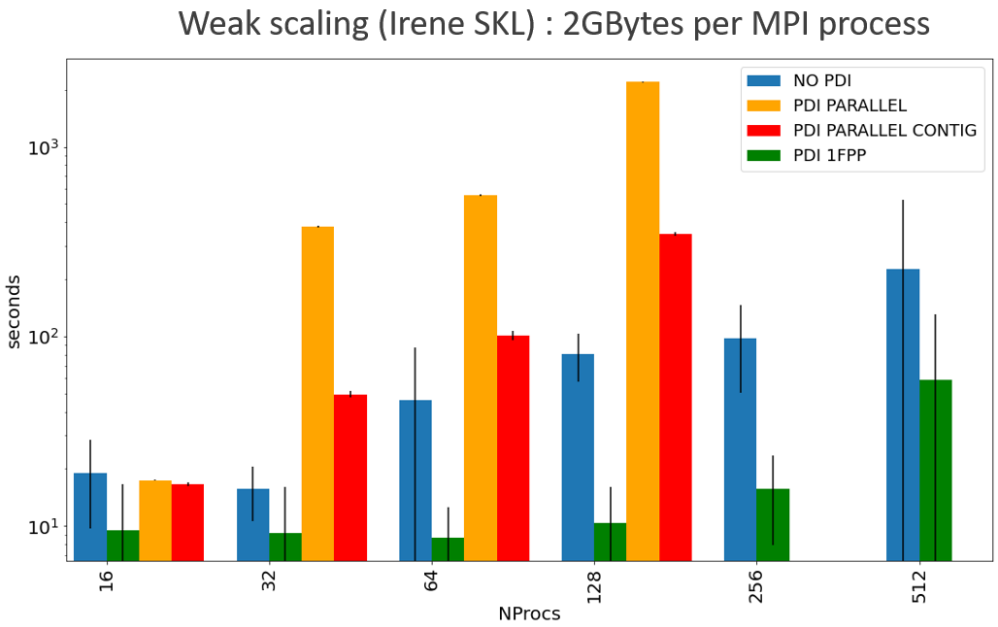


Figure 51: CPU times for HDF5 restart file writing versus the number of processors. Weak scaling performed from 16 processors to 512 processors on Irene-SKL partition. HDF5 sequential writing (without PDI *in blue and with* PDI *in green) are compared to HDF5 parallel writing (with* PDI *in orange and with* PDI *by taking into account data contiguity in red).*

PDI for more modularity in the diagnostic treatment In the end, out of the Petabytes of data manipulated during a GYSELA simulation, only a few Terabytes are saved due to storage capacity limits. This data reduction is based on saving at fixed time steps a number of mainly 3D fluid quantities. These diagnostics are directly integrated in the code and not executed as post-processing because the amount of data to write before reduction would be unreasonable and make the code I/O bound by a large factor. Knowing that there is a growing gap between CPU performance and I/O bandwidth on large-scale systems, this post-hoc approach is already very constraining and will become even more so.

The objective of this sub-task was to add more modularity in the diagnostic computing via PDI. This activity started at M18. Since then all the diagnostics have been refactored to be saved via PDI. PDI is now fully implemented in GYSELA (i.e restart files + all diagnostics).

A first proof of principle of in-situ diagnostics has recently been realized in GYSELA via Deisa [33] (Dask-enabled in situ analytics) a tool whose development is carried out at MdlS in collaboration with the INRIA Datamove team. The 5D distribution function was exposed during the simulation via PDI and a principal component analysis (PCA) written as a Python script based on Dask was performed. The use of Dask [34] allows the script to be run in parallel and Deisa avoids the need to write the 5D data to disk, which would be prohibitively expensive. This in-situ approach opens the field of possibilities. One of the first objectives will be to embed in-situ diagnostics for automatic detection of anomalies or rare events. For simulations that may run for several hours on several hundred thousand cores, automatic anomaly detection is a crucial issue to limit the number of CPU hours consumed unnecessarily. A data backup optimised according to the detection of rare events could also allow a significant gain in terms of storage.

9.4.2 Multi-resolution

The large temperature variation – typically by 2 orders of magnitudes – from the far edge to the very core of tokamak plasmas requires refined meshes. Multi-resolution and/or multi-patch approaches then reveal mandatory to avoid wasting large amounts of CPU time and memory resources. In the context of reduced manpower, we had to abandon the multi-patch strategy initially proposed because it would have required an almost complete rewriting of the code. Therefore, it has been decided to treat this intrinsic difficulty by using non-equidistant splines.

As described in Task T1.5.2-2, the 2D mini-application VOICE was used as numerical testbed to simulate plasma sheath. As a reminder, VOICE is a 1D-1V kinetic FORTRAN code based on the same numerical scheme than the GYSELA code, most of its modules having been extracted from GYSELA. The implementation of a semi-Lagrangian scheme for non-equidistant mesh in VOICE was part of Emily Bourne's PhD while the exploiting for plasma sheath understanding was part of Yann Munschy PhD. All this work is the subject of two papers (one submitted very recently by Emily Bourne [35] and the other in the process of being finalized by Yann Munschy).

At M18, the non-equidistant spline module developed in the SELALIB numerical library (collaboration CEA-IRFM Cadarache and MPG-IPP Garching) had been coupled to VOICE.

Since M18, the non-equidistant semi-Lagrangian as been implemented and successfully validated in VOICE leading to very good conservation properties (see Bourne's paper [35]) . These very encouraging results reinforce the idea that non-equidistant splines are a viable solution to improve core-edge coupling in the GYSELA code. Even in 2D, the CPU time required to perform relevant physical simulation was long. A first acceleration of the VOICE code was performed with the adding of OPENMP parallelism. The GYSELA team had the opportunity to take part in the NVIDIA-GPU Hackathon at IDRIS in May 2021. This hackathon was an opportunity to learn more about GPU parallelization and to port the first VOICE modules to GPU via the use of OPENACC. Since then, VOICE has been efficiently be ported and regurlarly run on GPU systems.

Fig. 52 shows the time taken for each advection step of the simulation for different parallelisation methods and spline degrees, for uniform and non-uniform splines. The OPENMP tests were run on a SkyLake processor with 192 GB of RAM. The OPENACC tests were run on a NVIDIA V100 GPU with 380GB of RAM. On CPU, we note that non-uniform splines are significantly more costly than uniform splines as expected. This cost (simulation time) increases with the spline degree. However we can also see that the scheme scales well when it is parallelised using OPENMP. This can allow the cost to be attenuated somewhat. We also see that GPUs present themselves as the natural solution to this problem. On GPU, the simulation time to simulate large simulations using the non-uniform splines of degree 7 (NU-7 in Fig. 52) is just slightly more costly than the uniform splines of degree 1 (U-1 in Fig. 52). Effective parallelisation through the use of GPUs leads to non-uniform simulations running 5.5 times faster than uniform simulations providing equivalent results.

9.4.3 Complex Geometry

The objective of this task was to implement a more realistic magnetic configuration based on a Culham equilibrium to be able to address ITER relevant D-shape magnetic geometries. So far, GYSELA could only handle circular cross-sections. This major task required rewriting most of the operators.

