Report on porting and performance analysis

Table of Contents

List of Figures

List of Tables

Executive Summary

The previous Deliverable D2.1 presented best practice guidelines and tutorials for the various heterogeneous High-Performance Computing systems available to the European Center of Excellence in Exascale Computing “Research on AI- and Simulation-Based Engineering at Exascale” (CoE RAISE). The present Deliverable D2.2 is the first of a series of three Deliverables, namely D2.2, D2.3 and D2.4, all reporting on porting codes to heterogeneous systems, performing code optimizations, and analyzing code performance. Key results of this work are enhancements of the performance of specific numerical components embedded in simulation frameworks and their demonstration. These performance enhancements are key to the success of all the use cases proposed in Work Package 3 *“Compute-Driven Use-Cases at Exascale”* and Work Package 4 *“Data-Driven Use-Cases at Exascale”* of CoE RAISE. In this Deliverable, these successes are demonstrated for the multi-physics simulation codes Alya from the Barcelona Supercomputing Center and m-AIA from RWTH Aachen University. This is complemented by an extensive scaling and accuracy study of the most important AI frameworks.

1 Introduction

The architectures of next-generation supercomputers with Exascale power will evolve around the current modular and heterogeneous setups. These systems will consist of multiple interconnected components with each component suited for a specific set of tasks and with an immense computational power. Such a modular approach is especially suited for compute- and data-centric workflows that may require different High-Performance Computing (HPC) architectures for the various potentially concurrently running workflow components.

The different use-cases of the European Center of Excellence in Exascale Computing "Research on AI- and Simulation-Based Engineering at Exascale" (CoE RAISE) in Work Package 3 (WP3) "*Compute-Driven Use-Cases at Exascale*" and in WP4 "*Data-Driven Use-Cases at Exascale*" require workflows belonging to this class. They intertwine HPC methods for simulation and data processing with Artificial Intelligence (AI) technologies at Exascale to reduce the time-to-solution while retaining a high accuracy. The size of such simulations and data-driven workflows in terms of computational resources and amount of data is expected to reach unprecedented levels. Thus, porting existing codes to new architectures and new systems as well as optimizing code is required at all levels of the workflow. This involves data management (file transfer, data repositories), the modification of computational kernels of simulation and data processing codes, Input/Output (I/O) tuning, and the optimization of workflow management itself.

The Task T2.1 "*Modular and heterogeneous supercomputing architectures*", which corresponds to this Deliverable, involves heterogeneous architectures and targets porting and optimization of the codes that contribute to the complex workflows of the use-cases in CoE RAISE. The systems considered here include the heterogeneous HPC systems found at the Tier-2 and Tier-3 centers of the consortium (University of Iceland - UOI, RWTH Aachen University - RWTH, and Riga Technical University - RTU), the cutting-edge HPC systems of the Tier-0 and Tier-1 providers (Forschungszentrum Jülich – FZJ, and Barcelona Supercomputing Center - BSC), as well as the resources granted by the CoE's access call of PRACE. These resources are targeted for development, optimization, debugging, and performance analysis purposes.

Obviously, before exploring the performance of the codes, a porting phase is necessary, which ensures that the different codes can run on the target systems. The compilation on a new architecture requires adjustments to the compilation options and may involve changes to the code, e.g., to account for the specific compiler version. Only then, performance analyses and code optimizations can be performed to test the codes and to achieve the highest possible performance on the available systems. All optimizations will involve both Central Processing Units (CPUs) and accelerators. However, at this stage of the project, it is not clear which simulation codes of WP3 will eventually run concurrently with the AI tools on General Purpose Graphics Processing Units (GPGPUs). This will become clear in the second year of the project when full integrations of AI tools and HPC codes are achieved. All the activities of Task 2.1 correspond to the preparation of the codes to make efficiently use of the upcoming Exascale systems.

This Deliverable is organized as follows. Section 2 describes the computational resources available to the partners. This includes not only core-hours on European systems, but also the data repository provided and managed by FZJ. Then the porting, optimization, and performance analysis work is structured in two different parts. On the one-hand, Sec. 3 and Sec. 4 provide details in this respect on two of the simulation codes involved in WP3, i.e., Alya

from BSC and m-AIA from RWTH. Section 5 presents the work carried out on important AI tools considered in CoE RAISE. Finally, Sec. 6 summarizes the work performed and draws some conclusions.

2 Access to supercomputing resources

To perform code analysis, engineering and tuning, to run large-scale simulations and data analysis, and to share data with the partners and the community, it is necessary to provide the developers in CoE RAISE with access to supercomputing resources. Such resources can be classified into four main categories:

- compute time on prototype systems to give developers the opportunity to port to and test software on new hardware
- compute time on large-scale production systems to port to and scale software on real production systems
- compute time for application cases to perform domain-specific research and data analysis
- data repositories that can be used to share data and AI models also with respect to performance engineering.

These resources need to be further sub-classified into resources that can be provided by CoE RAISE for all partners to be shared and those that need to be acquired by the individual partners. All smaller development resources and shared data spaces fall into the first sub-category. Partners need to take care of their own computing time when it comes to their specific science and large-scale computing requests, i.e., such resources fall into the second sub-category.

In the following, an overview of the computational resources available within CoE RAISE is given. This includes compute time on prototype systems, see Sec. 2.1, on production systems for performance engineering and application cases, see Sec. 2.2 and Sec. 2.3, as well as data repositories, see Sec 2.4.

2.1 Compute time on prototype systems

The CoE RAISE partners have access to various prototype systems that are available at the HPC centers involved in the project. An overview of the available systems, their specifications, accessibility, and usage for production and development can be found in Deliverable D2.5.

2.2 Compute time on production systems for performance engineering

The Partnership for Advanced Computing in Europe (PRACE)[1] offers limited resources to all CoEs via their Rapid Access Program. In 2021, CoE RAISE has applied twice for PRACE resources through this program. In the first and the second round, compute time on main European systems has been granted as listed in Table 1. An overview of the resources and the systems is available for the CoE RAISE partners on the project's Basic Support for Cooperative Work (BSCW) server[2]. The first round 2021-1 period was from 01/04/2021 to 31/09/2021, the second round 2021-2 started on 01/10/2021 and runs until 31/03/2022.

round	location	system	core-h
2021-1	CINECA	Marconi100	275k[3]
	FZJ	JUWELS-BOOSTER	38k

[1] PRACE https://prace-ri.eu
[2] BSCW compute time https://bscw.zam.kfa-juelich.de/bscw/bscw.cgi/3599273
[3] 781 node hours = 270k cumulative hours (unit used by PRACE)/25k local hours

	FZJ	JUWELS-CLUSTER	40k
	HRLS	Hawk	280k
	CSCS	Piz Daint	510k
	BSC	Marenostrum4	100k
	CEA	Joliot-Curie Rome	150k
	CEA	Joliot-Curie SKL	87k
	CEA	Joliot-Curie KNL	94k
	LRZ	SuperMUC-NG	65k
2021-2	CINECA	Marconi100	300k
	FZJ	JUWELS-BOOSTER	16k
	FZJ	JUWELS-CLUSTER	45k
	HLRS	Hawk	144k
	CSCS	Piz Daint	10k
	BSC	Marenostrum4	92k

Table 1: Supercomputing resources granted to CoE RAISE from PRACE.

2.3 Compute time for application cases

Since production runs for the use cases may require a large amount of computing resources, they cannot be offered directly by the CoE RAISE such that the partners have to apply for them individually on the systems of their choice. An overview of how to apply for scientific computing resources is given in Deliverable D2.1.

As an example, FZJ and RWTH jointly applied for the compute time project *"Reconstruction of actuated turbulent boundary layers using neural networks"* related to Task 3.1 "*AI for turbulent boundary layers"*. The proposal requested 8.6 Mio. core-h on the GPU partition of the Jülich Research on Exascale Cluster Architectures (JURECA) system. The resources for the compute time project, running from Nov. 2021 to the end of Oct. 2022, were fully granted after a successful technical and scientific review.

2.4 Data repositories

There have been requests from various CoE RAISE partners to share simulation and measurement data. These requests mainly came from partners jointly working on a specific use case. A shared data space also allows to reuse data in use cases that they have originally not been intended for, which is especially favorable when the corresponding compute time projects are running at the same HPC facility. Furthermore, in the context of porting and performance analysis, such a joint project space can be used to share performance analysis data, e.g., log- and trace-files, as well as simulation setups for scaling analyses.

To provide users with space to perform such activities, the CoE RAISE management from FZJ applied for a 200TB data project at the Jülich Supercomputing Centre (JSC)[4] that was granted in July / 2021. CoE RAISE partners are invited to use this data space to share their data. They can apply for an account through the JUDOOR system[5], which has intensively been described in Deliverables D2.1 *"Best practice guidelines/tutorials for MSA / heterogenous systems"* and D2.5 *"Best practice guidelines / tutorials prototype".* Once the user has registered, he or she can join the data project `raise`.

In the project folder, a subfolder `performance_engineering` has been created, where the contributors to Task 2.1, Task 2.2, and also the main code developers of the simulation and AI tools can place, e.g., their testing setups or mini apps for performance analysis or porting, and results from such analyses.

The data space features the capability to also share data with the community. For this purpose, a folder `open_data` has been created, which, for the time of writing this document, holds data from the two WP3 use cases "*AI for turbulent boundary layers"* (RWTH, approx. 15TB) and "*AI for data-driven models in reacting flows"* (CERFACS, approx. 400MB). The data is available for download through CoE RAISE's website[6]. More information on the integration into CoE RAISE's website and the dataset details provided by the owners is given in Deliverable D6.9 *"Visual Identity"*.

The data is made available through `UFTP` (UDP - User Datagram Protocol - File Transfer Protocol). To make use of this sharing capability, the user needs to prepare the supercomputing environment. A detailed description on this and more information on `UFTP` can be found on JSC's website[7].

[4] JSC data projects https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/NewUsageModel/DataProjects.html
[5] JUDOOR https://judoor.fz-juelich.de
[6] CoE RAISE open data website https://www.coe-raise.eu/open-data
[7] `UFTP` https://apps.fz-juelich.de/jsc/hps/judac/uftp.html

3 Porting and performance analysis of Alya from BSC

Some porting activities have already been carried out in the context of CoE RAISE on some machines (DEEP-EST, JUAWEI, CT-ARM, and CT-AMD) and results have been provided in project month M6 in Deliverable D2.6 *"Support report"* as part of Task 2.2 *"Hardware prototypes".*

Subsequently, an overview of Alya is first provided in Sec. 3.1 before in Sec. 3.2 different porting activities are described. This is followed by a description of optimization activities in Sec. 3.3 and results of performance analyses in Sec. 3.4.

3.1 Overview

Alya [1] is a simulation code based on Fortran 2008 and is developed by BSC. Alya solves coupled multi-physics problems using HPC techniques for distributed and shared memory supercomputers, together with vectorization and optimization at the node level.

Strong scalability has been established for years, and recent efforts have mainly been devoted to node-level performance and parallel efficiency. In this sense: (i) A co-execution model has been developed to fully exploit heterogeneous resources and therefore enhance resource usage. (ii) An intra-node dynamic load balance strategy was implemented to correct load imbalances using the Dynamic Load Balancing (DLB) library developed at BSC[8]. (iii) At the inter-node level, a runtime redistribution mechanism based on real timings was implemented as well as a partition independent I/O strategy. (iv) To further enhance efficiency when solving multi-physics problems, an oversubscription strategy has been developed to avoid idle cores. (v) A continuous monitoring of the code enables to obtain the exact parallel efficiency, as a combination of communication efficiency and load balance, with the Tracking Application Low-level Performance library (TALP) developed at BSC.

To monitor the progress, BSC has developed a performance suite, run whenever a new version of the code is available (several times a week). In addition, this suite allows the development team to detect failures in the performance. It is fully integrated into the Continuous Integration / Continuous Delivery (CI/CD) approach the team follows. Therefore, any advances in the code can be compared to previous versions and quantified. As far as weak scalability is concerned, external weak-scalable solvers (multigrid, domain decomposition) were integrated into the code, mainly in the course of the Energy Oriented Center of Excellence-II (EoCoE-II) project[9]. Scalability has been demonstrated by the different Unified European Applications Benchmark Suite (UEABS) reports up to 32k cores (although strong scalability was established on Blue Waters in 2014 up to 100k cores for production multi-physics runs). The speed-up obtained is usually over 80% and of course, depends on the load per core. But in such tests, the speed-up is normalized using the lowest core count possible on the machine, which is sometimes quite high.

