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), various high-performance computing prototypes are made available to the developers in the project and beyond. 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 hosts of the prototype systems. This Deliverable is the first in a line of two.

1 Introduction

High-Performance Computing (HPC) prototypes are key to new developments on different implementation levels. From a hardware perspective, they might unite novel and experimental components, for which the efficiency of their interplay may be unknown and is waiting to be explored. Middleware, as the connecting element between complex hardware and application software, is able to hide the complex interplay between such hardware components. New hardware is hence also a driver for new middleware technologies. Obviously, efficient hard- and middleware form the basis for the core component: software to be executed on such systems. In the end, it is indeed the problem that can be solved by such software that justifies the effort put into new hard-, middle-, and software developments. Prototypes, as potential predecessors of next-generation production systems, form a playground for all these developments. Working jointly on a shared testbed prototype allows to directly interact and exchange information with or between the regular users of the system and prototype developers. What is even more important is the direct interaction with the system support and the system providers/vendors in the sense of co-design, i.e., to give feedback on the performance of certain system components with respect to specific applications, and if needed, to request further software or middleware required for specific tasks or projects.

In the European Center of Excellence in Exascale Computing “Research on AI- and Simulation-Based Engineering at Exascale” (CoE RAISE), different prototype systems are available for porting available software and to test and tweak their performance on new architectures. The CoE RAISE partners have the opportunity to use two prototype systems at Forschungszentrum Jülich (FZJ), i.e., the Dynamical Exascale Entry Platform - Prototype System (DEEP-EST)[1] and the HUAWEI-based JUAWEI system[2]. At the Barcelona Supercomputing Center (BSC), the Cluster de Tecnologies Emergents (CTE) machines, i.e., CTE-ARM[3] and CTE-AMD[4] systems are available. These systems are rather small and meant for joint testing and analyzing novel hardware technologies.

To briefly summarize, the DEEP-EST prototype system comprises a Modular Supercomputing Architecture (MSA) system with a Cluster-Booster technology, i.e., it consists of an Intel Cluster and an NVIDIA General-Purpose Graphics Processing Unit (GPGPU) Booster, a Data Analytics Module with NVIDIA GPGPUs and STRATIX Field Programmable Gate Arrays (FPGAs), and some smaller modules with Intel Knight’s Landing (KNL) accelerators, Network Attached Memory (NAM) modules, etc. The JUAWEI system holds HUAWEI Taishan servers with Advanced Reduced Instruction Set Computer (RISC) Machines (ARM) technologies. The CTE-ARM system also consists of ARM-based components, i.e., it contains Fujitsu A64FX Central Processing Units (CPUs). In contrast, the CTE-AMD machine holds the latest AMD EPYC processors. 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].

In Task 2.2 (Hardware prototypes) of Work Package 2 (WP2) “AI- and HPC-Cross Methods at Exascale”, both FZJ and BSC provide support for the developments made in CoE RAISE on the corresponding prototype systems. The present document reports on early support activities performed until month six of the project. In this period, the main supporting work has

[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 D2.5 download https://www.coe-raise.eu/downloads

been dedicated to the Multiphysics-Aerodynamic Institute Aachen (m-AIA) simulation code, developed by RWTH Aachen University (RWTH)[6] and to BSC's simulation code Alya[7]. These two codes have been selected for detailed investigations as they are two of the main frameworks used in WP3 of CoE RAISE (Compute-Driven Use-Cases towards Exascale). They both have demonstrated exceptional scalability and are easily accessible.

For m-AIA, the work performed in Task 2.2 covers porting work to and performance analyzes on all available prototype systems at FZJ, see Sec. 2. Alya has been ported to and analyzed on both FZJ's and BSC's systems, see Sec. 3. Modifications to the codes have directly been committed to dedicated branches or forks of the code repositories. Finally, Sec. 4 additionally presents some first analyses of Artificial Intelligence (AI) / Machine Learning (ML) algorithms on the DEEP-EST system and Sec. 5 provides a summary and an outlook.

[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

2 Support activities for m-AIA from RWTH

The developers of the m-AIA code from RWTH have been supported in compiling, installing, running, and benchmarking their simulation code on the DEEP-EST and JUAWEI systems. In the following, an overview of m-AIA is given in Sec. 2.1. This includes a description of the application as given by Task 3.1 of CoE RAISE (AI for turbulent boundary layers) and is followed by a report on the support activities on the DEEP-EST and JUAWEI systems in Sec. 2.2 and Sec. 2.3.

2.1 Overview of m-AIA

The simulation code m-AIA is a multi-physics framework, based on C++, developed at RWTH and supported by FZJ. It contains several different modules to solve, e.g., compressible and incompressible flow, particle-laden flow, aeroacoustics, and moving boundaries problems. The framework operates on hierarchical Cartesian meshes that are generated with a massively parallel grid generator as part of m-AIA.

The different solvers in m-AIA, i.e., a finite-volume, discontinuous Galerkin, lattice-Boltzmann, level-set, and Lagrange solver yield a weak- and strong-scale performance towards hundreds of thousands of compute-cores [1]. For example, the lattice-Boltzmann solver showed an excellent scaling performance up to 2^{18} cores of the Jülich Blue Gene/Q system JUQUEEN[8] and 2^{13} Intel Xeon Phi cores on the Jülich Research on Exascale Cluster Architectures (JURECA) Booster[9] at FZJ, where the node-based communication was the limiting factor. The finite-volume solver showed to scale up to 91,872 cores on the HAZEL HEN machine[10], the predecessor of the HAWK machine[11] at the High-Performance Computing Center Stuttgart (HLRS). The scalability of m-AIA on HAZEL HEN was also shown for the discontinuous Galerkin solver, which furthermore also showed an excellent scaling performance up to the full JUQUEEN system with 458,752 cores. The grid generator achieved a monotonically increasing speed-up up to 2^{18} cores of the JUQUEEN system and up to the full HERMIT[12] machine with 2^{17} cores, the predecessor of the HAZEL HEN machine. The performance of the m-AIA code has continuously been tweaked for years. Especially, hybrid parallelism with mixed Message Passing Interface (MPI) / Open Multiprocessing (OpenMP) implementations have been integrated and the code has in parts been ported to General Purpose Graphics Processing Units (GPGPUs) and Intel Xeon Phis, ARMv7, and ARMv8 architectures. The framework also includes a dynamic load-balancing mechanism, which reacts to load-imbalances due to dynamic mesh refinement. The mesh and geometry are fully parallelized, parallel input/output (I/O) is used with the Hierarchical Data Format version 5 (`HDF5`)[13] or with the parallel Network Common Data Form (`parallel-NetCDF`)[14] libraries, and an in-situ interface has been integrated to connect to in-situ data processing tools.

[8] JUQUEEN https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/de-installedSystems/JUQUEEN/JUQUEEN_node.html
[9] JURECA Booster https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/ JURECA/JURECA_node.html
[10] Hazel Hen https://www.hlrs.de/systems/cray-xc40-hazel-hen/
[11] Hawk https://www.hlrs.de/systems/hpe-apollo-hawk/
[12] HERMIT https://www.hlrs.de/systems/cray-xe6-hermit/
[13] `HDF5` https://www.hdfgroup.org/solutions/hdf5/
[14] `parallel-NetCDF` https://parallel-netcdf.github.io

For performance and scalability analyzes as well as parallel debugging, multiple tools such as m-AIA-internal counters, LIKWID[15], Scalasca[16], Score-P[17], CUBE[18], CrayPat[19], or TotalView[20] are used.

Within Task 3.1 of CoE RAISE, m-AIA is planned to be used for the prediction of turbulent boundary layer flows, with the aim to analyze the impact of various actuation methods on viscous drag and noise emission. To produce a large enough data set for AI training, a large number of simulations on the order of O(1,000) have to be conducted. These simulations will be performed with a farming approach, where many medium-sized flow problems will be simulated simultaneously on a large number of computing cores. The size of the computational mesh for a single simulation is approximately O(100) million cells. The corresponding simulations will use O(1,000) computing cores. For such problem sizes and core numbers, the parallel scalability of m-AIA has already been demonstrated to be high enough for the efficient use of computing resources.

If the findings obtained by the AI methods with respect to minimum drag or minimum noise emission should be verified for realistic technical applications such as a wing of an aircraft, much larger problem sizes have to be solved. The corresponding cell count for such cases will increase dramatically. The simulations need to be performed on a much larger number of computing cores, i.e., O(100,000) cores, where the minimum number of cores is determined by the available memory per core and the computing speed of the specific core type. This defines the wall clock time. For presently available HPC hardware, such as the HAWK system installed at HLRS Stuttgart, and the flow problem of a turbulent external flow, typically twice the number of cores than it is necessary from the constraint of sufficient memory is used. To reduce the wall clock time to reach convergence of a turbulence scale-resolving simulation result in a statistical sense, a larger number of computing cores must be used. For such simulations a good strong scaling becomes important. Therefore, the major focus will be put on the improvement of the strong scaling performance of m-AIA within CoE RAISE.

2.2 m-AIA on the DEEP-EST system at FZJ

The m-AIA code can be compiled using various compilers, where a list of available compilers is given in Table 1. It should be noted that older compiler versions may work but might require optimizations. The `Intel` compiler and the GNU Compiler Collection (`GCC`) in their more up-to-date versions are preinstalled on the DEEP-EST system as modules.

The inter-process communication is taken care of using the `ParaStationMPI` implementation[21], which is a derivative of the most recent freely available and portable `MPICH`[22] implementation in its version 3.3. The m-AIA code requires a minimum version of 4.0 for the `ParaStationMPI` library, where a more recent version exists as a module on the DEEP-EST system. The available compilers with `ParaStationMPI` implementation support C/C++ and Fortran languages, yielding no issues to compile and run m-AIA on the DEEP-EST system.

[15] LIKWID https://github.com/RRZE-HPC/likwid
[16] Scalasca https://www.scalasca.org
[17] Score-P https://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/docs/scorep-6.0/html/index.html
[18] CUBE https://hbp-hpc-platform.fz-juelich.de/?hbp_software=cube-score-p-scalasca
[19] CrayPat https://docs.nersc.gov/tools/performance/craypat/
[20] TotalView https://totalview.io
[21] `ParastationMPI` https://docs.par-tec.com/html/psmpi-userguide/index.html
[22] `MPICH` https://www.mpich.org

Compiler	Minimum tested version
AMD	-
Clang	9
Cray	-
Fujitsu	-
GCC	9.1
IBM	-
Intel	19.1
PGI	19.2

Table 1: List of compilers and minimum tested versions for m-AIA.