At M18 the magnetic equilibrium initialization had been modularized to facilitate the implementation of the new Culham equilibrium. Let us introduce some useful notations for the following. We consider a set of coordinates labelled $\{x^i\}$, the metric tensor $\{g_{ij}\}$ is the product of the transposed Jacobian matrix J^T and the Jacobian matrix J, *i.e* $\{g_{ij}\} = J^T J$. For a set of cartesian coordinates X^i, the elements J_{ij} of the Jacobian matrix are defined as $J_{ij} = \partial_{x^j} X^i$. Let g represents the determinant of the met-

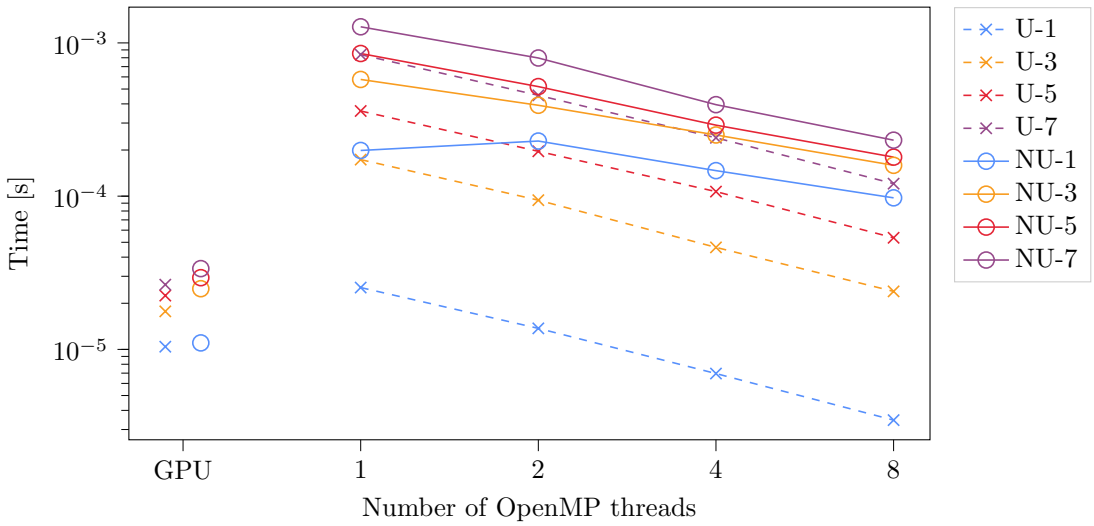


Figure 52: The time required to run an advection step for a grid with 2048 grid points for uniform (x) and non-uniform (○) splines of various degrees on code accelerated with OpenMP *for multi-threading or* OpenACC *for GPUs. Tests were run at the Centre de Calcul Intensif d'Aix Marseille.*

ric tensor (*i.e* $g = \det\{g_{ij}\}$), then the Jacobian in space J_{x} is defined as $J_{\mathrm{x}} = \sqrt{g}$ and is equal to $J_{\mathrm{x}} = \left[\left(\boldsymbol{\nabla} x^1 \times \boldsymbol{\nabla} x^2\right) \cdot \boldsymbol{\nabla} x^3\right]^{-1}$, *i.e* the volume element is $J_{\mathrm{x}}\, \mathrm{d}^3\mathbf{x}$. The contravariant metric tensor $\{g^{ij}\}$ is the inverse of the metric tensor $\{g_{ij}\}$. The covariant tensor and its inverse, namely the contravariant metric tensor, associated to Culham transformation had been analytically derived (see Gysela [28] documentation).

Since M18, they are now completely implemented in a new module dedicated to Culham equilibrium initialization and all the steps identified have now been completed. Namely,

- The Vlasov equations were expressed in terms of contravariant components to be able to switch from the circular geometry to the D-shape geometry via the computing of the contravariant metric tensor components.
- The gyroaverage operator was completely rewritten from scratch.
- The quasi-neutrality equation was currently solved by projecting in Fourier space in the poloidal direction and by using finite differences in the radial direction. This strategy is no more applicable in the case of non-circular geometry. The Poisson solver has been extracted to be replaced by a spline-based Poisson solver developed for this purpose in task T1.5.1-2.
- All the diagnostics have been modified. The flux surface diagnostics in the code have been generalised for both geometry via the computing of the Jacobian in space. The Python post-processing diagnostics have been updated to be more versatile (able to handle 2D and 3D datasets and compatible with non-circular geometry) by Baptiste Legouix during his 6 month internship at CEA-IRFM. The cartesian-to-toroidal numerical transformation is now performed via the using high-performance tools like Pyccel[5] (Python to FORTRAN translator) with possibility to resample the data in order to select a subsets of nodes and lighten the computational cost of the image (or movie) generation.

This work was part of Emily Bourne PhD (European NUMERICS PhD funding (2019-2021)) and Kevin Obrejan post-doc (EoCoE-II post-doc hired for this task for 18 months since April 2019). The validation

[5] Pyccel - Python extension language using accelerators `https://github.com/pyccel`

(see section task T1.5.1-2 in WP1) has been done in strong collaboration with physicists: Xavier Garbet (IRFM/CEA), Peter Donnel (IRFM/CEA) and David Zarzoso (CNRS/Aix-Marseille).

Modification of gyroaverage operator GYSELA's previous gyroaveraging operator, used in the circular geometry, was based on averaging over sample points on circular trajectories, a method that is in principle compatible with any geometry. However, the implementation itself relied on several assumptions (θ-invariance of the sample points' relative positions, Larmor radius constant in space, simple $(r, \theta) \mapsto (R, Z)$ mapping) that are incompatible with general geometries. A new gyroaveraging module was thus written from scratch, where the sample points are located on the mesh using an efficient search routine, only requiring that the mesh to be convex. Fig. 53 illustrates this search algorithm on a purposely very large and very pathological circle, showing in black the cells visited when searching for the green sample point and in blue the cells where a sample point was found. The assumptions made the previous gyroavaraging

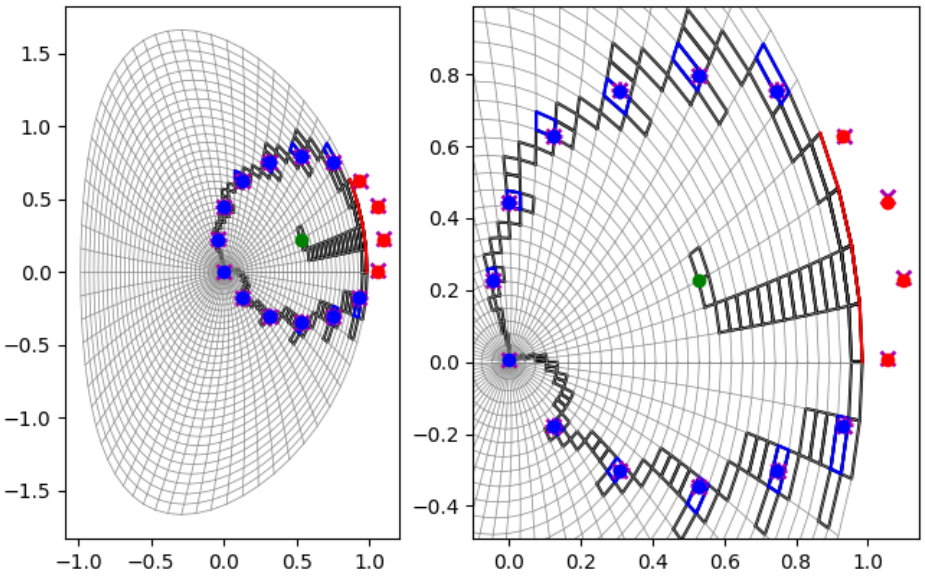


Figure 53: Search algorithm

module also allowed for lighter computations and several optimisations were required to reduce the new operator's computation time. The location for the sample points –or rather the interpolation points derived from them– during the code's initialisation phase is an inherently heavy process. However it is also a local process as the sample points are close to the circle's centre allowing for very short search paths and efficient OPENMP parallelisation. In addition, several optimisations were used to keep the computational cost of the gyroaveraging itself low: loop blocking in r and θ, batching in $v_{//}$ and φ to improve vectorisation, *etc.* Overall, the new gyroaveraging module incurs little time increase on GYSELA's initialisation phase and none to the computation kernel itself.