To know the real parallel efficiency, BSC integrated the TALP library into Alya. One of the objectives is to include this library for the testing of the UEABS, to get the correct parallel efficiencies (which are in general different from the one stated by the classical normalization mentioned before). The code has been tested using a hybrid MPI/OpenMP approach in combination with the DLB library. A co-execution model enables the code to take advantage of both CPU and GPGPU heterogeneous architectures.

[8] DLB library: https://pm.bsc.es/dlb
[9] EOCOE-2 project: https://www.eocoe.eu

CoE RAISE aims at introducing AI technologies into the code while conserving the scalability enhancements made in Task 2.1 *"Modular and heterogeneous supercomputing architectures"* and Task 3.2 *"AI for wind farm layout optimization"* of CoE RAISE. In the course of the project, the focus is therefore on the implementation of efficient AI surrogates as well as their integration into the simulation workflow. Furthermore, the AI training phase will require specific tools to be implemented in the code which need to satisfy the Exascale requirements. It should be noted that at this stage it is not known if the Computational Fluids Dynamics (CFD) component of Alya will run in a co-execution mode together with AI tools during the high-fidelity training phase as well as during the simulation of the wind farm including the surrogates.

Finally, it should be noted that all the optimizations and developments carried out in the context of RAISE can be identified in Alya GitLab in the branches with label `project:raise`[10].

3.2 Porting

Porting Alya to GPGPUs already started in 2018. A first version was published in [2], where a co-execution model is presented, which enables to *almost* fully exploit a heterogeneous node. The main assembly kernel of the Navier-Stokes equations has then been optimized in the context of the EoCoE-II project. Despite the high gains obtained for this kernel, Amdahl's law exhibited some new bottlenecks in subroutines not already ported, like the loop over elements to compute the critical time step. The corresponding subroutines have thus been modified to remove this constraining bottleneck and a performance analysis has been performed. To further accelerate the execution on GPGPUs, physical properties can now be computed on the fly during the assembly phase, instead of transferring them from the main memory. In the following, these optimization activities are described. It should, however, be noted that the performance analysis could not be finalized on time for this Deliverable. The complete results will hence be included in the subsequent version of this document, i.e., in Deliverable D2.3, which is due in project month M24.

3.2.1 Porting to JUWELS-BOOSTER

Thanks to the high portability of Alya, the program can run on the Jülich Wizard for European Leadership Science (JUWELS) system and on other supercomputers by only changing a few parameters and initial settings in the configure file and loading the proper environment modules.

For the JUWELS-BOOSTER, the first step is to load these two environment modules:

```
module load NVHPC/21.5-GCC-10.3.0 OpenMPI/4.1.1
```

Then, once the source code is downloaded using git, we have to define in the `Executables/unix` folder a configure file `config.in` as shown next. Alya can compile as well using cmake, and all the options will soon be implemented for their use with `cmake`.

[10] Developments and optimizations implemented in Alya in the context of CoE RAISE on Alya gitlab: https://gitlab.com/bsc-alya/alya/-/merge_requests?scope=all&state=all&label_name[]=project%3Araise

```
#########################################################################
#                          PGI CONFIGURE                                #
#########################################################################
F77      = OMPI_FC=pgfortran mpif90
F90    =   OMPI_FC=pgfortran   mpif90
FCOCC  = cc -c
FCFLAGS = -c -fast -Minfo=all -acc -ta=tesla:cuda11.3 -Mpreprocess -
I./Objects_x/ -Mbackslash  -Mextend -Mnoopenmp -Munroll -Mnoidiom -module $O
EXTRALIB = -lc
Fa2p     = pgfortran -c -x f95-cpp-input -DMPI_OFF -J../../Utils/user/alya2pos -
I../../Utils/user/alya2pos
Fa2plk   = pgfortran


#########################################################################
#                            PERFORMANCE FLAGS                          #
#########################################################################
FOPT    = OMPI_FC=pgfortran mpif90
CSALYA := $(CSALYA) -DNDIMEPAR -DOPENACCHHH -DSUPER_FAST -DDETAILED_TIMES -
DSPLITDO
CSALYA := $(CSALYA) -DVECTOR_SIZE=32000
```

3.2.2 Porting to Rudens at RTU HPC

The RTU HPC cluster Rudens is using CentOS[11], EPEL[12], and the OpenHPC[13] software package repositories. The GPGPU nodes on Rudens feature the NVIDIA A100 GPGPUs which benefit from the most recent version of the CUDA toolkit[14].

The RTU HPC environment uses a queuing system to match users' jobs with available computing resources. Users submit their programs to the job scheduler (Portable Batch System, PBS[15]), which maintains a queue of jobs and distributes them on the compute nodes according to the server status, scheduling policies, and jobs parameters (number of compute nodes / cores, estimated execution time, required memory, etc.).

RTU is the AI partner of BSC in CoE RAISE, i.e., this porting is key for the developments and testing of the proposed AI strategies. To compile Alya in the RTU HPC environment, it is necessary to load these modules:

```
module load cmake/3.15.4 gnu8/8.3.0 mpi/openmpi-4.1.1
```

[11] CentOS https://www.centos.org
[12] EPEL https://docs.fedoraproject.org/en-US/epel/
[13] OpenHPC https://openhpc.community
[14] CUDA toolkit https://developer.nvidia.com/cuda-toolkit
[15] PBS https://www.pbspro.org

After the source code is downloaded from the git repository, it is necessary to configure and compile Alya by following the CMake[16] configuration principles.

In the Alya directory, a new `build` directory needs first to be created:

```
mkdir build
cd build
```

To configure CMake using the command line, the following needs to be executed:

```
cmake ..
```

By default, the compiled Alya executables will be optimized for the processor architecture used by the login node. To adapt to a specific architecture, it is possible to modify the corresponding CMake configuration flags in `config/gnu.cmake` file. For example, to tune executables for AMD EPYC architecture, the following flags can be used:

```
set(CUSTOM_Fortran_FLAGS_ARCHITECTURE "-march=znver1 -mtune=znver1")
```

To start the compilation process, the following command is issued:

```
make
```

Multiple threads for the compilation can be employed via (4 threads in the given example):

```
make -j4
```

To install Alya in the `build` directory, type the following command:

```
make install
```

3.3 Optimization

Any code optimization should begin by analyzing the execution times for the problem of study. Here, the focus is on the analysis of those parts of the code that take most of the time in the execution phase of Alya to study the viability of carrying out code changes. It is the aim to reduce the execution time by minimizing the effort of changing the code.

As mentioned before, one of the goals is the use of AI tools for wind farm layout optimization. For this reason, a potential wind farm layout at the Bolund mountain is considered. The simulation is based on the solution of the Navier-Stokes equations using an LES turbulence model, where the computational mesh consists of 30 million elements. A snapshot of the solution is shown in Figure 1.

[16] CMake https://cmake.org

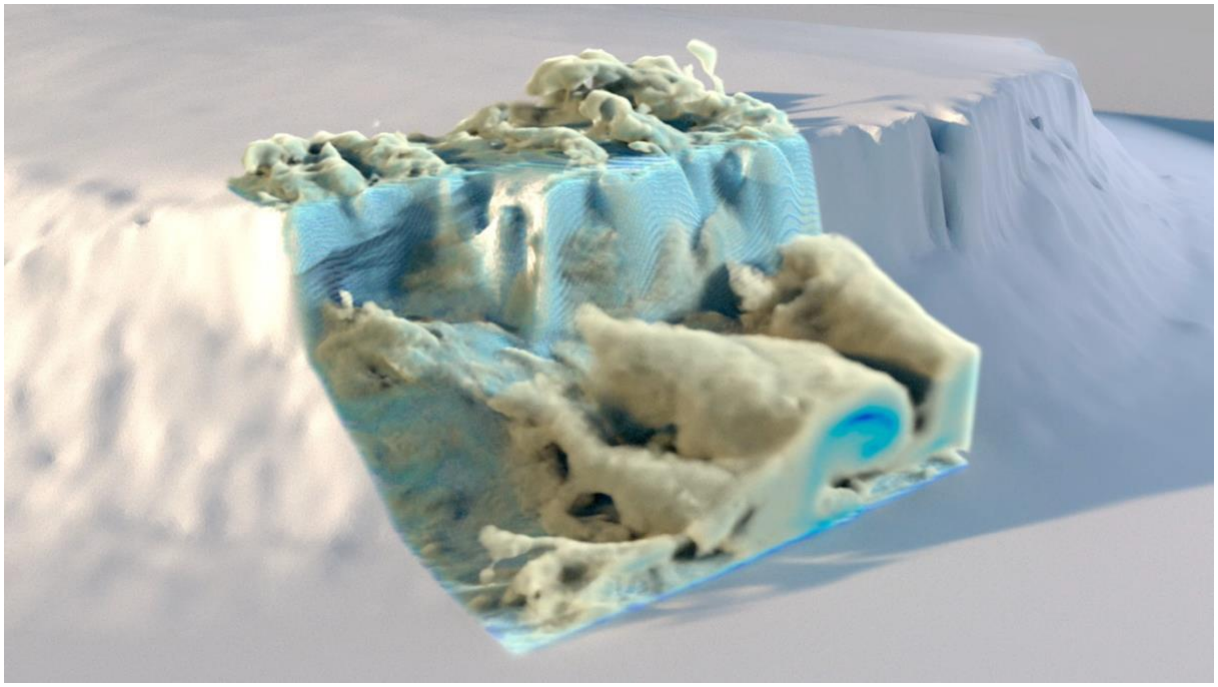

Figure 1: Snapshot of the flow solution (Q-vorticity contours) obtained on the Bolund mountain case with Alya.

As the computational pattern is repeated over time, only short executions on the JUWELS system, using 48 CPUs and 4 GPUs, are performed. The execution times for the 10 first solver iterations are shown in Figure 2.

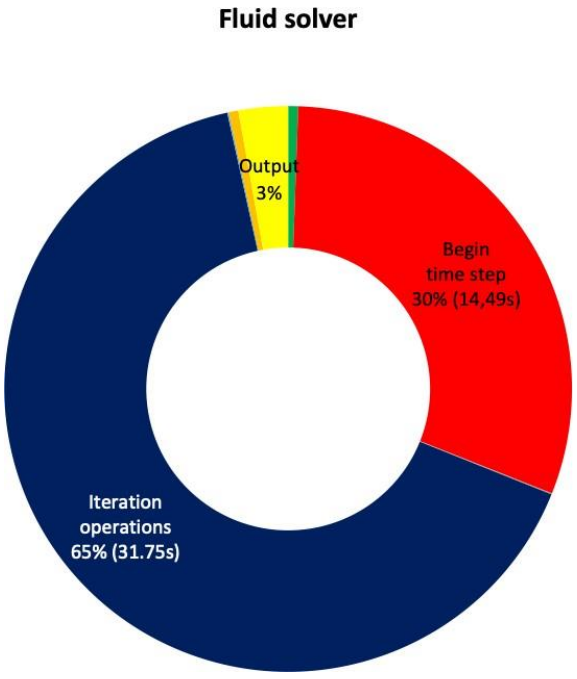

Figure 2: Alya CPU time before optimization of time step computation.

As it can clearly be seen, there are two main parts of the code that take almost all the time for solving the governing equations: the beginning of a time step in red and the iteration operations in blue.

In the context of this project, we use a fractional step method to solve the Navier-Stokes equations [3]. The finite element assembly consists of a loop over the elements of the mesh to

compute elemental right-hand sides, which are further assembled into a global right-hand side. With respect to the *Iteration operations*, most of the time is consumed in this right-hand side assembly operations (shown in blue). In the *Begin time step* part, the calculation of the critical time step (shown in red) takes almost all of execution time. This calculation is performed exclusively on CPUs.

In the following Sec. 3.3.1 and Sec. 3.3.2 the optimization of the critical time step computation and the element assembly are presented.

3.3.1 Optimization of the critical time step computation

As described in [2], both vectorization and OpenACC[17] portings are based on the same code, defining a `VECTOR_SIZE` at compilation time. The subsequently displayed algorithm illustrates the concept for the calculation of the element mass matrix `Ae`, where `Jac` is the Jacobian of the iso-parametric transformation including the weight of the Gauss-point `g` out of `ngaus` and `N(i,g)` is the shape function of node `i` out of `nnode` nodes at Gauss-point `g`. On the one hand, when using OpenACC (`OPENACC` is defined), the loop over elements is parallelized as `DEF_VECT=ivect;`. On the other hand, when vectorizing for CPUs, a bunch of `VECTOR_SIZE` elements is assembled at the same time, as defined by `DEF_VECT=VECTOR_SIZE`.

```
#ifdef  OPENACC
#define DEF_VECT ivect
use openacc
#else
#define DEF_VECT 1:VECTOR_SIZE
#endif

#ifdef OPENACC
!$acc data create(…)  &
!$acc copyin(…)
!$acc parallel loop gang vector default(present)
do ivect = 1,VECTOR_SIZE
#endif

do g = 1,ngaus
  do j = 1,nnode
    do i = 1,nnode
      Ae(DEF_VECT,i,j) = Ae(DEF_VECT,i,j)+Jac(DEF_VECT,g)*N(i,g)*N(j,g)
    end do
  end do
 end do

#ifdef OPENACC
end do
!$acc end parallel loop
!$acc end data
#endif
```

Thus, the parameter `VECTOR_SIZE` corresponds to the number of elements assembled at the same time on CPUs, and to the number of elements used to parallelize the OpenACC element

[17] OpenACC https://www.openacc.org

loop on GPGPUs. This value is on the order of 32-64 on CPUs to trigger vectorization on superscalar cores, and it is on the order of 10^5-10^6 on GPGPUs to guarantee efficient parallelism.