The discrete Fourier transformations required by m-AIA are handled with an external software library, the Fastest Fourier Transform in the West (FFTW)[23] with a minimum version number of 3.3.2. This library with a more recent version exists as a module on the DEEP-EST system. The post-processing libraries required by m-AIA, see Sec. 2.1, are the parallel-NetCDF library (more recent than version 1.9) and the HDF5 library in its parallelized implementation (more recent than version 4.0). Both libraries are available through modules. Instead of using the module system, the aforementioned software libraries can also be compiled, installed, and included in the m-AIA code locally. For this purpose, the home directory /p/home/user of the user can be used. For compilation and installation of external libraries, a significant workload and compile duration are required. Therefore, they are compiled in the work or scratch directories using the slurm job scheduler. This is to avoid overloading the login nodes on the DEEP-EST system, which share only a 1GB/s connection.

Efforts to port m-AIA to the DEEP-EST system are discussed in Sec. 2.2.1. Subsequently, Sec. 2.2.2 presents how to run a simulation case. Results of performance investigations on the DEEP-EST system are presented in Sec. 2.2.3. Using different compilers may result in performance differences. Hence, the performance using the Intel and GCC compilers is also analyzed here. Furthermore, the results are juxtaposed to results obtained on the Jülich Wizard for European Leadership Science (JUWELS)[24] system at FZJ.

2.2.1 Porting support

To ensure that porting-related code changes are integrated into the code base of m-AIA, a branch forked from the main m-AIA development repository, which can only be accessed through an invitation by the RWTH group from the given link[25], has been created. The supporting developments described in the following can be found in the new branch FZJ_RAISE of the GIT repository[25]. The changes to the master-version are kept to a minimum,

[23] FFTW http://www.fftw.org
[24] JUWELS https://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUWELS/JUWELS_news.html
[25] https://git.rwth-aachen.de/aia

thus, merging of the created branch with the master version should be seamless. Henceforth, the following commands can be issued on the login nodes of that machine unless otherwise specified.

The DEEP-EST system consists of different development stages. These stages are accessible via

```
$ module use $OTHERSTAGES
```

The stages are packages of modules that are made available for specific time periods. Currently, the oldest stage is denoted as `2018b`, whereas the most recent stage is `Devel-2020a`. It should be noted that the DEEP-EST system is periodically updated in the scope of this project. To port m-AIA to the DEEP-EST system, the stable stage `2020` is used instead of the latest development stage as the latter is still not yet finalized. This stage is loaded with

```
$ module –force purge
$ module use $OTHERSTAGES
$ module load Stages/2020
```

In this stage, frameworks and libraries are available in their recent versions. For instance, a small list of available modules is visible through

```
$ module avail
  intel-para/2020
  CMake/3.18.0
  EasyBuild/4.3.1
  pscom/5.4-default
  GCC/9.3.0
  Intel/2020.2.254-GCC-9.3.0
```

Compilation of m-AIA with the GNU compiler suite

m-AIA is first tested using `GCC`, which has several different versions in stage `2020`. The latest loadable version of `GCC` is 10.2.0. In this stage, the rest of the dependencies, such as `HDF5` are not available for this version and need to be compiled manually. In contrast, loading `GCC` version 9.3.0 also provides access to most of the required dependencies in their recent versions. A `GCC` with the desired version can be loaded via

```
$ module load GCC/<VERSION_NUMBER>
```

The available compatible version of the `ParaStationMPI` library is 5.4.7, which can be loaded through

```
$ module load ParaStationMPI/5.4.7-1-mt
```

Loading `ParaStationMPI` explicitly loads the required `pscom` for the low-level communication layer. To check which version number of a module is currently loaded, the `module show` command can be issued, as:

```
$ module show pscom

  /usr/local/software/skylake/Stages/2020/UI/Tools/pscom/.5.4- \
  default.lua:
```

FFTW and the two post-processing dependencies parallel-NetCDF and HDF5 can then be loaded using the following command:

```
$ module load FFTW parallel-netcdf HDF5
```

Issuing this command automatically selects versions 3.3.8 for FFTW, 1.12.1 for parallel-NetCDF, and 1.10.6 for HDF5. These versions are relatively up-to-date, hence, no explicit version updates are needed. Local compilation of these dependencies is discussed later in this section.

Finally, the compilation process of m-AIA is automated using the CMake software, which can be loaded via

```
$ module load CMake
```

Loading these modules to compile m-AIA implicitly includes various other modules, e.g., the Zlib library for compressing the output of HDF5 files. If the aforementioned steps are performed, the following modules should be visible:

```
$ module list

  Currently Loaded Modules:
    1) Stages/2020       7) CMake/3.18.0           13) Szip/.2.1.1
    2) GCCcore/.9.3.0    8) numactl/2.0.13         14) mpi-settings/plain
    3) zlib/.1.2.11      9) nvidia-driver/.default 15) ParaStationMPI/5.4.8-1
    4) binutils/.2.34    10) CUDA/11.0             16) HDF5/1.10.6
    5) GCC/9.3.0         11) UCX/1.9.0             17) FFTW/3.3.8
    6) ncurses/.6.2      12) pscom/.5.4-default    18) parallel-netcdf/1.12.1
```

In this list of modules, the version numbers beginning with a '.' indicate that the corresponding module is hidden. Hidden modules are either dependencies for other modules or deprecated older versions of programs and/or libraries. In the present case, the hidden modules are dependencies for other modules. It can be seen that the loaded modules consist of libraries that satisfy the requirements of m-AIA to be compiled on the DEEP-EST system.

In case the HDF5 and/or parallel-NetCDF libraries are required for Input/Output (I/O), the initial step is to pre-compile the iolib package that is included in m-AIA's source code directory. This package (iolib_maia) is compiled in a separate location different from the main m-AIA directory using the CMake script that is included with the following commands:

```
$ cmake -DOPTION1=value1 -DOPTION2=value2 $PATH_TO_IOLIB_MAIA
$ make
```

Possible options are listed in the subsequent Table 2.

Options	Descriptions
AIA	Compile on JUQUEEN (requires to be turned on)
JURECA	Compile on JURECA (requires to be turned on)
CLAIX	Compile on CLAIX (requires to be turned on)
HAWK	Compile on HAWK (requires to be turned on)
PARALLEL	Parallelized I/O (default)
PNETCDF	Use parallel-NetCDF (default)

Table 2: List of options for the compilation of `iolib_maia`.

The script is extended to work on the DEEP-EST system, where no additional modules apart from the aforementioned are required. The script copies the libraries to a local directory and compiles this package there. This script is made available in the FZJ_RAISE repository under $MAIA/scripts, where the command to initiate the script is

```
$ ./install.sh
```

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 detects systems around the world frequently used by the m-AIA community. This automated script is extended to include the DEEP-EST system. Therefore, the $MAIA/cmake/GetHost.cmake file is modified issuing the commands below.

```
$ cd $MAIA/cmake/
$ sed -i.bak \
  '33i\    elseif (${_raw_host} MATCHES "deepv")\' GetHost.cmake
$ sed -i.bak \
  '34i\      set(${_var} "deepv" PARENT_SCOPE)\' GetHost.cmake
```

A pre-modified file is also available in the FZJ_RAISE branch. A new file in the $MAIA/auxiliary/hosts/ directory must be generated using the initials of the DEEP-EST system – in this case deepv.cmake. This file must include the system-specific information for the locations of the dependencies and the default compilation options in case not specified during configuration. 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.

During the compilation, several failed compile attempts were encountered when using GCC with a null-pointer dereference error. A fix has been provided through a DEEP-EST system-specific modification of the compile options in the file $MAIA/auxiliary/compilers/GNU.cmake, which is also available in the FZJ_RAISE branch.

The configuration script includes several options that can be specified. Possible options can be viewed using the following command.

```
$ ./configure.py --help
```

It should also be noted that this script requires `Python` with a version of 3.x. Some important options are listed below in Table 3.

Options	Descriptions
`--with-hdf5`	Link against `HDF5` library and enable `HDF5`-related code
`--with-hdf5iolib`	Link against `HDF5` IOLib and enable I/O for the structured m-AIA solver
`--disable-openmp`	Disable OpenMP extensions

Table 3: Important options to the configure.py script of m-AIA.

The configuration script can be initiated using

```
$ ./configure.py ? ?
```

This command is sufficient for a compilation of the code using the default settings as specified in `deepv.cmake`. However, the default options do not include the `HDF5` library. It hence needs to be specified explicitly. Therefore, the default configuration command on the DEEP-EST system using the `GCC` is updated to read

```
$ ./configure.py 1 2 --reset --with-hdf5iolib --with-hdf5
```

This command resets any previous options and compiles m-AIA in its production build type. Different build types for m-AIA such as `debug`, `production`, `extreme`, `prototyping`, `sanitize`, and `coverage` exist. It should also be noted that all possible compile and build options can be accessed by simply calling

```
$ ./configure.py
```

Finally, if the configuration is successful, m-AIA can be compiled using the `make` command. It is recommended to issue `make -j<NUMBER_OF_CORES>` with a wisely chosen processor number to decrease the compilation time. This option should not be used on the login node due to its limited computing power and bandwidth. A fully automated compilation and installation script for m-AIA on the DEEP-EST system using `GCC` is included in the `FZJ_RAISE` branch under `$MAIA/scripts`, where this script can be issued by

```
$ ./run.sh GNU
```

Compilation of m-AIA using the Intel compiler suite

The `Intel` compiler is based on the `GCC` project with additional optimizations for CPUs with Intel architectures. That is, the basis of m-AIA using `GCC` on the DEEP-EST system is similar

to using the Intel compiler. All of the modules required for the Intel compiler can be loaded using the commands below.

```
$ module --force purge
$ module use $OTHERSTAGES
$ module load Stages/2020
$ module load intel-para/2020-mt FFTW HDF5 parallel-netcdf
```

The intel-para module provides Intel compilers, ParaStationMPI, and the Intel Math Kernel Library (imkl). The Intel and ParaStationMPI have version numbers 2020.2.254 and 5.4.7, respectively. This Intel compiler is based on the implementation of GCC with version number 9.3.0.