Modification of the quasi-neutrality solver As a reminder, the quasi-neutrality equation that needs to be solved is of the generic form

$$-\nabla \cdot (\alpha \nabla \phi) + \beta \left(\phi - \gamma \langle\, \phi \,\rangle_{\mathrm{FS}}\right) = \mathrm{RHS} \tag{5}$$

where the differential operators expressed using $\nabla_\perp \simeq \nabla$ are now defined as:

$$\nabla\phi = \frac{\partial \phi}{\partial x^j} g^{jk} e_k = \left(\frac{\partial \phi}{\partial r} g^{rr} + \frac{\partial \phi}{\partial \theta} g^{\theta r}\right) \nabla r + \left(\frac{\partial \phi}{\partial r} g^{r\theta} + \frac{\partial \phi}{\partial \theta} g^{\theta\theta}\right) \nabla \theta \tag{6}$$

and

$$\nabla \cdot \mathbf{X} = \frac{1}{\sqrt{|g|}} \frac{\partial}{\partial x^j} \left(\sqrt{|g|} \mathbf{X}^j \right) = \frac{1}{\sqrt{|g|}} \frac{\partial}{\partial r} \left(\sqrt{|g|} \mathbf{X}^r \right) + \frac{1}{\sqrt{|g|}} \frac{\partial}{\partial \theta} \left(\sqrt{|g|} \mathbf{X}^\theta \right) \tag{7}$$

where g is the metric tensor for the coordinates of the poloidal cross-section. As discussed in sub-task WP1-T1.5.1-2, the advantage of using a Culham equilibrium is that the Jacobian matrix of coordinate transformation can be derived analytically .

To solve equation (5) a spline-based Finite Element solver (see WP2-T1.5.1-2 for more detailed) as been extracted from SELALIB library and coupled to the GYSELA code. The SELALIB solver is available for splines of arbitrary degree but we only use the cubic splines to be consistent with the rest of the GYSELA code. The spline solver relies on conjugate gradient methods whose number of iterations and thus computation time can vary depending on calls. Fortunately, the timesteps required for the overall stability and accuracy of the simulation ensure that the electric potential does not change much between 2 consecutive timesteps, thus allowing to re-use the solution from a timestep as initial guess in the following one. This effect is clearly visible on Fig. 54, which compares performance from the spline solver to the original one for 2 resolutions. After an initial phase, the number of iterations of the spline solver until convergence and computation time drop by one order of magnitude, remaining low for the rest of the simulation. Despite the speedup granted by this efficient choice of the initial guess, the spline solver remains slower than the previous solver –although the gap shrinks at higher resolutions– when using a single OPENMP thread. However, its many matrix/vector operations allows for better OPENMP parallelisation than what was possible with the Fourier solver. Fig. 55 shows a comparison with 12 OPENMP threads, where the spline solver is now faster than the Fourier solver.

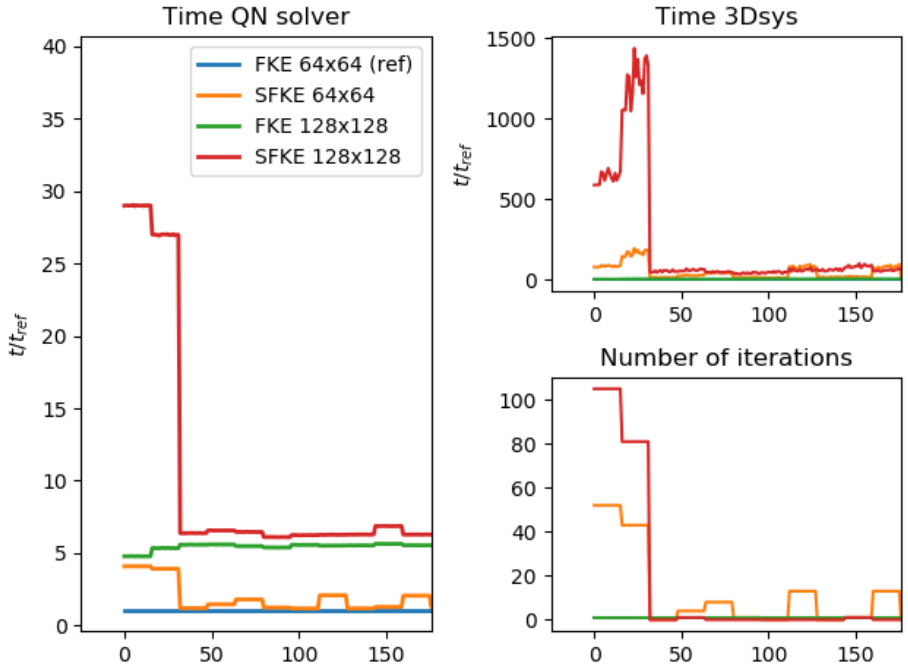


Figure 54: Comparison in the sequential case (1 OPENMP thread) of the computation time for the whole Poisson solver (Time QN solver), the linear solver itself (Time 3Dsys), and the number of iterations for the original Fourier solver (FKE) and new spline solver (SFKE) at 2 resolutions. The computation times are normalised to the time taken by the Fourier solver in the 64x64 case.

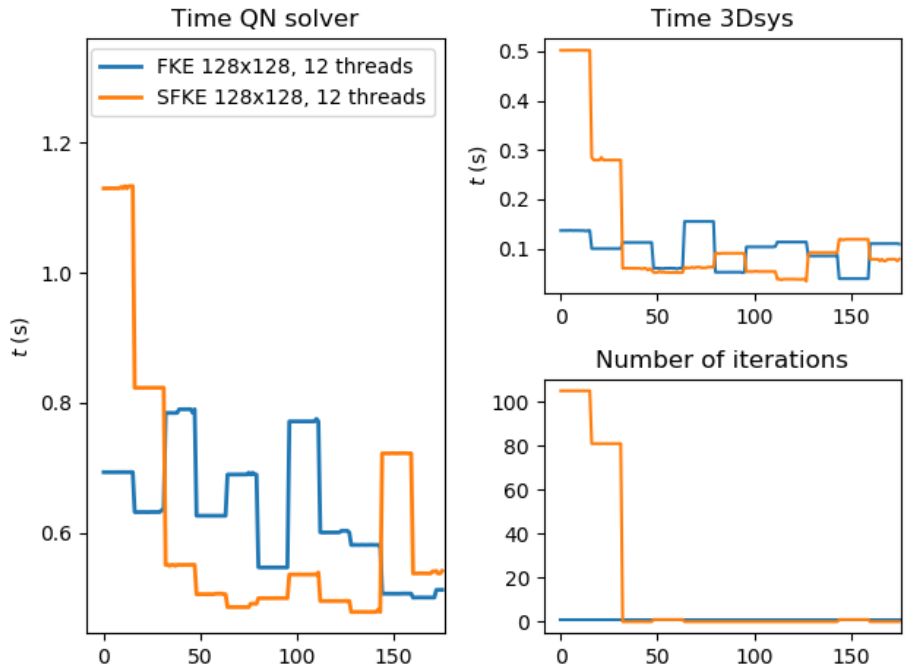


Figure 55: Same comparison as in Fig. 54 but where 12 OpenMP *threads were used and only the 128x128 case is shown. Notice that the times are given in seconds rather than normalised.*

9.4.4 Optimisation of GyselaX code on pre-exascale architectures

The Gysela code, developed in FORTRAN and based on a hybrid MPI/OpenMP parallelization, runs efficiently on more than 100'000 cores on current standard architectures. The code is one of the selected codes for the benchmarks in the framework of the EPI (European Processor Initiative). However, the new architectures planned for the exascale, based on heterogeneous accelerated computing nodes, are less favourable to applications such as Gysela which require a lot of memory and parallel communications. Achieving performance on exascale computers is a major challenge for Gysela. All the optimisation efforts on pre-exascale machines carried out during the EoCoE-II project have enabled major advances in this race to exascale. As already mentioned, the targeted architecture chosen within this project was the Fujitsu A64FX processors (ARM-based processor developed by Fujitsu and used in the Fugaku super-computer). This choice was made because we imagined that porting a code with the complexity of Gysela would be much easier on this type of architecture and therefore more in line with the duration of the project than on GPU-based architectures. But in the end, after feedback, obtaining performance on A64FX turned out to be much more complicated than expected.