At present, the same vector size is used for CPU and GPGPU parallelization. Therefore, the acceleration of the element assembly (in blue) is carried out at the expense of the critical time step calculation (in red) which saturates the memory bandwidth by using a way too high `VECTOR_SIZE`. To overcome this issue, a second `VECTOR_SIZE` named `VECTOR_SIZE_CPU`, which is exclusive to CPU-based subroutines and used in the time step computation, has been defined in Alya. Note that in the future, this subroutine may be ported to OpenACC, as it was done for the element assembly, see next section.

3.3.2 Optimization of the element assembly

For the right-hand side assembly, physical properties are required at the Gauss-points of the elements. In the current version of Alya, these are computed at each time step in a separate loop, and then gathered from global arrays to element arrays during the assembly. To reduce the constraining data movement between CPU and GPGPU that is necessary for the assembly phase, on-the-fly calculations of such properties `Prop` have been implemented. The implementation follows the previously described parallelization/vectorization strategy. This strategy is shown in subsequently displayed algorithm. Corresponding results of a performance analysis will be reported in next Deliverable D2.3.

```
#ifdef  OPENACC
#define DEF_VECT ivect
use openacc
#else
#define DEF_VECT 1:VECTOR_SIZE
#endif

#ifdef OPENACC
!$acc data create(…)  &
!$acc copyin(…)
!$acc parallel loop gang vector default(present)
do ivect = 1,VECTOR_SIZE
#endif

do g = 1,ngaus
  Compute properties Prop(DEF_VECT,g)
end do

#ifdef OPENACC
end do
!$acc end parallel loop
!$acc end data
#endif
```

3.4 Performance analysis

The following Sec. 3.4.1 and Sec. 3.4.2 report on first performance analysis results for the optimized version of the time step computation in Alya and on the whole Alya workflow.

3.4.1 Alya: time step computation

For the performance analysis of the time step computation, the same problem as described in the previous section is considered. In this simulation, `VECTOR_SIZE` = 512000 is set, which is used by both the CPUs and GPGPUs. By changing the vector size of the CPUs to `VECTOR_SIZE_CPU` = 64, the times of executions as shown in Figure 3 are obtained.

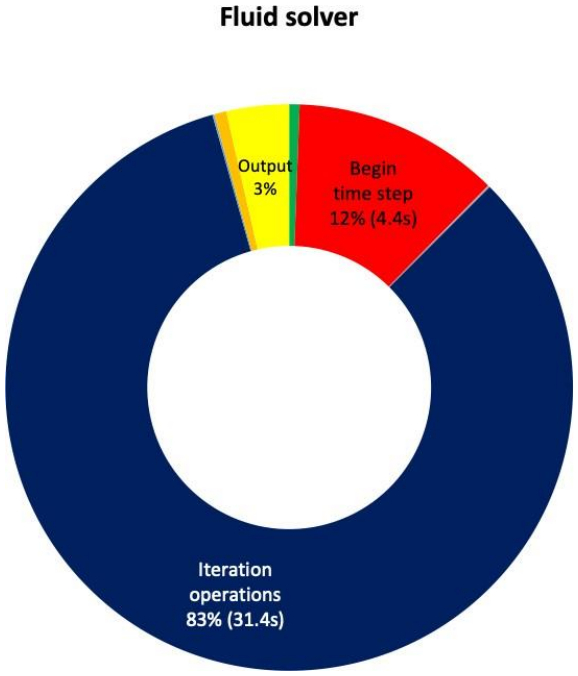

Figure 3: **Alya CPU time after optimization of the time step computation.**

Obviously, the optimization leads to a reduction of the time of the calculation of the time step by a factor of three with respect to the original version, which employed the same `VECTOR_SIZE` for both CPUs and GPGPUs. Considering that this operation is performed at each time step and that usually, for production runs, the simulations take thousands of time steps to finish, these code modifications will lead to a significant reduction of the total execution time of the use cases.

3.4.2 Workflow

The goal of using Alya on RTU's HPC system is to train a Machine Learning (ML) model that is able to predict wind turbine sink parameters `(x,y,z)` in such a way that the difference between the simulation results of Alya and the reference data (`label.data`) is minimized, see Figure 4.

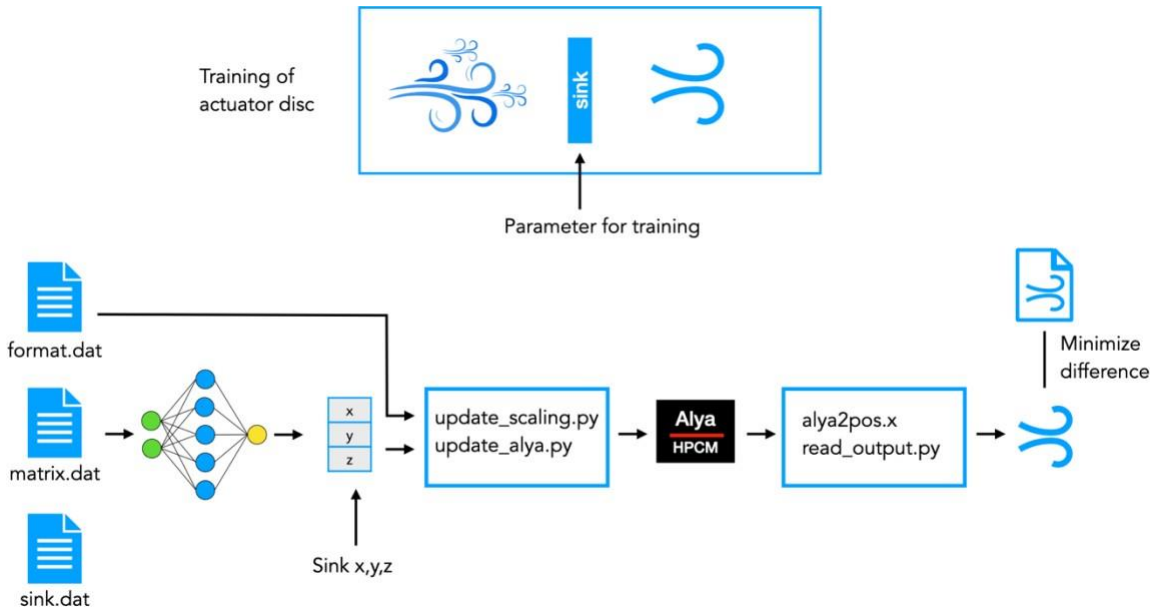

Figure 4: Alya workflow for the training phase.

At the current stage, the Alya workflow is tested without involving ML computations. To run the Alya workflow, the following batch script should be submitted to the HPC's batch system (example for running case-075 test case on 8 EPYC cores using PBS):

```
#!/bin/bash
#PBS -N alya_workflow
#PBS -q batch
#PBS -A coe_raise
#PBS -l nodes=1:ppn=64,feature=epyc
#PBS -l walltime=00:30:00
#PBS -j oe

cd $PBS_O_WORKDIR
./alya_workflow.sh
```

The content of the file `alya_workflow.sh` is as follows:

```
#!/bin/bash

module load gnu8/8.3.0
module load mpi/openmpi-4.1.1

STARTTIME=$(date +%s)

python3 update_scaling.py
python3 update_alya.py

...
```

```
...

cd case/case-075
mpirun ./alya case-075

./alya2pos case-075

cd ../..
python3 read_output.py

ENDTIME=$(date +%s)
echo "It takes $(($ENDTIME - $STARTTIME)) seconds to complete alya
task..."
```

The average execution times of the Alya core workflow on RTU's HPC system are shown in Table 2 for varying CPU core counts. It should be pointed out that the mesh of this test case is coarse, i.e., it only contains 41,805 hexahedra. This case has been selected for development and not for optimization purposes.

CPU cores	Execution time	Speed-up ratio	Parallel efficiency
1	996	1	-
4	437	2.3	57%
8	223	4.5	56%

Table 2: Preliminary performance testing results of the Alya workflow on RTU HPC EPYC nodes using case-075 test case data.

A second series of tests has been carried out on the Bolund mesh. The performance results on 16, 32, and 64 CPU cores and shown in Table 3. Obviously, this case is much better suited for a large number of cores. As expected, the parallel efficiency is higher than in the previous case.

CPU cores	Execution time	Speed-up ratio	Parallel efficiency
16	408	1	-
32	207	1.97	99%
64	125	3.26	82%

Table 3: Performance testing results of Alya on RTU HPC EPYC nodes using Bolund test case data.

4 Porting and performance analysis of m-AIA from RWTH

The m-AIA code developed at RWTH was ported to GPGPUs on heterogeneous systems such as the JURECA-DC and JUWELS-BOOSTER modules at JSC, FZJ. The shared memory OpenMP parallelization of the structured Finite Volume (FV) solver of m-AIA was replaced by an implementation using the Parallel Standard Template Library (PSTL)[18].

The PSTL is an implementation of the C++ standard library algorithms extending the execution policies by parallel and Single Instruction-Multiple Data (SIMD) optimizations. It offers efficient support for both a parallel and vectorized execution of algorithms.

A brief overview of m-AIA, a description of the necessary porting environments, and details on the PSTL implementations are subsequently given in Sec. 4.1, Sec. 4.2 and Sec. 4.3. Section 4.4 presents the results of a performance analysis, juxtaposing the performance of a pure CPU implementation to the accelerated GPGPU port.

4.1 Overview of m-AIA

The simulation code m-AIA is a multi-physics framework based on C++. It is developed at RWTH and FZJ provides support by means of further numerical method implementations and performance engineering. Detailed information on m-AIA can be found in Deliverable D2.6 “Support report” of CoE RAISE. The m-AIA code contains several different modules to solve, e.g., compressible and incompressible flows, particle-laden flow, aeroacoustics, and moving boundary problems. The framework operates both on hierarchical Cartesian meshes that are generated with a massively parallel grid generator as part of m-AIA as well as with structured curvilinear meshes.

The computation of turbulent boundary layer flows controlled by active wall movements is performed using the structured FV framework of m-AIA. For this application, the grid generation is simple, and the structured memory layout allows for a straightforward optimization of the numerical algorithm. The objective of the performance engineering of m-AIA within Task 2.1 is to enable m-AIA for GPGPU execution. Therefore, all routines called from the main loop, i.e., functions that are called in every iteration step, are ported to a GPU-parallelization-ready structure. Above all, this involves replacing the original loops over all cells with new PSTL loops. The details of this porting activity are found below.

4.2 Setting up the environment for porting m-AIA to GPGPUs

The GNU Compiler Collection (GCC)[19] from the NVIDIA HPC (NVHPC)[20] Software Development Kit SDK is used to compile m-AIA with GPGPU acceleration. The inter-process communication is taken care of using the ParaStation MPI[21] implementation. The discrete Fourier transformations required by m-AIA are handled with an external software library, the Fastest Fourier Transform in the West (FFTW)[22]. On the JURECA-DC and JUWELS-BOOSTER heterogeneous systems, NVHPC with ParaStation MPI can be loaded as modules with the command below:

[18] Parallel STL
https://www.intel.com/content/www/us/en/developer/articles/guide/get-started-with-parallel-stl.html/
[19] GCC https://gcc.gnu.org/
[20] NVHPC https://developer.nvidia.com/hpc-sdk/
[21] Parastation MPI https://docs.par-tec.com/html/psmpi-userguide/index.html
[22] FFTW http://www.fftw.org/

```
ml NVHPC/21.1-GCC-9.3.0 ParaStationMPI
```

Loading these modules enables the system to use NVIDIA's optimized GNU compiler `nvc++`, tuned especially for heterogeneous HPC systems. Note that a newer NVHPC/21.9-GCC-10.3.0 exist in the software stack of the system but requires CUDA[23] with a version 11.4 that is not yet available in the software stack. Parallel I/O is performed using the Hierarchical Data Format version 5 (HDF5[24]) or the parallel Network Common Data Form (parallel-NetCDF[25]) libraries. Furthermore, an in-situ interface has been integrated to connect to in-situ data processing tools. The compilation process of m-AIA is automated using the CMake software. The aforementioned software can be loaded on JURECA-DC and/or JUWELS-BOOSTER via:

```
ml FFTW HDF5 parallel-netcdf CMake
```

The configuration of m-AIA with NVHPC and HDF5 support can be performed by issuing the Python-3.x-based configuration script, via:

```
python3 configure.py nvhpc production --enable-pstl=ampere \
                                      --with-hdf5iolib
```

This command uses the settings as specified in the `cmake` file corresponding to the present system, e.g., the `jureca.cmake` file, which can be found in `$PATH_TO_MAIA/auxiliary/hosts` directory of the m-AIA source code directory. This modified configuration script is made available for further use as part of a GIT-branch created for CoE RAISE, which is forked from the main m-AIA development repository, accessed through an invitation by the RWTH group from the given link[26]. Further information on how to create and/or modify this settings file can be found in D2.6. Upon successful configuration, the `make -j` command is issued to compile the m-AIA code.

