Support Report

Table of Contents

List of Figures

List of Tables

Executive Summary

Within the European Center of Excellence in Exascale Computing “Research on AI- and Simulation-Based Engineering at Exascale” (CoE RAISE), the developers have access to various high-performance computing prototypes. The main systems are provided by the project partners Barcelona Supercomputing Center (BSC), Spain, and Forschungszentrum Jülich (FZJ), Germany. These prototypes are comprised of components that have the potential to become part of future supercomputing systems. They are hence of special interest to CoE RAISE’s developers. This document reports on the supporting activities provided to the developers in the project by the hosting entities of the prototype systems. This Deliverable is the second in a line of two. The first Deliverable D2.6 that was submitted to the EU in month M6 of the project presented porting and performance analysis activities mainly for the two simulation codes m-AIA from RWTH Aachen University and Alya from BSC. Furthermore, a first analysis of Machine Learning (ML) frameworks was performed. Between M6 and M18, the activities continued and this Deliverable reports on further activities on m-AIA, Alya, and different ML frameworks and tools. In addition, another simulation code, the open-source Computational Fluid Dynamics (CFD) code Basilisk which is used by the project partner Cyprus Institute (CYI) for the wetting hydrodynamics use case in Work Package (WP) 2 of CoE RAISE, was ported and analyzed and the results are discussed in this document.

1 Introduction

The prototype systems that are available to the European Center of Excellence in Exascale Computing "Research on AI- and Simulation-Based Engineering at Exascale" (CoE RAISE) are of experimental nature. They represent ideal playgrounds to test available codes on hardware that is not available in production High-Performance Computing (HPC) systems. These systems hence contribute to the evolvement of simulation and data processing codes, and they have the potential to influence the co-design of next-generation HPC systems by providing feedback to the system maintainers and operators.

In CoE RAISE, various prototype systems hosted by Forschungszentrum Jülich (FZJ) and the Barcelona Supercomputing Center (BSC) are available, i.e., the Dynamical Exascale Entry Platform – Prototype System (DEEP-EST)[1], the Huawei-based JUAWEI system[2], and the Cluster de Technologies Emergents (CTE) machines, i.e., the CTE-ARM[3] and the CTE-AMD[4] systems. These systems are rather small and meant for joint testing and analyzing novel hardware technologies. For more details on the hardware specifications and the access rules for these machines, the reader is referred to Deliverable D2.5 (Best practice guidelines/tutorials prototype) of CoE RAISE which is also available on CoE RAISE's website[5].

Two main code categories are considered in this document. The first category is represented by three major simulation codes that are used in Work Package 3 (WP3) *"Compute-Driven Use-Cases Towards Exascale"*:

- The multi-physics simulation framework m-AIA (Multiphysics-Aerodynamic Institute Aachen)[6] is developed by the Institute of Aerodynamics and Chair of Fluid Mechanics (AIA) RWTH Aachen University (RWTH). It is used in Task T3.1 *"AI for turbulent boundary layers"* in WP3 for the simulation of turbulent boundary flows over an actuated surface.
- The Alya code[7] is also a multi-physics simulation code and is developed by BSC. It is used in Task T3.2 *"AI for wind farm layout optimization"* for simulating the flow over individual wind turbines as well as over a whole wind park.
- The Basilisk code[8] is an open-source software for the solution of partial differential equations based on adaptive Cartesian computational domains. It is used by the partner Cypress Institute (CYI) in Task T3.5 *"AI for wetting hydrodynamics"* for the simulation of droplet behavior on various surfaces.

The second category is represented by Machine Learning (ML) codes and related tools. More specifically, the investigations presented in this document concentrate on the following two main subcategories:

- Different ML frameworks that are openly available to the community but which might not be available on the prototype systems;
- Open-source hyperparameter tuning frameworks.

[1] DEEP-EST system https://fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/DEEP-EST/_node.html
[2] JUAWEI system https://trac.version.fz-juelich.de/armlab/wiki/Public/JUAWEI
[3] CTE-ARM system https://www.bsc.es/user-support/arm.php
[4] CTE-AMD system https://www.bsc.es/user-support/amd.php
[5] CoE RAISE https://www.coe-raise.eu/_files/ugd/248388_1e59578dfa4e4a04920be850626439dc.pdf
[6] m-AIA https://git.rwth-aachen.de/aia/MAIA (code will be made open source in the course of the project)
[7] Alya https://gitlab.com/bsc-alya/alya
[8] Basilisk http://basilisk.fr

These codes are used across all CoE RAISE use-cases and are jointly further developed, applied, and tested in strong collaborations between members of Work Package 2 (WP2) *"AI- and HPC-Cross Methods at Exascale"*, WP3, and WP4 *"Data-Driven Use-Cases Towards Exascale"*.

The activities this Deliverable reports on are part of Task T2.2 (Hardware prototypes) of the cross-sectional WP2, where both, FZJ and BSC provide support for the developments made in CoE RAISE on the corresponding prototype systems. This covers porting and performance analysis activities in continuation to the previous activities performed between project months M1 and M6. It hence builds upon the work presented in Deliverable D2.6 that was submitted to the European Commission (EC) in M6 and which is available for download on CoE RAISE's website[9]. That is, this document presents the ongoing work that was performed in T2.2 between M6 and M18. Note that this is the last in a row of two Deliverables for this Task as T2.2 ends at the end of M18. Where possible the activities will be continued and will also include further codes developed in WP4 that are at present in a development state that needs to be further advanced to make porting and performance analyses on the prototype systems feasible.

The document is structured as follows. Section 2 reports on support activities that relate to ML frameworks and tools. This is followed by a discussion on the Basilisk code in Sec. 3 which has not been covered by the previous Deliverable D2.6. Subsequently, the continuous support activities for the Alya and the m-AIA codes are reported in Sec. 4 and Sec. 5. The document closes with a summary and conclusion of the T2.2 activities in Sec. 6.

[9] Deliverable D2.6 https://www.coe-raise.eu/_files/ugd/248388_25d91c9cc0d74de386c27fe7999c2a30.pdf

2 Support activities for ML technologies

It is one of the objectives of CoE RAISE to couple the Computational Fluid Dynamics (CFD) / multi-physics codes introduced in the previous Deliverables D2.2 and D2.6 to AI frameworks to achieve intertwined and efficient simulation, surrogate and modeling, and data processing workflows. This section reports on such activities that have taken place in the project months M6 to M18 of CoE RAISE as a continuation of the work from M1 to M6 which was presented in Sec. 4 of Deliverable D2.6. In D2.6, the first analysis of ML frameworks on the available prototype systems was presented.

In the following, an overview of the various datasets used in ML applications is first given in Sec. 2.1, and the parallelization strategy used in ML training on HPC platforms is briefly discussed in Sec. 2.2. Subsequently, Sec. 2.3 discusses strategies for porting to and running such AI frameworks on the prototype systems alongside a description of the external libraries the AI frameworks depend on when a high-performance and parallel execution is desired. The presentation of these strategies is complemented by the results of performance analyses of the AI frameworks in Sec. 2.4. As hyperparameter tuning is a key element to many CoE RAISE applications, Sec. 2.5 discusses how to port the widely used tuning framework Ray Tune[10] to the prototype systems and how it performs for a specific case.

2.1 Overview of employed datasets for ML applications

The datasets used to test, train, and validate have previously been described in detail in Deliverable D2.2. 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”.* For completeness, 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 [1]. 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 to which these images are assigned. 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. ImageNet has become the most used benchmarking dataset for computer vision applications.

The ResNet model is used to train the ImageNet dataset. This Convolutional Neural Network (CNN)-based model was initially introduced as AlexNet in [2]. Various improvements were made, leading to the ResNet architecture [3]. 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 [4] consists of a collection of handwritten digital images used for character recognition. This dataset is an extension of the original dataset available from the National Institute of Standards and

[10] Ray https://www.ray.io/

Technology (NIST)[11]. The dataset consists of 60,000 sample digital images for training and 10,000 samples for testing purposes. Each sample image is centered and represented in fixed 28x28 black and white pixels. The grayscale levels are introduced to each image using anti-aliasing techniques. A CNN-based classification method is an optimal tool to train the MNIST dataset, as mentioned in [4].

ATBL: CAE

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 to simulate the flow over a moving geometry. This creates 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 D3.1 *"Report on outcomes of WP3 use-cases"*[12].

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).

A Convolutional Auto-Encoder (CAE) model is applied to train on this ATBL dataset. CAEs are unsupervised neural network models that summarize the general properties of the input dataset in fewer parameters, while learning to reconstruct the input again after compression in a decompression process. Due to their simple implementation, CAEs are widely used for reducing the dimensionality of any dataset or to remove noise. More information on this topic can be found in the Deliverables D2.2 and D2.14 *"Report on novel AI technologies"*[13].

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 Graphics Processing Units (GPUs), where the trainable parameters between the GPUs are exchanged occasionally. This method is explained in the following Sec. 2.2.

2.2 Parallelization strategy of ML training

A common parallelization strategy to train ML models with large datasets is to distribute the input dataset to separate GPUs, where the trainable parameters between the GPUs are exchanged occasionally. This method is called Distributed Data Parallelization (DDP) and greatly reduces the training time. Depending on the data exchange rate between the workers (in this case GPUs), this type of parallelized training can scale to many workers. Currently, the GPUs require the CPUs to access the input data, therefore the transfer of the input data to the workers is, in this case, the bottleneck of such training. Alternatively, only CPUs could be employed to train ML models but the GPU architecture enables shorter training durations due to much faster ML-related matrix operations.

[11] NIST https://www.nist.gov
[12] Note that D3.1 is confidential. A public version will be made available at the end of the project within the context of Deliverable D3.3.
[13] Note that D2.14 is confidential. A public version will be made available at the end of the project within the context of Deliverable D2.16.

As discussed in Deliverable D2.2, the sole drawback of DDP is the loss in training accuracy caused by the large total batch sizes. As each GPU is set to a fixed mini-batch size, increasing the number of GPUs yields an increase in the total batch size. For training on a large number of GPUs, this may hence lead to reduced training accuracies. This limits the scaling performance of the ML training, as the training accuracy becomes the decisive factor when investigating the scaling performance of a training. A common strategy is to tune other hyperparameters keeping the accuracy of the training at an acceptable level which has intensively been addressed in the literature [5,6,7] and in Deliverable D2.2.

2.3 Porting existing ML frameworks to CoE RAISE prototype systems

In this section, the focus is on porting existing ML frameworks to the heterogeneous prototype systems DEEP-EST and CTE-AMD. This DDP strategy can be applied to the prototype systems via several existing frameworks. The following list presents the frameworks that have been tested on the available prototype systems, noting that a change in the structure of these frameworks is not required, as they are already optimized for both, CPU and/or GPU architectures (detailed information can be found in Deliverable D2.2):

- The Distributed Data Parallel (DDP) module, as part of a PyTorch package [8];
- The Horovod distributed training package, developed by Uber [9];
- The Helmholtz Analytics Toolkit HeAT, a project of the Helmholtz association [10];
- The DeepSpeed framework, developed by Microsoft [11].

At a macroscopic scale, these frameworks operate similarly. However, minor optimizations in each framework lead to different scaling performances and accuracies. These frameworks share the same open-source host library, namely PyTorch[14] which is used here in version 1.11.0.

The following Sec. 2.3.1 and Sec. 2.3.2 report on how to port these frameworks to the DEEP-EST and CTE-AMD machines. Subsequently, it is explained how to initialize the frameworks on these machines and how to employ external libraries in Sec. 2.3.3 and Sec. 2.3.4.

2.3.1 Porting to DEEP-EST at FZJ

Porting these frameworks to heterogeneous prototype systems is mostly similar. However, both, DEEP-EST and CTE-AMD consists of different GPU vendors, hence, different drivers and software stacks. Therefore, it is necessary to show the porting steps individually for both systems, starting in this section with the DEEP-EST system.

The existing frameworks require a communication library for parallelism and Python with a version of *3.x*. Three communication libraries can be selected: the Message Passing Interface (MPI), the NVIDIA Collective Communications Library (NCCL)[15], and Gloo[16]. To access the fast InfiniBand (IB) connection, either the MPI or NCCL library should be selected. Moreover, the Compute Unified Device Architecture (CUDA)[17] library for NVIDIA GPUs or the RadeonOpenCompute (ROCm)[18] library for AMD GPUs is also required if GPUs are to be used.

[14] PyTorch https://pytorch.org/
[15] NCCL https://developer.nvidia.com/nccl
[16] Gloo https://github.com/facebookincubator/gloo
[17] CUDA https://developer.nvidia.com/cuda-toolkit/
[18] ROCm https://github.com/RadeonOpenCompute/ROCm

On DEEP-EST, the Lmod[19] package as part of the EasyBuild[20] system is used to load and link necessary libraries to the system, as described extensively in Deliverable D2.6. The necessary latest software stack `Stages/2022` is first loaded via:

```
$ ml use $OTHERSTAGES
$ ml Stages/2022
```

The following command loads all the required modules, i.e., the GNU Compiler Collection (GCC)[21], the inter-process communication library OpenMPI[22], the NVIDIA CUDA Deep Neural Network library (cuDNN)[23], NCCL, Python, and CMake[24].

```
$ ml GCC/11.2.0 OpenMPI/4.1.2 cuDNN/8.3.1.22-CUDA-11.5 \
  NCCL/2.12.7-1-CUDA-11.5 Python/3.9.6 CMake/3.21.1
```

Each existing framework can be compiled on DEEP-EST from source. However, the Python package installer pip[25] or the open-source package management system Conda[26] greatly reduce the complexity to compile such frameworks. Alternatively, the current software stack includes the latest PyTorch 1.11.0. In the following, pip is chosen as an example in case another PyTorch version is required. The Python environment variable `<env_name>` can be created and sourced via:

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

PyTorch with DDP

PyTorch can be compiled noting that the installed CUDA version on the system is 11.5, by:

```
$ pip3 install torch==1.11.0+cu115 torchvision==0.12.0+cu115 \
  torchaudio===0.11.0+cu115 \
  -f https://download.pytorch.org/whl/torch_stable.html
```

The PyTorch framework contains all required libraries for the DDP module to run, i.e., no additional compilation of this module is necessary. The current PyTorch version does not detect the correct IB utilized Internet Protocol (IP) addresses. This issue arises from the distributed communication package c10d[27] of PyTorch which connects the communication library with DDP by selecting one of the compute nodes as the host to handle the parallel communication. In the current version of PyTorch 1.11.0, an external Python library sockets[28]

[19] Lmod https://lmod.readthedocs.io/en/latest/
[20] EasyBuild https://easybuild.io/
[21] GCC https://gcc.gnu.org/
[22] OpenMPI https://docs.par-tec.com/html/psmpi-userguide/index.html
[23] cuDNN https://developer.nvidia.com/cudnn
[24] CMake https://cmake.org
[25] pip https://pypi.org/project/pip/
[26] Conda https://docs.conda.io/
[27] c10d https://pytorch.org/docs/stable/distributed.html
[28] sockets https://docs.python.org/3/library/socket.html

included in c10d fails to fetch the IB's IP address of the host compute node. PyTorch then appoints the IP addresses from a slower ethernet connection which is not optimized for node-to-node communication. At the time of writing this document, the PyTorch developers only released a manual fix to this problem which modifies the `torchrun` source file:

```
import re, sys
from torch.distributed.run import main
from torch.distributed.elastic.agent.server import api as sapi

def new_get_fq_hostname():
    return _orig_get_fq_hostname().replace('.', 'i.', 1)

if __name__ == '__main__':
    _orig_get_fq_hostname = sapi._get_fq_hostname
    sapi._get_fq_hostname = new_get_fq_hostname
    sys.argv[0] = re.sub(r'(-script\.pyw|\.exe)?$', '', sys.argv[0])
    sys.exit(main())
```

This modified file converts the ethernet to an IB IP address. However, the `torchrun` command requires the following additional flags with new definitions to detect which compute node is the host:

```
rdzv_conf=is_host=\$(((SLURM_NODEID)) && echo 0 || echo 1)
rdzv_endpoint='$(scontrol show hostnames "$SLURM_JOB_NODELIST" | head \
  -n 1)'i:29500
```

These flags include environment variables available to the Slurm[29] job scheduler. Therefore, further changes are necessary in case a different job scheduler is used. A complete Slurm command can be found in Sec. 2.3.3. This issue has officially been reported[30].