The deepv.cmake file must be adjusted similarly to the GCC version described above for the Intel compiler. A pre-modified version of this file is available in the FZJ_RAISE branch. The default configuration command on the DEEP-EST system using the Intel compiler is

```
$ ./configure.py 2 2 --reset --with-hdf5iolib --with-hdf5
```

Here, the only difference between the Intel and GCC compiler is the first wrapper – the first option "2" specifies that the compiler is of Intel type. Similarly, upon successful configuration, m-AIA can be compiled using the make -j<NUMBER_OF_CORES> command. A fully automated compilation and installation script for m-AIA suited for the DEEP-EST system and using the Intel compiler is available in the FZJ_RAISE branch under $MAIA/scripts. The script can be issued by

```
$ ./run.sh Intel
```

In most cases, the modules available on the DEEP-EST system are sufficient to compile m-AIA. However, there is also an option to locally compile, install, and include external libraries, which becomes necessary if, e.g., a newer, faster, or bug-free version of a dependency is released but not yet included as a module.

An example is the recently released HDF5 library with version number 1.12. For instance, this library can be compiled and installed in the directory $HOME/local/hdf5 with the following commands.

```
$ wget https://support.hdfgroup.org/ftp/HDF5/releases/hdf5-1.12/hdf5 \
  -1.12.0/src/hdf5-1.12.0.tar.bz2
$ tar xfvj hdf5-1.12.0.tar.bz2
$ cd hdf5-1.12.0
$ ./configure --prefix=$HOME/local/hdf5 --enable-parallel
$ make -j <NUMBER_OF_CORES>
$ make install
```

These series of commands create four folders bin, include, lib, and share in the directory $HOME/local/hdf5. Note that the number of cores needs to be provided to the make command. The include and lib folders must be linked to the CMake file of the iolib_maia

dependency and the configuration file of m-AIA. The `CMake` file of the `iolib_maia` dependency is extended by including the following lines.

```
set(INCLUDE_DIRS ${INCLUDE_DIRS} ~/local/hdf5/include )
set(LIBRARY_DIRS ${LIBRARY_DIRS} ~/local/hdf5/lib )
```

Example `CMake` files are available in the `FZJ_RAISE` branch in the directory file `$MAIA/scripts/iolib_maia`. The `deepv.cmake` file of the m-AIA code for automated configuration requires similar modifications, where an example can be found in the `$MAIA/auxiliary/hosts/Host.cmake.in` directory.

Finally, a complete script that compiles, installs, and links different dependencies such as `FFTW`, `parallel-NetCDF`, `HDF5`, and `iolib_maia` on the DEEP-EST system can be found in the `$MAIA/scripts/installApps_deepv.sh` directory of the `FZJ_RAISE` branch.

2.2.2 Execution of m-AIA

Each test case that uses m-AIA consists of a properties file, usually denoted as `properties_run.toml`. This file includes the specific computation parameters such as the simulation run time or the Reynolds number, etc. A simulation can be initiated with the following line.

```
./$PATH_TO_MAIA properties_run.toml
```

On the DEEP-EST system, the initial step is to enable support to Unified Communication X (`UCX`)[26] in InfiniBand networks (see Deliverable 2.5 for more details) by issuing the `PSP_UCP=1` command. The execution of m-AIA can be initiated using the `srun` command from the `slurm` workload manager by

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

This command initiates the execution of m-AIA in real-time using `2` nodes and `8` tasks over `5` minutes of wall-time. The `-p` option specifies the partition to be used for the submitted job, in this case, the Cluster partition (`dp-cn`). Alternatively, a batch script can be used to submit m-AIA for later execution using the `sbatch` command. An exemplary batch script is given below.

```
# General configuration of the job
#SBATCH --job-name=<name>
#SBATCH --account=<account_name>
#SBATCH --output=job.out
#SBATCH --error=job.err
#SBATCH --time=00:05:00

...
```

[26] `UCX` https://www.openucx.org

```
...

# Configure node and process count
#SBATCH --partition=dp_cn
#SBATCH --nodes=1
#SBATCH --ntasks=24
#SBATCH --ntasks-per-node=24
#SBATCH --cpus-per-task=1

# Execute
export HDF5_USE_FILE_LOCKING="FALSE"
PSP_UCP=1 srun ./$PATH_TO_MAIA properties_run.toml
```

This script can be submitted to `slurm` by

```
sbatch $NAME_OF_BATCH_SCRIPT
```

In the current development stage of the DEEP-EST system, the `HDF5` dependency has a software issue, where m-AIA cannot read or write `HDF5` files. This issue has been acknowledged by the main developers of the `HDF5` software, where a current fix is to include the following line of code before the execution of m-AIA.

```
export HDF5_USE_FILE_LOCKING="FALSE"
```

This explicitly included command does not affect the performance of m-AIA on the DEEP-EST system. A more detailed batch script with various options to run m-AIA on the DEEP-EST system can be found in the `$MAIA/scripts/startScript_deepv` directory of the `FZJ_RAISE` branch. For each finalized simulation, m-AIA generates a log file `maia_log` that includes computational details and profiled timers of that specific run. Some important parameters included in the `maia_log` file are given in Table 4.

Options	Descriptions
`Total memory`	Allocated memory
`Total wall-time`	Total run-time
`Total CPU time`	Total computational time
`Init`	Loading previously generated files, e.g., restart files
`Constructor`	Reading the computational domain and computation of prerequisites
`Save output`	Writing the final post-processing step
`Exchange`	MPI communications

Table 4: General information to the log file generated by the m-AIA code.

2.2.3 First base performance analyses

Three benchmark cases are chosen to test the performance of m-AIA on the DEEP-EST system. These cases are three-dimensional simulations of (i) a Taylor-Green Vortex (TGV) with 17 million computational elements and two Turbulent Boundary Layers (TBLs) with (ii) 2.9 million and (iii) 27 million computational elements. These simulations employ the finite-volume solver of m-AIA and use a structured computational grid. Hereinafter, the TBL with 2.9 million, and the TBL with 27 million computational elements are denoted as TBL-small, and TBL-large, respectively.

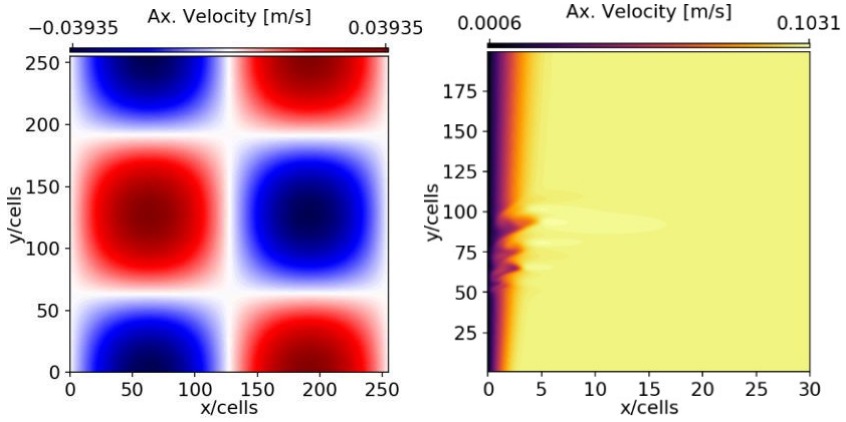

Figure 1: Contour plots of the axial velocity on middle planes of the TGV (left) and the TBL-small (right) simulation results.

Figure 1 shows the contour plots of the axial velocity on the middle plane from the TGV and TBL-small simulation results using m-AIA compiled with GCC. It can be seen that these simulations are able to produce results on the DEEP-EST system as intended. It should be noted that only a limited number of iterations are performed and that hence no (statistically) converged solutions are obtained. Furthermore, the results are not validated since the agreement of the simulation results to the evidence is not in the scope of this report.

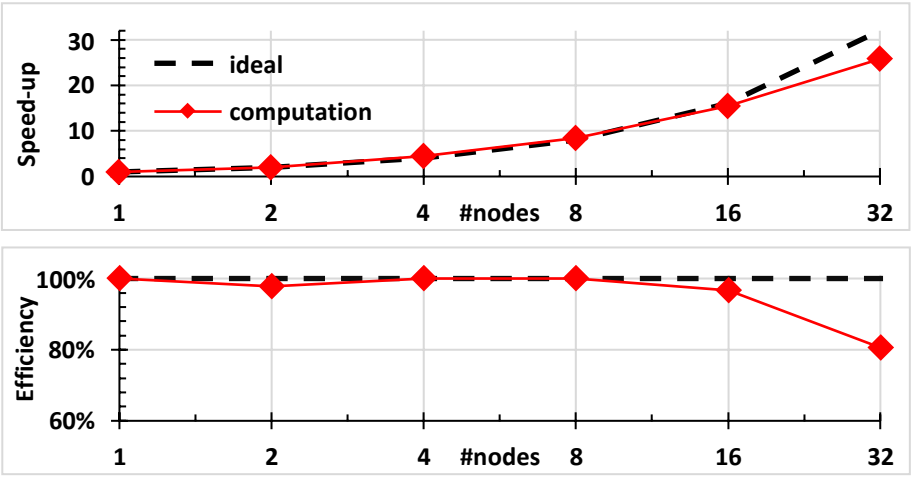

Figure 2: Strong scaling of m-AIA compiled with GCC on the DEEP-EST system for the TGV simulation (red line) compiled with GCC. The ideal scaling and efficiency are represented by the black dashed line. The achieved speed-up (top) and the efficiency (bottom) are shown. The output routines are disabled.

Figure 2 shows the strong scaling results of m-AIA compiled with GCC on the DEEP-EST system for the TGV simulation. The speed-up is computed as the inverse of the normalized computational time per iterative step of m-AIA, i.e., it measures the time for a complete Runge-Kutta loop to integrate the time derivative of the employed transport equation. More information

on the numerical treatment of this time derivative of the transport equations can be found in classical textbooks on Computational Fluid Dynamics (CFD), e.g., in [2]. The efficiency is estimated as the speed-up over the used nodes per simulation. The compute time using the minimum number of nodes is used for normalization, i.e., the results obtained on a single node. All of the post-processing routines are disabled. On the DEEP-EST system, there are 24 cores per node available (see Deliverable D2.5). 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 consists of a total of 50 nodes. It is visible in Figure 2 that the TGV simulation scales well on the DEEP-EST. However, the speed-up of the code is still not ideal when 32 nodes are used. It is expected that a better scaling performance can be achieved for cases with much more computational elements – this bold argument is discussed later, based on the results shown in Figure 3. Good efficiency is achieved up to 16 nodes, whereas the efficiency is still over 80% for 32 nodes.