4.3 Optimization of m-AIA via GPGPU porting

In the structured FV module of m-AIA, only a few functions in the main loop occupy most of the computational time. These functions include the computation of the convective and viscous flux terms on the governing equations, the time integration of the governing equations using the Runge-Kutta method, and the exchange of information between computational discretized domains (namely subdomains) via MPI. In the following, the exemplary modification of PSTL to the function that computes the convective fluxes is shown, noting that the procedure is equivalent for all aforementioned functions.

Originally, the convective flux is computed within four nested for-loops, the most outer one looping over the spatial dimensions (in this case three). All following loops iterate over the number of computational grid points (namely cells) in the respective direction `I,J,K`. This allows these loops to cover the active inner cells and to omit all non-active ghost cells on the

[23] CUDA https://developer.nvidia.com/cuda-toolkit/
[24] HDF5 https://www.hdfgroup.org/solutions/hdf5/
[25] parallel-NetCDF https://parallel-netcdf.github.io/
[26] GIT for mAIA with pSTL https://git.rwth-aachen.de/aia/MAIA/Solver/-/tree/structured_pstl

boundaries. In an ideal case, the PSTL-parallelized loop should iterate over a large number of elements, e.g., all cells in all directions. The nested loops, however, inherently lead to an iteration over small chunks of elements. Below is an example of such a function, where first the cell-surfaces values are reconstructed using a Monotone Upstream Centered Scheme for Conservation Laws (MUSCL) type strategy. Then, the cell-surface flux is computed with the Advection Upstream Splitting Method (AUSM).

```
for(MInt dim = 0; dim < nDim; dim++) {
  for(MInt k = 1; k < noCellsK-1; k++) {
    for(MInt j = 1; j < noCellsJ-1; j++) {
      for(MInt i = 1; i < noCellsI-1; i++) {

        // extrapolation of variables to cell-surfaces (MUSCL)
        …

        // computation of convective flux with extrapolated
        // cell-surface values (AUSM)
        …
        }
    }
  }
}
```

Here, the number of cells in a direction is denoted as `noCells` with `I,J,K`. This nested loop structure has been replaced by two loops as it is shown in the code below, i.e., loop unrolling has been performed.

```
for(MInt dim = 0; dim < nDim; dim++) {
  …
  for(MInt I=start1D; I < end1D; ++I) {
    // MUSCL
    // AUSM
  }
}
```

For each direction, an individual start and endpoint are determined, and the inner loop then iterates over nearly all cells, omitting only a few unnecessary cells at the very beginning and at the end. Although this is less efficient than the four nested loops shown above as `start1D>(noCellsI*noCellsJ*noCellsK)`, since the flux is also computed unnecessarily for several non-active ghost-cells, this loop structure is easily parallelizable with PSTL, as seen below:

```
for(MInt dim = 0; dim < nDim; dim++) {
  ...
#ifdef MAIA_NVHPC_COMPILER
  auto begin_ = thrust::counting_iterator(MInt{start1D});
#else
  auto begin_ = std::ranges::views::iota(MInt{start1D}).begin();
#endif

...
```

```
...

  std::for_each_n(std::execution::par_unseq, begin_,
    end1D, [=](MInt I) {
     // MUSCL
     // AUSM
   });
}
```

The inner `for`-loop has been replaced by the `for_each_n`-loop together with the `par_unseq` parameter such that the inner loop content is converted to a lambda function, which can be executed in parallel and in a non-sequential order on the GPGPU. When using GPGPUs, it is recommended to employ fewer CPU ranks to allow the PSTL-loop to iterate over several million elements.

All other `for`-loops in functions called from the main-loop of the m-AIA structured FV solver are converted similarly. Due to the acceleration of compute-intensive functions via GPGPU execution, functions previously consuming only a fraction of the overall main-loop time now exert a larger weight. That is, every non-PSTL-treated `for`-loop over all elements being executed solely on the CPU (such as resetting the right-hand side to zero) turns into the new execution bottleneck. Therefore, all functions looping over a significant number of elements need to be PSTL-parallelized to achieve a global speedup.

4.4 Performance analysis of the m-AIA GPGPU port

For the sake of the length of this document, only a few important results are presented in the following. First, the strong scaling results of m-AIA using only CPUs or the CPU / GPGPU acceleration on the JUWELS-BOOSTER module are discussed. A Taylor-Green Vortex (TGV) case with 16.7 million computational elements in three directions is chosen for the benchmarks. This simulation employs the finite-volume solver of m-AIA and uses a structured computational mesh. The contour plot of such a TGV simulation's axial velocity data is presented in Figure 5.

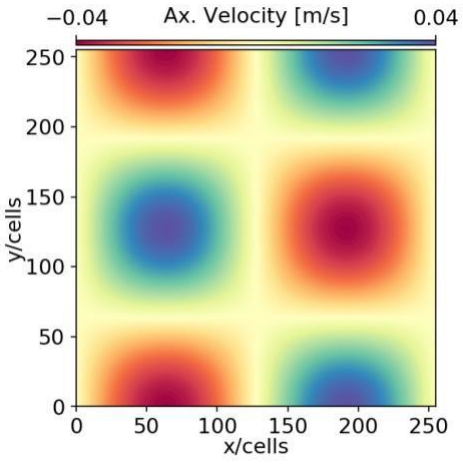

Figure 5: Contour plots of the axial velocity on the middle plane of the TGV with 16.7 million computational elements.

Figure 6 shows the strong scaling results of the TGV simulation using m-AIA compiled with NVHPC on the JUWELS-BOOSTER module. The computational time per iterative step is plotted over the number of ranks, i.e., the time for a complete Runge-Kutta loop to integrate the time derivative of the employed transport equation is measured. More information regarding this topic can be found in classical textbooks, e.g., in [4]. The relative performance is computed based on the slowest simulation. Presented are the performances of the pure CPU computation (denoted as m-AIA CPU) and the GPGPU accelerated computation (denoted as m-AIA GPU). In an ideal case, the computational domain is decomposed into subdomains, where each subdomain is attained to a CPU thread. In the current implementation, each subdomain employs a single GPGPU. Thus, this implementation does not utilize the complete number of available CPU threads. For example, a JUWELS-BOOSTER node consists of two CPUs with a total of 48 CPU hardware threads and 96 Simultaneous Multithreading (SMT) threads. To use all resources efficiently, the computational domain should be subdivided into 96 subdomains. As each node features four GPGPUs, the current implementation makes only use of four threads yielding unoccupied 92 SMT threads. This

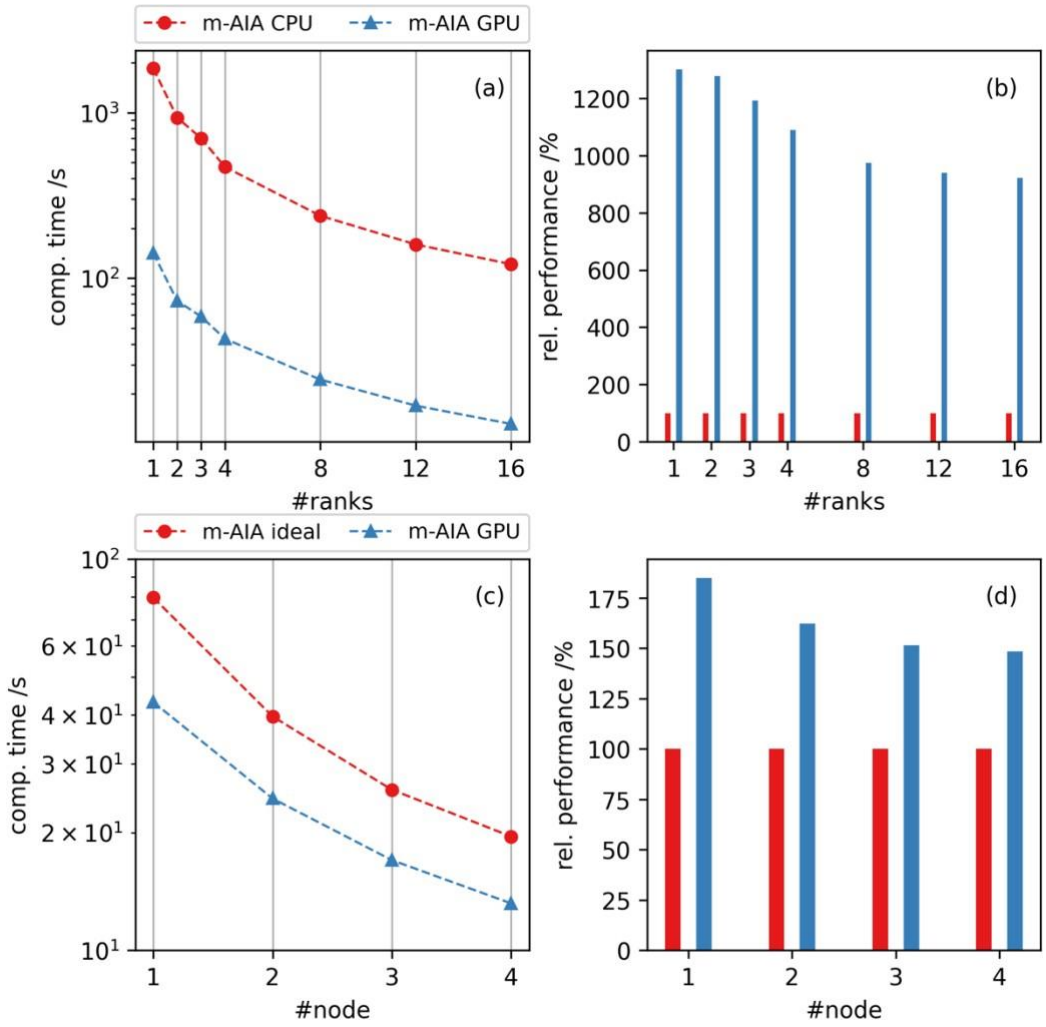

Figure 6: Performance of m-AIA to simulate a TGV case with 16.7 million elements on the JUWELS-BOOSTER module using only CPUs (m-AIA CPU and m-AIA ideal) and both CPUs with GPGPU acceleration (m-AIA GPU). The m-AIA CPU in the top row employs the same number of threads as the m-AIA GPU variant. The ranks denote the number of subdomains, i.e., each rank uses a single SMT thread with a single GPGPU. The bottom row utilizes all the SMT threads (in this case 96). Note the bottom row has the number of nodes on the x-axis.

limitation is a result of the current implementation, where each subdomain can only utilize a single SMT thread and a single GPGPU. Note that developments for co-execution, i.e., using both resources CPUs and GPGPUs more efficiently in a concurrent manner, are ongoing. For a fair comparison, the results of the m-AIA CPU case use only four CPU threads, noting that allocating multiple SMT threads to a single subdomain is currently not possible when GPGPU acceleration is utilized.

It is evident from Figure 6 that employing the GPGPU acceleration greatly reduces the computational time. The relative performance plot depicted in Figure 6b reveals the m-AIA GPU version to be at least a factor of 10 faster than the conventional m-AIA CPU version when the number of CPU threads is set equally. An additional analysis is performed by comparing the GPGPU accelerated m-AIA GPU code with the standard m-AIA implementation that utilizes 96 SMT threads per node, i.e., 96 subdomains per node. The latter is denoted as m-AIA ideal. Utilizing all of the CPU threads in a m-AIA CPU execution (m-AIA ideal) greatly reduces the overall computation time, see Figure 6c. The time-to-solution using the GPGPU acceleration is, however, still shorter. The relative performance of m-AIA with GPGPU acceleration to m-AIA with pure CPU computation utilizing all the CPU threads shows 50% better performance, see Figure 6d. The power consumption using these additional GPGPUs is yet to be investigated. Different co-execution strategies to utilize the unused CPU threads are currently being discussed.