Horovod

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
$ export HOROVOD_WITH_MPI=1
$ export HOROVOD_MPI_THREADS_DISABLE=1
$ export HOROVOD_CUDA_HOME=$cuda_dir
$ export HOROVOD_NCCL_HOME=$nccl_dir
$ export TMPDIR=$tmp_dir
$ pip3 install horovod
```

The first three environment variables setup CUDA for GPUs, NCCL as communication backend between workers, and PyTorch for the host library. It should be noted that the first

[29] Slurm https://slurm.schedmd.com/
[30] Report link https://github.com/pytorch/pytorch/issues/73656

two flags are only necessary for GPU-accelerated training, and for AMD GPUs, ROCm, and the ROCm Communication Collectives Library (RCCL)[31] replace CUDA and NCCL. Horovod uses both, the NCCL and MPI communication libraries. The former is used for `allreduce` commands and the latter for the data loader. The fourth environment variable `HOROVOD_WITH_MPI=1` enables MPI. The multi-thread MPI operation is not required and is hence disabled via the fifth environment variable `HOROVOD_MPI_THREADS_DISABLE=1`. Optionally, CUDA, NCCL, and temporary folder (in case of limited disk space) directories can be defined with the remaining environment variables. Furthermore, the variables listed in Table 1 are available.

Options	Descriptions
`HOROVOD_DEBUG`	install a debug build of Horovod with checked assertions, disabled compiler optimizations, etc.
`HOROVOD_BUILD_ARCH_FLAGS`	additional C++ compilation flags to pass in for your build architecture
`HOROVOD_CUDA_HOME`	path where CUDA `include` and `lib` directories can be found
`HOROVOD_BUILD_CUDA_CC_LIST`	List of compute capabilities to build Horovod CUDA kernels for, e.g., `HOROVOD_BUILD_CUDA_CC_LIST=60,70,75`
`HOROVOD_ROCM_HOME`	path where ROCm `include` and `lib` directories can be found
`HOROVOD_NCCL_HOME`	path where NCCL `include` and `lib` directories can be found
`HOROVOD_NCCL_INCLUDE`	path to NCCL `include` directory
`HOROVOD_NCCL_LIB`	path to NCCL `lib` directory
`HOROVOD_NCCL_LINK`	opt.: {`SHARED, STATIC`}. Mode to link NCCL library. Defaults to `STATIC` for CUDA, `SHARED` for ROCm
`HOROVOD_WITH_GLOO`	require that Horovod is built with Gloo support enabled
`HOROVOD_WITH_MPI`	require that Horovod is built with MPI support enabled
`HOROVOD_GPU`	opt.: {`CUDA`, `ROCM`}. Framework to use for GPU operations

[31] RCCL https://github.com/ROCmSoftwarePlatform/rccl

Options	Descriptions
HOROVOD_GPU_OPERATIONS	opt.: {NCCL, MPI}. Framework to use for GPU tensor allreduce, allgather, and broadcast
HOROVOD_GPU_ALLREDUCE	opt.: {NCCL, MPI}. Framework to use for GPU tensor allreduce
HOROVOD_GPU_ALLGATHER	opt.: {NCCL, MPI}. Framework to use for GPU tensor allgather
HOROVOD_GPU_BROADCAST	opt.: {NCCL, MPI}. Framework to use for GPU tensor broadcast
HOROVOD_ALLOW_MIXED_GPU_IMPL	allow Horovod to install with NCCL allreduce and MPI GPU allgather / broadcast. Not recommended due to a possible deadlock
HOROVOD_CPU_OPERATIONS	opt.: {MPI, GLOO, CCL}. Framework to use for CPU tensor allreduce, allgather, and broadcast
HOROVOD_CMAKE	path to the CMake binary used to build Horovod
HOROVOD_WITH_TENSORFLOW	require Horovod to install with TensorFlow[32] support
HOROVOD_WITH_PYTORCH	require Horovod to install with PyTorch support
HOROVOD_WITH_MXNET	require Horovod to install with MXNet[33] support

Table 1: Important environment variables to configure Horovod installation and their descriptions.

HeAT

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

```
$ 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[34] or parallel-NetCDF[35] formats. The I/O libraries need to be preloaded by issuing:

```
$ ml HDF5/1.12.1-serial PnetCDF/1.12.2
```

[32] TensorFlow https://www.tensorflow.org/
[33] MxNet https://mxnet.apache.org/versions/1.9.0/
[34] HDF5 https://www.hdfgroup.org/solutions/hdf5
[35] Parallel NetCDF https://trac.mcs.anl.gov/projects/parallel-netcdf

DeepSpeed

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

```
$ export DS_BUILD_FUSED_ADAM=1
$ export DS_BUILD_UTILS=1
$ export DS_BUILD_AIO=0
$ export TMPDIR=<tmp_dir>
$ pip3 install DeepSpeed
```

The first two environment variables are responsible to compile helper utilities of DeepSpeed required during the CAE training. Unfortunately, DeepSpeed on the DEEP-EST system has asynchronous input/output (I/O) issues, hence, disabling this via the third environment variable is currently a workaround, and does not affect the current ML training. Additionally, DeepSpeed has the following options listed in Table 2.

Options	Descriptions
DS_BUILD_OPS	toggles all below operations
DS_BUILD_CPU_ADAM	builds the CPUAdam op.
DS_BUILD_FUSED_ADAM	builds the FusedAdam op.
DS_BUILD_FUSED_LAMB	builds the FusedLamb op.
DS_BUILD_SPARSE_ATTN	builds the sparse attention op.
DS_BUILD_TRANSFORMER	builds the transformer op.
DS_BUILD_TRANSFORMER_INFERENCE	builds the transformer-inference op.
DS_BUILD_STOCHASTIC_TRANSFORMER	builds the stochastic transformer op.
DS_BUILD_UTILS	builds various optimized utilities
DS_BUILD_AIO	asynchronous I/O op. (for NVMe disks)

Table 2: Important environment variables to configure DeepSpeed installation and their descriptions.

Further pre- and post-processing, and advanced CNN commands

For pre- and post-processing, the data, and for advanced CNN commands (such as three-dimensional convolutions), the additional Python libraries Pillow[36], pyparsing[37], python-

[36] Pillow https://pillow.readthedocs.io/en/stable
[37] pyparsing https://github.com/pyparsing/pyparsing

dateutil[38], matplotlib[39], h5py[40], and pytorch-nlp[41] need to be included to the Python environment by:

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

An alternative approach to compile these existing frameworks on the DEEP-EST system is via using the Conda package manager. Conda can be compiled in any `conda_env` directory on DEEP-EST following these lines of codes:

```
# download latest Miniconda
$ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux- \
  x86_64.sh

# compile and source Miniconda
$ bash Miniconda3-latest-Linux-x86_64.sh -p $conda_env/miniconda3 -b
$ source $PWD/miniconda3/etc/profile.d/conda.sh
$ conda activate
```

After this step necessary libraries required by PyTorch (with CUDA 11.5 support) can be compiled into the Conda environment by:

```
# standard libraries
$ conda install -y astunparse numpy pyyaml mkl mkl-include setuptools \
  cffi typing_extensions future six requests dataclasses Pillow \
  --force-reinstall

# For CUDA 11.5 - check system version!
$ conda install -c pytorch -y magma-cuda115 --force-reinstall
$ conda install -y pkg-config libuv --force-reinstall
```

The PyTorch library can now be compiled using the Conda environment via:

```
# clone latest PyTorch repository
$ git clone --recursive https://github.com/pytorch/pytorch pytorch

# update repository
$ cd pytorch
$ git submodule sync
$ git submodule update --init --recursive

# install PyTorch
$ export CMAKE_PREFIX_PATH=${CONDA_PREFIX:-"$(dirname $(which \
  conda))/../"}
$ export TMPDIR=${iDir}/tmp
```

[38] python-dateutil https://dateutil.readthedocs.io/en/stable/
[39] matplotlib https://matplotlib.org
[40] h5py https://www.h5py.org
[41] pytorch-nip https://pytorch.org/tutorials/beginner/deep_learning_nlp_tutorial.html

```
$ python setup.py clean
$ CMAKE_C_COMPILER=$(which mpicc) CMAKE_CXX_COMPILER=$(which mpicxx)\
  USE_DISTRIBUTED=ON USE_MPI=ON USE_CUDA=ON NCCL_ROOT_DIR=$EBROOTNCCL \
  USE_NCCL=ON USE_GLOO=ON CUDNN_ROOT=$EBROOTCUDNN USE_CUDNN=ON python \
  setup.py install
```

This compiles PyTorch with all available worker-to-worker communication libraries, MPI, NCCL, and Gloo. The last step is to compile the TorchVision package as part of the PyTorch library:

```
# clone latest TorchVision repository
$ git clone --recursive https://github.com/pytorch/vision.git torchvision

# update repository
$ cd torchvision
$ git submodule sync
$ git submodule update --init --recursive

# install TorchVision
$ export CMAKE_PREFIX_PATH=${CONDA_PREFIX:-"$(dirname $(which \
  conda))/../"}
$ export TMPDIR=${iDir}/tmp
$ python setup.py clean
$ CMAKE_C_COMPILER=$(which mpicc) CMAKE_CXX_COMPILER=$(which mpicxx) \
  FORCE_CUDA=ON python setup.py install
```

Horovod, HeAT, and DeepSpeed can be compiled to the Conda environment following similar steps as described above for pip to both, the DEEP-EST and the CTE-AMD prototypes.

2.3.2 Porting to CTE-AMD at BSC

The Lmod package as part of the EasyBuild system (see Sec. 2.3.1) is again used to load and link necessary libraries on the CTE-AMD system. The following command loads all the required modules, i.e., the GNU Compiler Collection (GCC), the communication library OpenMPI, ROCm, Python, and CMake:

```
$ ml bsc/current gcc/10.2.0 openmpi/4.0.5 rocm/4.0.1 python/3.9.1 \
  cmake/3.18.2
```

Details on these modules can be found in Deliverable D2.6. An important module here is AMD's ROCm library which handles GPU operations. Using these modules a Python environment can simply be created. This Python module does not automatically link the `pip3` installer, `libtinfo`, and `libpython` to the created environment on the CTE-AMD machine. This can, however, be realized by:

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

# libtinfo and libpython
```

```
$ ln -s /usr/lib64/libtinfo.so.6 $PWD/lib/libtinfo.so.5
$ ln -s $PYTHONPATH/GCC/lib/libpython3.9.so \
  $PWD/<env_name>/lib/libpython3.9.so

# copy pip to environment
$ cp $PYTHONPATH/GCC/bin/pip3.9 $PWD/<env_name>/bin/

# modify headers
$ var="#!$PWD/<env_name>/bin/python3.9"
$ sed -i "1s|.*|$var|" $PWD/<env_name>/bin/pip3.9

# link library path
$ export LD_LIBRARY_PATH=$PWD/<env_name>/lib:$LD_LIBRARY_PATH
```

As stated earlier in Deliverable D2.6, incoming communication to the CTE-AMD nodes from an external source is forbidden. Hence, each necessary Python library needs to be transferred to the system individually via the `sshfs` client in either wheel or tar-ball format. Then, these libraries can be installed in the Python environment via:

```
$ pip3 install --no-cache-dir --no-index --find-links --force-reinstall \
  $wheels_dir -r reqs.txt
```

The wheels should be in the wheel directory `wheels_dir` and the name of the library should be stated in the `reqs.txt` file. Note that https://pypi.org/ is the official portal to download Python libraries.

PyTorch

PyTorch can be compiled noting the installed ROCm version 4.0.1 on the CTE-AMD system, via editing this `reqs.txt` file as below, copying the wheels or tar-balls to the `wheels_dir`, and issuing the pip command above:

```
torch==1.9.0+rocm4.0.1
torchvision==0.10.0
wheel==0.37.0
```

Horovod

Horovod can be compiled with these commands:

```
$ export HOROVOD_GPU=ROCM
$ export HOROVOD_GPU_OPERATIONS=NCCL
$ export HOROVOD_WITH_PYTORCH=1
$ export HOROVOD_WITH_MPI=1
$ export HOROVOD_ROCM_HOME=$rocm_dir
$ export TMPDIR=$tmp_dir
$ pip3 install --no-cache-dir --no-index --find-links --force-reinstall \
  $wheels_dir -r $wheels_dir/horovod-0.24.3.tar.gz
```

These environment variables have been explained earlier for the DEEP-EST system (see Sec. 2.3.1), however, now ROCm is used. As RCCL is currently not an option when compiling Horovod, a second environment variable `HOROVOD_GPU_OPERATIONS` needs to be set to `NCCL`. This workaround does not cause any issue as ROCm handles the conversation between NCCL and RCCL.

HeAT

The HeAT framework requires HDF5 and NetCDF libraries which depend on Intel-related modules, loaded to CTE-AMD via:

```
$ ml bsc/current rocm/4.0.1 intel/2018.4 impi/2021.5.1 hdf5/1.12.1 \
  netcdf-c/4.8.1 pnetcdf/1.12.2 python/3.9.1 cmake/3.18.2
```

Alternatively, the input and output capabilities of HeAT can be excluded from the compilation, where an external I/O library can be implemented (not discussed here). Similar to the Horovod example above, HeAT with a version number 1.1.1 can be compiled with pip:

```
$ pip3 install --no-cache-dir --no-index --find-links --force-reinstall \
  $wheels_dir -r $wheels_dir/ heat-1.1.1_mod.tar.gz[hdf5,netcdf]
```

DeepSpeed

The DeepSpeed framework, noting the ROCm compatibility, can be compiled with these commands:

```
$ export DS_BUILD_FUSED_ADAM=1
$ export DS_BUILD_UTILS=1
$ pip3 install --no-cache-dir --no-index --find-links --force-reinstall \
  $wheels_dir -r $wheels_dir/DeepSpeed_rocm.tar.gz
```

The first two environment variables are responsible to compile the helper utilities of DeepSpeed that are required during the CAE training.

Further pre- and post-processing, and advanced CNN commands

For pre- and post-processing, the data, and for advanced CNN commands, the `reqs.txt` file can be extended, e.g., by:

```
...
matplotlib==3.4.3
pyyaml==5.4.1
h5py==3.4.0
mpi4py==3.1.1
ninja==1.10.2
...
```

An important issue on the CTE-AMD machine was identified, where three-dimensional convolutions yielded an error "`The Forward Convolution cannot be executed due`

`to incorrect params`". This issue has been reported, where the problem could be from the MIOpen[42] library that is included with the ROCm framework. The more recent ROCm version 4.5.0 is also added as a module on CTE-AMD for testing purposes which can be loaded via:

```
$ ml rocm/4.5.0-testOnly
```

This issue can be reproduced by calling the `torch.nn.Conv3d` function in the ML network in a PyTorch framework. At the time of writing this document, no solution to this issue is available.

2.3.3 Initialization of used frameworks

The initialization of the frameworks is similar to the one described for the JURECA-DC system which was thoroughly introduced in Deliverable D2.2. Hence, only important changes to the prototype systems DEEP-EST and CTE-AMD are given in following.

Initialization of ML frameworks on DEEP-EST at FZJ

The four distributed data parallelization frameworks are initialized using various methods. For the DDP framework, a *host-based non-elastic launch* framework that is a part of the PyTorch library is used. This type of launch framework uses a preset host node to access the rest of the nodes and transfers the data efficiently between the nodes. For DEEP-EST, an example batch script is given below with the help of Slurm's environment variables:

```
#!/bin/bash

# general configuration of the job
#SBATCH --job-name=<name>
#SBATCH --account=<account>
#SBATCH --time=<time>
#SBATCH --partition=dp-esb
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=16
#SBATCH --gpus-per-node=1
#SBATCH --exclusive

# set modules
ml  --force purge
ml use $OTHERSTAGES
ml Stages/2022 GCC/11.2.0 OpenMPI/4.1.2 cuDNN/8.3.1.22-CUDA-11.5 \
  NCCL/2.12.7-1-CUDA-11.5 Python/3.9.6 CMake/3.21.1