At M18, the Gysela code had been successfully compiled on the INTI cluster (ATOS prototype equipped of Thunder X2 ARM-based processors) and small test cases had shown promising results when compared to results obtained on Broadwell processors of OCCIGEN Tiers-1 HPC machine located at CINES (Montpellier - France).

Since M18, Gysela was ported on two supercomputer equipped of Fujitsu ARM-A64FX processors:

- The CEA-RIKEN collaboration gives us the opportunity to access the pre-exascale Fujitsu super-computer Fugaku [36].
- A preparatory access was obtained via GENCI technology watch unit on the prototype ARM A64FX (Atos FX700) partition (80 nodes) installed in August 2021 at TGCC/France.

and optimisation efforts have considerably intensified thanks to the involvement of many people via new collaborations. This dynamic, which would not have been possible without the EoCoE-II project, has clearly enabled the GYSELA team to overcome its lack of human resources as HPC specialists and to increase its skills in this field following the departure of its HPC specialist at the beginning of the project:

- Monthly regular meetings were organized with HPC experts from R-CCS (Mitsuhisa Sato, Hitoshi Murai and Miwako Tsuji), from JAEA/Japan (Yuuichi Asahi) to help for the porting of GYSELA on FUGAKU supercomputer.
- Weekly regular meetings were organized with FAU university (Georg Hager, Markus Wittmann, Tobias Klöffel) in the framework of the WP3 and POP3 Centre of Excellence (Brian Wylie) dedicated to training in handling profiling tools (Score-P[6], Scalasca[7] and LIKWID tools[8]) and to help for node-level optimisation.
- Financial support from GENCI help for the optimisation of GYSELA on Joliot-Curie/Irene ARM-A64FX partition. Bi-monthly meeting were organised since January 2022 with ATOS experts (Antoine Morvan, Stephan Jaure and Christophe Bethelot), with ARM experts (Conrad Hillairet and Fabrice Dupros), with Pierre Lagier from Fujitsu and with TGCC high level support team (Laurent Nguyen and Bruno Froge).
- Kevin Obrejan was hired in October 2021 at IRFM-CEA after his EoCoE post-doc to reinforce the new HPC group created at IRFM in January 2021 and led since by Dorian Midou. Since his hiring Kevin has dedicated 50% of his time to the optimisation of GYSELA.

Strong efforts of optimisation performed on all the kernels of GYSELAX Detailed analysis of the different computational kernels using profiling tools have demonstrated that the code was not extensively and properly using vectorization as imagined. Indeed, the vectorisation efforts that were made to obtain good performances on the many-core architectures (Intel KNC and KNL) in 2017 proved insufficient. Fig. 56 shows the average vector length used for arithmetic operations in the full code and most time-consuming kernels (Advection, quasi-neutrality (QN) solver, collisions, Krook, diagnostics). This vector length is based on hardware counters (extracted from Score-P results) and computed as:

$$\text{Avg. vector size} = \frac{\#\text{scalar} + 2 \cdot \#\text{vec128b} + 4 \cdot \#\text{vec256b} + 8 \cdot \#\text{vec512b}}{\#\text{scalar} + \#\text{vec128b} + \#\text{vec256b} + \#\text{vec512b}} \tag{8}$$

It clearly highlights that vectorisation in the GYSELA code was poorly used in the version of September 2021 (blue dotted-line). Indeed, the average vector length of arithmetic operations was close to 1 (close to the scalar situation) for all the main components of the code. where $\#scalar$ is the number of non-vectorised operations on double precision floating point numbers while $\#vec128b$, $\#vec256b$ and $\#vec512b$ are the number of operations on vectors containing respectively 2, 4 and 8 of such numbers. At this stage it was thought that this low vectorisation could be one of the causes of poor performance on the ARM-A64FX architecture. We therefore worked on improving vectorisation thinking it would benefit to all architectures. Therefore, we spent more than 6 months improving the vectorisation of the main kernels of the code, leading to an average vector size larger than 2.2 for the global code in May 2022 (see orange dotted-line in Fig. 56) on the Skylake processor. In Fig. 56, the difference between the orange and green line is the vectorisation register size respectively 4 double-precision floats (AVX2) for the orange one and 8 double-precision floats (AVX512) for the green one. On recent Intel processors, the clock frequency

[6] Score-P - Scalable Performance Measurement Infrastructure for Parallel Codes: `https://www.vi-hps.org/projects/score-p/`

[7] Scalasca is a software tool that supports the performance optimisation of parallel programs by measuring and analyzing their runtime behaviour: `https://www.scalasca.org/`