5 Porting and performance analysis of AI technologies

One of the goals of CoE RAISE is to couple the aforementioned CFD / multi-physics codes to AI frameworks to achieve intertwined and efficient simulation, surrogate and modeling, and data processing workflows. This section first gives an overview of the various datasets used in ML applications in Sec. 5.1. Section 5.2 discusses strategies of porting such AI frameworks to heterogeneous systems such as the JUWELS, JURECA, and the DEEP-EST systems at the Jülich Supercomputing Centre (JSC), HAWK at the High-Performance Computing Center Stuttgart (HLRS), PizDaint at the Swiss National Supercomputing Center (CSCS), and the CT-AMD system at BSC. The presentation of these strategies is complemented by the results of performance analyses of the AI frameworks in Sec. 5.3.

5.1 Overview of datasets used in ML benchmarks

For the sake of the length of this document, only three important datasets are introduced in this section. The first two datasets ImageNet and modified National Institute of Standards and Technology (MNIST) are two famous datasets freely available for ML research. The third dataset is an example from a compute-driven use-case introduced in Task 3.1 *"AI for turbulent boundary layers".* A brief overview of these datasets is given below.

ImageNet: ResNet

The ImageNet dataset was first introduced as a benchmarking dataset at the ImageNet Large Scale Visual Recognition Challenge in 2012 [5]. It contains 1,281,167 images in the training set and 50,000 images in the validation set. In total, there are 1,000 classes that these images are assigned to. The main goal of the challenge is to train algorithms on the training set to accurately predict the classes of the images in the validation set. Uncompressed, the total dataset has a size of approximately 300 GB. Over the past few years, it has become the most used benchmarking dataset for computer vision applications.

Convolutional Neural Networks (CNNs) have shown great performance on the task of predicting the ImageNet classes. From the original structure presented in the AlexNet [6], continuous improvements were made, yielding the ResNet architecture [7]. Residual Nets feature so-called short skip connections that omit some layers, making it possible to train deeper networks that would otherwise suffer from degradation [7]. The original ResNet50 architecture has 50 layers of neurons and is despite recent advancements in the field of Transformers still one of the standard benchmarking architectures in computer vision.

MNIST: CNN

The modified National Institute of Standards and Technology (MNIST) dataset [8] consists of a collection of handwritten digital images used for character recognition. This dataset is an extension to the original dataset available from the National Institute of Standards and Technology (NIST)[27]. The dataset consists of 60,000 example digital images for training and 10,000 examples for testing purposes. Each example image is centered and represented in fixed 28x28 black and white pixels. The grayscale levels are introduced to each image using anti-aliasing techniques.

Many ML techniques have been tested to train this dataset, such as linear classifiers, k-nearest neighbors, boosted, stumps, non-linear classifiers, support vector machines, neural nets, and

[27] NIST https://www.nist.gov

CNNs, where an overview can be found in [8]. As this dataset has proven to be useful in testing ML performance, this work uses this dataset for CNN benchmarks.

ATBL: Auto-encoder

The Actuated Turbulent Boundary Layer (ATBL) dataset is generated in the compute-driven use-case Task 3.1 *"AI for turbulent boundary layers".* Briefly, the training dataset is generated using a high-resolution large-eddy simulation (LES) approach with a moving geometry to create oscillating boundary layers, which reduces the friction drag of the turbulent boundary layer. Further information on the geometrical setup, the simulation method, and the first results can be found in Deliverable 3.1.

To train networks via ML, an 8.3 TB dataset is extracted from the LES of the ATBL. Moreover, a smaller dataset is generated based on this large dataset for development purposes. This smaller dataset consists of 21 GB of data for training and additional 1.2 GB of data for testing purposes (a total of 22.2 GB).

The initial ML framework to be tested is a Convolutional Auto-Encoder (CAE). CAEs are unsupervised neural network models that summarize the general properties of the input dataset in fewer parameters while learning how to reconstruct the input again after compression, namely decompression [9]. Due to their simple implementation, CAEs are widely used for reducing the dimensionality of any dataset. More information on this topic can be found in Deliverable D2.14, where the details are omitted for brevity.

Training a CAE with large datasets is computationally challenging and can only be performed efficiently when parallelization strategies are exploited. A common parallelization strategy is to distribute the input dataset to separate GPGPUs, where the trainable parameters between the GPGPUs are exchanged occasionally. This method is called distributed data parallelization and greatly reduces the training time. Depending on the data exchange rate between the CPUs and/or GPGPUs, this type of parallelized training can scale to many workers of CPUs or GPGPUs. Currently, the GPGPUs require the CPUs to access the input data. Since the data needs to be transferred from the CPUs to the GPGPUs, the performance of the CPUs becomes the limiting factor in CAE training.

It should be noted that CAE training can be performed using only CPUs. It is, however, preferred to run this on GPGPUs as their architecture allows for much faster ML-typical matrix-matrix operations. This way, in slow training times can be avoided.

The only drawback of the data parallelization strategy is the loss in training accuracy that is caused by the increased batch sizes. As the input dataset is distributed to separate workers, the total batch size linearly increases by the number of workers - even though the batch size per worker remains fixed. That is, in a data-parallel training with a large number of workers, the batch size inevitably becomes large, which leads to reduced training accuracies. This limits the linear scaling performance of the CAE training and renders the training accuracy an important factor when investigating the scaling performance of a CAE training.

The loss of training accuracy and how to cope with it has intensively been addressed in the literature [10,11,12]. Tuning other hyperparameters such as the learning rate, batch size per worker, and the number of epochs can be adjusted to keep the training accuracy at a certain level when the number of workers is increased, as done in previous studies [10,11,12].

5.2 Porting existing ML frameworks to heterogeneous systems

This part focuses on porting existing ML frameworks to different heterogeneous systems. A change in the structure of these frameworks is not required, as these frameworks are already optimized for both CPU and/or GPGPU. The CAEs using distributed data parallelization methods are developed with the open-source framework PyTorch 1.10.0[28]. There are several frameworks that integrate distributed data parallelization to PyTorch, where the most popular ones are ported and investigated in this project:

- Distributed Data Parallel (DDP) module as part of a PyTorch package [13]
- Horovod distributed training package, developed by Uber [14]
- Helmholtz Analytics Toolkit HeAT, a project of the Helmholtz association [15]
- DeepSpeed, developed by Microsoft [16]

At a macroscopic scale, these distributed data parallel frameworks operate similarly. However, at a microscopic scale, each framework is optimized differently leading to different scaling performances and training accuracies for individual cases.

In the following, a brief overview of the used frameworks is given in Sec. 5.2.1. Subsequently, the necessary porting of these methods is exemplarily explained for the systems JURECA in Sec. 5.2.2. Finally, general initialization methods are presented in Sec. 5.2.3 along with a specific configuration for Rudens at RTU. It should be noted that porting these frameworks to further systems is similar.

5.2.1 A brief overview of used ML frameworks

PyTorch Distributed Data Parallel (PyTorch-DDP) [13]

PyTorch is an open-source ML framework, mainly developed by Facebook AI Research. The PyTorch-DDP module features a built-in way to run distributed training of neural networks on multiple workers. Similar to Horovod, the PyTorch-DDP library also uses an `AllReduce` operation for gradient reduction. Additionally, the gradients are not synchronized individually. Instead, several gradients are collected in ‘buckets’ and are only communicated when the buckets are sufficiently filled. This reduces the total number of communication operations. In contrast to Horovod, PyTorch-DDP uses a simple heuristic for determining the reduction order on a local worker level: The gradients are bucketed in the reverse order of their computation in the forward pass of the network. This is motivated by the fact that the last layers of a network are likely the first to finish computation during the backward pass. To further speed up the training process, PyTorch-DDP offers a `no_sync` option to only execute an `AllReduce` operation every few iterations.

Horovod [14]

Horovod is an open source distributed Deep Learning (DL) library originally developed by Uber for TensorFlow. It supports most other DL frameworks such as PyTorch, TensorFlow[29], Keras[30], and Apache MXNet[31]. Horovod can be included into existing training script with only a few lines of code changes rendering it a popular choice when it comes to distributed DL. It was one of the first libraries to make use of a decentralized `Ring_AllReduce` scheme for the

[28] PyTorch https://pytorch.org/
[29] TensorFlow https://www.tensorflow.org
[30] Keras https://keras.io
[31] MXNet https://mxnet.apache.org

synchronization of the gradients, whereas most of the other frameworks at the time were using a centralized parameter server. On a local worker level, Horovod handles communication operations asynchronously via a separate background thread. To ensure consistency across all workers on a global scale, a consistency protocol is enforced. In literature, it has been shown that the combination of background thread and consensus protocol leads to huge communication overhead, acting as a drain on performance [17].

HeAT [15]

The Helmholtz Analytics Toolkit HeAT is an open-source distributed DL library developed by partners from Deutsches Zentrum für Luft- und Raumfahrt - German Aerospace Center (DLR), FZJ, and Karlsruhe Institute of Technology (KIT). HeAT is a flexible and seamless open-source software for high-performance data analytics and ML. It provides highly optimized algorithms and data structures for tensor computations using CPUs, GPGPUs, and distributed cluster systems using MPI for communication. The objective of HeAT is to fill the gap between data analytics and ML libraries with a strong focus on single-node performance on the one hand, and traditional HPC on the other [12].

DeepSpeed [16]

DeepSpeed is an open-source distributed DL library originally developed by Microsoft. The library is designed to reduce computational effort and memory usage, and to train large distributed models with better parallelism on existing computer hardware. DeepSpeed is optimized for low latency, high throughput training. It includes the Zero Redundancy Optimizer (ZeRO)[32] for training models with 100 billion or more parameters, especially useful for CNNs and Natural Language Processing (NLP).

5.2.2 Porting ML frameworks

Porting these frameworks to heterogeneous systems is similar. Therefore, necessary porting steps are in the following exemplarily shown on the DC module of the JURECA system at FZJ. This system consists of 192 accelerated compute nodes, each equipped with dual AMD EPYC 7742 CPUs and four NVIDIA A100 GPU - a total of 24,576 cores and 768 GPGPUs. The nodes are connected to each other with dual InfiniBand HDR switches.

The distributed data parallel frameworks require the MPI library and Python with a version *3.x*. In case the ML training is to be accelerated with GPGPUs, either CUDA or ROCm[33] libraries are required, depending on the GPGPU's manufacturer. On the JURECA-DC system, GCC, Parastation MPI, CMake, and Python[34] frameworks are available to the user as modules, and loaded as:

```
ml GCC ParaStationMPI Python
```

This command implicitly loads GCC 10.3, ParaStationMPI 5.4.10-1, Python 3.8.5, CUDA 11.3 frameworks, and explicitly loads UCX 1.10.1[35], CUDA 11.3 and the cuDNN 8.2.1.32 libraries[36].

[32] ZeRO https://www.deepspeed.ai/tutorials/zero/
[33] ROCm https://github.com/RadeonOpenCompute/ROCm
[34] Python https://www.python.org/
[35] UCX https://openucx.org/
[36] cuDNN https://developer.nvidia.com/cudnn

These modules alone satisfy the basic installation of the distributed data parallel frameworks. However, to enable all the framework options such as the NCCL communication backend[37], the following module needs to be loaded:

```
ml NCCL
```

This provides the user with NCCL 2.10.3 with CUDA support. Each parallel framework can be compiled on JUWELS from the source code. However, the Python package installer pip[38] or the open-source package management system Conda[39] greatly reduce the complexity to compile such frameworks. Alternatively, a slightly older PyTorch 1.8.1 is available as a module on the system. In the following, pip is chosen for the sake of simplicity. Initially, it is wise to create a Python environment with a name `<env_name>` and source it:

```
python3 -m venv <env_name>
source <env_name>/bin/activate
```

This keeps the management of the compiled frameworks simple. The first required library is PyTorch, which can be compiled with these commands (noting the installed CUDA version of 11.3 on the system):

```
pip3 install torch==1.10.0+cu113 torchvision==0.11.1+cu113 \
     torchaudio===0.10.0 \
     -f https://download.pytorch.org/whl/torch_stable.html
```

The PyTorch framework already contains all the required libraries for the DDP module to run, i.e., no additional compilation of this module is necessary.