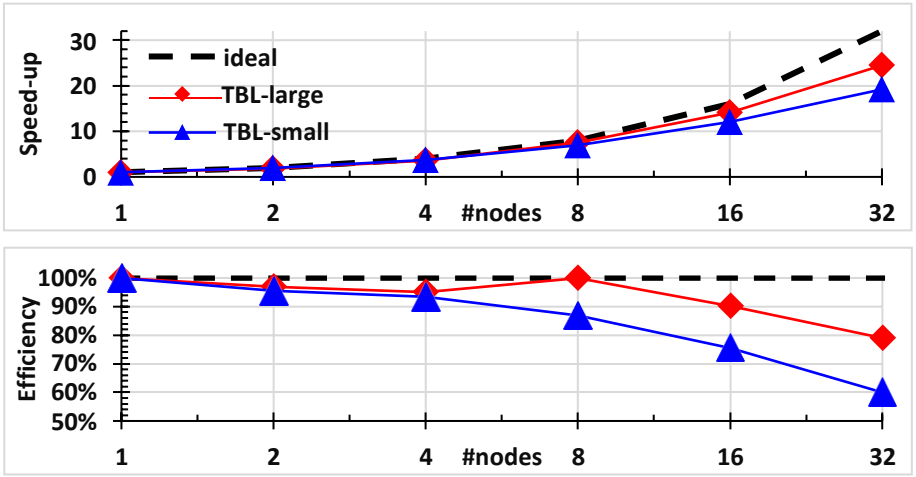

Figure 3: Strong scaling of m-AIA compiled with GCC on the DEEP-EST system for the TBL-large (red line) and TBL-small (blue line) simulations. The ideal scaling and efficiency are represented by the black dashed line. The achieved speed-up (top) and the efficiency (bottom) are shown. The output routines are disabled.

Figure 3 shows the results of a strong scaling test compiled with GCC for the TBL-large and TBL-small cases. It can be observed that the TBL-small simulation scales well up to four nodes. The speed-up performance drops when 8 or more nodes are employed. This is due to the small problem size of this simulation with 2.9 million computational elements. Consecutively, increasing the number of nodes yields too small number of computational elements per MPI rank. Having such a small computational mesh per rank inevitably limits the CPU computation, whereas the parallelized simulation is restrained to the MPI communication capabilities. It is also worth mentioning that the efficiency of m-AIA for the TBL-small simulation drops to as low as 60% when 32 nodes are used.

In contrast to the TBL-small cases, the TBL-large mesh consists of a factor of nine more computational elements. As expected, the TBL-large simulation shows much better scalability performance than the smaller TBL-small case. This is primarily due to the larger problem size with more computational elements per MPI rank. Compared to the efficiency values of the TBL-small simulation, the TBL-large case achieves an efficiency of over 90% on up to 16 nodes. Consistent with the performance figures of the TGV simulations presented in Figure 2, the efficiency drops to 79% when 32 nodes are employed. A strong presumption can be made that

when even more computational elements are used, an increase of the speed-up and the efficiency values is possible.

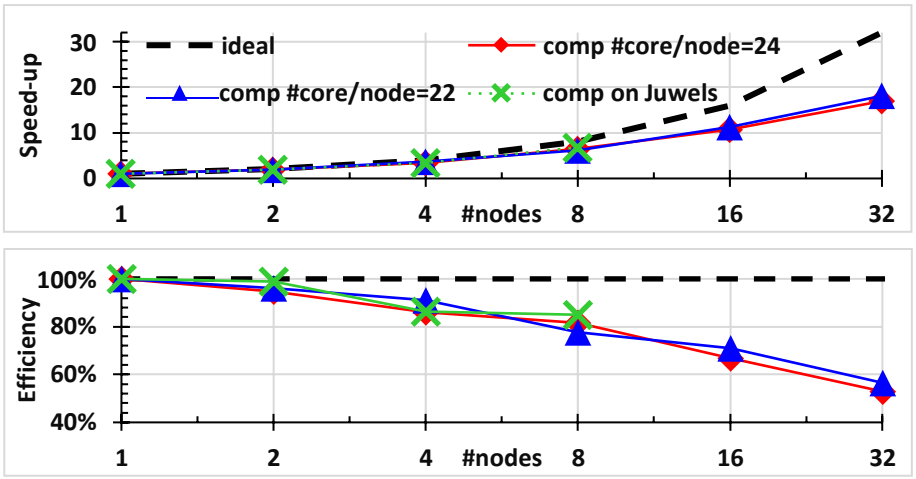

Figure 4: Strong scaling of m-AIA compiled with `GCC` on the DEEP-EST system for the TBL-small computation using 24 cores per node (red line), using 22 cores per node (blue line), and on the JUWELS system (green line). The ideal scaling and efficiency are represented by the black dashed line. The achieved speed-up (top) and the efficiency (bottom) are shown. The output routines are disabled.

Figure 4 shows the results of the strong scaling tests of running m-AIA compiled with `GCC` for three different TBL-small computations, i.e., using 24 cores per node, 22 cores per node, and, for comparison, on the JUWELS system (using 24 cores per node). An attempt is made to improve the scaling performance of the TBL-small simulation by allocating two free cores to handle the MPI communications. It is evident from Figure 4 that both speed-up and efficiency values improve when 22 cores per node are employed. This is especially true when 16 or more nodes are used. This proves that the TBL-small simulation with too many nodes is restrained to the MPI communication. Hence, the aforementioned strong presumption made for the TBL-large case still holds.

The same scaling performance analysis as for the TBL-small simulation is repeated on the JUWELS system using the Cluster partition and the development queue `devel`, where the results are shown in Figure 4, noting that the JUWELS system is only presented as a reference. The analysis includes up to eight nodes due to the limitations of JUWELS' test queue. Further simulations with more nodes are planned. The scaling performance of m-AIA on the JUWELS system is very similar to the performance on the DEEP-EST system for the TBL-small case.

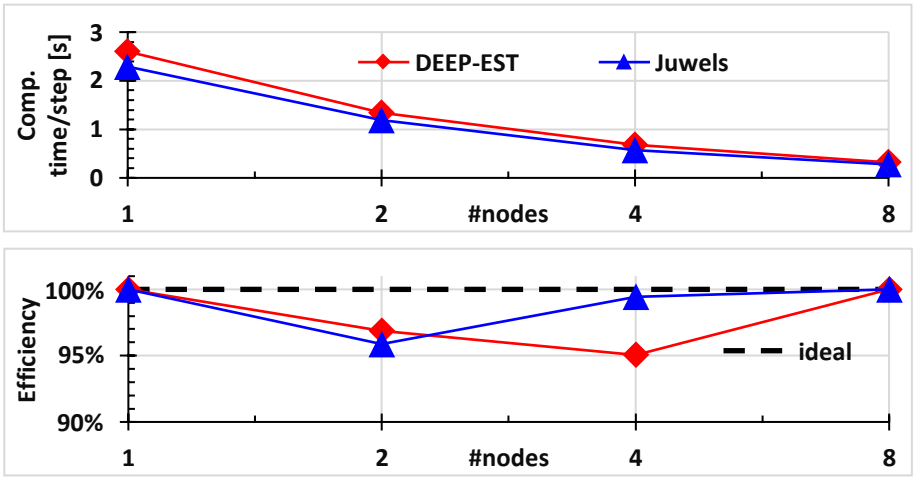

Figure 5: Performance results of the TBL-large case using m-AIA compiled with `GCC` on the DEEP-EST and JUWELS systems. The ideal scaling and efficiency are represented by the black dashed line. The computation time of a single computational loop (top) and the efficiency (bottom) are shown. The output routines are disabled. Note the y-scale.

Figure 5 compares two TBL-large simulations run on the DEEP-EST and JUWELS systems using m-AIA compiled with `GCC`. As mentioned earlier, the strong scaling is performed on up to eight nodes. The computational time of a single computational loop, i.e., the time for a complete full Runge-Kutta loop, and the efficiencies are juxtaposed. Quantitatively, m-AIA on the JUWELS system is around 10-14% faster than on the DEEP-EST system, which can be explained by the different CPU architectures. That is, the DEEP-EST system features Intel Xeon Skylake Gold 6146 CPUs while JUWELS is equipped with Intel Xeon Platinum 8168.

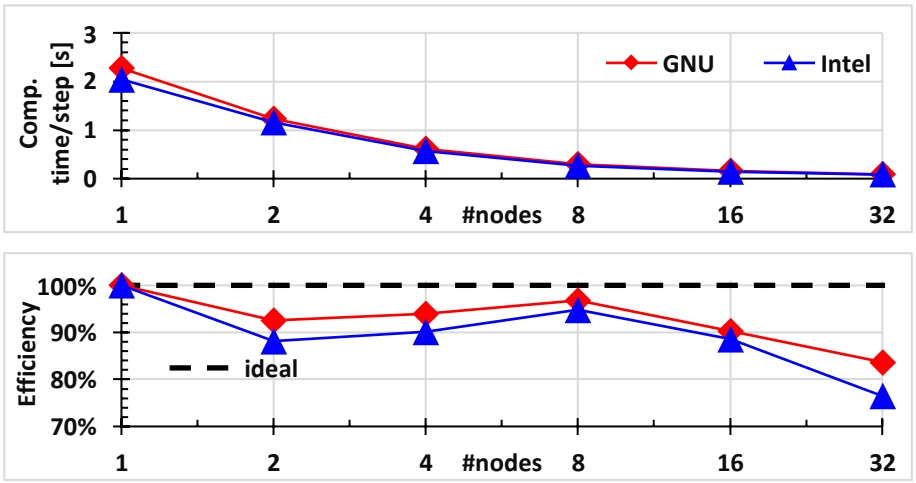

Figure 6: Performance results of the TBL-large case using m-AIA compiled with `GCC` and `Intel` compiler on the DEEP-EST systems. The ideal scaling and efficiency are represented by the black dashed line. The computation time of a single computational loop (top) and the efficiency (bottom) are shown. The output routines are disabled. Note the y-scale.