[8] LIKWID Performance Tools `https://hpc.fau.de/research/tools/likwid/`

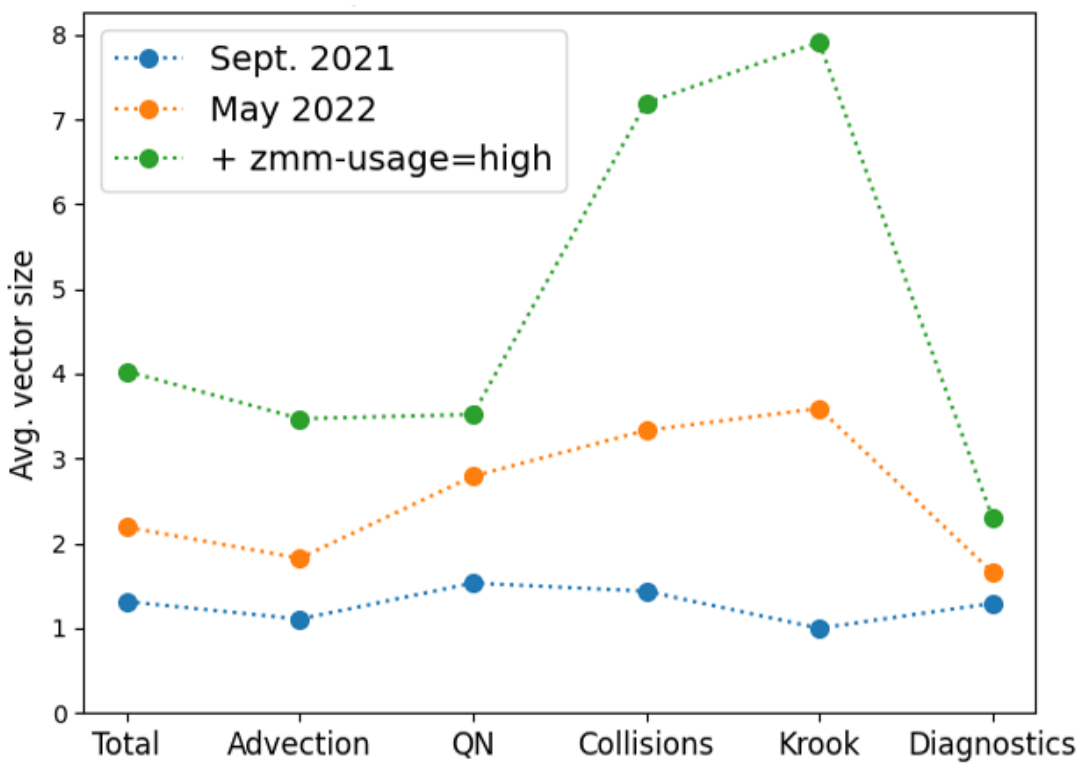


Figure 56: GYSELAX *vectorisation results on SKL architecture: Comparison of the average vector length (see Eq.*(8)*) for the main kernels of the code between a version of September 2021 (blue line) and version after vectorisation optimisations in May 2022 (orange dotted-line and green dotted-line with the* `-qopt-zmm-usage=high` *Intel compiling option).*

varies depending on the number of cores and the vector size for consumption and thermic issues. The frequency using AVX512 and all core is the slowest one. Using AVX2, the cores can run at a faster clock speed. In the case of GYSELA, forcing the compiler to use AVX512 instructions through the compilation option `-qopt-zmm-usage=high` did not yield any speedup to the code, despite enabling a vector size of 8 double-precision floats instead of 4 (see green dotted-line Fig. 56). Another consequence of that is that hyperthreading which was the standard mode for GYSELA when available has no more benefit. This is another proof, if needed, of a better vectorisation of the GYSELA code.

Since 2021, many other numerical improvements have been made in the code, the main ones being the following:

- Complete refactoring of collision operator including blocking-cache optimisation,
- Complete rewriting of a more complex gyroaverage operator to tackle D-shape magnetic configuration with an optimised search algorithm (see section 9.4.3),
- More complex quasi-neutrality solver for trapped kinetic electrons taking into account a limiter configuration via penalisation technique with asynchronous MPI communications during the calculation of the RHS, and
- Strong simplification of the source terms

All these efforts were rewarded by reducing the CPU time of a GYSELA simulation by over 70% on the Skylake architecture as shown in Fig. 57. Tests on Irene-AMD partition have confirmed the same gain on AMD EPYC-Rome nodes.

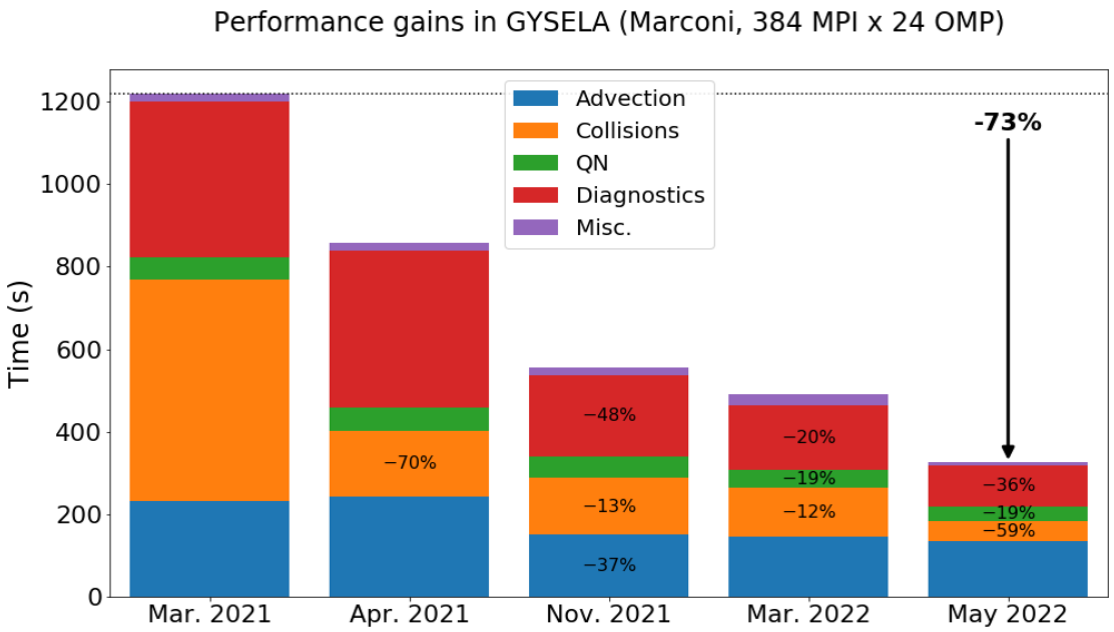


Figure 57: Optimisation performed between March 2021 and May 2022 on the main kernels of GYSELA *code (advection in blue, collisions in orange, quasi-neutrality in green and diagnostics in red) leading to a global gain of 73%. Tests performed on Fusion partition of CINECA/Marconi supercomputer on SKL architecture for a relevant simulation on 384 MPI processes with 24 threads.*

Adaptation to ARM-A64FX architecture Despite all the efforts, succeed in obtaining performance on an Fujitsu A64FX type architecture is not straightforward for a code such as GYSELA. Almost all improvements performed on the Skylake (SKL) architecture had a beneficial impact on A64FX architecture. Fig. 58 shows that almost all optimisations performed between September 2021 and May 2022 for the SKL architectures have provided speed-ups on A64FX architecture except for the quasi-neutrality (QN) solver. The porting of the GYSELA code on the new Irene-ARM partition of Joliot-Curie supercomputer at TGCC/France in August 2021 has confirmed the same behaviour. The reason for this degradation is under investigation. The first possibility, which was the fact that asynchronous communications were added to the calculation of the Right-Hand-Side of the QN equation, does not fully explain this increase in CPU time. However Fig. 59 shows that the GYSELA code is still 3 times slower on A64FX than on Intel SKyLake architecture.

The optimisation work will continue in the coming year as part of the CEA-RIKEN collaboration. Now that vectorisation has been added wherever possible in the main kernels and has proven to be effective on the SKL and AMD-Milan architectures, it remains to be understood why it is not effectively taken into account on Fujitsu-A64FX. Another problem that has been identified is the fact that the Fujitsi compiler cannot handle inlined functions, whereas they are handled by the Intel and GNU compilers on more standard architectures. It is not at all possible to replace by hand the call to these functions with the corresponding lines of code. The possibility of doing this automatically thanks to the metaprogramming framework MetaX is under investigation. RIKEN teams are currently developing this metaprogramming framework for existing HPC languages, including FORTRAN based on Omni[9] to improve the productivity of HPC programs (see Murai's paper [37] for more details). Preliminary tests have been carried out on 2D advection routines in GYSELA by Hitoshi Murai but have not yet been finalised due to lack of time and human resources. It is hoped that the use of MetaX will improve the performance of the code as we have chosen not to rewrite specific kernels optimised for A64FX. This would require the rewriting of most of the computational kernels.

[9]Omni is a compiler infrastructure based on source-to-source translation for FORTRAN and C developed by RIKEN and the University of Tsukuba.