The second distributed data parallelization framework Horovod can be compiled with these commands:

```
export HOROVOD_GPU=CUDA
export HOROVOD_GPU_OPERATIONS=NCCL
export HOROVOD_WITH_PYTORCH=1
pip3 install horovod
```

The first three environmental variables setup CUDA for GPGPUs, NCCL for communication backend between workers, and PyTorch for the host language. It should be noted that the first two flags are only necessary for GPGPU accelerated training, and for AMD GPGPUs, ROCm replaces CUDA and RCCL[40] replaces NCCL.

The third distributed data parallelization framework HeAT can be compiled with these commands:

[37] NCCL https://developer.nvidia.com/nccl
[38] pip https://pypi.org/project/pip/
[39] Conda https://docs.conda.io/
[40] RCCL https://github.com/ROCmSoftwarePlatform/rccl

```
pip3 install heat[hdf5,netcdf]
```

The optional arguments `hdf5` and `netcdf` denote the input and output capabilities of HeAT. These arguments are important since most of the input dataset is stored in HDF5 or parallel-NetCDF formats. The I/O libraries need to be preloaded by issuing:

```
ml HDF5 parallel-netcdf
```

The final distributed data parallelization framework DeepSpeed can be compiled with these commands:

```
export DS_BUILD_FUSED_ADAM=1
export DS_BUILD_UTILS=1
pip3 install DeepSpeed
```

The first two environmental variables are responsible to compile helper utilities of DeepSpeed required during CAE training. For pre- and post-processing the data, and for advanced CNN commands (such as three-dimensional convolutions), the additional Python libraries Pillow[41], pyparsing[42], python-dateutil[43], matplotlib[44], h5py[45], and pytorch-nlp[46] need to be included to the Python environment by

```
pip3 install Pillow pyparsing python-dateutil matplotlib h5py \
     Pytorch-nlp
```

5.2.3 Initialization of used frameworks

The following paragraph, describes the general approach on the JURECA-DC system to initialize ML frameworks. Next paragraph provides examples on how to perform parallel training using PyTorch-DDP on the Rudens system at RTU.

Initialization of ML frameworks on JURECA-DC at FZJ

These four distributed data parallelization frameworks are initialized using various methods. For the DDP framework, an *elastic launch* framework that is a part of the PyTorch library is used (previously known as *distributed launch*). This elastic launch framework enables distributed training jobs to be executed on multiple workers on a single or multiple-node, where in case of a failed worker or node, a new worker replaces the faulty one. Unfortunately, this type of initialization is not favored by the slurm[47] job scheduler (typically found in many HPC systems) as the user must reserve the workers and nodes in advance. For this purpose, the

[41] Pillow https://pillow.readthedocs.io/en/stable
[42] pyparsing https://github.com/pyparsing/pyparsing
[43] python-dateutil https://dateutil.readthedocs.io/en/stable/
[44] matplotlib https://matplotlib.org
[45] h5py https://www.h5py.org
[46] pytorch-nip https://pytorch.org/tutorials/beginner/deep_learning_nlp_tutorial.html
[47] Slurm https://slurm.schedmd.com/

initialization without elasticity can be used, as given below with the help of slurm's environment variables:

```
torchrun \
  --nnodes=$SLURM_NNODES \
  --nproc_per_node=$SLURM_GPUS_PER_NODE \
  --rdzv_id=$SLURM_JOB_ID \
  --rdzv_backend=c10d \
  --rdzv_endpoint=$SLURMD_NODENAME.jureca:<free_TCP_port> \
  <training_Script>.py (arguments)
```

Several keyword arguments must be provided to the `torchrun` framework, such as the number of nodes, the number of GPGPUs per node, a unique job identification number, the used collective communication library `c10d`[48], and the host (or master) node address with a free Transmission Control Protocol (TCP) port, in respective order.

The DeepSpeed framework utilizes a similar initialization method as the PyTorch-DDP framework, i.e., the `deepspeed.launcher` framework, where an example is given below:

```
python3 -m deepspeed.launcher.launch \
  --node_rank $SLURM_PROCID \
  --master_addr $SLURMD_NODENAME \
  --master_port <free_TCP_port> \
  --world_info <list_of_workers_nodes_in_Base64> \
  <training_Script>.py (arguments) \
  --deepspeed_mpi \
  --deepspeed_config <DS_config.json>
```

Here, the `deepspeed.launcher` framework requires several keyword arguments such as the rank of the node, the host (or master) node address, a free TCP port of the host (master) node, and the list of the workers and nodes in `Base64` format. The latter two arguments tell the DeepSpeed framework to run in parallel and provide it with a special configuration file `<DS_config.json>`. The `--world_info` argument with the list of the workers and nodes in *Base64* format can be defined in the batch script used on the JURECA system by:

```
sysN=$(eval "scontrol show hostnames")
for i in $sysN; do
  x+=\"$i\":[$CUDA_VISIBLE_DEVICES],
done
list_of_workers_nodes_in_Base64=`echo {${x::-1}} | base64 -w 0`
```

The other two distributed data parallelization frameworks Horovod and HeAT do not require a special initialization, i.e., the ML training scripts can simply be executed by the command:

```
python3 -u <training_Script>.py
```

[48] `c10d` https://pytorch.org/docs/stable/distributed.html

Example CAE training scripts using various datasets such as MNIST, Image Net, and the ATBL datasets are documented and can be found in the master branch of a GIT repository[49]. This repository also contains machine-specific batch scripts for the heterogeneous systems JURECA, JUWELS, DEEP-EST, CT-AMD, and automated compilation scripts for the distributed data parallelization frameworks mentioned above.

Using PyTorch-DDP on the Rudens system at RTU HPC

To reduce the training time, the PyTorch models are mostly trained on multiple GPGPUs within a single node or across different nodes. An example to run PyTorch-DDP on two nodes, each having two Tesla V100 Volta GPGPU, using the batch script `pytorch_multinode.sh` and helper script `pytorch_worker.sh` is given below. The batch script `pytorch_multinode.sh` executes (spawns) the PyTorch-DDP code on all defined nodes under the control of PBS. The content of `pytorch_multinode.sh` is:

```
#!/bin/sh
#PBS -N pytorch_multinode_job
#PBS -q batch
#PBS -A coe_raise
#PBS -l nodes=2:ppn=2:gpus=2,feature=v100
#PBS -j oe

pbsdsh -u -v "$PBS_O_WORKDIR/pytorch_worker.sh"
```

Several keyword arguments for `qsub` are required, such as the following:

- `PBS -N` - declares the name for the job.
- `PBS -q` - defines the destination of the job. The destination names a queue, a server, or a queue at a server.
- `PBS -A` - defines the project account to allocate the computational resources.
- `PBS -l` - denotes the requested computational resources.
- `PBS -j` - declares if the standard error stream of the job will be merged with the standard output stream of the job. An option argument value of `oe` directs that the two streams will be merged, intermixed, as standard output.

The batch script `pytorch_multinode.sh` calls the helper script `pytorch_worker.sh` on each node for PyTorch-DDP to be executed on that node. The content of `pytorch_worker.sh` is:

```
#!/bin/sh

module load cuda/cuda-10.2 conda
source activate raise_conda_env

cd $PBS_O_WORKDIR
python3 <ddp_gpu.py> > <log_file>
```

[49] GIT trainig scripts https://gitlab.jsc.fz-juelich.de/CoE-RAISE/FZJ/ai-for-hpc.git

The script `pytorch_worker.sh` sources Conda environment that includes required Python libraries. Creating the Conda environment can be performed simply by initiating the commands below.

```
module load cuda/cuda-10.2 conda
conda activate
conda install -y tempfile pytorch torchvision torchaudio /
      cudatoolkit=10.2 -c Pytorch
```

The content of the exemplary training script `ddp_gpu.py` is:

```
#!/usr/bin/env python

import os, sys, tempfile, torch
import torch.distributed as dist
import torch.nn as nn
import torch.optim  as  optim
import torch.multiprocessing as mp
from torch.nn.parallel import DistributedDataParallel as DDP

def setup(rank, world_size):
  os.environ['MASTER_ADDR'] = 'localhost'
  os.environ['MASTER_PORT'] = '12355'

  # initialize the process group
  dist.init_process_group("nccl", rank=rank,
    world_size=world_size)

def cleanup():
  dist.destroy_process_group()

class NeuralNetModel(nn.Module):
  # Model implementation
  # ...

def run_model(rank, world_size):
  print(f"Running DDP model on rank {rank}.")
  setup(rank, world_size)

  # create model and move it to GPU with id rank
  model = NeuralNetModel().to(rank)
  ddp_model = DDP(model, device_ids=[rank])

  # implementation of model optimization
  # ...

  # clean the process group
  cleanup()

...
```

```
...

def run_multi_model(demo_fn, world_size):
  mp.spawn(demo_fn, args=(world_size,), nprocs=world_size,
           join=True)

if__name__== "__main__":
  n_gpus = torch.cuda.device_count()
  assert n_gpus >= 2,
    f"Requires at least 2 GPUs to run, but got {n_gpus}"
  world_size = n_gpus
  run_multi_model(run_model, world_size)
```

Finally, the job is executed on multiple nodes by submit the batch script to PBS, via:

```
qsub -V <pytorch_multinode.sh>
```

The flag `-V` above for the `qsub` command declares that all environment variables in the `qsub` commands are exported to the batch job.

5.3 Performance analysis of existing ML frameworks on heterogeneous systems

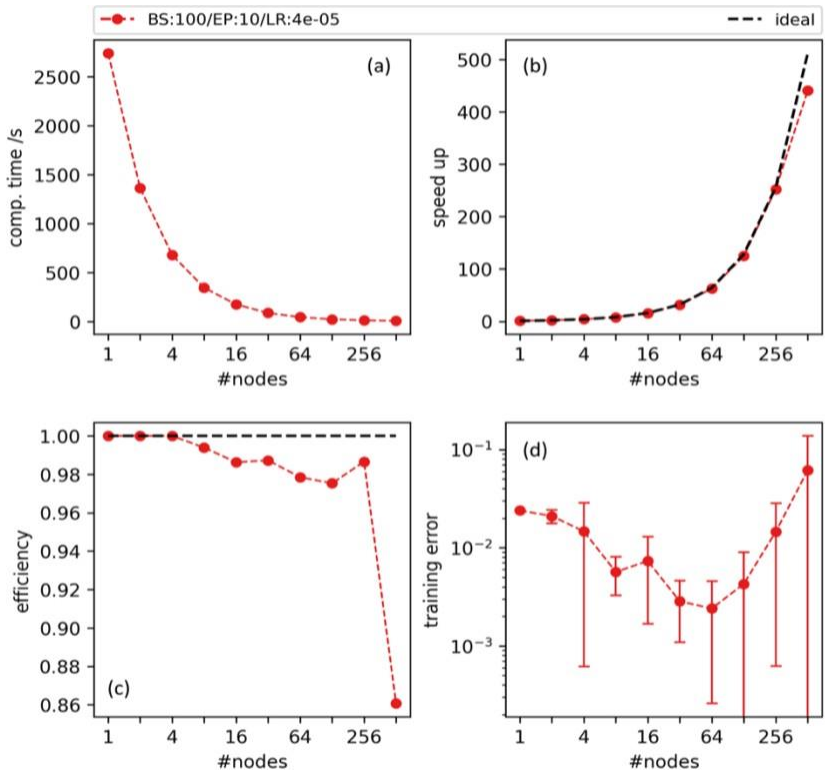

Figure 7: Performance of the PyTorch-DDP framework for training on a small version of the ATBL dataset on the JUWELS-BOOSTER. Each node consists of four NVIDIA A100 GPGPUs. Depicted are the compute time over the number of nodes (a), the strong-scaling performance (b), the code efficiency with increasing node number (c), and the corresponding training error (d). The configurable hyperparameter learning rate is linearly scaled. The black dashed lines represent the ideal scenario. Note the exponential scales.

This part focuses on the scaling performances of various ML frameworks that are ported to different heterogeneous systems without modifying the original code. For the sake of the length of this document, only a few important results are presented in this section. Initially, the strong scaling results of the PyTorch-DDP framework on the JUWELS system are discussed.

Figure 7 shows the performance of the PyTorch-DDP framework for training on the small version of the ATBL dataset (21 GB) on the JUWELS-BOOSTER system. To reduce the computational time, a total of 10 training epochs *E=10* is performed. The batch size per GPGPU is set to *B=100*. The hyperparameter learning rate *L* is linearly scaled with the number of nodes. The learning rate is set to *L=4e-5* and *L=0.02* for a single and 512 nodes. The training error is adopted from the work by Jin et al. [18] and is computed as the difference of values between the input and reconstructed data. The CAE training employs the Adam optimization algorithm [19] with a weight decay parameter of *W=0.003*.