Figure 6 compares the performance of the TBL-large case using m-AIA compiled with `GCC` and the `Intel` compilers (see Sec. 2.2.1 for more information). As mentioned earlier, the CPUs of the DEEP-EST system are based on Intel's architecture. Thus, it is expected that using the `Intel` compiler increases the performance of any program on this system. Up to now, the basis of the benchmarks relies on the `GCC` libraries since many prototype systems provided in the scope of this project employ both Intel and non-Intel-based CPUs. However, it is evident from Figure 6 that employing the `Intel` compiler for m-AIA on the DEEP-EST system provides

an up to 11.26% performance increase. Quantitatively, the TBL-large simulation using 32 nodes is only 1.67% faster when the `Intel` compiler is used. An increase in the number of nodes monotonically decreases the performance difference between the `GCC` or `Intel` compiled versions. Thus, m-AIA compiled with `GCC` achieves despite its lower node-based performance slightly better efficiencies than its `Intel`-compiled counterpart. An explanation for this could be that when more nodes are employed, the scaling performance is limited to the MPI communications (as in the TBL-small case in Figure 4). Hence, the performance difference between the compilers becomes irrelevant. On the other hand, it can be seen from Figure 6 that the efficiency of m-AIA using `GCC` achieves slightly higher values, especially when 32 nodes are employed.

2.3 m-AIA on the JUAWEI system at FZJ

The m-AIA code can be compiled using `GCC` for both `ARM` and `x86`-based CPUs. `ARM`-based CPUs are tested for the first time for m-AIA. To compile m-AIA for `ARM`-based CPUs, the `ARM` developer node of the JUAWEI system (`juawei-a28`) must be used. In contrast, the `Intel` developer node (`juawei-x12`) needs to be used to compile and run m-AIA on `x86`-based CPUs. More details on the node configuration of the JUAWEI system can be found in Deliverable D2.5 of CoE RAISE. To compile for both `ARM` and `x86`-based CPUs, a `GCC` compiler in an up-to-date version is preinstalled on the JUAWEI system as a module.

Efforts to port m-AIA to the JUAWEI system are discussed in Sec. 2.3.1. Running a case on the JUAWEI system is explained in Sec. 2.3.2. Results of performance analyses are presented in Sec. 2.3.3, where this section also compares `ARM` and `x86`-based CPUs on the JUAWEI system. The following discussion is based on Sec. 2.2, hence, only major differences are presented.

2.3.1 Porting support

The JUAWEI system uses `EasyBuild` to organize its software stack (see Deliverable 2.5). The software stack is not enabled by default. To switch to the new software stack, the file `/opt/ohpc/pub/easybuild/switch_to_eb_sw_stack.sh` needs to be sourced via

```
$. /opt/ohpc/pub/easybuild/switch_to_eb_sw_stack.sh
```

Similar to the DEEP-EST system described in Sec 2.1, The JUAWEI system consists of different development stages. These stages are interchangeable via the following command.

```
$ . /opt/ohpc/pub/easybuild/switch_stage.sh -s <STAGE_NAME>
```

Currently, there are three stages available: `2019b`, `2020a`, and `2021a`. The m-AIA code can only be compiled using the earliest `2019b` stage. In this development stage, framework and library versions satisfy the requirements to compile m-AIA. A shortlist of available modules is visible through

```
$ module avail

  CMake/3.16.1
  EasyBuild/4.1.1
  OpenMPI/4.0.1
  GCC/9.2.0
```

The m-AIA code can be compiled using the GNU compiler suite with its `GCC` compiler at version 9.2.0. Loading the corresponding module provides the user with a set of libraries, which fulfills most of m-AIA's dependencies. The inter-process communication is taken care of with the `OpenMPI` implementation[27]. External dependencies to compile m-AIA such as `CMake`, `FFTW`, and `HDF5` can be loaded directly, where their version numbers are 3.16.1, 3.3.8, and 1.10.5, respectively. All of the required libraries can be loaded by issuing the following command.

```
$ module load GCC/9.2.0 OpenMPI CMake FFTW HDF5
```

Loading these modules automatically includes various modules, e.g., the `Zlib` library for compressing the output of `HDF5` files, see Sec. 2.1. However, the JUAWEI system does not include the `parallel-NetCDF` library, which needs to be locally compiled, installed, and included with the following commands (`<NUMBER OF CORES>` specifies the number of cores to execute the `make` command with).

```
$ wget https://parallel-netcdf.github.io/Release/pnetcdf-1.12.2.tar.gz
$ tar xfvj pnetcdf-1.12.2.tar.gz
$ cd pnetcdf-1.12.2
$ ./configure --prefix=$HOME/local/hdf5 --disable-fortran
$ make -j <NUMBER_OF_CORES>
$ make install
```

These and further steps to include locally compiled `parallel-NetCDF` and `iolib` libraries are the same as for the DEEP-EST system, see Sec. 2.1. A detailed explanation is therefore omitted for brevity. However, it should be noted that compiling `parallel-NetCDF` on `ARM` CPU modules results in compile errors when stage `2019b` is selected. This issue can be circumvented by using stage `2020a` to compile `parallel-NetCDF` for `ARM` CPU modules (using `GCC/9.3.0` library). A complete script that compiles, installs, and links different dependencies such as `FFTW`, `parallel-NetCDF`, `HDF5`, and `iolib_maia` explicitly for the JUAWEI system can be found in `$MAIA/scripts/installApps_juawei.sh` of the `FZJ_RAISE` branch.

2.3.2 Execution of m-AIA

On the JUAWEI system, m-AIA can be initiated using the `srun` command from the `slurm` workload manager as

```
$ srun --mpi=pmix_v3 -p x86-normal -N 2 -n 8 -t 00:05:00 \
  ./$PATH_TO_MAIA properties_run.toml
```

[27] `OpenMPI` https://www.open-mpi.org

This command initiates the execution of m-AIA in real-time using `2` nodes and `8` tasks over `5` minutes of wall-time. The `-p` option denotes the partition for the submitted job, in this case, the CPU partition with x86 architecture (`x86-normal`). As mentioned earlier, the CPU partition with ARM architecture (`arm-hi1616`) will be tested once the system is ready. Alternatively, a batch script can be used to submit m-AIA for later execution using the `sbatch` command, similar to the one provided in the `$MAIA/scripts/startScript _juawei` directory of the `FZJ_RAISE` branch. Further details on executing m-AIA on the JUAWEI system are the same as executing it on the DEEP-EST system. The reader is hence referred to Sec. 2.2.2 for more information. For each finalized simulation, m-AIA generates a log file `maia_log`, cf. Sec. 2.2.2.

2.3.3 First base performance analyses

The TGV case with 17 million computational elements is chosen to analyze the scaling performance of m-AIA on the JUAWEI system. Initially, the m-AIA code using `x86`-based CPUs and `GCC` on the JUAWEI system is compared against its counterpart on the DEEP-EST system.

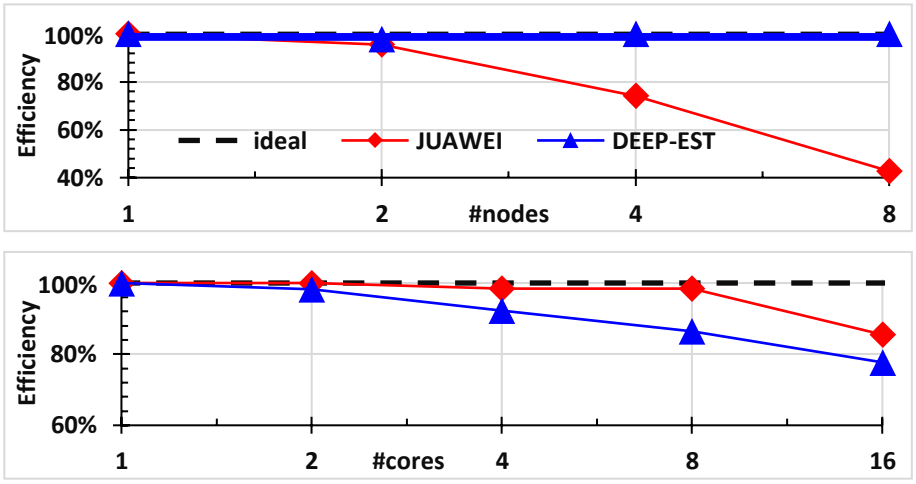

Figure 7: Performance results of the TGV case using m-AIA compiled with `GCC` and run on `x86`-based CPUs of the JUAWEI system (red line). The ideal efficiency is given by the dashed black line. The efficiency of using multiple nodes (top) and a single node (bottom) are shown. For reference, the results on the DEEP-EST system are also shown (blue line). The output routines are disabled.

Figure 7 shows the scaling results of the TGV case using m-AIA compiled with `GCC` on `x86`-based JUAWEI CPUs employing multiple nodes and a single node. The same study is performed on the DEEP-EST system for reference. The efficiency of m-AIA on the JUAWEI system achieves moderate values when four (or more) nodes are employed. It should be reminded that the JUAWEI system is a prototype and hence node-based communication may suffer from performance issues. Therefore, Figure 7 also presents results of small-scaling tests on up to 16 cores on a single node. Obviously, m-AIA scales well on a single node, where the efficiency values are above 85% even when up to 16 cores are used. In fact, running m-AIA on a single node of the JUAWEI system achieves better efficiencies than on the DEEP-EST system.

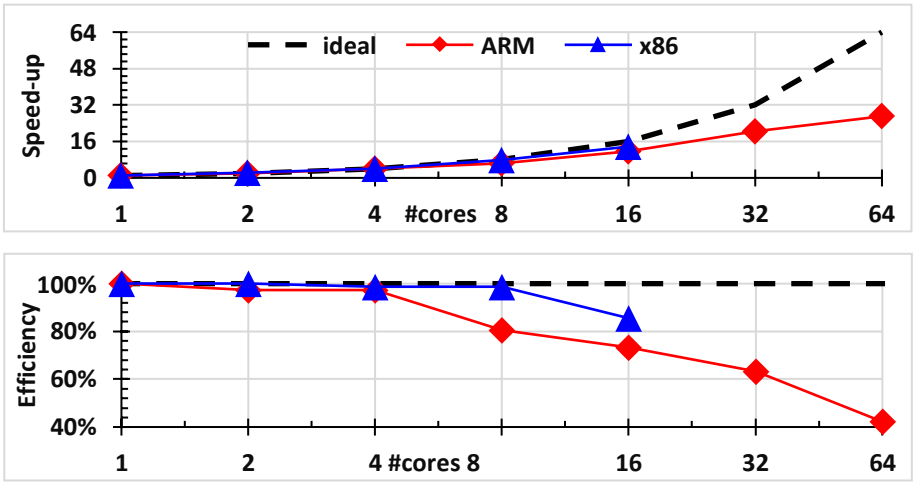

Figure 8: Scaling results of the TGV case using m-AIA compiled with `GCC` and run on either a single `ARM` (red line) or `x86`-based (blue line) node of the JUAWEI system. The ideal efficiency is given by the dashed black line. The achieved speed-up (top) and the efficiency (bottom) are shown. The output routines are disabled.