# set environment
source <env_dir>/bin/activate

# debug
export NCCL_DEBUG=INFO

# important environment variables
export CUDA_VISIBLE_DEVICES="0"
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
```

[42] MiOpen https://github.com/ROCmSoftwarePlatform/MIOpen

```
srun bash -c "torchrun \
  --log_dir='logs' \
  --nnodes=$SLURM_NNODES \
  --nproc_per_node=$SLURM_GPUS_PER_NODE  \
  --rdzv_id=$SLURM_JOB_ID \
  --rdzv_conf=is_host=\$(((SLURM_NODEID)) && echo 0 || echo 1) \
  --rdzv_backend=c10d \
  --rdzv_endpoint='$(scontrol show hostnames "$SLURM_JOB_NODELIST" | \
    head -n 1)'i:29500 \
  <training_Script> --arguments"
```

This script uses the extreme scale booster module (`dp-esb`) of the DEEP-EST system to run PyTorch. Here, only one keyword argument must be provided to the `torchrun` command, the training script (and possible flags to this script). The rest of the flags are automatically set. Moreover, the data loader in PyTorch-DDP's `training_script` should be defined as:

```
data_loader = torch.utils.data.DataLoader(<data_set>, \
  num_workers=args.nworker, **kwargs )
```

Here, the `args.nworker` argument needs to match the `cpus-per-task` argument to utilize fast multi-thread data loading. This utilization of the data-loader alone may already drastically reduce the ML training times if a large data set is used.

The DeepSpeed framework utilizes a similar initialization method as the PyTorch-DDP framework, i.e., the `deepspeed.launcher` framework, see Deliverable D2.2. However, it has been identified that the `node_rank` argument passed to the `deepspeed.launcher` requires attention. If not set correctly, the `node_rank` argument is not unique to different compute nodes (as this argument is evaluated before the `srun` command). A solution is to either wrap the `srun` command by a `bash -c` command so the Slurm's environment variable `SLURM_PROCID` becomes unique:

```
srun bash -c "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> --arguments \
  --deepspeed_mpi \
  --deepspeed_config <DS_config.json>”
```

or, by modifying the `<env_name>/site-packages/deepspeed/launcher/launch.py` file by adding `args.node_rank=int(os.environ.get("SLURM_PROCID",0))` after the definition of `args = parse_args()`. The remaining keyword arguments were already defined in Deliverable D2.2 and here only a summary is given. 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 are the required keyword arguments, whereas, 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 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`
```

Horovod handles the initialization internally. As Horovod uses MPI communication, the `srun` command requires PMIx[43] support which can be enabled via including the `--mpi=pspmix` flag:

```
srun –mpi=pspmix –cpu-bind=none python3 -u <training_Script>.py
```

Finally, HeAT does not require any special initialization, i.e., the ML training scripts can simply be executed conveniently by the command:

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

Initialization of ML frameworks on CTE-AMD at BSC

As the CTE-AMD machine also employs Slurm, the four distributed data parallelization frameworks are initialized similarly to the DEEP-EST system. Standard ML networks utilize convolution calls, i.e., CAE training employ such convolutions in both, decoder and encoder parts to gather a set of convolution algorithms in the form of an array of structures. This process is time-consuming because it requires online benchmarking of competing algorithms. In the recent MIOpen version 2.0, a look-up algorithm approach is introduced which uses a database of these algorithms to be looked-up during training. This look-up database is named Find-Db and using it greatly reduces the computational overhead. However, at this stage, turning on this option creates issues on the CTE-AMD system and it hence must be turned off via:

```
$ export MIOPEN_DEBUG_DISABLE_FIND_DB=1
```

Example ML training scripts using various datasets such as MNIST, Image Net, and the ATBL datasets on prototype systems are documented and can be found in the master branch of a GIT repository[44]. This repository also contains machine-specific batch scripts for several heterogeneous systems available to RAISE members, such as Piz Daint[45], JURECA, JUWELS, DEEP-EST, CTE-AMD, and automated compilation scripts for the DDP frameworks mentioned above.

[43] PMIx https://pmix.github.io
[44] GIT training scripts https://gitlab.jsc.fz-juelich.de/CoE-RAISE/FZJ/ai-for-hpc.git
[45] Piz Daint https://www.cscs.ch/computers/piz-daint/

2.3.4 External tools

Jupyter kernels on DEEP-EST at FZJ

JupyterLab[46] (notebook) is a popular tool, heavily used by ML developers to create and share their training scripts, pre-process datasets, and post-process the training results. The Jupyter framework is already available on DEEP-EST through JSC's Jupyter Hub[47]. However, these Jupyter notebooks require a Jupyter kernel to compile the code written in these notebooks – the standard Jupyter kernel configuration does not include any ML library such as PyTorch, TensorFlow, or MxNet. Hence, the Python environments as created by the tutorials in Sec. 2.3.1 and Sec. 2.3.2 can be used for this Jupyter kernel. The initial step is to install ipykernel (a package of interactive Python IPython[48]) via pip after loading the Python environment:

```
# create Jupyter kernel configuration directory and files
$ pip3 install --ignore-installed ipykernel --no-cache-dir
$ $VIRTUAL_ENV/bin/python3 -m ipykernel install --name=<KERNEL_NAME> \
  --prefix $VIRTUAL_ENV
```

A `kernel.sh` configuration file depending on the system is then created by:

```
# create kernel.sh in Python environment
$ echo '#!/bin/bash'"

# load modules
ml ...

# Activate your Python virtual environment
source $VIRTUAL_ENV/bin/activate

# update path for Python environment
export PYTHONPATH=$VIRTUAL_ENV/lib/python3.9/site- \
  packages:"'$PYTHONPATH'"

exec python3 -m ipykernel "'$@' > $VIRTUAL_ENV/kernel.sh

# give permission to kernel.sh
$ chmod +x $VIRTUAL_ENV/kernel.sh
```

The kernel is then registered by creating a `kernel.json` file with specific kernel information:

```
echo '{
  "argv": [
    "'$VIRTUAL_ENV/kernel.sh'",
    "-m",
    "ipykernel_launcher",
    "-f",
```

[46] Jupyter https://jupyter.org/
[47] FZJ-JSC Jupyter Hub https://jupyter-jsc.fz-juelich.de/hub/login
[48] IPython https://ipython.org/

```
    "{connection_file}"
  ],
  "display_name": "'$<KERNEL_NAME>'",
  "language": "python"
}' > $VIRTUAL_ENV/share/jupyter/kernels/$<KERNEL_NAME>/kernel.json
```

Finally, the kernel files are moved to the user's local Jupyter directory:

```
$ mkdir -p $HOME/.local/share/jupyter/kernels
$ cd $HOME/.local/share/jupyter/kernels
$ ln -s $VIRTUAL_ENV/share/jupyter/kernels/$<KERNEL_NAME> .
```

The kernels become available from the drop-down menu when a Jupyter notebook is opened under the `<KERNEL_NAME>`.

JUBE on prototype systems

The JUBE benchmarking environment[49] is an automated benchmarking suite which is actively developed by the Jülich Supercomputing Centre (JSC) at FZJ. Automating benchmarks is important for the reproducibility of the benchmark tests and the comparability of the benchmarking results. However, managing different combinations of parameters takes a significant amount of time for benchmarks with large parameter spaces. For example, in ML trainings the change in the number of GPUs or the setting of different hyperparameters drastically affects the training. Here JUBE provides a script-based framework to easily create benchmark sets, run those sets on different computer systems and evaluate the results.

The JUBE benchmarking environment is available on FZJ's systems (also on DEEP-EST) and can be loaded via:

```
$ ml Python JUBE
```

In case JUBE is not available, it can be compiled to the `jube_dir` directory using Python via:

```
$ wget http://apps.fz-juelich.de/jsc/jube/jube2/download.php?version= \
  2.4.2 -O JUBE-2.4.2.tar.gz
$ tar xzf JUBE-2.4.2.tar.gz
$ cd JUBE-2.4.2
$ python3 setup.py install --prefix=$jube_dir
```

JUBE supports two different types of input formats: XML- and YAML-based files which provide the same options. As an example, a (trimmed) configuration file to run an ML training using the PyTorch-DDP framework on the DEEP-EST system is given below:

[49] JUBE https://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/JUBE/_node.html

```
<?xml version="1.0" encoding="UTF-8"?>

<jube>
  <benchmark name="benchmark" outpath="bench_run">
    <comment>example benchmark script</comment>

    <!-- bench configuration -->
    <parameterset name="paramset">
      <parameter name="iterNO" type="int">1,2,4,8,16</parameter>
    </parameterset>

    <!-- job configuration -->
    <parameterset name="executeset">
      <parameter name="submit_cmd">sbatch</parameter>
      <parameter name="nodes">$iterNO</parameter>
      <parameter name="ready_file">ready</parameter>
      <parameter name="job_file">batch_script</parameter>
      <parameter name="script">training_script.py</parameter>
    </parameterset>

    <!-- load jobfile -->
    <fileset name="files">
      <copy>$job_file</copy>
      <link>$script</link>
    </fileset>

    <!-- substitute jobfile -->
    <substituteset name="sub_job">
      <iofile in="${job_file}" out="$job_file" />
      <sub source="#NODES#" dest="$nodes" />
    </substituteset>

    <!-- operation/execution of bench -->
    <step name="submit">
      <use>param_set</use>
      <use>executeset</use>
      <use>files,sub_job</use>
      <do done_file="$ready_file">$submit_cmd $job_file</do>
    </step>

    <!-- results -->
    <patternset name="pattern">
      <pattern name="ID" type="int">${jube_wp_id}</pattern>
      <pattern name="Nnodes" type="int">${nodes}</pattern>
    </patternset>

    <!-- analyse -->
    <analyzer name="analyse" >
      <use>pattern</use>
      <analyse step="submit">
        <file>job.out</file>
      </analyse>
    </analyzer>

    <!-- create result table -->
    <result>
      <use>analyse</use>
      <table name="result" style="pretty" sort="jube_wp_id">
```

```
        <column>ID</column>
        <column>Nnodes</column>
      </table>
    </result>

  </benchmark>
</jube>
```

In the batch script, the `#SBATCH --nodes=#NODES#` command is then modified for JUBE to work. Issuing `jube run <JUBE_configuration.xml>` would then submit five ML training each with a different number of nodes defined in the configuration file's `iterNO` parameter. When the jobs are finished the benchmark results are returned to the user:

```
$ jube result -a bench_run --id last
  result:
  | ID | Nnodes |
  |    |        |
  |  0 |      1 |
  |  1 |      2 |
  |  2 |      4 |
  |  3 |      8 |
  |  4 |     16 |
```

For each existing framework, JUBE configuration files are available in the master branch of the GIT repository[50].

2.4 Performance analysis of existing ML frameworks on prototype systems

This section focuses on the scaling performances of various ML frameworks that are ported to the DEEP-EST (Sec. 2.4.1) and CTE-AMD (Sec. 2.4.2) prototype systems. For the sake of the length of this document only the most important results are discussed.

2.4.1 Performance analysis on the DEEP-EST system

Figure 1 shows the performance of the existing distributed data parallel frameworks for training on a small version of the ATBL dataset (21 GB) on the DEEP-EST system. For each framework a total of 10 epochs *E=10* is performed, the batch size per GPU is set to *B=96*, and the hyperparameter learning rate is *L=0.01*. The CAE training employs the Adam optimization algorithm [12] with a weight decay parameter of *W=0.003*. The training error (or accuracy) is computed as presented in [13]. The relative speed-up is computed as the computational time of a framework on a particular number of GPU to the slowest framework in percentage. Not to be confused with the speed-up over the number of GPUs, here, the speed-up by selecting a different framework is identified. Similarly, the relative accuracy is computed as the training error of a framework on a particular number of GPU to the framework with the worse training error in percentage.

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

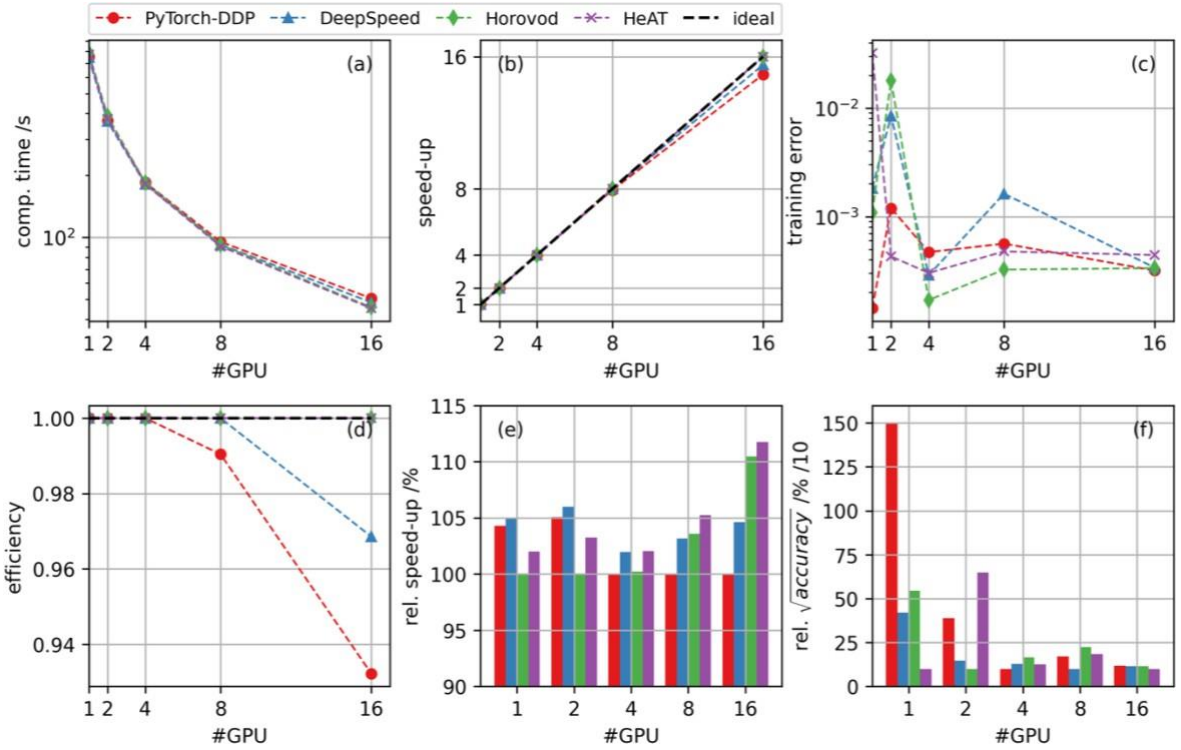

Figure 1: Performance of the existing distributed data parallel frameworks for training on a small version of the ATBL dataset on the DEEP-EST system at FZJ. Each node consists of a single NVIDIA V100 GPU. Depicted are the compute time over the number of GPUs (a), the strong-scaling performance (b), the corresponding training error (c), the code-efficiency under increasing GPU count (d), the relative speed-up (speed-up relative to the slowest among the tested frameworks) (e), and the relative square root of the accuracy (or training error) divided by 10 (f). The configurable hyperparameters for each framework are fixed. The black dashed lines represent the ideal scenario. Note the logarithmic scales.

The tested ML frameworks perform similarly on the DEEP-EST system as evident in Figure 1(a). The training times are halved when the number of GPUs is doubled, indicating almost perfect scaling performance, see Figure 1(b). This leads to high efficiencies of the tested frameworks, way above *e>0.93* even with 16 GPUs, see Figure 1(d). The relative speed-up between these frameworks shows that DeepSpeed is faster by ~5% when a single GPU is used, and HeAT is faster by ~11% when 16 GPUs are employed, see Figure 1(e). This marginal performance difference is expected since the frameworks use similar data loading methods. The major difference between the frameworks is the different intercommunication strategies. For example, PyTorch-DDP relies on the gradient bucketing approach, whereas Horovod applies a decentralized `Ring_AllReduce` scheme for the synchronization of the gradients – see [14] or Deliverable D2.2 for more details. Therefore, the scaling performance of each framework tested here would be different if a larger ML network (or smaller data set) was chosen. Furthermore, a substantial difference in the training accuracies is evident between these frameworks due to these different communication strategies from Figure 1(c), nothing that *E=10* is not sufficient for a converged training. On DEEP-EST Horovod has been found to be the most accurate framework (for a non-converged training) when more than four GPUs were employed, see Figure 1(f), and fast enough to be able to scale to a large number of GPUs. Interestingly, the node-based communication issues that were reported in Deliverable D2.2 for Horovod seem not to influence the performance on the DEEP-EST system.