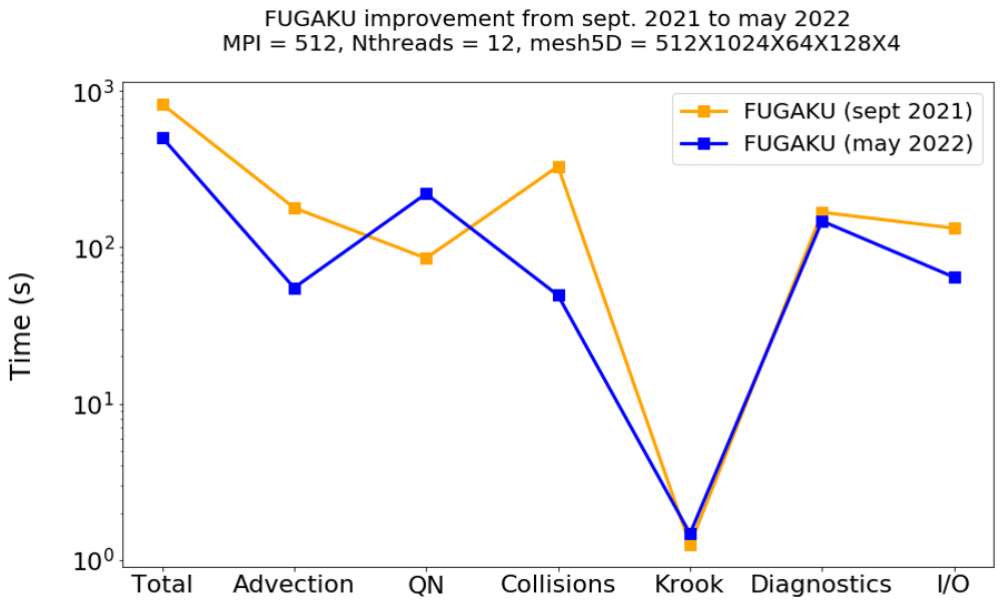


Figure 58: Total CPU time of the GYSELA *code and of its main kernels (Advections, Quasi-neutrality (QN), Collisions and Krook operators, diagnostics and I/O) for a simulation parallelised with* 512 *MPI process and* 12 OPENMP *thread on FUGAKU supercomputer. Results obtained in September 2021 (orange line) are compared to the results of May 2022 (blue line).*

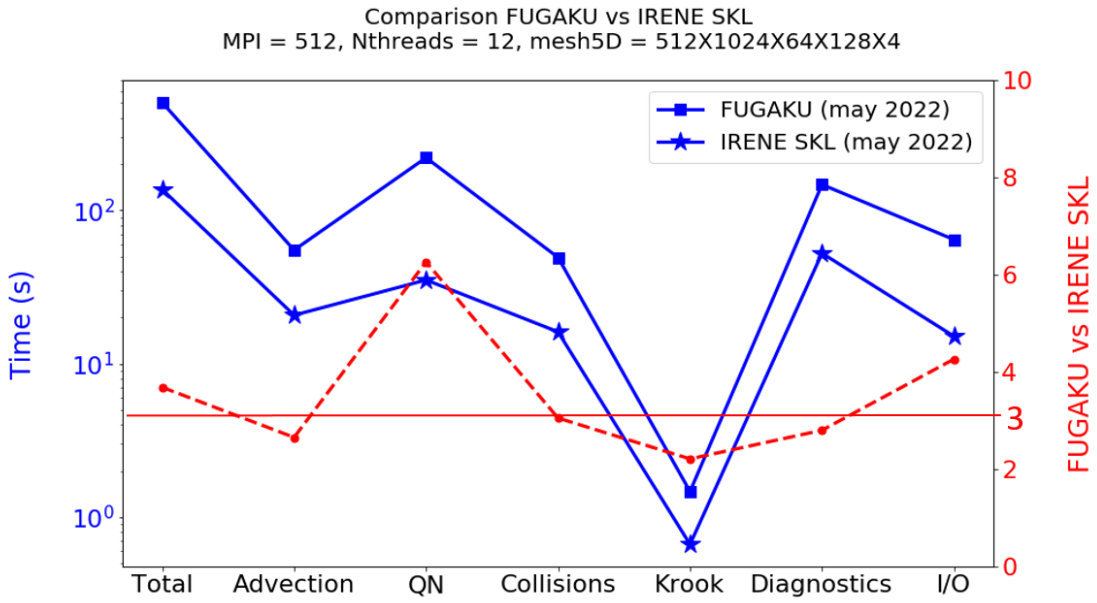


Figure 59: Comparison of GYSELA *CPU time (Total time and time of the main kernels) on A64FX (blue stars-line) and SKyLake (blue squares-line) architectures. Both simulations were performed for a 5D mesh of size 512*$\times$ *1024* $\times$ *64* $\times$ *128* $\times$ *4 with the same parallelism (512 MPI process and 12* OPENMP *threads) respectively on FUGAKU and Joliot-Curie/Irene SKL supercomputers. The red dashed-line represent the ratio between FUGAKU and SKL which values are given by right axis.*

The current feedback at the end of EOCOE-II is that the strategy we have adopted so far, consisting of developing a single code while accepting the fact that it performs a little less well on certain architectures,

does not seem to be viable in the long term given the increasing heterogeneity of the computing nodes. This becomes even clearer with the porting of the code on GPU which began in late 2021 via a contract of progress CINES (2021-2022). The GYSELAX code is one of the 5 codes selected to get help from HPE teams for its porting on the future supercomputer Adastra –based on AMD Instinct MI250X accelerator– which will be commissioned at CINES in late 2022.

This porting to GPU is not discussed here because it is not part of the EOCOE-II project. But it is part of the same long term strategy which is the preparation of the GYSELAX for exascale simulations. The fact that the porting to both architectures has to be done at the same time with very limited human resources for the GYSELA team explains the fact that the optimisation on ARM has been reduced since September 2021.

GYSELAX weak scaling up to 729 088 cores A code such as GYSELAX is definitely more adapted to standard architectures such as those to which we had access until now. This was proved again very recently with the performance we managed to obtain on the CEA-HF machine installed in early 2022 at Bruyère-le-Chatel in France. It is composed on 6330 AMD-Milan (EPYC 7763) nodes. The code GYSELAX is one of the few civil codes that had the opportunity to access this French defence supercomputer during "Grand Challenge" campaign. This opportunity gave us access to a number of processors that we had never been able to access before. These scaling tests up to 5696 nodes (90% of CEA-HF nodes) were carried out thanks to the HPC experts of the TGCC center and more particularly Laurent Nguyen (CEA, DAM/DIF). The tests were performed for meshes with $(N_r, N_\theta, N_\varphi, N_{v_\parallel}) = 512 \times 1024 \times 128 \times 128$ and N_μ varying from 32 to 178. The smallest 5D mesh $(N_r, N_\theta, N_\varphi, N_{v_\parallel}, N_\mu) = 512 \times 1024 \times 128 \times 128 \times 32$, corresponding to 275 billions of points, was the largest mesh used so far with the GYSELA code. The largest 5D mesh which corresponds to a mesh of 1.5 trillions of points is a new record for the GYSELAX code. The scaling was performed varying the number of MPI processes from 8192 to 45568 and using 32 OPENMP Threads per MPI process. As a successful result of the EOCOE-II project, the GYSELAX code run efficiently on more than 500k cores. Indeed, as seen in Fig. 60 the GYSELAX code exhibits a relative efficiency of 85% on more than 500k cores and 63% on 729 088 cores for a weak scaling from 1024 to 5696 nodes. The two main bottlenecks are the quasi-neutrality solver and the I/O. Concerning the QN solver the gap from 4096 to 5696 nodes (green histogram in Fig. 60) is due to huge MPI communications required for the computation of the RHS of the equation. This is unfortunately one of the well known bottleneck of the GYSELA code difficult to overcome. The lower efficiency of the diagnostics is largely due to the writing to disk of the huge amount of data for the restart files. The writing of the restart files was done without PDI (see paragraph dedicated above) because the access window to the supercomputer was too short to port both GYSELAX and PDI in this new machine. We hope that we will be able to have other slots soon to validate writing via PDI and thus test writing via parallel HDF5. The results show a relative efficiency of 55% on 3072 nodes which deteriorates very strongly when the number of nodes is doubled. There is no measurement for 5696 nodes due to a crash during the writing. These results must be taken as preliminary because the tests were performed when the file system was not completely tuned. However, this writing to disk remains an achievement in itself as we managed to write 13.2 TB of data distributed in 24576 files that were simultaneously written to disk. The fact that this writing took 1173 seconds remains a bottleneck that needs to be removed. As already discussed above, writing efficiently to disk huge amounts of data will be a critical points for future exascale file systems.