Figure 8 shows the results of strong scaling experiments on a single node of m-AIA on the JUAWEI system. m-AIA is compiled using `GCC` and is run on either `ARM` or `x86`-based CPUs. It should be noted that the `ARM` and `x86`-based CPUs each consists of 64 and 20 cores, respectively. It is visible that both `ARM` and `x86`-based CPUs perform similarly up to four cores. However, m-AIA run on `ARM`-based CPUs suffers from low-efficiency values when more than 8 cores are employed. It is observed that the efficiency of m-AIA run on `ARM`-based CPUs drops to as low as 40% when all 64 cores are used. It is also witnessed that the performance of m-AIA executed on `ARM`-based CPUs unexpectedly fluctuates. In tests, the run times on an `ARM`-based CPU using four cores varied between 1,721 and 2,802 seconds (with a standard deviation of 514 seconds). A similar test using 16 cores achieved a run time between 571 and 819 with a standard deviation of 102 seconds. Noting again that the JUAWEI system is a prototype, further investigations will be performed regarding the fluctuating `ARM`-based CPU performance.

#Nodes / CPU	1	2	4	8
`x86`	67.34 /s	35.24 /s	22.7 /s	19.7 /s
`ARM`	162.08 /s	90.45 /s	47.44 /s	*87.04 /s*

Table 5: Comparison of m-AIA runtime for a single timestep on `ARM`and `x86`-based CPUs of JUAWEI system per node. The runtimes are multiplied by the number of cores on that node.

The run times of m-AIA for the TGV case are quantitatively compared in Table 5 to assess the performance difference between `ARM` and `x86`-based CPUs on the JUAWEI system. In both scenarios, `GCC` is used to compile m-AIA. The run times are multiplied by the number of cores per node on each system. This way, the results in Table 5 are independent of the different number of cores of each `ARM`– (64 cores) and `x86`-based (20 cores) CPUs. It is observed that the `x86`-based CPUs are up to 2.5 times faster than the `ARM`-based CPUs. Using eight nodes with `ARM`-based CPUs, communication bottlenecks, which yielded exceptionally long run times, were encountered. Hence, this result should be approached with caution.

3 Support activities for Alya from BSC

Within this task, the Alya code, developed by BSC, has been ported and tested on various prototype systems available to the developers within CoE RAISE. In the following, Sec. 3.1 provides an overview of Alya and explains how it is used in CoE RAISE. This is followed by a description of the activities that have been performed on the DEEP-EST and JUAWEI systems at FZJ in Sec. 3.2 and Sec. 3.3. Furthermore, details of Alya on the CTE-ARM and CTE-AMD machines at BSC are provided in Sec. 3.4 and Sec. 3.5.

3.1 Overview of Alya

Alya [3] is a simulation code based on Fortran and C languages and is developed by BSC. Alya solves coupled multi-physics problems using HPC techniques for distributed and shared memory supercomputers, together with vectorization and optimization at the node level. Strong scalability has been established for years, and recent efforts have mainly been devoted to node-level performance and parallel efficiency. In that sense: (i) A co-execution model has been developed to fully exploit heterogeneous resources and therefore enhance resource usage. (ii) An intra-node dynamic load balance strategy was implemented to correct load imbalances using the Dynamic Load Balancing (DLB) library[28] developed at BSC. (iii) At the inter-node level, a runtime redistribution mechanism based on real timings was implemented as well as a partition-independent I/O strategy. (iv) To further enhance efficiency when solving multi-physics problems, an oversubscription strategy has been developed to avoid idle cores. (v) A continuous monitoring of the code enables to obtain the exact parallel efficiency, as a combination of communication efficiency and load balance, through the use of the TALP library developed at BSC. In order to monitor the progress, BSC has developed a performance suite, run whenever a new version of the code is available (several times a week). In addition, this suite allows the development team to detect failures in the performance. It is fully integrated into the Continuous Integration / Continuous Delivery (CI/CD) approach the team follows. Therefore, any advances in the code can be compared to previous versions and quantified. As far as weak scalability is concerned, external weak-scalable solvers (multigrid, domain decomposition) were integrated into the code, mainly in the course of the Energy Oriented Center of Excellence-II (EoCoE-II)[29] project.

Scalability has been demonstrated by the different Unified European Applications Benchmark Suite (UEABS) reports up to 32k cores (although strong scalability was established on Blue Waters[30] in 2014 up to 100k cores for production multi-physics runs). The speed-up obtained is usually over 80% and of course, depends on the load per core. But in such tests, the speed-up is normalized using the lowest core count possible on the machine, which is sometimes quite high. To know the real parallel efficiency, BSC integrated the TALP library into Alya. One of our objectives is to include this library for the testing of the UEABS, to get the right parallel efficiencies (which are in general very different from the one stated by the classical normalization mentioned before). The code has been tested using a hybrid MPI/OpenMP approach in combination with the DLB library. A co-execution model enables the code to take advantage of both CPU and GPGPU heterogeneous architectures.

CoE RAISE aims at introducing AI technologies into the code while conserving the scalability enhancements made in Task 2.1 (Modular and heterogeneous supercomputing architectures)

[28] DLB https://pm.bsc.es/dlb
[29] EoCoE-II https://www.eocoe.eu
[30] Blue Waters http://www.ncsa.illinois.edu/enabling/bluewaters

and Task 3.2 (AI for wind farm layout optimization) of CoE RAISE. In the course of the project, the focus is therefore on the implementation of efficient AI surrogates as well as their integration into the simulation workflow. Furthermore, the AI training phase will require specific tools to be implemented in the code which need to satisfy the Exascale requirements.

3.2 Alya on the DEEP-EST system at FZJ

Alya requires either a `GCC` or `Intel` compiler and a related MPI communication library such as `ParaStationMPI` or `OpenMPI`. Both compilers are preinstalled on the DEEP-EST system, where solely the `GCC` compiler is tested.

The subsequent discussion is similar to that of Sec. 2.2. Therefore, only major code-specific differences are explained in more detail. Requirements to port Alya to the DEEP-EST system are discussed in Sec. 3.2.1. Section 3.2.2 explains how to run a case using Alya on the DEEP-EST system. Finally, the results of the first performance analyses are presented in Sec. 3.2.3.

3.2.1 Porting support

The Alya source code can be accessed from a repository provided by BSC[31], which is available through invitation. The support developments described in the following can be found in the new branch `FZJ_RAISE`. The changes to the master-version are kept to a minimum, thus, merging of the created branch to the master version will be seamless.

As already described in Sec. 2.2.1, stage `2020` is loaded by

```
$ module –force purge
$ module use $OTHERSTAGES
$ module load Stages/2020
```

As a next step, the GNU compiler suite with `GCC`, `ParaStationMPI`, `pscom`, and `CMake` modules have to be loaded by issuing the command below.

```
$ module load GCC ParaStationMPI pscom CMake
```

Alya depends on the graph partitioning library `metis`[32] if run in parallel. This library is used to solve the complex geometrical problem of partitioning the mesh under a reduction of the contact area between sections and balancing the number of computational elements. This library is included in the Alya repository. An installation of this library is performed by issuing the commands

```
$ cd $ALYA/Executables/unix
$ cp configure.in/<selfConfig.in> config.in
$ ./configure -x -parall
```

where `selfConfig.in` is the file for the chosen compiler and CPU architecture. The option `-x` indicates that Alya is compiled in its production stage. An detailed overview of the configuration of Alya can be found on Alya's Wiki site[33].

[31] https://gitlab.com/bsc-alya/projects/alya-raise
[32] `metis` http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
[33] https://gitlab.com/bsc-alya/alya/-/wikis/User-Documentation/Configure.

The `make metis4` and `make -j <NUMBER_OF_CPUs>` commands are issued to compile the `metis` library and Alya, respectively. Alternatively, the `CMake` software can be used to compile Alya as described below. Preferably in a separate directory, Alya is installed in the directory `$ALYA/install` with the command below.

```
$ cmake -DCMAKE_INSTALL_PREFIX=$ALYA/install $ALYA
```

Alya uses the newest `Fortran` language standard of 2008. As the DEEP-EST system has issues using this Fortran standard, a minor fix due to a syntactical issue in a header file was necessary. Alya must be then compiled with the modified header file by

```
$ cmake -DCMAKE_INSTALL_PREFIX=$ALYA/install -DSHARED='ON' \
  -DCMAKE_Fortran_COMPILER=$HOME/local/psmpi/bin/mpif90 $ALYA
```

Alternatively, the `Fortran` language standard 2008 can be disabled by issuing the option below, which then compiles Alya on the DEEP-EST system using `Fortran 95`.

```
$ cmake -DCMAKE_INSTALL_PREFIX=$ALYA/install -DWITH_STD2008='OFF' $ALYA
```

The performance difference using different `Fortran` language standards is yet to be investigated. If the `Fortran` language standard 2008 must be considered on the DEEP-EST system, the `$ALYA/scripts/installApps.sh` script in the `FZJ_RAISE` branch takes care of the aforementioned changes automatically. Furthermore, some important options to compile Alya are listed in Table 6.

Options	Descriptions	Possibilities	Default
CMAKE_BUILD_TYPE	Build options	Release/Debug	Release
CMAKE_INSTALL_PREFIX	Install directory	Path, /usr/local	/usr/local
INTEGER_SIZE	Integer size	4/8	4
VECTOR_SIZE	Vectorization size	Int/16	16
WITH_MPI	Enable/disable MPI	ON/OFF	ON
WITH_OPENMP	Enable/disable OpenMP	ON/OFF	OFF
OPTIMIZATION_LEVEL	Optimization level	0-3	3
SHARED	Use shared libraries	ON/OFF	OFF
WITH_GPU	Use GPUs	ON/OFF	OFF
WITH_STD2008	Fortran language standard 2008	ON/OFF	ON
CMAKE_Fortran_COMPILER	Fortran compiler path	Path	/usr/

Table 6: Compile options for the Alya code.