2.4.2 Performance analysis on the CTE-AMD system

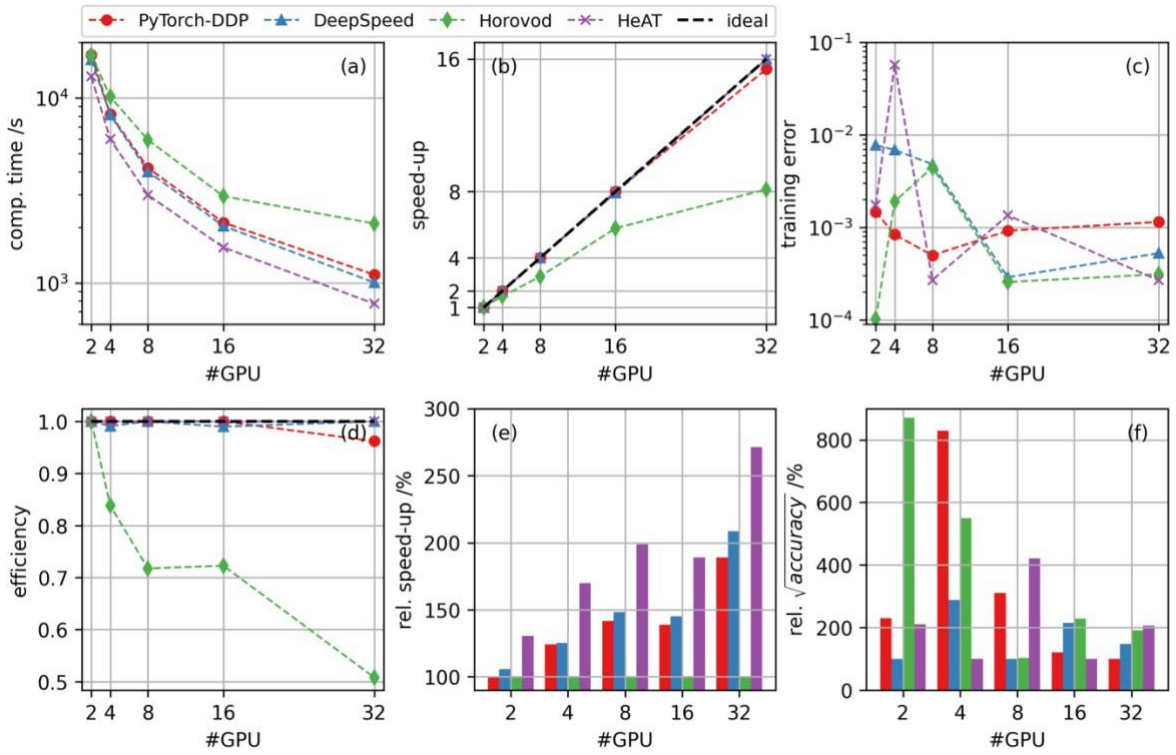

Figure 2: Performance of the existing distributed data parallel frameworks for training on a small version of the ATBL dataset on the CTE-AMD system at BSC. Each node consists of two AMD MI50 GPUs. Depicted are the compute time over the number of GPUs (a), the strong-scaling performance (b), the corresponding training error (c), the code-efficiency under increasing GPU count (d), the relative speed-up (speed-up relative to the slowest among the tested frameworks) (e), and the relative square root of the accuracy (training error) (f). The configurable hyperparameters for each framework are fixed. The black dashed lines represent the ideal scenario. Note the logarithmic scales.

Figure 2 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 for NVIDIA-based systems. A similar analysis was previously presented in Deliverable D2.2. To achieve better statistics the number of epochs was increased by a factor of three as compared to the previous studies.

For each framework, a total of 30 epochs *E=30* is performed, the batch size per GPU is set to *B=96*, and the hyperparameter learning rate is *L=0.01*. Again, the CAE training employs the Adam optimization algorithm [12] with a weight decay parameter of *W=0.003*.

From Figure 2 it is obvious that HeAT clearly outperforms the other distributed data parallel framework due to its scaling performance, see Figure 2(b), and Figure 2(e), and its superior accuracies, see Figure 2(c), and Figure 2(f) which was also documented earlier in Deliverable D2.2. Figure 2(d) shows that the reported efficiencies of the tested frameworks - except for Horovod - are much higher (*e>0.95*) than the efficiencies reported in D2.2 (*e<0.9* for 32 GPUs). Interestingly, the efficiency of Horovod is now even lower compared to the reported efficiency in D2.2 (then: *e=0.61*, now: *e=0.5*). A possible explanation is that only Horovod among these frameworks relies on both, the MPI and RCCL communication libraries rather than solely on

RCCL. Hence, there might be a driver issue between MPI and the ROCm platform. Moreover, yet again, Horovod shows excellent training accuracies for training with a large number of GPUs, see Figure 2(f).

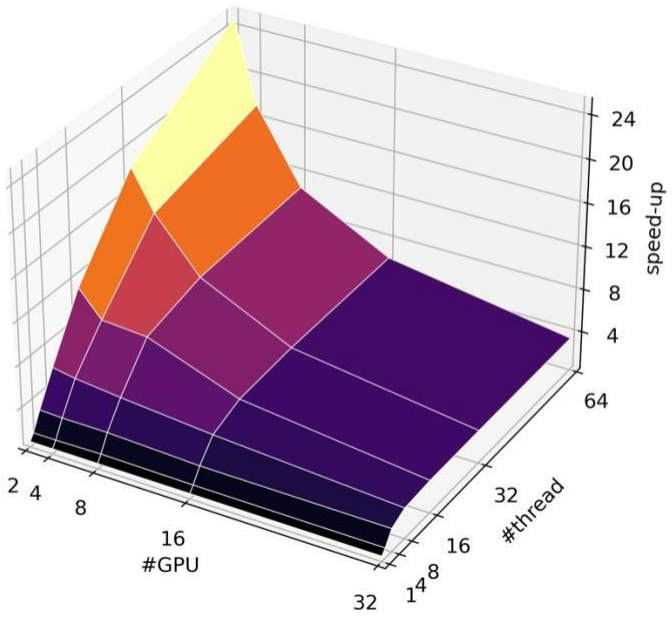

Figure 3: The impact of multi-threaded data loading algorithm on the training durations for training a small version of the ATBL dataset at CTE-AMD. The x-axis and y-axis denote the number of GPUs and the CPU threads per node, respectively. The z-axis shows the speed-up achieved.

Figure 3 shows the impact of the multi-thread data loading algorithm on the ML training durations for the training on a small version of the ATBL dataset (21 GB) on the CTE-AMD machine. An ML training with an exceptionally large dataset suffers from the additional overhead of transferring the data from the disk to the workers. This operation is even more costly for GPU utilization, as the data is first transferred to the system memory controlled by the CPU and then to the GPU memory. To realize this efficiently a good workload balance between the CPUs and GPUs is required. Recent advancements enable the GPU memory to directly access the disk which is unfortunately at present not possible on the prototype systems. Thus, the workload of the CPU can be increased by the multi-threaded data loader option by assigning several CPU threads to each GPU such that the flow of the data from disk to GPU becomes seamless. The data loader library provides comprehensive optimization flags that need to be adjusted for each dataset to balance the CPU and GPU workloads, but the current example in Figure 3 uses the default options. Here, it is visible that employing more CPU threads for a smaller number of GPUs speeds up the ML training up by a factor of 25. For a larger number of GPUs a speed-up of a factor of three is achieved when 16 threads are employed but no additional speed-up is achieved for more threads. As the dataset is split into more GPUs the data loading becomes less of a bottleneck, hence, in this example, employing more CPU threads does not further reduce the training times when many GPUs are used. However, it can be safely assumed that ML training with larger datasets would definitely benefit from using more CPU threads.

2.5 Hyperparameter tuning on prototype systems

The performance of ML models is highly dependent on the hyperparameters that must be manually set before starting the training process. These decisions involve not only the general architecture of the network and the parameters of the optimizer but also the selection of parameters for data pre-processing and regularization methods. Finding the best combination of hyperparameters (here called a “configuration”) is challenging due to several reasons: the evaluation of a single parameter combination (here called a “trial”) requires a complete model training run to be evaluated which for large models or huge datasets demands high computational resources. Furthermore, the search space that the hyperparameters are sampled from is frequently complex and high-dimensional, often including both, continuous variables (e.g., the learning rate), and discrete ones (e.g., the number of layers). Additionally, these hyperparameters change under model and dataset selection, making generalization difficult. On a cluster system like DEEP-EST it is possible to run this hyperparameter optimization in a distributed way with multiple trials in parallel on multiple nodes, thus reducing the overall runtime of the process.

2.5.1 Hyperparameter tuning with Ray Tune

The open-source Ray library provides a universal Python API for adapting single machine code with few code changes to run in a distributed way on multiple machines. Ray is compatible with all common ML frameworks including TensorFlow and PyTorch. It also supports distributed deep learning libraries like Horovod and PyTorch-DDP. The focus of the sub-package Ray Tune is on running distributed hyperparameter tuning scaled to many workers (in this case GPUs). As hyperparameter tuning involves running a lot of trials with different sets of hyperparameters, allocating and launching each trial manually is inefficient. With Ray Tune only a head node needs to be launched and all the worker nodes via a `slurm` script. The head node then connects to the worker nodes and launches the trials. During training the worker nodes report their current status including performance metrics to the head node that terminates low-performing trials or launches new ones. The user specifies the number of resources to use per trial, the hyperparameters and their range, and a scheduling or optimization algorithm. The head node takes care of communication and scheduling tasks.

2.5.2 Porting Ray Tune to DEEP-EST

For installing Ray Tune on DEEP-EST the following modules need to be loaded first:

```
$ ml Stages/2022
$ ml load GCC/11.2.0 OpenMPI/4.1.2 cuDNN/8.3.1.22-CUDA-11.5 \
  NCCL/2.12.7-1-CUDA-11.5 Python/3.9.6 CMake/3.21.1
```

A virtual environment can be created, then Ray and Ray Tune can be installed with the following commands:

```
$ python3 -m venv <env_name>
$ source <env_name>/bin/activate
$ pip3 install ray ray[tune]
```

A Ray Tune job can be submitted to the DEEP-EST system’s job scheduler. An example Slurm script given below.

```
#!/bin/bash
#SBATCH --job-name=<name>
#SBATCH --account=<account>
#SBATCH --output=Ray_tune_test.out
#SBATCH --error=Ray_tune_test.err
#SBATCH  --partition=dp-esb
#SBATCH --nodes=16
#SBATCH --cpus-per-task=8
#SBATCH --gpus-per-node=1
#SBATCH --tasks-per-node=1
#SBATCH --time=01:00:00
#SBATCH --exclusive

# load modules
ml --force purge
ml use $OTHERSTAGES
ml Stages/2022 GCC/11.2.0 OpenMPI/4.1.2 cuDNN/8.3.1.22-CUDA-11.5 \
  NCCL/2.12.7-1-CUDA-11.5 Python/3.9.6 CMake/3.21.1

# source the environment
source <env_name>/bin/activate

# export CUDA settings
export CUDA_VISIBLE_DEVICES="0"

# set maximum pending trials
export TUNE_MAX_PENDING_TRIALS_PG="32"

####### this part is Ray Tune Example slurm script #####
set -x

# __doc_head_address_start___
# Getting the node names
nodes=$(scontrol show hostnames "$SLURM_JOB_NODELIST")
nodes_array=($nodes)
head_node=${nodes_array[0]}
head_node_ip=$(srun --nodes=1 --ntasks=1 -w "$head_node" \
  hostname --ip-address)

# __doc_head_ray_start___
echo "Starting HEAD at $head_node"
srun --nodes=1 --ntasks=1 -w "$head_node" \
     ray start --head --node-ip-address="$head_node"i --port=6379 \
     --num-cpus "${SLURM_CPUS_PER_TASK}" \
     --num-gpus "${SLURM_GPUS_PER_NODE}" --block &
# __doc_head_ray_end___
# __doc_worker_ray_start___
# number of nodes other than the head node
worker_num=$((SLURM_JOB_NUM_NODES - 1))

for ((i = 1; i <= worker_num; i++)); do
  node_i=${nodes_array[$i]}
  echo "Starting WORKER $i at $node_i"
```

```
  srun --nodes=1 --ntasks=1 -w "$node_i" \
       ray start --address "$head_node"i:"6379" \
       --redis-password='5241590000000000' \
       --num-cpus "${SLURM_CPUS_PER_TASK}" \
       --num-gpus "${SLURM_GPUS_PER_NODE}" --block &
  sleep 5
done

echo "Ready"

python -u ray_tune_script.py
```

This script will use the DEEP-EST's extreme scale booster module (`dp-esb`) to run Ray later. It is required to first launch the head node with the `ray start head` command and then afterward launch the worker nodes one by one with the `ray start` command, directing them to the correct head node address with the `address` parameter. Adding an "`i`" behind the name of the head node makes sure the communication goes via the IB network.

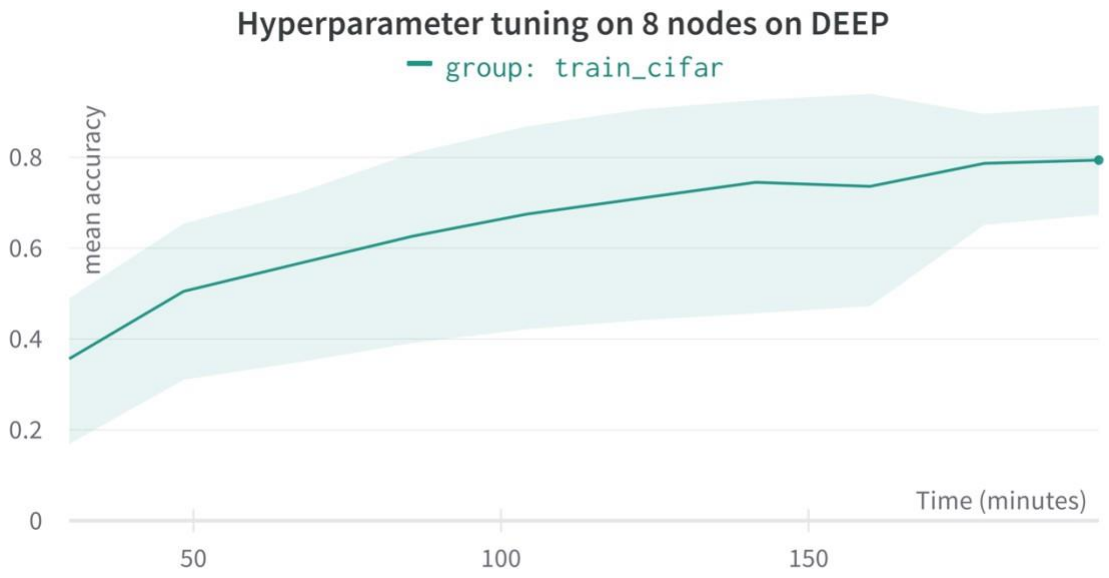

Figure 4: Hyperparameter tuning on the DEEP-EST system with 8 nodes in parallel using Ray Tune. The graph shows the normalized mean accuracy between zero and unity of all trials over time. The highlighted green area depicts the results of the different trials, and the green line depicts the trial-averaged normalized mean accuracy.

In the following the hyperparameters (learning rate and batch size) of a simple CNN on the cifar-10[51] dataset are tuned with the help of Ray Tune. The tuning uses eight nodes of the DEEP-EST in parallel with each trial using one GPU and 8 CPUs. Figure 4 shows how the mean accuracy increases over time as better hyperparameters are found. The mean accuracy is normalized between zero and unity. The mean accuracy before the tuning is at only 0.38 (38%) which increases up to 0.79 (79%) after running the hyperparameter tuning algorithm for three hours indicating the utmost importance of these tunings.

[51] cifar-10 https://www.cs.toronto.edu/~kriz/cifar.html

3 Support activities for Basilisk used by CYI

This section describes the support activities for the open-source software Basilisk[52]. Section 3.1 provides an overview of this software. Subsequently, porting activities, how the code can be executed, and results of first base performance analyses are presented for the machines DEEP-EST (Sec. 3.2), JUAWEI (Sec. 3.3), CTE-AMD (Sec. 3.4), and CTE-ARM (Sec. 3.5).