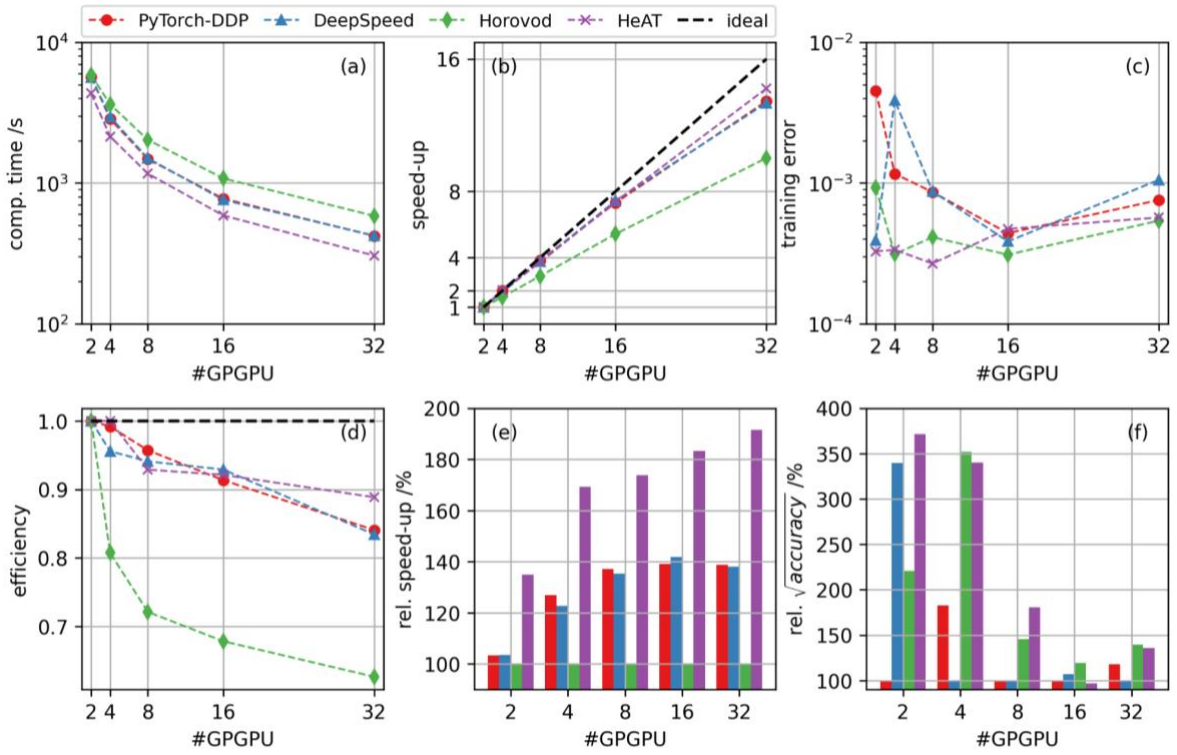

Figure 8: Performance of the existing distributed data parallel frameworks for training on a small version of the ATBL dataset on the CTE-AMD system at BSC. Each node consists of two AMD MI50 GPGPUs. Depicted are the compute time over the number of GPGPUs (a), the strong-scaling performance (b), the corresponding training error (c), the code efficiency under increasing GPGPU-count (d), the relative speed-up (e), and the relative square root of the training error (f). The configurable hyperparameters for each framework are fixed. The black dashed lines represent the ideal scenario. Note the exponential scales.

Figure 8 shows the performance of the existing distributed data parallel frameworks for training on a small version of the ATBL dataset (21 GB) on the CTE-AMD system, which employs the ROCm and RCCL libraries instead of the CUDA and NCCL libraries that are necessary on NVIDIA-based systems. For each framework, a total of 10 epochs *E=10* are performed for the sake of computational time, the batch size per GPGPU is set to *B=96*, and the hyperparameter learning rate is *L=0.01*. Similarly, the CAE trainings here employ the Adam optimization algorithm [19] with a weight decay parameter of *W=0.003*.

From Figure 8a, it is obvious that the HeAT framework is slightly faster than the rest of the frameworks, especially evident when more than two GPGPUs are employed for the CAE

training. In contrast, Horovod performs worst among the compared frameworks. As DeepSpeed shares most of the source code with the PyTorch-DDP framework, these two frameworks perform quite similarly. Except for Horovod, all investigated frameworks show a good scaling performance, see Figure 8b. As shown in Figure 8d, HeAT achieves an efficiency value of *e=0.89* when 32 GPGPUs are used, *e=0.83* is the efficiency value achieved by DDP and DeepSpeed. The Horovod framework only achieves an efficiency of *e=0.62* on 32 GPGPUs. Interestingly, the efficiency value of Horovod sharply reduces down to *e=0.81* when two nodes (or four GPGPUs) are employed. This indicates that Horovod might be experiencing node-based communication issues. Figure 8e shows the relative speed-up in percentiles, based on the slowest framework, indicating how much scaling performance could be gained by employing alternative distributed data parallel frameworks. Here, HeAT is evidently computationally faster and scales better to larger amounts of workers than the compared distributed data parallel framework.

The training error is computed as presented in [18]. Even though the scaling performance of Horovod is not satisfactory, a lower training error is achieved with this framework, see Figure 8c. Here, HeAT also achieves similar training errors. However, both PyTorch-DDP and DeepSpeed show larger training errors compared to HeAT and Horovod. Figure 8f shows the square rooted relative training error between the considered distributed data parallel frameworks in percentiles, based on the framework with the largest training error, i.e., the framework with the lowest accuracy shows the highest percentile. It can be seen that Horovod achieves the best relative square rooted training error on 32 GPGPUs. HeAT is, however, not far behind with similar training error values. It can be concluded that HeAT clearly outperforms the other distributed data parallel framework due to its scaling performance and excellent accuracies, noting that these results are performed on the CTE-AMD system.

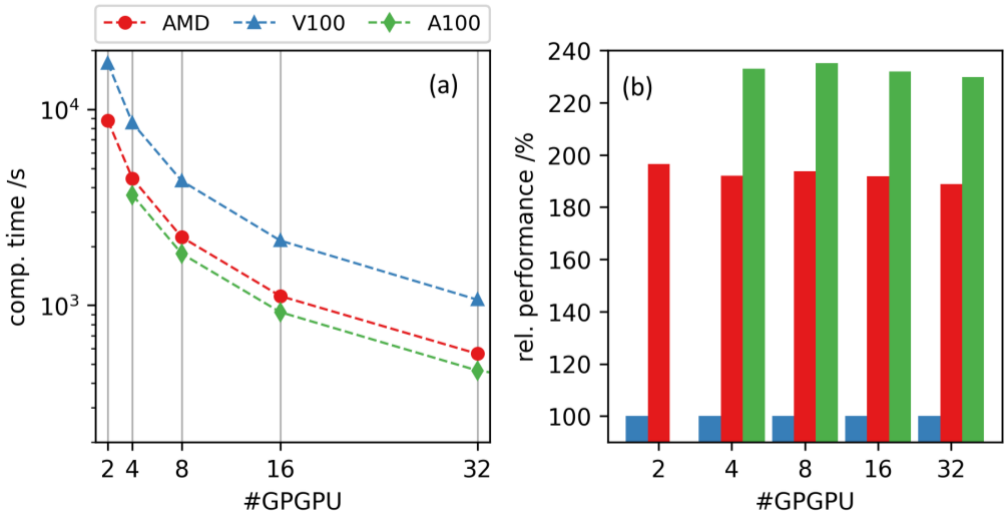

Figure 9: Performance of the PyTorch-DDP framework for training on a small version of the ATBL dataset on the CTE-AMD, DEEPEST, and JUWELS-BOOSTER systems. The CTE-AMD machine is equipped with AMD MI50 (red), the DEEP-EST system with NVIDIA V100 (blue), and the JUWELS-BOOSTER with NVIDIA A100 GPGPUs (green). Depcicted are the compute time over the number of GPGPUs (a) and the relative performance (b). Note the exponential scales.

Figure 9 shows the performance of the PyTorch-DDP for training on the small version of the ATBL dataset on the three different systems, each with different GPGPU-kinds.The CTE-AMD system is equipped with AMD MI50, the DEEP-EST system withNVIDIA V100, and JUWELS-BOOSTER with NVIDIA A100 GPGPUs, see Deliverables D2.5 and D2.6. That is, three GPGPU types are cross-compared. For each test, a total of 10 epochs *E-10* are performed for

the sake of computational time, the batch size per GPGPU is set to *B=96*, and the hyperparameter learning rate to *L=0.01*. Again, the CAE trainings employ the Adam optimization algorithm [19] with a weight decay parameter of *W=0.003*. It should be noted that these systems use slightly different node-based InfiniBand (IB) connections - the slowest being the DEEP-EST system with InfiniBand-Enhanced Data Rate (IB-EDR) connection.

The PyTorch-DDP framework shows similar scaling performances across all systems, shown in Figure 9 (not shown but marginally visible from computational times in Figure 9a - the training times reduce in similar order at each system). Figure 9 also shows the relative performance in percentiles, based on the slowest GPGPU, indicating how much performance could be gained by switching the systems. The new generation NVIDIA GPGPU A100 is approximately 2.3 times faster than the previous generation V100s. The performance of the AMD MI50 GPGPUs is close to that of the A100s, but 15% slower. As sole performance values are meaningless, further investigations considering the energy consumption are planned soon.

no.	*Horovod*				*Pytorch-DDP*			
GPUs	U [%]	e	T [s]	DT [i/s]	U [%]	e	T [s]	DT [i/s]
4	72.29	1.00	59,221	1,929	83.15	1.00	51,939	2,219
8	69.65	0.88	32,850	3,430	82.29	1.00	25,829	4,456
16	69.71	0.85	17,415	6,621	82.16	0.99	13,065	8,825
32	69.72	0.80	9,314	12,379	81.18	0.99	6,533	17,648
64	70.62	0.72	5,142	22,423	82.04	0.99	3,245	35,154
128.	70.60	0.64	2,913	39,582	83.32	0.99	1,622	70,523
256	70.96	0.59	1,560	73,930	74.14	0.99	811	142,001
512	61.76	0.62	747	154,357	68.37	0.99	405	284,210
1,024	54.53	0.47	488	236,281	63.86	0.97	203	567,390

Table 4: Performance of Horovod and PyTorch-DDP on the JUWELS-BOOSTER system; *U*: percentage of average GPGPU usage at training in %; *e*: parallel efficiency; *T*: run time in seconds; *DT*: data throughput in images per second.

To enable a more general comparison, the Horovod and PyTorch-DDP frameworks are also evaluated on the default ImageNet benchmark, training a ResNet50 architecture on the dataset on up to 1,024 GPGPUs on the JUWELS-BOOSTER system. The run time of training the network for *E=90* epochs with a batch size of *B=64* per GPGPU is measured. The data throughput (images per second) is shown in Figure 10 (left). Overall, PyTorch-DDP performs much better than Horovod and achieves a higher data throughput on all instances with an increasing difference under an increasing number of GPGPUs. On 1,024 GPGPUs, PyTorch-DDP finishes its training in 203s and is thus more than twice as fast as Horovod, which requires 488s for the same task. This is also evident from the plot of the parallel efficiency *e* in Figure 10 (right). It is confirmed that PyTorch- DDP scales almost perfectly with $e > 0.96$ up to 1,024 GPGPUs. In contrast, the efficiency of Horovod already drops below $e = 0.90$ at 8 GPUs. The drop in efficiency continues up to 256 GPGPUs, before a small increase appears at 512 GPUs, followed by a drop again at 1,024 GPUs. The general low scaling performance of Horovod is likely due to the consensus protocol and master thread that Horovod has to run in the background. Exact numbers are reported in Table 4. Apparently, Horovod is not able to fully utilize the GPGPUs, as it never reaches more than ~70% utilization while PyTorch-DDP utilizes more than ~80% for most runs.