A script is made available that compiles and installs Alya on the DEEP-EST system to the `$ALYA/install` directory. This script `run.sh` is located in the `$ALYA/scripts` directory of the `FZJ_RAISE` branch.

3.2.2 Execution of Alya

To run a test case with Alya, the case directory has to include the following files

```
$ ls -1 $PATH_TO_CASE

casename.dat            //case global parameters
casename.dom.dat        //domain input data
casename.geo.dat        //mesh definition
casename.set.dat        //set data: groups for post-processing, optional
casename.fix.dat        //boundary conditions data
casename.ker.dat        //kernel parameters (common to all modules)
casename.'module'.dat   //physics modules: 'nsi' (NASTIN), 'tur' (TURBUL)
casename.post.alya.dat  //post process parameters used by alya2pos app
```

Launching Alya on the DEEP-EST system is similar to launching m-AIA, cf. Sec. 2.2.2. The `srun` command from the `slurm` workload manager can be used to run Alya:

```
$ srun $PATH_TO_ALYA_BIN/alya $PATH_TO_CASE/CASE_NAME
```

Resources for later execution of Alya on the DEEP-EST system can be allocated via batch scripts. An example batch script for running Alya on `3` nodes and `72` cores over `20` minutes on the `dp-cn` partition is given below (similar to Sec. 2.2.2).

```
# General configuration of the job
#SBATCH --job-name=<name>
#SBATCH --account=<account_name>
#SBATCH --output=job.out
#SBATCH --error=job.err
#SBATCH --time=00:20:00

# Configure node and process count
#SBATCH --partition=dp_cn
#SBATCH --nodes=3
#SBATCH --ntasks=72
#SBATCH --ntasks-per-node=24
#SBATCH --cpus-per-task=1

# Execute
srun $PATH_TO_ALYA_BIN/alya $PATH_TO_CASE/CASE_NAME
```

This script can be submitted to `slurm` by

```
sbatch $NAME_OF_BATCH_SCRIPT
```

For each finalized simulation, Alya generates several log files as listed in Table 7.

Options	Descriptions
CASE_NAME.log	Complete simulation information including the profiled timers
CASE_NAME-git.log	The version number of Alya and the Git repository
CASE_NAME-system.log	Compiler information
CASE_NAME.ker.log	Used solver
CASE_NAME.nsi.log	Profiled timers on the numerical methods
CASE_NAME.par.log	Profiled timers on the parallelization
CASE_NAME.post.alyalog	Simulation time-steps and post-processed times

Table 7: Generated Alya log files and their descriptions.

3.2.3 First base performance analyses

To assess the scaling behavior of Alya on the DEEP-EST system, a benchmark developed by BSC is used. The test case can be accessed through BSC's GitLab repository[34]. It simulates the turbulent flow over a sphere with a computational domain consisting of 16.7 million computational elements and 2.9 million nodes. To compute the solution, a third-order Runge-Kutta operation for the time integration and conjugate gradient method for solving the system of equations are used.

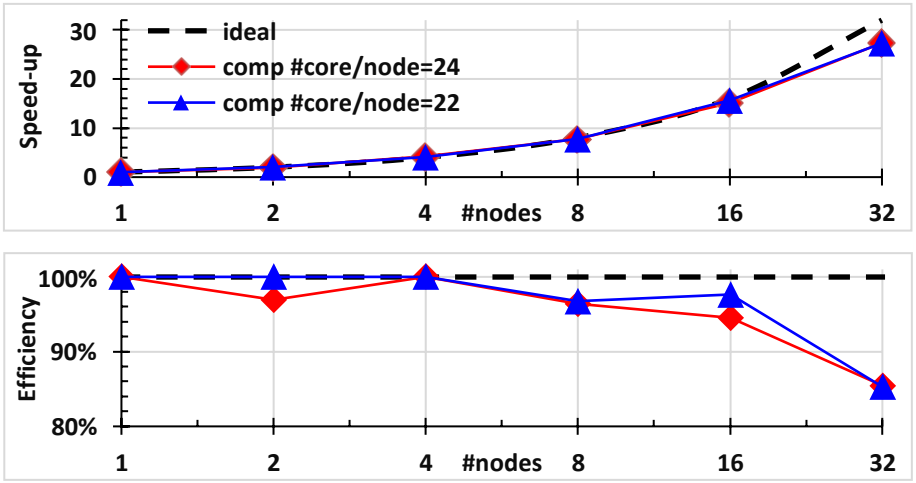

Figure 9: Speed-up of Alya on the DEEP-EST system for a benchmark with 16.7 million computational elements. Strong-scaling results using nodes consisting of 24 cores are represented by the red line. Scaling results with nodes consisting of 22 cores are depicted in blue. The ideal computation and efficiency are represented by the black dashed lines. The efficiency of using multiple nodes (top) and a single node (bottom) are shown. The output routines are disabled.

A strong scaling test using up to 32 compute nodes using a hybrid MPI/OpenMP execution is performed. The DEEP-EST system consists of 24 cores per node. For testing purposes, a scaling analysis using 22 cores per node is also presented. The results are shown in Figure 9. Here, Alya scales very well on the DEEP-EST system and is close to the ideal scaling behavior.

[34] https://gitlab.bsc.es/alya/benchmarks/sphere-16M

Using 22 cores per node in contrast to 24 cores has only a minor effect on the performance, where a slight efficiency increase is observed when two and 16 nodes are used.

3.3 Alya on the JUAWEI system at FZJ

Compiling and running Alya on the JUAWEI system is similar to running it on the DEEP-EST system. The following discussion is similar to Sec. 2.3. Therefore, only major differences are given.

Efforts to port Alya to the JUAWEI system are discussed in Sec. 3.3.1. Section 3.3.2 explains how to run a case. Finally, the results of the first performance analyses are presented in Sec. 3.3.3.

3.3.1 Porting support

In addition to the modifications to Alya to run it on the DEEP-EST system, the `FZJ_RAISE` branch also holds updates to the code to compile it on the JUAWEI system. Alya can be compiled on the JUAWEI system using the newest software stack `2021a`, which is loaded via

```
$ . /opt/ohpc/pub/easybuild/switch_to_eb_sw_stack.sh
$ . /opt/ohpc/pub/easybuild/switch_stage.sh -s 2021a
```

Furthermore, the `GCC` compiler and `CMake` library, as well as the `OpenMPI` library for MPI support can be loaded by

```
$ module load GCC CMake OpenMPI
```

The script `run.sh` is located in the `$ALYA/scripts` directory of the `FZJ_RAISE` branch is also capable of compiling and installing Alya on the JUAWEI system.

3.3.2 Execution of Alya

The execution of Alya on the DEEP-EST system is described in Sec. 3.2.2, and generally running a code on the JUAWEI system is discussed in Sec. 2.3.2. These descriptions are also the base for executing Alya on the JUAWEI system and hence, only the information specific to the execution of Alya on the JUAWEI system are presented in the following. The reader is kindly referred to the corresponding sections for more information.

On the JUAWEI system, Alya can be initiated using the `srun` command from the `slurm` workload manager by

```
srun --mpi=pmix_v3 $PATH_TO_ALYA_BIN/alya $PATH_TO_CASE/CASE_NAME
```

This command initiates the execution of Alya in real-time. Alternatively, a batch script can be used to submit Alya for later execution using the `sbatch` command, cf. Sec. 3.2.2. Again, the `srun` command in the batch script needs to be provided with the type of MPI:

```
$ tail $EXAMPLE_BATCH_SCRIPT

  #SBATCH --cpus-per-task=1

  # Execute
  srun --mpi=pmix_v $PATH_TO_ALYA_BIN/alya $PATH_TO_CASE/CASE_NAME
```

This script can be submitted to `slurm` by executing

```
sbatch $NAME_OF_BATCH_SCRIPT
```

3.3.3 *First base performance analyzes*

To assess the scaling behavior of Alya on the JUAWEI system, the same test case as on the DEEP-EST system is used as a benchmark, i.e., the turbulent flow over a sphere with a mesh consisting of 16.7 million computational elements is simulated.

As the system is small, a strong scaling test using up to only 8 compute nodes with MPI/OpenMP is performed using `x86`-based CPUs. The speed-up is plotted in Figure 10. The results show that Alya also scales well on the JUAWEI system, performing close to the ideal scaling behavior on up to 4 nodes, where a superlinear speed-up is achieved. This behavior can be explained by a high computational workload per core on a single node and a reduced communication effort. Furthermore, caching effects might be the cause. In the eight-node case, a drop in performance of around 20% compared to the ideal speed-up is observed. This also translates that the efficiency drops to 77% when eight nodes are employed.

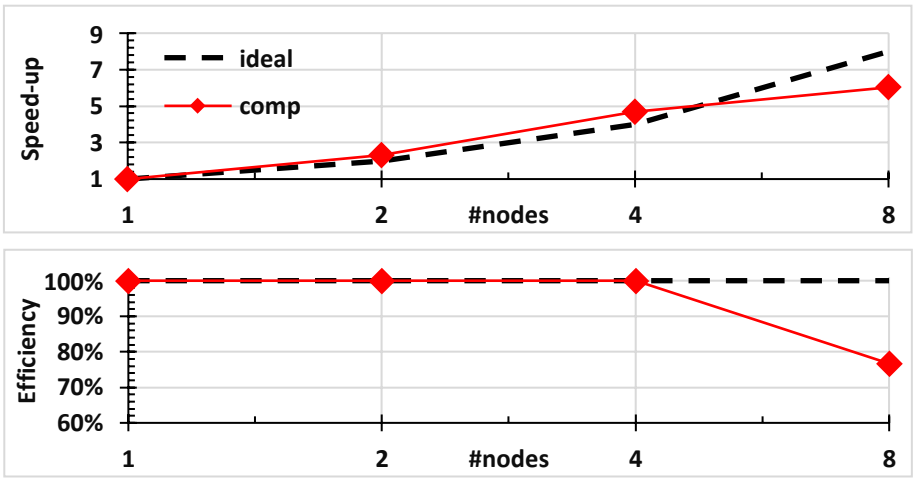

Figure 10: Speed-up of Alya on the `x86`-based CPUs of JUAWEI for a benchmark with 16.7 million elements. Strong-scaling results obtained by using nodes consisting of 20 cores are depicted in red. The ideal computation is represented by the black dashed lines. The achieved speed-up (top) and the efficiency (bottom) are shown. The output routines are disabled.