3.1 Overview of Basilisk

Basilisk is a free software for the solution of partial differential equations on adaptive Cartesian computational domains [15] which evolved from the Gerris CFD code [16]. It follows a second-order accurate, finite volume methodology coupled with an iterative multigrid solver written in C. It only requires a C99-compliant compiler and an MPI implementation, e.g., OpenMPI, for parallelization beyond a single CPU node.

Specifically for multiphase flows the code is using the volume-of-fluid method to distinguish between the liquid and the gas phases, retaining the effects of viscosity and density differences as well as surface tension and gravity [17]. The quadtree grid construction allows for increased grid resolution in the region of the two-fluid interface. The time-integration of the Navier-Stokes equations follows a second-order predictor-corrector scheme[53] and the resulting Poisson equation for the pressure field is solved with a build-in multigrid solver[54]. A more detailed description of the code routines can be found on the code's website. The code performance and scalability were analyzed in various HPC centers with results published on the Basilisk website for the case of the Occigen[55] and Irene[56] supercomputers.

Basilisk will be used to facilitate the generation of CFD samples in wetting hydrodynamics scenarios in Task T3.5 of CoE RAISE. Specific to this research area a number of studies [18-21] validated and verified Basilisk, while the code's website provides further proof of the consistency and stability of the overall computational method (see also the bibliography associated with Basilisk[57]). Other tools that can be used in wetting hydrodynamics setups are phase-field methods, see, e.g. [22,23], and the general CFD solver OpenFOAM[58]. Phase-field methods typically suffer from the excessive diffusion of the two-fluid interface which increases in time and can lead to unphysical solutions especially for contact line problems, where the movement of the droplet is largely driven by the diffusion of the interface along the contact line. Additionally, Basilisk's adoption of the "octree" grid refinement approach and the conservative geometric VOF was found to be advantageous when compared against OpenFOAM, both in terms of accuracy and efficiency [24].

Within Task T3.5 Basilisk will be used to simulate cases of initially spherical droplets that travel across chemically heterogeneous surfaces. The generated datasets will be used for undertaking an exploratory investigation of developing AI-assisted surrogate models in the domain of wetting hydrodynamics, namely by augmenting low-accuracy models with a data-driven part as done, for example, in other contexts, see, e.g. [25,26]. As a large number of simulations are required to gather an adequate sample size for AI training (O(100) cases), each case will consider surfaces with different chemical heterogeneity profiles. More

[52] Basilisk http://basilisk.fr/
[53] Basilisk predictor scheme http://basilisk.fr/src/navier-stokes/centered.h
[54] Basilisk solver http://basilisk.fr/src/poisson.h
[55] Occigen http://basilisk.fr/src/test/mpi-laplacian.c#how-to-run-on-occigen
[56] Irene http://basilisk.fr/src/examples/isotropic.c#scalability-on-irene
[57] Basilisk bibliography http://basilisk.fr/Bibliography
[58] OpenFOAM https://openfoam.org/

specifically, the goal is to learn the non-linear mappings between droplet trajectories and surface features in order to (i) enhance insights about the morphology of surfaces and how droplets evolve on them complementing related experimental efforts, and (ii) help accelerate parametric studies aimed towards designing surfaces in applications for controllable droplet transport.

Within Task T2.2 Basilisk has been ported to and tested on several prototype systems. In the following the corresponding work on the DEEP-EST system (see Sec. 3.2), on the ARM-based JUAWEI system at FZJ (see Sec. 3.3), on the CTE-AMD machine (see Sec. 3.4), and on the CTE-ARM machine (see Sec. 3.5) is reported on.

3.2 Basilisk on the DEEP-EST system at FZJ

This section concentrates on bringing the Basilisk code to the DEEP-EST system. The porting activities are presented in Sec. 3.2.1. This is followed by an explanation of how to execute the code in Sec. 3.2.2. Finally, the results of the first base performance analyses are presented in Sec. 3.2.3.

3.2.1 Porting support

For the compilation the main requirements are a C99-compliant compiler and the CMake utility. Basilisk is installed on the DEEP-EST system using the GCC compiler. First the DEEP-EST system's various development stages are made accessible and the stable stage `Stage/2022` is loaded with:

```
$ ml use $OTHERSTAGES
$ ml Stages/2022
```

This way, several useful libraries are made available in their recent versions. The C compiler is GCC at version number 11.2.0 and the compatible OpenMPI library provides the framework for inter-process communication. A few other modules such as Python, flex[59], and Bison[60] need to be loaded, while the overall compilation process is controlled using CMake. All the modules are loaded as follows:

```
$ ml GCC/11.2.0 OpenMPI/4.1.2 Python CMake/3.21.1 flex/2.6.4 Bison/.3.7.6
```

Currently, DEEP-EST's software stack does not include the required SWIG[61] library which can simply be compiled (exemplary for a version number 4.0.2) on the system to a local directory defined via `--prefix` argument via:

```
$ wget 'https://sourceforge.net/projects/swig/files/swig/swig-
  4.0.2/swig-4.0.2.tar.gz'
$ tar xzf swig-4.0.2.tar.gz
$ cd swig-4.0.2
$ mkdir build
$ ./configure --prefix=$PWD/swig-4.0.2/build --without-pcre
```

[59] flex https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.html
[60] bison https://www.gnu.org/software/bison/
[61] SWIG https://swig.org/download.html

```
$ make -j
$ make install
```

The compiled library must be added to the `PATH` via `export` command. The next step is to download and unpack Basilisk using:

```
$ wget http://basilisk.fr/basilisk/basilisk.tar.gz
$ tar xzf basilisk.tar.gz
```

Once all the files are in place the Basilisk path is exported and the GCC-based configuration file `config.gcc` is linked to the `config` file which is used to guide the compilation. The compilation is completed by running the command `make` two times, first with the `-k` option to continue as much as possible after errors and create dependencies, and then run `make` again without any options to finalize the compilation:

```
$ cd basilisk/src
$ export BASILISK=$PWD
$ export PATH=$PATH:$PWD
$ ln -s config.gcc config
$ make -k
$ make
```

Optionally, the Basilisk path can be added to `.bash_profile` to avoid having to re-type the two `export` commands above at every login:

```
$ echo "export BASILISK=$PWD" >> ~/.bash_profile
$ echo "export PATH=\$PATH:$BASILISK" >> ~/.bash_profile
```

For testing purposes Basilisk is also compiled using the previous software stage `Stage/2020` with an older version of GCC. This stage also includes a ParaStation MPI[62] implementation for inter-process communication, hence, both, OpenMPI and ParaStation MPI can be tested. Similar to stage `Stages/2022`, stage `Stages/2020` is loaded as:

```
$ ml use $OTHERSTAGES
$ ml Stages/2020
```

This stage already contains all the required software and can be loaded with the command below:

```
$ ml GCC/10.3.0 ParaStationMPI/5.4.9-1-mt Python/3.8.5 CMake/3.18.0 \
  flex/2.6.4 Bison/.3.7.6 SWIG/.4.0.2-Python-3.8.5
```

The compilation process of Basilisk is the same as for stage `Stages/2022`, hence, it is not given here.

[62] Parastation MPI https://docs.par-tec.com/html/psmpi-userguide/index.html

3.2.2 Execution of Basilisk

Each test case that uses Basilisk consists of a compliable configuration file, usually denoted as `drop.c`. This file includes all of the computational routines such as the solver, simulation parameters, boundary conditions, etc. This file is compiled using the Basilisk's built-in compiler with MPI support accessed via `qcc` command, as:

```
$ CC99='mpicc -std=c99' qcc -O2 -Wall -D_MPI=1 drop.c -o run -lm
```

This command creates the executable `run` and Basilisk can be executed via `srun` on DEEP-EST:

```
$ srun --mpi=pspmix -K1 -n $SLURM_NTASKS ./run 2> log > out
```

The details of the `srun` flags have been given in Deliverable D2.6. Alternatively, a batch script can be used to submit Basilisk for later execution using the `sbatch` command, where an example is given below:

```
#!/bin/bash
#SBATCH --job-name=<name>
#SBATCH --account=<account>
#SBATCH --output=basilisk.out
#SBATCH --error= basilisk.err
#SBATCH --partition=dp-cn
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=24
#SBATCH --time=00:30:00
#SBATCH --exclusive

# modules
ml ...

# export external libs
export PATH="${path_to_swig}/bin:$PATH"

# UCX settings in DEEP-EST Cluster module
export PSP_OPENIB=1
export PSP_UCP=0

# compile drop.c if needed
CC99='mpicc -std=c99' $path_to_qcc -O2 -Wall -D_MPI=1 drop.c -o run -lm

# run
srun -K1 -n $SLURM_NTASKS ./run
```

This script will use the DEEP-EST's cluster module (`dp-cn`) to run Basilisk later.

3.2.3 First base performance analysis

To assess the scaling behavior of Basilisk on the DEEP-EST system a simple benchmark of a droplet that spreads over a chemically heterogeneous surface developed by CYI is used. A

cubic computational domain with 16 million grid points (256 grid points towards each direction) is selected. The benchmarks are performed using both, the newest `Stages/2022` and older `Stages/2020` stages, noting that the newer stage uses OpenMPI and the older stage uses ParaStation MPI. To run performance analyses the benchmarking tool JUBE is used. Its setup can be found in Sec. 2.3.4.

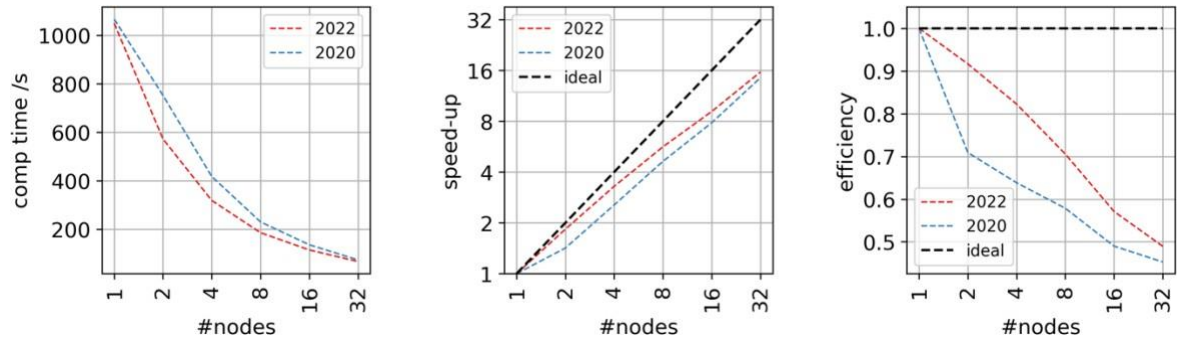

Figure 5: Speed-up of Basilisk on the DEEP-EST system for a benchmark with 16 million computational elements. Strong-scaling results obtained by using nodes consisting of 2 CPUs with each having 12 cores are depicted in red for the newer software stage `Stages/2022` and depicted in blue for the older software stage `Stages/2020`. The ideal scaling and efficiency are represented by the black dashed line. Computational time (left) achieved speed-up (middle) and the efficiency (right) are shown. The output routines are disabled.

Two strong scaling tests using up to 32 compute nodes using both, newer and older software stack stages, are performed on the DEEP-EST cluster module. This cluster module consists of two Intel Xeon Skylake Gold 6146 CPUs with each having 12 cores/24 threads (a total of 24 cores/48 threads per node). The computational domain of Basilisk is decomposed into 24 MPI ranks per node, where CPU's hyperthreading support is enabled and no shared-memory parallelization strategy is considered. The results are shown in Figure 5. Ideally, the computational time is halved when the number of cores (or in this case nodes) is doubled which translates into a doubling of the speed-up. A total of six simulations with various node sizes are tested, ranging from a single node to 32 nodes. It should also be noted that the Cluster module on the DEEP-EST system (`dp-cn`) consists of a total of 50 nodes. Here Basilisk scales well on the DEEP-EST system up to 16 nodes, as the efficiency remains higher than *e>0.5*. For 32 nodes the efficiency is just under the *e>0.5* threshold. As expected the newer software stack is faster and scales better than the older software stack.

3.3 Basilisk on the JUAWEI system at FZJ

Compiling and running Basilisk on the JUAWEI system is similar to running it on the DEEP-EST system. The following discussion is similar to Sec. 3.2. Therefore, only major differences are given. Section 3.3.1 provides an overview of the porting support, Sec. 3.3.2 describes what is necessary to execute the code, and Sec. 3.3.3 presents results of first base performance analyses on the JUAWEI system.

3.3.1 Porting support

The JUAWEI system already includes all of the necessary libraries to compile Basilisk. These libraries can be loaded for ARM-based CPUs using the newest software stack `2021a` by:

```
$ . /opt/ohpc/pub/easybuild/switch_to_eb_sw_stack.sh
$ . /opt/ohpc/pub/easybuild/switch_stage.sh -s 2021a
$ ml GCC/11.1.0 OpenMPI/4.1.2 CMake/3.20.0 Python/3.9.4 flex/2.6.4 \
  Bison/3.7.6 SWIG/4.0.2
```

And for x86-based CPUs the only change is the OpenMPI version number:

```
$ ml GCC/11.1.0 OpenMPI/4.1.0 CMake/3.20.0 Python/3.9.4 flex/2.6.4 \
  Bison/3.7.6 SWIG/4.0.2
```

The rest of the compilation process is the same as given in the previous Sec. 3.2.

3.3.2 Execution of Basilisk

Similar to DEEP-EST the compliable configuration file `drop.c`. is compiled on the JUAWEI system using the Basilisk's built-in compiler with MPI support accessed via `qcc` command, as:

```
$ CC99='mpicc -std=c99' qcc -O2 -Wall -D_MPI=1 drop.c -o run -lm
```

This command creates again the executable `run` and Basilisk can be executed via `srun` on JUAWEI:

```
$ srun --mpi=pmix_v3 -K1 -n $SLURM_NTASKS ./run 2> log > out
```

Alternatively, a batch script can be used to submit Basilisk for later execution using the `sbatch` command. The exemplary batch script as given for DEEP-EST can also be used here after JUAWEI-specific parameters are applied, see Deliverable D2.5.

3.3.3 First base performance analysis

To assess the scaling behavior of Basilisk on the JUAWEI system the same benchmark developed by CYI is used but a smaller domain is considered; a cubic computational domain with 2 million grid points (128 grid points towards each direction).

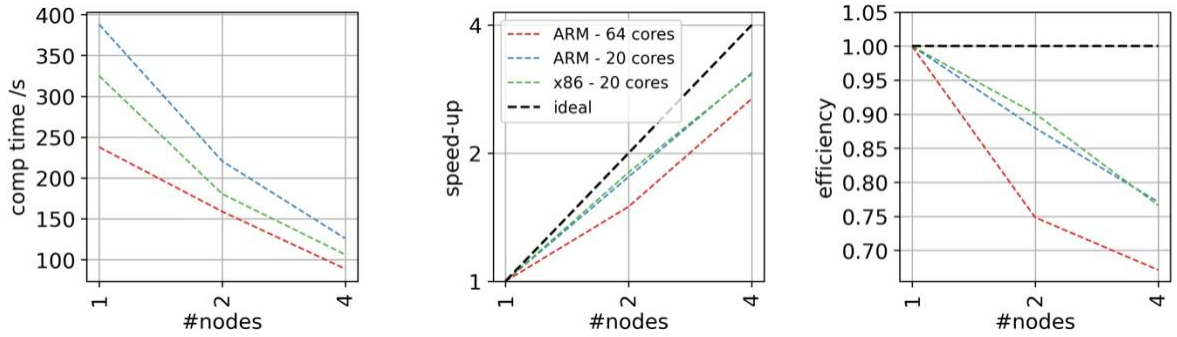

Figure 6: Speed-up of Basilisk on the JUAWEI system for a benchmark with 2.1 million computational elements. Strong-scaling results obtained by using the `Hi1616` module consisting of ARM-based CPUs, while using all 64 cores are depicted in red, and, while using 20 cores are depicted in blue. Results obtained by using the `Haswell` module consisting of x86-based CPUs, while using all 20 cores are depicted in green. The ideal scaling and efficiency are represented by the black dashed line. Computational time (left) achieved speed-up (middle) and the efficiency (right) are shown. The output routines are disabled.