9.5 Conclusion

The clear ambition for Gysela within EOCOE-II was to prepare the code for future Exascale machines while improving algorithms and the structure. The EOCOE-II project clearly enabled a detailed study on the best solutions to adopt and those to discard for porting the code. Numerous obstacles and unforeseen events (departure of an important collaborator, COVID) have forced the team to adapt and change the

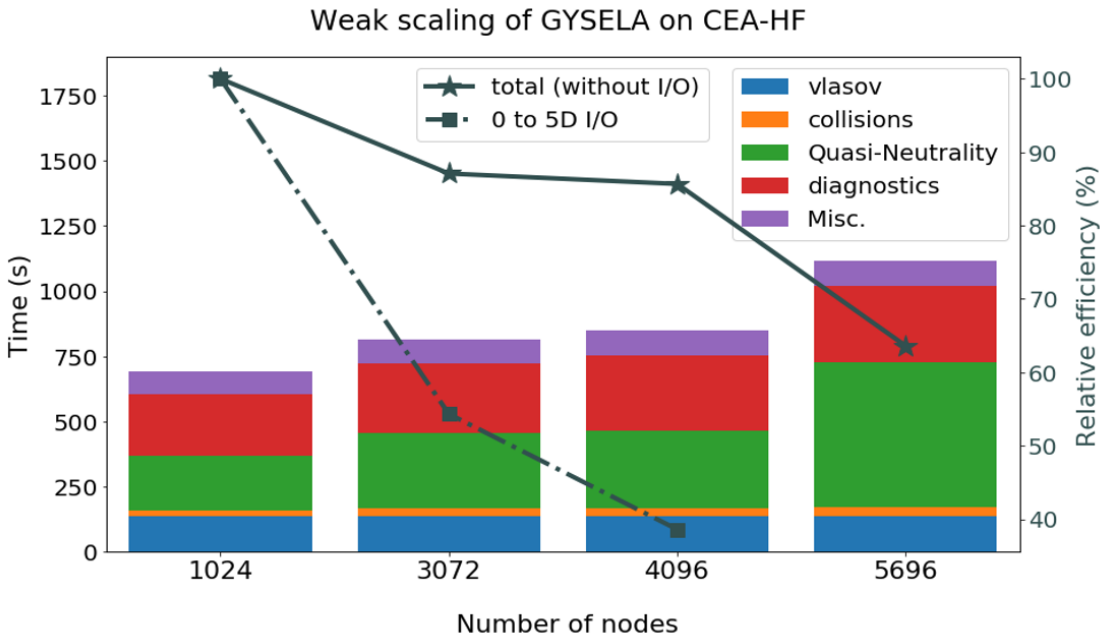


Figure 60: GYSELAX *weak scaling from 1024 to 5696 nodes of the CEA-HF supercomputer (6330 AMD EPYC 7763 nodes = 810 240 cores). CPU time (left vertical axis) measured for 4 iterations is shown: (i) for the main kernels of the code (Vlasov solver in blue, collision operator in orange, quasi-neutrality solver in green and diagnostics in red) and (ii) for the rest of the code in purple. Relative efficiency is plotted (right vertical axis) for the code without I/O (solid-star line) and separately for the 0 to 5D I/O (dashed-square line).*

initial plans. In the end, the project enabled an important global dynamic to be initiated. This dynamic was also achieved thanks to the many collaborations and external fundings that came in synergy with the project. The modernisation of the code has started. Thanks to the porting on ARM-based processors, many bottlenecks were understood and the global performance of the code has been improved on many architectures (Intel and AMD x86 processors). The GPU porting is also underway and will exploit the new modern C++ based foundation.

10 Conclusion

Within work package 2, the aim of the EoCoE project was to assist developers in optimising selected codes on existing architectures and to prepare them for future architectures. The ultimate goal is to prepare the codes for exascale. In the end, all codes benefited from significant acceleration on x86 CPU architectures. In addition to this, many codes have improved their scaling to accommodate the increased number of nodes. These optimisations were made possible thanks to the collaboration with the project's HPC experts (FAU, BSC, CEA) and external collaborations (PoP, Atos, ARM, Fujitsu, RIKEN). However, in order to achieve exascale, optimisation on x86 processors is not enough. More and more supercomputers are now equipped with GPU accelerators. In addition to this, ARM processors are gaining in power in the HPC market. As a result, some developement teams have focused on GPU porting (ALYA, LIBNEGF, PARFLOW) during the project. The GYSELA code was ported to ARM architecture. The ALYA and LIBNEGF codes are both in the process of being ported but are well underway and already showing encouraging results. They have both chosen to use CUDA and OPENACC, limiting them to NVIDIA GPUs for the time being with potentially higher performance. The ParFlow application is the most successful on this point with GPU version used in production using CUDA and KOKKOS. The CUDA backend provides the best performance

resultst on NVIDIA GPUs. The KOKKOS backend provides performance portability on major GPU cards of the market including technologies in coming exascale systems. The Adaptive Mesh Refinment based on P4est is partially implemented. The preconditioners have to be updated.

In conclusion, for many applications, the EoCoE-II project has started the necessary shift to Exascale. Not all codes are ready yet, but some are almost ready. Others are now clearly in this optimization dynamic. It is clear, however, that the non-renewal of the project will have a negative impact on further activities.

References

[1] "DDC's' website." https://github.com/Maison-de-la-Simulation/ddc. Accessed on 2022-06-30.

[2] "Top500 website." https://www.top500.org/. Accessed on 2022-06-30.

[3] "Kokkos github page." https://github.com/kokkos/kokkos. Accessed on 2022-06-30.

[4] "Raja github page." https://github.com/LLNL/RAJA. Accessed on 2022-06-30.

[5] C. Roussel, K. Keller, M. Gaalich, L. Bautista Gomez, and J. Bigot, "PDI, an approach to decouple I/O concerns from high-performance simulation codes," *hal-01587075*, vol. 1, no. 1, p. 1–12, 2017.

[6] J. BIGOT, "PDI's website." https://pdi.dev. Accessed on 2022-06-30.

[7] "Libkomp website." https://gitlab.inria.fr/openmp/libkomp. Accessed on 2022-06-30.

[8] M. Lobet and G. Hager, "Eocoe performance evaluation workshop registration website." https://indico.math.cnrs.fr/event/4587/. Accessed on 2022-06-30.

[9] "Pop center of excellence." https://pop-coe.eu/. Accessed on 2022-06-30.

[10] J. G. Georg Hager, Jan Treibig, "Eocoe performance evaluation workshop recorded videos." https://public.weconext.eu/eocoe2/2019-10-07/index.html. Accessed on 2022-06-30.