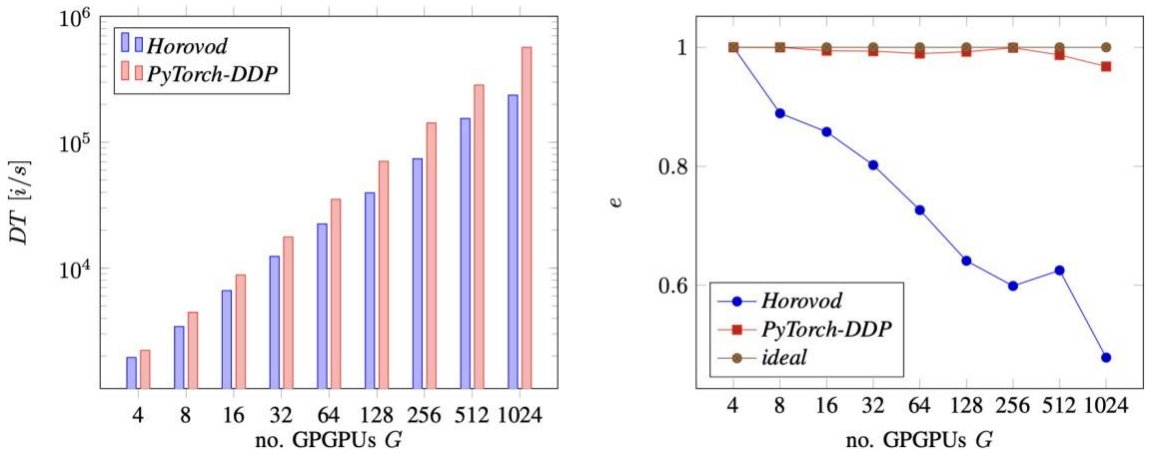

Figure 10: Performance of Horovod and PyTorch-DDP on the ImageNet benchmark on the JUWELS-BOOSTER module for an increasing number of GPGPUs *G*. Left: Comparison of the data throughput *DT* in images *i* per second. Right: Comparison of the parallel efficiency *e*.

As the batch size *B* increases with the number of workers, a drop in validation accuracy can be expected once *B* passes a certain threshold. Figure 11 shows that this drop occurs at a *B=16,000* for Horovod and already at B=*2,000-4,000* for PyTorch-DDP. It is interesting to note that overall Horovod retains a better validation accuracy than PyTorch-DDP.

The results clearly show the superiority of the PyTorch-DDP framework over *Horovod* in terms of scalability. With a parallel efficiency of over 0.96 across all instances, PyTorch-DDP is close to the best-case scenario of linear scaling. However, the validation accuracy suffers from this approach and the training already diverges at medium-sized batch sizes. It is hence recommended to use PyTorch-DDP if scalability is in focus and large HPC systems are employed, and to use Horovod if high accuracy is demanded.

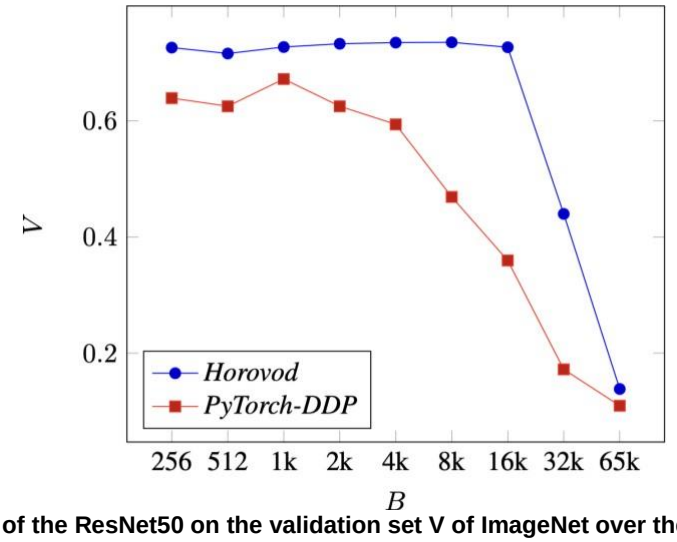

Figure 11: Accuracy of the ResNet50 on the validation set V of ImageNet over the overall batch size B.

6 Summary and conclusions

Several activities with respect to porting simulation and AI codes to different architectures and systems, and code optimizations were carried out in the first phase of the project. This included both simulation codes used in WP3, namely Alya and m-AIA, and a series of ML frameworks. The benefit of the optimizations was demonstrated through performance analysis that were performed on various supercomputers on different Tiers.

Alya was ported to the Tier-0/1 system JUWELS-BOOSTER at FZJ and to the Tier-2 system Rudens at RTU. There, first performance results were obtained.

For RWTH's code m-AIA, the computational times were drastically reduced via a structural change of the source code. The overhead of the computationally expensive subroutines was transferred from CPUs to GPGPUs. For this purpose, the loops with OpenMP acceleration were replaced with Parallel Standard Template Library (PSTL) variants. The performance of the GPGPU-accelerated m-AIA code was cross-compared with the original pure CPU implementation of the code. This performance analysis of m-AIA was performed on the JURECA-DC and JUWELS-BOOSTER supercomputers at FZJ.

Existing ML frameworks were ported to RTU's Rudens system, to BSC's CTE-AMD machine, and to FZJ's DEEP-EST, JURECA-DC, and JUWELS-BOOSTER systems without any fundamental structural code changes. That is, each ML framework was ported to the heterogeneous systems by issuing simple commands. Therefore, the focus of the activities was on the analysis of the relative and scaling performance on different heterogeneous systems. With the help of an analysis of the relative performance, cross-comparisons were possible, whereas scaling analyses assessed the performance of these frameworks on different heterogeneous systems. From these analyses, it was evident that each of the considered frameworks could achieve either exceptional scaling performance or good training accuracy. It was found that blindly adding more GPGPUs to the ML training indeed reduced the training times. This was, however, accompanied by unacceptable training accuracies due to very large total batch sizes. A careful analysis of each ML framework should be performed to find a balance between the scale of the training (number of workers) and training accuracy. This is part of the ongoing work.

All past and ongoing porting and optimization activities aim at bringing complete use-case-specific workflows including AI components to next-generation supercomputers. As the HPC landscape in Europe is continuously changing, more porting and optimization activities are planned for the second year, likely on the largest supercomputers. They will be reported in the next Deliverable D2.3 of this series of Deliverables in project month M24.

References

[1] Vázquez, M., Houzeaux, G., Koric, S., Artigues, A., Aguado-Sierra, J., Arís, R., … Valero, M. (2016). Alya: Multiphysics engineering simulation toward exascale. Journal of Computational Science, 14, 15–27. https://doi.org/10.1016/j.jocs.2015.12.007

[2] Borrell, R., Dosimont, D., Garcia-Gasulla, M., Houzeaux, G., Lehmkuhl, O., Mehta, V., … Oyarzun, G. (2020). Heterogeneous CPU/GPU co-execution of CFD simulations on the POWER9 architecture: Application to airplane aerodynamics. Future Generation Computer Systems, 107, 31–48. https://doi.org/10.1016/j.future.2020.01.045

[3] Lehmkuhl, O., Houzeaux G., Owen, H., Chrysokentis, G. and Rodríguez, I. (2019). A low-dissipation finite element scheme for scale resolving simulations of turbulent flows. Journal of Computational Physics, 390, 51-65. https://doi.org/10.1016/j.jcp.2019.04.004

[4] Poinsot, T., Veynante, D., (2005). Theoretical and numerical combustion. RT Edwards, Inc. https://doi.org/10.1016/j.combustflame.2005.11.002

[5] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., … Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision, 115(3), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[6] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. Communications of the ACM, 60(6), 84–90. https://doi.org/10.1145/3065386

[7] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 770–778. https://doi.org/10.1109/CVPR.2016.90

[8] Li Deng. (2012). The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. IEEE Signal Processing Magazine, 29(6), 141–142. https://doi.org/10.1109/MSP.2012.2211477

[9] Gallinari, P., Lecun, Y., Thiria, S. and Soulie, F.F. (1987). Mémoires associatives distribuées: une comparaison (distributed associative memories: a comparison). In Proceedings of COGNITIVA 87, Paris, La Villette, May 1987. Cesta-Afcet.

[10] Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., … He, K. (2017). Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. Retrieved from http://arxiv.org/abs/1706.02677

[11] You, Y., Gitman, I., & Ginsburg, B. (2017). Large Batch Training of Convolutional Networks. Retrieved from http://arxiv.org/abs/1708.03888

[12] Yamazaki, M., Kasagi, A., Tabuchi, A., Honda, T., Miwa, M., Fukumoto, N., … Nakashima, K. (2019). Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds. Retrieved from http://arxiv.org/abs/1903.12650

[13] Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., … Chintala, S. (2020). PyTorch Distributed: Experiences on Accelerating Data Parallel Training. Retrieved from http://arxiv.org/abs/2006.15704

[14] Sergeev, A., & Del Balso, M. (2018). Horovod: fast and easy distributed deep learning in TensorFlow. Retrieved from http://arxiv.org/abs/1802.05799

[15] Götz, M., Debus, C., Coquelin, D., Krajsek, K., Comito, C., Knechtges, P., … Streit, A. (2020). HeAT – a Distributed and GPU-accelerated Tensor Framework for Data Analytics. 2020 IEEE International Conference on Big Data (Big Data), 276–287. https://doi.org/10.1109/BigData50022.2020.9378050

[16] Rasley, J., Rajbhandari, S., Ruwase, O., & He, Y. (2020). DeepSpeed. Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, 3505–3506. https://doi.org/10.1145/3394486.3406703

[17] Pumma, S., Buono, D., Checconi, F., Que, X., & Feng, W. (2020). Alleviating Load Imbalance in Data Processing for Large-Scale Deep Learning. 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), 262–271. https://doi.org/10.1109/CCGrid49817.2020.00-67

[18] Jin, X., Cheng, P., Chen, W.-L., & Li, H. (2018). Prediction model of velocity field around circular cylinder over various Reynolds numbers by fusion convolutional neural networks based on pressure on the cylinder. Physics of Fluids, 30(4), 047105. https://doi.org/10.1063/1.5024595

[19] Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. Retrieved from http://arxiv.org/abs/1412.6980

List of Acronyms and Abbreviations

AI	Artificial Intelligence
ATBL	Actuated Turbulent Boundary Layer
AUSM	Advection Upstream Splitting Method
BS	Batch Size
BSC	Barcelona Supercomputing Centre, Spain
BSCW	Basic Support for Cooperative Work
CAE	Convolutional Auto-Encoder
CEA	Comissariat à l'énargie atomique et aux énergies alternatives
CERFACS	Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique, France
CFD	Computational Fluids Dynamics
CINECA	Consorzio Interuniversitario del Nord est Italiano Per il Calcolo Automático
CI/CS	Continuous Integration / Continuous Delivery
CNN	Convolutional Neural Network
CoE RAISE	Center of Excellence "Research on AI- and Simulation-Based Engineering at Exascale"
CPU	Central Processing Unit
CSCS	Centro Svizzero di Calcolo Scientifico
DDP	see PyTorch-DDP
DL	Deep Learning
DLB	Dynamic Load Balance library
DLR	German Aerospace Center
EoCoE-II	Energy Oriented Centre of Excellence
FFTW	Fastest Fourier Transform in the West
FV	Finite Volume
FZJ	Forschungszentrum Jülich GmbH, Jülich, Germany
GCC	GNU Compiler Collection
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HDF5	HDF5 high performance data software library and file format to manage, process, and store your heterogeneous data
HeAT	Helmholtz Analytics Toolkit
HLRS	High-Performance Center Stuttgart
HPC	High-Performance Computing
IB	InfiniBand
IB-EDR	InfiniBand-Enhanced Data Rate
I/O	Inpout/Output
JSC	Jülich Supercomputing Centre
JUDAC	Jülich Data Access system
JUDOOR	Portal for managing accounts, projects and resources at JSC
JURECA	Jülich Research on Exascale Cluster Architectures
JUWELS	Jülich Wizard for European Leadership Science
KIT	Karlsruhe Institute of Technology
LES	Large-Eddy Simulations
ML	Machine Learning
MNIST	Modified National Institute of Standards and Technology

MPI	Message Passing Interface, API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages
MUSCL	Monotone Upstream Centered Scheme for Conservation Laws
NCCL	NVIDIA Collective Communication Library
NIST	National Institute of Standards and Technology
NLP	Natural Language Processing
NVHPC	NVIDIA HPC SDK. A Comprehensive Suite of Compilers, Libraries and Tools for HPC
OpenACC	Programming standard for parallel computing
PRACE	Partnership for Advanced Computing in Europe (EU project, European HPC infrastructure)
PBS	Portable Batch System
PSTL	Parallel Standard Template Library
PyTorch-DDP	PyTorch Distributed Data Parallel
RAISE	see CoE RAISE
RCCL	ROCm Communication Collectives Library
RTU	Rigas Tehniska Universitate, Latvia
RWTH	Rheinisch-Westfälische Technische Hochschule Aachen, Germany
SIMD	Single Instruction-Multiple Data
SMT	Simultaneous Multithreading
TCP	Transmission Control Protocol
TGV	Taylor-Green Vortex
TALP	Tracking Application Low-level Performance library
UDP	User Datagram Protocol
UEBAS	The Unified European Applications Benchmark Suite
UFTP	User Datagram Protocol - File Transfer Protocol
UOI	Háskóli Íslands – University of Iceland, Iceland
WP	Work Package
ZeRO	Zero Redundancy Optimizer