A smaller computational domain is considered due to slower hardware compared to the other prototype systems. Once again, the benchmarking tool JUBE is used.

Strong scaling tests using up to four compute nodes are performed on the JUAWEI system, where the results are presented in Figure 6. This system consists of two modules; the first module (`Hi1616`) consists of two ARM-based HiSilicon Hi1616 CPUs with each having 32 cores, whereas the second module (`Haswell`) consists of two x86-based Intel Xeon E5-2660 with each having 10 cores. For more information on this system please refer to Deliverables D2.5 and D2.6. For testing purposes the comparison includes the `Hi1616` module with all 64 cores and also with 20 cores, so that the number of cores matches the `Haswell` module. The computational domain is then decomposed so that each CPU core is given an MPI rank, CPU's hyperthreading support is disabled and no shared-memory parallelization strategy is considered. The sheer performance of the `Hi1616` module with all 64 cores is better than the `Haswell` module with fewer cores. However, strong scaling results in Figure 6(b) show that the `Hi1616` module with all 64 cores does not scale well from a single to two nodes but scales well from two to four nodes, also visible from efficiencies in Figure 6(c), where e_1-e_2=*0.25* efficiency drop is visible from single (e_1=*1*) to two (e_2=*0.75*) nodes, and only e_2-e_4=*0.07* efficiency drop is visible from two (e_2=*0.75*) to four (e_4=*0.68*) nodes. This is due to the too few computational elements of the chosen test case, meaning that each MPI rank is more busy communicating between nodes than computing. On the other hand it is likely that the `Hi1616` module with 64 cores with a larger benchmark would also perform well.

A better comparison can be made from the `Hi1616` module with 20 cores, where the scaling performance is on par with the `Haswell` module, noting that now the computational times of the `Hi1616` module are slower than the `Haswell` module, e.g., see Figure 6(a). In both modules the efficiencies remain higher than *e>0.77*. This concludes that Basilisk performs well not only on x86-based CPUs but also on ARM-based CPUs.

3.4 Basilisk on the CTE-AMD system at BSC

Compiling and running Basilisk on the CTE-AMD system is slightly different than on DEEP-EST. As mentioned in Sec. 2.3.2 the source files have to be transferred to CTE-AMD via the `sshfs` command. Other than that the structure of this section is the same as in Sec. 3.2, i.e., Sec. 3.4.1 provides an overview of the porting support, Sec. 3.4.2 describes what is necessary to execute the code, and Sec. 3.4.3 presents results of first base performance analyses on the CTE-AMD machine.

3.4.1 Porting support

The CTE-AMD system includes most of the necessary libraries to compile Basilisk. These libraries can be loaded by:

```
$ ml rocm/4.0.1 python/3.9.1 gcc/10.2.0 openmpi/4.0.5 \
  spack-apps/current swig/4.0.2-gcc-tb3 cmake/3.18.2
```

Currently, CTE-AMD's software stack does not include the required `flex` and `bison` libraries. They can, however, be compiled manually on the system to a local directory via (exemplary shown for `flex` with a version number of 2.6.4):

```
$ tar xzf flex-2.6.4.tar.gz
$ cd flex-2.6.4
$ mkdir build
$ ./configure --prefix=$PWD/flex-2.6.4/build
$ make -j
$ make install
```

These compiled libraries are then added to the `PATH` via the `export` command before compiling Basilisk. The rest of the process to compile Basilisk on CTE-AMD is the same as on DEEP-EST.

3.4.2 Execution of Basilisk

Execution of Basilisk on CTE-AMD system is exactly the same as on the DEEP-EST system, i.e.:

```
$ srun -K1 -n $SLURM_NTASKS ./run 2> log > out
```

Alternatively, a batch script can be used to submit Basilisk for later execution using the `sbatch` command. An exemplary batch script has been given in Deliverable D2.5 for the CTE-AMD system.

3.4.3 First base performance analysis

To assess the scaling behavior of Basilisk on the CTE-AMD system the same benchmark developed by CYI is used, where a cubic computational domain with 16 million grid points (256 grid points towards each direction). Once again, the benchmarking tool JUBE is used.

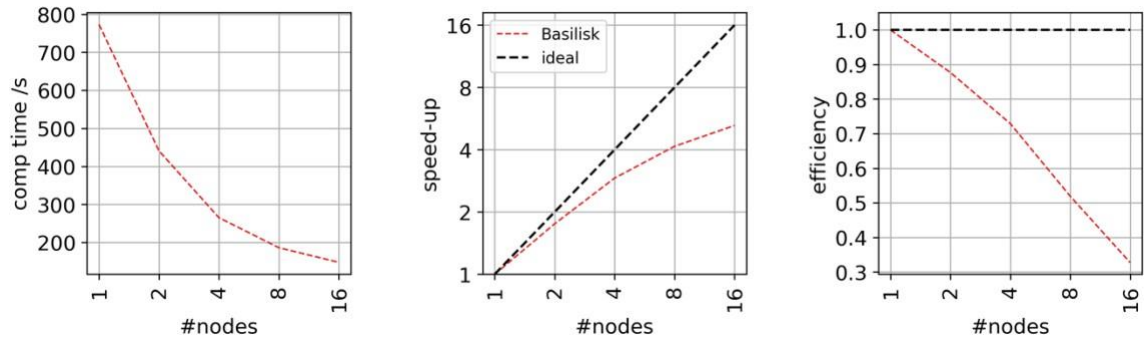

Figure 7: Speed-up of Basilisk on the CTE-AMD system for a benchmark with 16 million computational elements. Strong-scaling results obtained by using nodes consisting of an AMD CPU with 64 cores and 128 threads are depicted in red. The ideal scaling and efficiency are represented by the black dashed line. Computational time (left) achieved speed-up (middle) and the efficiency (right) are shown. The output routines are disabled.

Strong scaling tests on using up to 16 compute nodes are performed on the CTE-AMD system. This system consists of an AMD EPYC 7742 CPU which has 64 cores and 2 threads/core (total of 128 threads per node). The computational domain of Basilisk is decomposed into 128 MPI ranks per node, where CPU's hyperthreading support is enabled and no shared-memory parallelization strategy is considered. The results are shown in Figure 7. A total of five simulations with various numbers of nodes are tested - ranging from a single node to 16 nodes.

It should also be noted that there are 33 compute nodes on the CTE-AMD machine. The scaling performance of Basilisk on CTE-AMD with AMD CPUs is acceptable up to 8 nodes, where the efficiency remains higher than *e>0.5*. For 16 nodes, the efficiency is as low as *e=0.33*.

3.5 Basilisk on the CTE-ARM system at BSC

Different ARM compilers exist to compile and run Basilisk on the CTE-ARM system. Currently, only GCC with OpenMPI can compile Basilisk on CTE-ARM, whereas the Fujitsu compiler, specially tailored for the CPUs in CTE-ARM (Fujitsu FX1000), showed compatibility issues. Hence, this section describes the Basilisk compilation with the GCC approach. However, it should be noted that node-based communication is not possible with GCC and the OpenMPI compilation method.

Other than that the structure of this section is the same as in Sec. 3.2, i.e., Sec. 3.5.1 provides an overview of the porting support, Sec. 3.5.2 describes what is necessary to execute the code, and Sec. 3.5.3 presents results of first base performance analyses on CTE-ARM.

3.5.1 Porting support

An important characteristic of the CTE-ARM system is that the login nodes do not share the same architecture as the computing nodes. The login nodes are comprised of Intel Xeon Silver 4216 CPUs, and thus, compiling for the ARM (a64fx) architecture requires cross-compilation. The recommended way to compile the code is to use an interactive session on a compute node or send a compilation job. Hence, the following commands are to be issued after an interactive session is created with the commands below, where the definitions of the flags are given in the next Sec. 3.5.2.:

```
$ pjsub --interact -L node=1 -L elapse=01:00:00 --mpi "proc=48" \
  --mpi rank-map-bynode -L rscunit=rscunit_ft02,rscgrp=large,freq=2200
```

The CTE-ARM system does not include the necessary libraries to compile Basilisk. Compiling them is, however, trivial. First GCC and OpenMPI are loaded to the CTE-ARM system by:

```
$ ml gcc/11.1 openmpi/4.0.5
```

The required SWIG, flex, and bison libraries can be compiled on CTE-ARM to a local directory after they are transferred to the system with the aforementioned `sshfs` method. Exemplary, the below code compiles flex with version number 2.6.4.:

```
$ tar xzf flex-2.6.4.tar.gz
$ cd flex-2.6.4
$ mkdir build
$ ./configure --prefix=$PWD/flex-2.6.4/build
$ make -j
$ make install
```

These compiled libraries are then added to the `PATH` via `export` command before compiling Basilisk. Subsequently, a slight modification is needed to the automatic configuration file of Basilisk `$Basilisk/src/config.gcc`, where the `CC` variable must be defined:

```
$ head ./basilisk/src/config.gcc
# -*-Makefile-*-

# add CC's definition for CTE-ARM
CC=mpicc

# how to launch the C99 compiler
CC99 = $(CC) -std=c99 -pedantic -D_GNU_SOURCE=1 -Wno-unused-result

# how to strip unused code
STRIPFLAGS = -s
...
```

The rest of the compilation process on CTE-ARM is the same as on CTE-AMD.

3.5.2 Execution of Basilisk

The execution of Basilisk on the CTE-ARM system is slightly different, as it uses PJM[63] as the workload manager. Therefore, the traditional Slurm commands must be replaced by PJM commands, see Deliverable D2.6. An interactive session to compile and execute Basilisk requires the following command to be issued:

```
$ pjsub --interact -L node=1 -L elapse=01:00:00 --mpi "proc=48" \
  --mpi rank-map-bynode -L rscunit=rscunit_ft02,rscgrp=large,freq=2200
```

This command will start an interactive session on ARM-based compute nodes of the CTE-ARM system with 48 cores for an hour. Alternatively, a batch script can be used to submit a job tailored for Basilisk for later execution using the same `pjsub` command. An exemplary batch script has been given in Deliverable D2.6.

3.5.3 First base performance analysis

To assess the scaling behavior of Basilisk on the CTE-AMD system the same benchmark developed by CYI is used, where the smaller cubic computational domain with 2 million grid points (128 grid points towards each direction). Once again, the benchmarking tool JUBE is used. As noted earlier, using GCC in combination with the OpenMPI library does not allow node-to-node communication, hence, only single node performance is discussed here.

Strong scaling tests using up to 48 cores of a single compute node are performed on the CTE-ARM system. This system consists of a A64FX CPU (Armv8.2-A + SVE) @ 2.20GHz (4 sockets and 12 CPUs/socket, total 48 CPUs per node) by Fujitsu (FX1000). The results are shown in Figure 8. A total of four simulations with a various number of cores on a single node are tested. It is shown that the scaling performance of Basilisk on CTE-ARM with ARM CPUs is acceptable even up to the full 48 cores, where the efficiency remains higher than *e>0.5*.

[63] PJM https://www.ibm.com/docs/en/was-zos/9.0.5?topic=application-parallel-job-manager-pjm

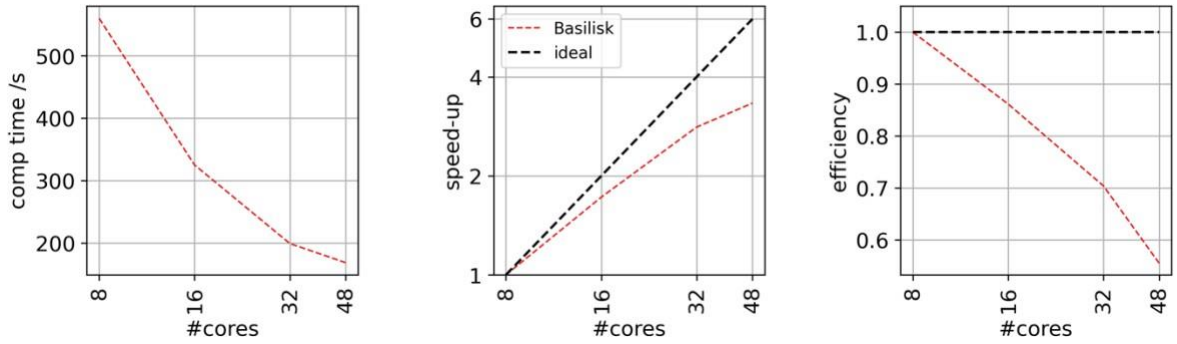

Figure 8: Speed-up of Basilisk on the CTE-ARM system for a benchmark with 2.1 million computational elements. The strong-scaling results (depicted in red) are obtained by using a single node consisting of a 48-core ARM CPU. The ideal scaling and efficiency are represented by the black dashed line. The computational time (left) achieved speed-up (middle), and the efficiency (right) are shown. The output routines are disabled.

4 Support activities for Alya from BSC

This section describes the support activities performed for the Alya code from BSC in project months from M6 to M18 or CoE RAISE in continuation of the activities reported in Deliverable D2.6.

In the following, initially, the Alya use cases are introduced in Sec. 4.1. Then, the porting activities and results of base performance analyses are presented for a Huawei cluster at BSC[64], and the CTE-AMD machine in Sec. 4.2, and Sec. 4.3, respectively.

4.1 Use cases for Alya

For the performance analyses two use cases are considered and the Alya code is executed for 20 timesteps. The number of timesteps is found to be enough for converged statistics, and no workload imbalances during the initialization phase of the simulations have been observed. The first case, referred to as *Wind farm 1* solves the Reynolds-Averaged Navier Stokes (RANS) equations for the simulation of the flow over the terrain. The *k-ε* turbulence model which is specially adapted for atmospheric boundary layers is used as a closure model. The RANS equations are solved with Alya's *Nastin* module, while the *Turbul* module takes care of the turbulence model operations. The solution method is implicit. The computational domain for this case consists of 9.2 million hexahedral computational elements. The geometry and the computational domain are shown in Figure 9.

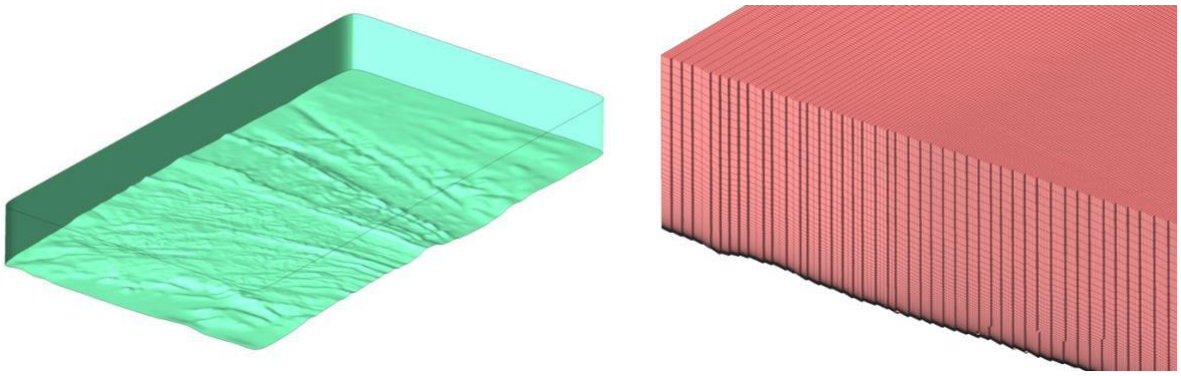

Figure 9: Wind farm 1 use case. Geometry and detail of the boundary layer computational domain. The length of the domain in the flow direction is 24kms, while the height is 3kms.

The second use case is referred to as *Wind farm 2*, where Large-Eddy Simulation (LES) of a thermal turbulent flow over the flat terrain is considered, and, where the turbulence is modelled via the *Vreman* LES turbulence model. The domain involves both, roughness and canopy models. In LES the time integration of the Navier-Stokes equations is performed using the semi-implicit fourth order Runge-Kutta method, whereas the pressure is corrected using an implicit pressure correction method with the fractional step approach. The Navier-Stokes equations are solved with Alya's *Nastin* module, while the temperature model is solved by the *Temper* module. With respect to the first use case, the workload of the CPU is lower as the transport equations for *k* and *ε* are replaced by a single transport equation for the pressure. The computational domain is composed of 25.6 million computational elements. Figure 10 shows the geometry and the computational domain.