[11] "ALYA's website." https://www.bsc.es/research-and-development/software-and-apps/software-list/alya. 2022-06-30.

[12] O. Lehmkuhl, G. Houzeaux, H. Owen, G. Chrysokentis, and I. Rodriguez, "A low-dissipation finite element scheme for scale resolving simulations of turbulent flows," *Journal of Computational Physics*, vol. 390, pp. 51–65, 2019.

[13] R. Borrell, D. Dosimont, M. Garcia-Gasulla, G. Houzeaux, O. Lehmkuhl, V. Mehta, H. Owen, M. Vázquez, and G. Oyarzun, "Heterogeneous cpu/gpu co-execution of cfd simulations on the power9 architecture: Application to airplane aerodynamics," *Future Generation Computer Systems*, vol. 107, pp. 31–48, 2020.

[14] "Alya performance suite." https://rooster.bsc.es/. Accessed on 2022-06-30.

[15] G. Houzeaux, M. Garcia, J. C. Cajas, A. Artigues, E. Olivares, J. Labarta, and M. Vázquez, "Dynamic load balance applied to particle transport in fluids," *International Journal of Computational Fluid Dynamics*, vol. 30, no. 6, pp. 408–418, 2016.

[16] G. Houzeaux, R. de la Cruz, H. Owen, and M. Vázquez, "Parallel uniform mesh multiplication applied to a navier–stokes solver," *Computers and Fluids*, vol. 80, pp. 142–151, 2013. Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics ParCFD2011.

[17] H. Schottenhamml, A. Anciaux-Sedrakian, F. Blondel, A. Borras-Nadal, P.-A. Joulin, and U. Rüde, "Evaluation of a lattice boltzmann-based wind-turbine actuator line model against a navier-stokes approach," *Journal of Physics: Conference Series*, vol. 2265, p. 022027, may 2022.

[18] C. Bak, F. Zahle, R. Bitsche, T. Kim, A. Yde, L. Henriksen, M. Hansen, J. Blasques, M. Gaunaa, and A. Natarajan, "The dtu 10-mw reference wind turbine," 2013. Danish Wind Power Research 2013.

[19] P. Benard, A. Viré, V. Moureau, G. Lartigue, L. Beaudet, P. Deglaire, and L. Bricteux, "Large-eddy simulation of wind turbines wakes including geometrical effects," *Comput. Fluids*, vol. 173, pp. 133–139, 2018.

[20] "Weather research and forecasting model." `https://www.mmm.ucar.edu/weather-research-and-forecasting-model`. Accessed on 2022-06-30.

[21] H. Elbern, A. Strunk, H. Schmidt, and O. Talagrand, "Emission rate and chemical state estimation by 4-dimensional variational inversion," *Atmos. Chem. Phys.*, vol. 7, pp. 1–59, 2007.

[22] C. J. Walcek, "Minor flux adjustment near mixing ratio extremes for simplified yet highly accurate monotonic calculation of tracer advection," *J. Geophys. Res.*, vol. 105, no. D7, pp. 9335–9348, 2000.

[23] "PARFLOW's website." `https://github.com/parflow/parflow/tree/adaptive`. 2022-06-30.

[24] J. Hokkanen, S. Kollet, J. Kraus, A. Herten, M. Hrywniak, and D. Pleiter, "Leveraging hpc accelerator architectures with modern techniques—hydrologic modeling on gpus with parflow," *Computational Geosciences*, vol. 25, no. 5, pp. 1579–1590, 2021.

[25] C. Burstedde, J. A. Fonseca, and S. Kollet, "Enhancing speed and scalability of the ParFlow simulation code," *Computational Geosciences*, vol. 22, no. 1, pp. 347–361, 2018.

[26] C. Clauser, *Numerical simulation of reactive flow in hot aquifers: SHEMAT and processing SHEMAT*. Springer Science & Business Media, 2003.

[27] V. Rath, A. Wolf, and H. Bücker, "Joint three-dimensional inversion of coupled groundwater flow and heat transfer based on automatic differentiation: Sensitivity calculation, verification, and synthetic examples," *Geophysical Journal International*, vol. 167, no. 1, p. 453–466, 2006.

[28] "GYSELA's website." `https://gyselax.github.io/`. Accessed on 2022-06-30.

[29] V. Grandgirard, J. Abiteboul, J. Bigot, T. Cartier-Michaud, N. Crouseilles, G. Dif-Pradalier, C. Ehrlacher, D. Esteve, X. Garbet, P. Ghendrih, G. Latu, M. Mehrenberger, C. Norscini, C. Passeron, F. Rozar, Y. Sarazin, E. Sonnendrücker, A. Strugarek, and D. Zarzoso, "A 5D gyrokinetic full-f global semi-lagrangian code for flux-driven ion turbulence simulations," *Computer Physics Communications*, vol. 207, pp. 35–68, 2016.

[30] V. Grandgirard, Y. Asahi, J. Bigot, E. Bourne, G. Dif-Pradalier, P. Donnel, X. Garbet, P. Ghendrih, Y. Güçlü, K. Kormann, D. Midou, Y. Munschy, K. Obrejan, C. Passeron, R. Varennes, and Y. Sarazin, "How to prepare the GYSELA-X code to future exascale edge-core simulations," in *PASC 2021 - The Platform for Advanced Scientific Computing Conference*, (Genève - E-Conference, Switzerland), Association for Computing Machinery (ACM) and the Swiss National Supercomputing Centre (CSCS), July 2021.

[31] V. Grandgirard, K. Obrejan, D. Midou, Y. Asahi, P.-E. Bernard, J. Bigot, E. Bourne, J. Dechard, G. Dif-Pradalier, P. Donnel, X. Garbet, A. Gueroudji, G. Hager, H. Murai, Y. Ould-Ruis, T. Padioleau, L. Nguyen, M. Peybernes, Y. Sarazin, M. Sato, M. Tsuji, and P. Vezolle, "New advances to prepare GYSELA-X code for exascale global gyrokinetic plasma turbulence simulations: porting on GPU and ARM architectures," in *PASC22 Conference - The Platform for Advanced Scientific Computing*, (Bâle (virtual event), Switzerland), the Association for Computing Machinery (ACM) and the Swiss National Supercomputing Centre (CSCS), June 2022.

[32] Y. Asahi, T. Padioleau, G. Latu, J. Bigot, V. Grandgirard, and K. Obrejan, "Performance portable vlasov code with c++ parallel algorithm," in *International Conference for High Performance Computing, Networking, Storage, and Analysis: Workshop P3HPC*, (Dallas, Texas), pp. 1–10, submitted to, Nov. 2022.

[33] A. Gueroudji, J. Bigot, and B. Raffin, "DEISA: dask-enabled in situ analytics," in *HiPC 2021 - 28th International Conference on High Performance Computing, Data, and Analytics*, (virtual, India), pp. 1–10, IEEE, Dec. 2021.

[34] "DASK website." `https://dask.org`. Accessed on 2022-06-30.

[35] E. Bourne, Y. Munschy, V. Grandgirard, M. Mehrenberger, and P. Ghendrih, "Non-Uniform Splines for Semi-Lagrangian Kinetic Simulations of the Plasma Sheath." working paper or preprint, Aug. 2022.

[36] "Fugaku supercomputer." `https://www.fujitsu.com/global/Images/supercomputer-fugaku.pdf`. Accessed on 2022-06-30.

[37] H. Murai, M. Sato, M. Nakao, and J. Lee, "Metaprogramming framework for existing hpc languages based on the omni compiler infrastructure," in *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, pp. 250–256, 2018.