[64] Huawei cluster at BSC: https://www.bsc.es/supportkc/docs/Huawei/intro

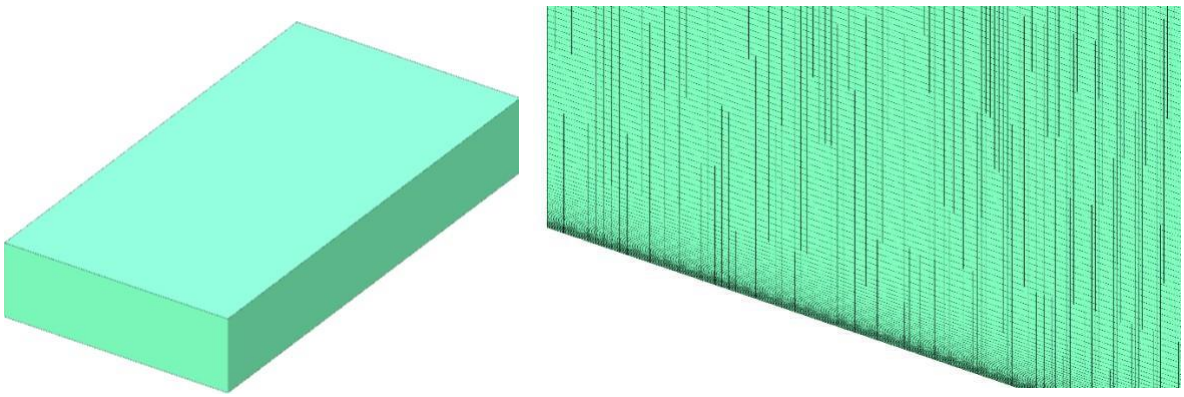

Figure 10: Wind farm 2 use case. Geometry and details of the boundary layer computational domain. The length of the domain in the flow direction is 26kms, while the height is 4kms.

4.2 Alya on the Huawei system at BSC

As the Huawei system has not been described previously, a brief overview of the system is given in Sec. 4.2.1. This is followed by a tutorial on how to port Alya to this system in Sec. 4.2.2. Finally, some first results of base performance analyses on the Huawei system are provided in Sec. 4.2.3.

4.2.1 Description of the Huawei cluster

The Huawei cluster is a supercomputer based on Kunpeng 920 processors which are based on the ARMv8.1 architecture and locate at BSC. It is a system using a Mellanox high-performance network interconnect and running CentOS 7.6 as the operating system. The computing elements of this cluster are split into 3 different modules:

1. General purpose computing module that has 16 nodes. Each node has two Kunpeng 920 CPUs (ARMv8.1) consisting of 64 cores each @ 2.6GHz and 256 GB of RAM.
2. Dedicated module for AI Training. This module has two Kunpeng 920 CPUs (ARMv8.1) consisting of 64 cores each @ 2.6GHz and 1 TB of RAM and eight Ascend 910A (Huawei Accelerators).
3. Dedicated module for AI Inference. This module has two Kunpeng 920 CPUs (ARMv8.1) consisting of 64 cores each @ 2.6GHz and 256 GB of RAM and five Atlas 300C (Huawei Accelerators based on IA Ascend 310 processors).

It should be noted that the BIOS and kernel reserve memory, i.e., the actual total usable RAM that commands like `free` or `lstopo` commands would report slightly lower values than the total theoretical amount of RAM.

4.2.2 Porting support

Alya can be compiled in two ways either using a configuration file or with CMake. For the moment Alya is ported using only the former method. Initially, the following modules have to be loaded:

```
$ ml openhpc/current gnu8 gcc openmpi/4.1.3 cmake/3.22.1
```

An `unset INCLUDE` command is mandatory because Alya's Makefile already uses it, while one of the modules imposes a specific path:

```
$ unset INCLUDE
```

Finally, the following flags are used in the configuration file `$path_to_alya/configure/config.in`:

```
FCFLAGS = -c -J$O -I$O -ffree-line-length-none -fimplicit-none \
  -DFORCE_END=3 -DALYA_CHM_FINITERATE_TIMMING \
  -fallow-argument-mismatch -gdwarf-4 -g

EXTRALIB = -lc -gdwarf-4 -g
```

4.2.3 First base performance analyses

First strong scalability analyses were performed using the *Wind farm 1* use case. Table 3 shows the corresponding timings obtained on the Huawei cluster.

	Huawei			
MPI tasks	128	256	384	512
Node(s)	1	2	3	4
Total time /s	1005	642	544	823
Iterations time /s	578	300	244	190

Table 3: Wind farm 1 use case. Timings for Huawei.

Figure 11 shows the speed-up obtain for this example. The top x-scale shows the load in terms of elements per core, while the lower x-scale represents the number of employed cores for the computation. Obviously at 256 cores the number of elements is still sufficient to yield a decent speed-up from 128 cores. Further increasing the number of cores also reduces the number of elements explaining the breakdown in the speed-up curve above 256 cores.

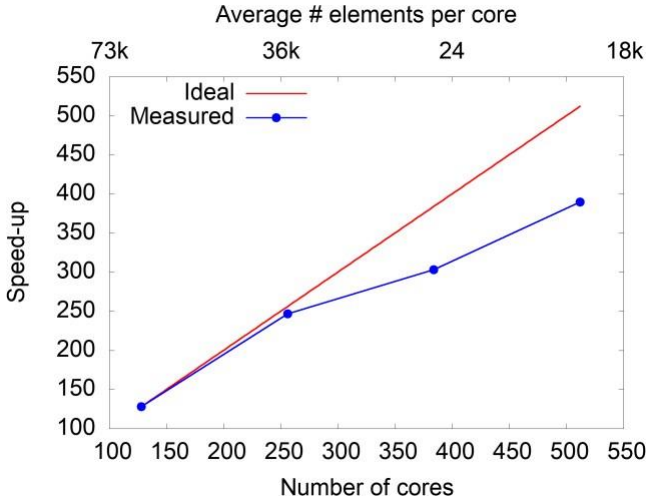

Figure 11: Wind farm 1 use case. Speed-up of Alya obtained on the Huawei cluster for Alya's *Nastin* module.

That is, the computation on a smaller amount of elements is much faster, and hence its share in the whole computational time is reduced. At the same time the share of the communication is increased.

4.3 Alya on the CTE-AMD system at BSC

This section first reports on the porting activities around the Alya code on the CTE-AMD machine in Sec. 4.3.1. This follows the first base performance analyses of the code on this system in Sec. 4.3.2.

4.3.1 Porting support

The necessary steps to port Alya to the CTE-AMD machine was previously presented in the previous Deliverable D2.6. In addition to these works two different Intel compilers which will be referred to as `intel/2018` and `intel/2022`, respectively, have been tested. With the help of the support team at BSC changes to the module definition were made. It is necessary to enable a proper definition of the compilers when using CMake. For `intel/2018` the following modules now need to be loaded:

```
$ ml --force purge
$ ml intel/2018.4 impi/2018.4 cmake
```

For `intel/2022` the following modules need to be used:

```
$ ml --force purge
$ ml intel/2022.0.2 impi/2018.4 cmake
```

The CTE-AMD nodes are comprised of one AMD EPYC 7742 CPU with 64 cores and two threads per core (total of 128 threads per node). To use 64 cores the batch script needs to contain `#SBATCH --cpus-per-task=2`. To do hyperthreading and use fewer nodes the variable should be set to `#SBATCH --cpus-per-task=1`. To enable hyperthreading, that is forcing 128 MPI tasks per node, the following batch script can be used:

```
#!/bin/bash
#SBATCH --job-name=<name>
#SBATCH --output=alya.out
#SBATCH --error=alya.err
#SBATCH --ntasks=128
#SBATCH --cpus-per-task=1
#SBATCH --time=00:30:00
#SBATCH --exclusive

export ALYA_DIR=$alya_dir
source $ALYA_DIR/env.sh
export ROMIO_HINTS=/apps/ALYA/hints/io_hints
export I_MPI_EXTRA_FILESYSTEM=gpfs
```

```
export I_MPI_EXTRA_FILESYSTEM=on

srun $ALYA_DIR/src/alya/alya wind-farm-1
```

To disable hyperthreading and use 64 MPI tasks per node `#SBATCH --cpus-per-task=2` needs to be used.

4.3.2 First base performance analyses

For the performance tests the same example as for the Huawei cluster, see Sec. 4.2.3, i.e., the *Wind farm 1 case* is used. Table 4 shows the results obtained for the timings.

	intel/2018			intel/2018			Intel/2022			Intel/2022		
CPU/task	2			1 (hyperthreading)			2			1 (hyperthreading)		
MPI tasks	64	128	256	64	128	256	64	128	256	64	128	256
Node(s)	1	2	4	1	1	2	1	2	4	1	1	2
Total time /s	1145	552	289	1442	1226	631	1200	577	302	1536	1287	653
Compute time /s	776	358	182	965	874	432	851	394	200	1085	958	470

Table 4: Wind farm 1 use case. Timings for AMD using the *Nastin* module.

From the results of Table 4 and the right graph in Figure 12 it is obvious that hyperthreading is not an efficient option for this use case. Consequently the effects of code vectorization will be examined more deeply as in principle; hyperthreading offers the possibility to use half of the resources. In contrast to hyperthreading the scaling performance without hyperthreading, as shown in the left graph of Figure 12, is with a super-linear scaling behavior extremely well from 64 to 256 cores.

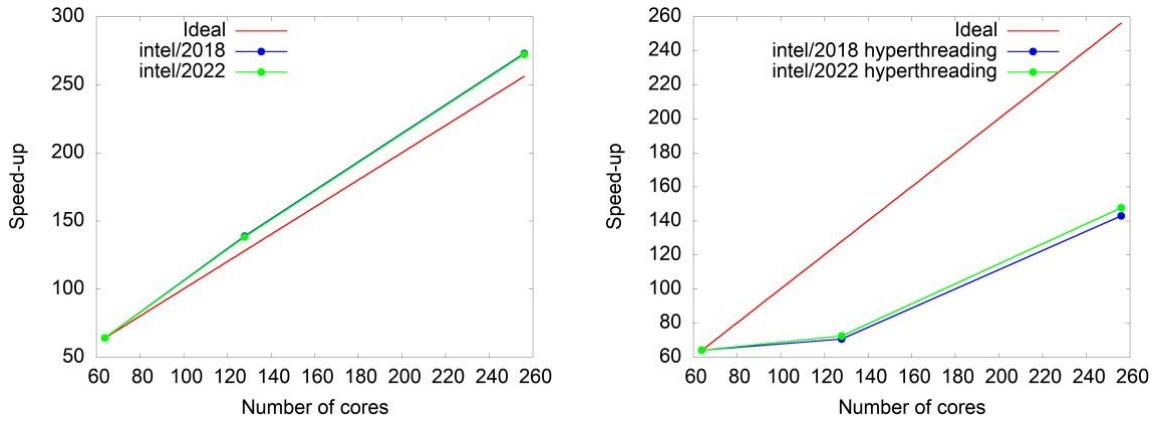

Figure 12: Wind farm 1 use case. Speed-up on CTE-AMD for Alya's *Nastin* module. Without hyperthreading (left) and with hyperthreading (right).

Figure 13 shows the time to the solution with and without using hyperthreading - this time in terms of the number of nodes. Again, the results show that hyperthreading is not an efficient option for this use case. Figure 14 shows the different relative timings for the *Wind farm 1* use case for simulations with 64 MPI tasks and 256 MPI tasks. Obviously not many

differences can be observed except for the simulation initialization (marked as *Init* in Figure 14), in which relative importance increases with an increasing number of cores.

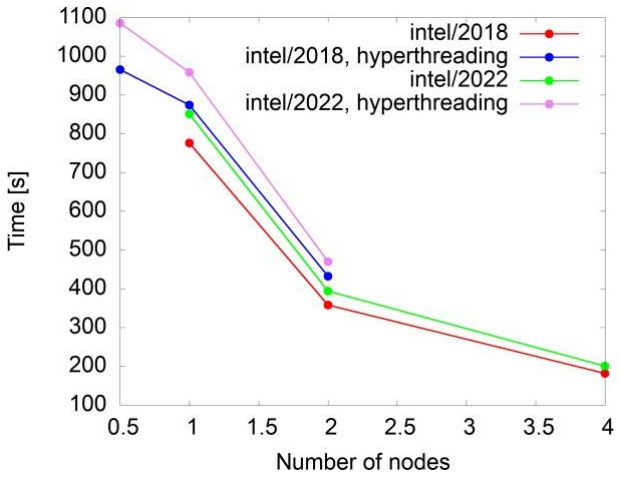

Figure 13: Wind farm 1 use case. Timings with and without hyperthreading on CTE-AMD for different number of nodes.

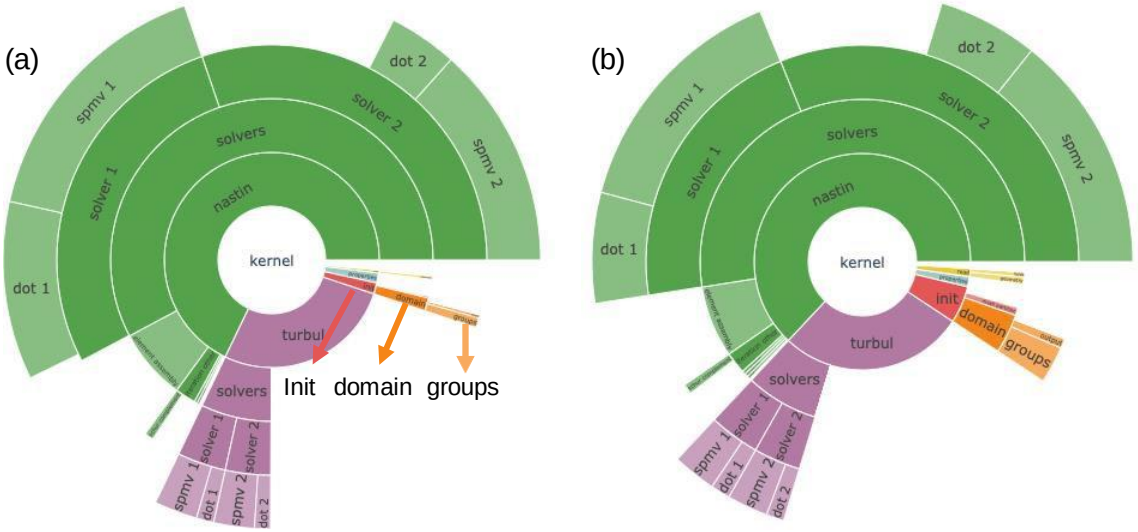

Figure 14: Wind farm 1 use case. Relative timings of the different operations using the intel/2018 compiler for simulations with 64 (a) and 256 (b) MPI tasks, respectively.

Finally, the *Wind farm 2* use case was analyzed on the CTE-AMD machine. Table 5 shows the timings obtained for different numbers of MPI tasks without using hyperthreading.

MPI tasks	128	256	512
Nastin time /s	445	219	110
Temper time /s	66	36	22
Iteration time /s	505	255	132
Total time /s	802	516	430

Table 5: Wind farm 2 use case. Timings for the different modules CTE-AMD.

Figure 15 shows the corresponding speed-up graphs. The speed-up of the temperature equation is quite worse than the one of *Nastin* which shows super-linear behavior. The

temporal scheme is explicit and should scale as well as the *Nastin* one (the fractional step method involves a pressure solver). At this stage we do not have any explanation for this behavior. However, the overall scaling performance of Alya is only marginally affected by the performance of the *Temper* module, as the time of *Temper* module is only a small fraction of the total iteration time (between 5-8%).

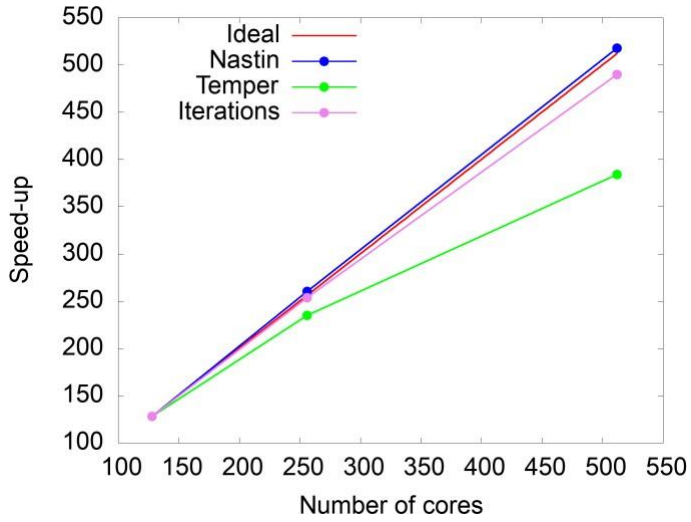

Figure 15: Wind farm 2 use case. Speed-up of the different modules and complete iterations on CTE-AMD.

5 Support activities for bringing m-AIA from RWTH to CTE-AMD

Instructions on how to port the m-AIA code to the DEEP-EST and JUAWEI systems were previously given in Deliverable D2.6. In continuation of this work m-AIA is additionally ported to the CTE-AMD system at BSC. This section reports on the corresponding activities by first presenting the necessary porting instruction in Sec. 5.1. This is followed by a description on how to execute the code on this specific machine in Sec. 5.2. Finally, some results of base performance analyses are presented in Sec. 5.3.

5.1 Porting support

Porting m-AIA to the CTE-AMD system is similar to the DEEP-EST system, previously presented in Deliverable D2.6. The following command loads all the required modules needed: the GNU Compiler Collection (GCC), the communication library OpenMPI, and CMake.

```
$ ml bsc/current gcc/10.2.0 openmpi/4.0.5 cmake/3.18.2
```

The discrete Fourier transformations required by m-AIA, e.g., for the initialization of turbulent fields, are handled by an external software library, the Fastest Fourier Transform in the West (FFTW)[65] which is not available on the CTE-AMD system with GCC. This library with version number 3.3.10 can, however, manually be compiled to the `fftw_build_dir` directory using the following commands:

```
$ tar xzf fftw-3.3.10.tar.gz
$ cd fftw-${verFF}
$ mkdir <fftw_build_dir>
$ ./configure –prefix=<fftw_build_dir> --enable-mpi --with-pic \
  --enable-openmp --disable-fortran --enable-shared --enable-threads \
  CC=mpicxx CXX=mpicxx
$ make -j
$ make install
```

The configuration flags enable an MPI and OpenMP parallelization for the `FFTW` library which are required for m-AIA. Similarly, the post-processing dependency parallel-NetCDF can be compiled externally for version number 1.12.3 into the directory given by the `--prefix` argument:

```
$ tar xzf pnetcdf-1.12.3.tar.gz
$ cd pnetcdf-1.12.3
$ mkdir <pnetcdf_build_dir>
$ ./configure –prefix=<pnetcdf_build_dir> MPICC=mpicxx MPICXX=mpicxx
$ make -j
$ make install
```

The corresponding tar-ball files need to be copied to the system via the `sshfs` client. The m-AIA code includes a configuration script (`configure.py`) that handles the compilation and installation of m-AIA to the subdirectory `$MAIA/src`. This configuration script automatically

[65] `FFTW` http://www.fftw.org

detects systems around the world frequently used by the m-AIA community. To add the CTE-AMD machine to the list of supported systems the `$MAIA/cmake/GetHost.cmake` file first needs to be modified:

```
$ cat $MAIA/cmake/GetHost.cmake
  ...
  elseif (${_raw_host} MATCHES "amd*")
    set(${_var} "CAMD" PARENT_SCOPE)
  ...
```

Then, a new file in the `$MAIA/auxiliary/hosts/` directory must be generated using the initials of the CTE-AMD system – in this case `CAMD.cmake`. This file must include the system-specific information for the locations of the dependencies and the default compilation options in case not specified at configuration time. A sample system file is provided through the file `$MAIA/auxiliary/hosts/Host.cmake.in`. This system file is optimized for the DEEP-EST system and is available in the *FZJ_RAISE* branch of the GIT repository[66], previously introduced in Deliverable D2.6. Finally, m-AIA is compiled on CTE-AMD with GCC:

```
$ ./configure.py 1 2 –reset
$ make -j
```

5.2 Execution of m-AIA

The execution of m-AIA on CTE-AMD system is exactly the same as on the DEEP-EST system, i.e.:

```
srun -p dp-cn -N 2 -n 8 -t 00:05:00 ./$PATH_TO_MAIA properties_run.toml
```

The details on the property file of m-AIA `properties_run.toml` were previously described in Deliverable D2.6. Moreover, a batch script can be used to submit m-AIA for later execution using the `sbatch` command. An exemplary batch script has been given in Deliverable D2.5 for the CTE-AMD system.

5.3 First base performance analyses

To assess the scaling behavior of m-AIA on the CTE-AMD system the Taylor-Green Vortex (TGV) case used also in Deliverable D2.6 is considered. A cubic computational domain with 16 million grid points (256 grid points towards each direction) is used. However, now the simulation employs the Lattice-Boltzmann solver of m-AIA and uses a structured computational grid. Therefore, both, MPI (distributed-memory) and OpenMP (shared-memory) parallelization methods could be used. The benchmarking tool JUBE is used.

Eight strong scaling tests using up to 16 compute nodes are performed on the CTE-AMD system, where the results are shown in Figure 16. Each scaling test uses a different set of MPI tasks and OpenMP threads. For example, the blue line in Figure 16 represents two threads meaning two MPI ranks per node, where each MPI rank has 64 CPU threads for OpenMP parallelization, leading to the maximum available core count per node in CTE-AMD (2*64=128

[66] m-AIA GIT-branch: https://git.rwth-aachen.de/aia

cores). For this exemplary case the best performance is achieved when 8 tasks and 16 threads per node are employed, where the efficiency is close to *e=0.8* up to 8 nodes. For 16 nodes the scaling results are not satisfactory where a larger test with more computational elements should have been used. As 16 nodes are half of the CTE-AMD system achieving a good scaling performance of up to 8 nodes should be considered as sufficient.

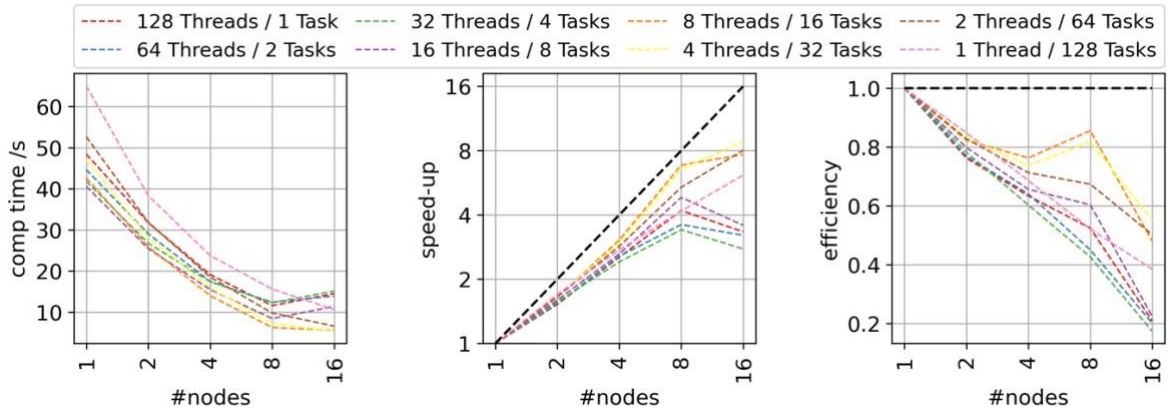

Figure 16: Speed-up of m-AIA on the CTE-AMD system for a benchmark with 16 million computational elements. Strong-scaling results obtained by using nodes consisting of an AMD CPU with various cores and threads are presented. The ideal scaling and efficiency are represented by the black dashed line. The computational time (left) achieved speed-up (middle), and the efficiency (right) are shown. The output routines are disabled.

6 Summary and conclusions

This document is the second and final document in a row of two reporting on support activities that took place within the scope of Task T2.2 in WP2. The second document covered a period of the past 12 months, between project months M6 and M18 – the first document reported on the activities from M1 to M6. Cases relevant to the use cases of WP3 and WP4 were considered.

Within the context of T2.2 existing ML frameworks were ported to the prototype systems DEEP-EST and CTE-AMD without any fundamental structural code changes. That is, each ML framework was ported by issuing simple commands. Therefore, the focus of the activities was on the porting, initialization, and first-base performance analysis of these prototype systems in correspondence to different AI tasks. Each step to port and initialize the tested ML frameworks to the prototype systems was carefully documented. The first-base performance analysis of these prototype systems showed that even production runs were possible due to the achieved good scaling efficiencies and bug-free environment.

The simulation code Basilisk used by the project partner CYI was compiled and run on the prototype systems DEEP-EST and JUAWEI at FZJ, and on CTE-AMD and CTE-ARM machines at BSC, where detailed instructions were provided. The required software libraries were identified and the scaling performance of Basilisk on these systems was analyzed, noting that node-based communication on the CTE-ARM system was not working. Hence, only single node results were provided. The code was able to show good scaling performance on these prototype systems.

In M6 – M18 the support for the Alya code from BSC was continued. The previous Deliverable D2.6 (M1 – M6) showed how Alya can be compiled and run on four different prototype systems. This Deliverable focused on the Huawei prototype system at BSC which was additionally made available to the CoE RAISE project, and on the analysis of the performance using a different software stack on the CTE-AMD system at BSC. Similarly, required software libraries were identified and the scaling performance was analyzed.

Finally, in continuation of the work reported in Deliverable D2.6, the Multiphysics simulation code m-AIA from RWTH was compiled and run on the prototype system CTE-AMD. Detailed instructions were provided, the required software libraries were identified, and the scaling performance of m-AIA was analyzed. A slightly complicated case that employed both, distributed and shared memory parallelization with MPI and OpenMP was selected. This test showed that a good distribution of CPU cores to individual MPI and OpenMP tasks can yield good scaling performance. However, unlike other prototype systems CTE-AMD showed promising performance close to a production system (e.g., JURECA-DC).

It was once again demonstrated that the ML and simulation frameworks relevant to WP3 and WP4 can benefit from using the prototype systems mentioned in this document. It was evident that both, the DEEP-EST and CTE-AMD prototype systems can even handle production runs if needed. Most importantly, the knowledge extracted when testing these systems plays a crucial role in CoE RAISE, especially for complex simulation- and AI workflows at Exascale. The documentation and results in both previous Deliverable D2.6 and this Deliverable D2.7 will be made available to system maintainers and operators, so that the best possible performance is extracted from such prototype systems and the obtained knowledge is transferred to production systems. And finally, through planned workshops, important feedback from system maintainers and operators will be gathered and evaluated.

References

[1] 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

[2] 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

[3] 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

[4] Deng, L. (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

[5] 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

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

[7] 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

[8] 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

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

[10] 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

[11] 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

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

[13] 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

[14] 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

[15] Popinet, S. (2015). A quadtree-adaptive multigrid solver for the Serre–Green–Naghdi equations. Journal of Computational Physics, 302, 336–358. https://doi.org/10.1016/j.jcp.2015.09.009

[16] Popinet, S. (2003). Gerris: a tree-based adaptive solver for the incompressible Euler equations in complex geometries. Journal of Computational Physics, 190(2), 572–600. https://doi.org/10.1016/S0021-9991(03)00298-5

[17] Tryggvason, G., Scardovelli, R., & Zaleski, S. (2011). Direct numerical simulations of gas–liquid multiphase flows. Cambridge university press. Retrieved from https://doi.org/10.1017/CBO9780511975264

[18] Fullana, T., Zaleski, S., & Popinet, S. (2020). Dynamic wetting failure in curtain coating by the Volume-of-Fluid method. The European Physical Journal Special Topics, 229(10), 1923-1934. https://doi.org/10.1140/epjst/e2020-000004-0

[19] Negus, M. J., Moore, M. R., Oliver, J. M., & Cimpeanu, R. (2021). Droplet impact onto a spring-supported plate: analysis and simulations. Journal of Engineering Mathematics, 128(1), 1–27. https://doi.org/10.1007/s10665-021-10107-5

[20] Dalwadi, M. P., Cimpeanu, R., Ockendon, H., Ockendon, J., & Mullin, T. (2021). Levitation of a cylinder by a thin viscous film. Journal of Fluid Mechanics, 917. https://doi.org/10.1017/jfm.2021.284

[21] Fudge, B. D., Cimpeanu, R., & Castrejón-Pita, A. A. (2021). Dipping into a new pool: The interface dynamics of drops impacting onto a different liquid. Physical Review E, 104(6), 065102. https://doi.org/10.1103/PhysRevE.104.065102

[22] Huang, Z., Lin, G., & Ardekani, A. M. (2020). Consistent, essentially conservative and balanced-force phase-field method to model incompressible two-phase flows. Journal of Computational Physics, 406, 109192. https://doi.org/10.1016/j.jcp.2019.109192

[23] Shahmardi, A., Rosti, M. E., Tammisola, O., & Brandt, L. (2021). A fully Eulerian hybrid immersed boundary-phase field model for contact line dynamics on complex geometries. Journal of Computational Physics, 443, 110468. https://doi.org/10.1016/j.jcp.2021.110468

[24] Frederix, E., Hopman, J. A., Karageorgiou, T., & Komen, E. M. (2020). Towards direct numerical simulation of turbulent co-current Taylor bubble flow. Retrieved from https://doi.org/10.48550/arXiv.2010.03866

[25] Wan, Z. Y., Vlachas, P., Koumoutsakos, P., & Sapsis, T. (2018). Data-assisted reduced-order modeling of extreme events in complex dynamical systems. PloS one, 13(5), e0197704. https://doi.org/10.1371/journal.pone.0197704

[26] Wan, Z. Y., & Sapsis, T. P. (2018). Machine learning the kinematics of spherical particles in fluid flows. Journal of Fluid Mechanics, 857. https://doi.org/10.1017/jfm.2018.797

List of Acronyms and Abbreviations

AI	Artificial Intelligence
AIA	Aerodynamic Institute Aachen (Institute of Aerodynamics and Chair of Fluid Mechanics)
ARM	Advanced RISC Machines
ATBL	Actuated Turbulent Boundary Layer
BS	Batch Size
BSC	Barcelona Supercomputing Center
BSCW	Basic Support for Cooperative Work
CAE	Convolutional Auto-Encoder
CERFACS	Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique, France
CFD	Computational Fluid Dyanmics
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
CTE	Cluster de Tecnologies Emergents
CYI	The Cyprus Institute
DEEP-EST	Dynamical Exascale Entry Platform - Prototype System
DDP	see PyTorch-DDP
DL	Deep Learning
DNN	Deep Neural Network
ESB	Extreme Scale Booster
FEFS	Fujitsu Exabyte File System
FFTW	Fastest Fourier Transform in the West
FZJ	Forschungszentrum Jülich
GCC	GNU Compiler Collection
GPFS	General Parallel File System
GPU	Graphics Processing Unit
HDF5	Hierarchical Data Format version 5
HeAT	Helmholtz Analytics Toolkit
HPC	High-Performance Computing
IB	InfiniBand
I/O	input/output
JSC	Jülich Supercomputing Centre
JURECA	Jülich Research on Exascale Cluster Architectures
JUWELS	Jülich Wizard for European Leadership Science
LES	Large-Eddy Simulations
m-AIA	multiphysics-AIA
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
NCCL	NVIDIA Collective Communication Library
NIST	National Institute of Standards and Technology

NLP	Natural Language Processing
OpenMP	Open Multiprocessing
Parallel-NetCDF	parallel Network Common Data Form
ParTec	ParTec Cluster Competence Center GmbH, Munich, Germany. Linked third Party of FZJ in CoE RAISE
PyTorch-DDP	PyTorch Distributed Data Parallel
RAISE	see CoE RAISE
RCCL	ROCm Communication Collectives Library
RWTH	RWTH Aachen University
TBL	Turbulent Boundary Layer
TCP	Transmission Control Protocol
TGV	Taylor-Green Vortex
UCX	Unified Communication X
WP	Work Package