The run times of Alya for a benchmark with 16.7 million computational elements are quantitatively compared in Table 8 to assess the performance impact of `ARM` and `x86`-based CPUs on the Alya code. In both scenarios, `GCC` is used to compile the Alya code. The runtimes are multiplied by the number of cores per node on each system (see Sec. 2.3.3 for further explanation). It is observed that the `x86`-based CPUs are up to four times faster than the `ARM`-

based CPUs when two or fewer nodes are employed. This performance difference is much larger than the estimated performance difference of m-AIA presented in Sec. 2.3.3 (the difference was 2.5 times). The reason for this could be that a problem size of 16.7 million computational elements is simply too large and is not favored on ARM-based CPUs.

#Nodes / CPU	1	2	4	8
x86	2332.9 /s	1006.1 /s	497.0 /s	386.9 /s
ARM	6367.9 /s	3846.9 /s	5207.1 /s	*4122.8 /s*

Table 8: Comparison of Alya run times for a single time step on ARM and x86-based CPUs of JUAWEI system per node. The run times are multiplied by the number of cores on that node.

Similar to m-AIA, see Sec. 2.3.3, employing ARM-based CPUs yields inconsistent benchmark results when four or more nodes are employed. The results should hence be approached with caution.

3.4 Alya on the CTE-ARM system at BSC

Alya is compiled and run on the CTE-ARM system. The compilation process is similar to that previously described in Sec. 3.2 and Sec. 3.3. Therefore, only core information is given below.

Efforts to port Alya to the CTE-ARM system are discussed in Sec. 3.4.1 and Sec. 3.4.2 explains how to run a case. The results of the performance analyses are presented in Sec. 3.4.3.

3.4.1 Porting support

One 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 based on Intel Xeon Silver 4216, and thus, compiling for the a64fx architecture requires cross-compilation. The recommended way to compile is to use an interactive session on a computing node or sending a compilation job. Another difference between CTE-ARM to the JUAWEI system is that it uses PJM[35] as the workload manager, and therefore, the traditional slurm commands must be replaced by pjm commands. The most common commands are:

- Submitting a job:

```
pjsub <BATCH_SCRIPT>
```

- Show all the submitted jobs and their statuses:

```
pjstat <BATCH_SCRIPT>
```

- Cancel a job from the queue:

```
pjdel <BATCH_SCRIPT>
```

- For an interactive session of 1 node over an hour, the following command is issued:

```
pjsub --interact -L node=1 -L rscgrp=small -L elapse=01:00:00
```

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

Ideally, the compilation should be done with the `Fujitsu` compiler. Unfortunately, this compiler is unable to compile Alya – the compilation duration can take more than 12 hours, hence, compilation becomes unpractical. Therefore, `GCC` is used instead. The required modules for compiling Alya can be loaded with the following command:

```
$ module load gcc/10.2.0 mpiwrap
```

The compilation has been tuned for the `a64fx` architecture by introducing the following options:

```
-march=armv8.2-a+sve -msve-vector-bits=512 -ffree-line-length-512
```

3.4.2 Execution of Alya

Note that the CTE-ARM parallel filesystem is based on the Fujitsu Exabyte File System (FEFS) that is shared among the computing nodes. The login nodes have access to the traditional General Parallel File System (GPFS) system that is shared among the rest of the BSC systems. The simulation files must be stored in the FEFS to use multiple nodes. The scratch directory (`/scratch/`) or its soft-link (`/fefs/scratch/`) can be used for this purpose. Additionally, for executing with MPI, it is necessary to load the following environment variables.

```
export PATH=/opt/FJSVxtclanga/tcsds-1.1.18/bin:$PATH
export LD_LIBRARY_PATH=/opt/FJSVxtclanga/tcsds-1.1.18/ \
       lib64:$LD_LIBRARY_PATH
```

Launching Alya with 48 CPUs on a node requires a `PJM` script such as the one below:

```
#!/bin/bash
#PJM -N parallel
#PJM -L rscgrp=small
#PJM -L node=1
#PJM -L elapse=05:30:00
#PJM --mpi "proc=48,max-proc-per-node=48"
#PJM -o job-%j.out
#PJM -e job-%j.err
export PATH=/opt/FJSVxtclanga/tcsds-1.1.18/bin:$PATH
export LD_LIBRARY_PATH=/opt/FJSVxtclanga/tcsds-1.1.18/lib64: \
       $LD_LIBRARY_PATH
module load gcc/10.2 mpiwrap
mpirun -np 48 ./Alya.x sphere-16M
```

3.4.3 First base performance analyses

The simulation of a flow around a sphere is used for assessing the performance of Alya on the CTE-ARM system. The results in terms of speed-up are presented in Figure 11. Running Alya on CTE-ARM using up to 16 nodes, a parallel efficiency above 70% is achieved in all the cases. Increasing the number of nodes to 32 nodes still accelerates the code but the efficiency decreases to 58%.

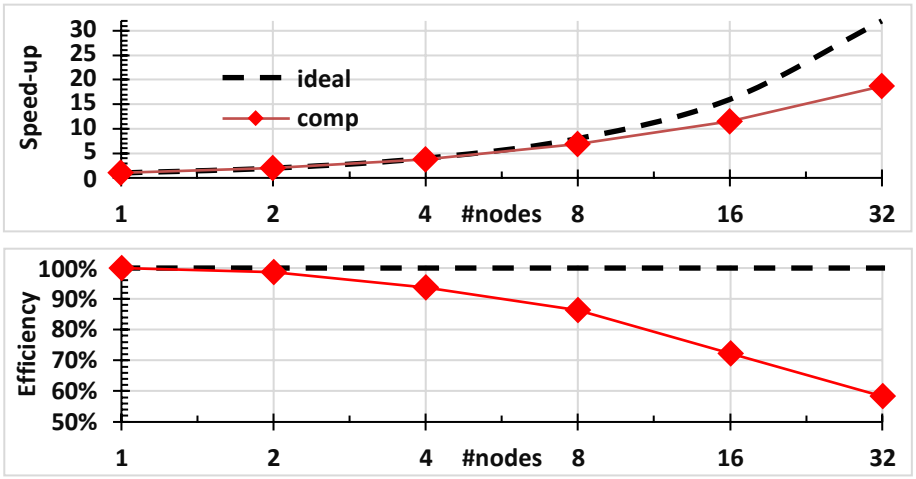

Figure 11: Speed-up of Alya on the CTE-ARM system for a benchmark with 16.7 million computational elements. Strong-scaling results obtained by using nodes consisting of 48 CPUs are depicted in red. The ideal scaling and efficiency are represented by the black dashed line. The achieved speed-up (top) and the efficiency (bottom) are shown. The output routines are disabled.

3.5 Alya on the CTE-AMD system at BSC

The structure of this section follows the one in Sec. 3.4. The porting efforts of compiling Alya on the CTE-ARM system are discussed in Sec. 3.5.1. This discussion is followed by a description of how to run a case using the Alya code on the CTE-ARM system in Sec. 3.5.2. Corresponding results of performance analyses are presented in Sec. 3.5.3. Again, only the major information differences are discussed. For any missing details, the reader is kindly referred to the previous Sec. 3.4.

3.5.1 Porting support

At present, Alya's parallelization does not support the AMD Radeon GPUs. Therefore, the compilation is based on the CPU-only version of Alya. The `GCC` and `Intel` compilers are tested. For the `Intel`-based compilation, the required modules are loaded via:

```
$ module load intel/2018.4 impi/2018.4
```

The only compilation option required is the `-xHost` option, which helps the compiler to recognize the AMD EPYC processor found in the CTE-AMD system. For the `GCC` compilation, the following modules need to be loaded:

```
$ module load gcc/10.2.0 impi/2018.4
```

Note that in both cases, the `impi` wrapper of MPI is used. In this case, the compilation is tuned for the AMD EPYC processor by introducing the option `-mtune=znver2`.

3.5.2 Execution of Alya

The AMD EPYC processor features 64 cores, each one capable of running two hardware threads. For `slurm`, those hardware threads count as MPI ranks. Thus, a single MPI process per core requires the following option.

```
--cpus-per-task=2
```

Submitting a case to the CTE-AMD system is similar to other platforms. For instance, the following script can be used to run Alya on a single node with 64 MPI processes.

```
#!/bin/bash
#SBATCH --job-name=test_parallel
#SBATCH --output=mpi_%j.out
#SBATCH --error=mpi_%j.err
#SBATCH --ntasks=64
#SBATCH --cpus-per-task=2
#SBATCH --time=01:30:00

module load intel/2018.4 impi/2018.4
srun ./Alya.x sphere-16M
```

3.5.3 First base performance analyzes

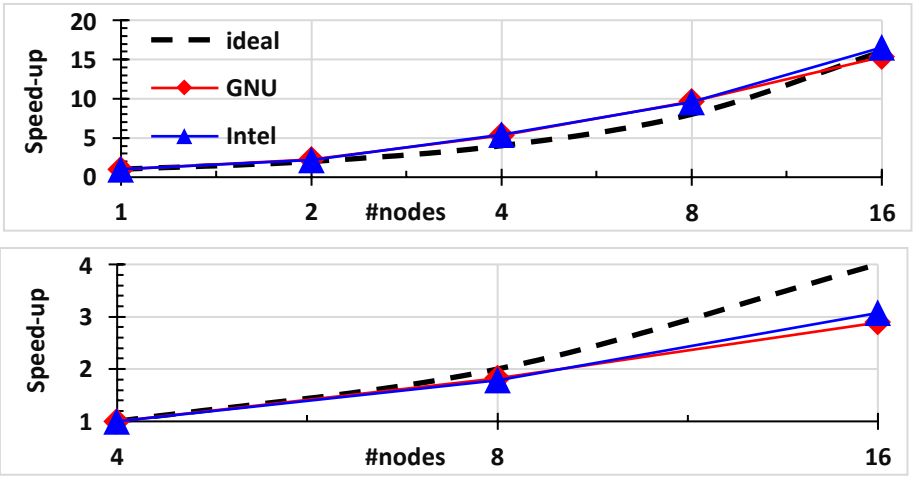

Figure 12: Speed-up of Alya compiled with the `GCC` (red line) and `Intel` compilers (blue line) on the CTE-AMD system for a benchmark with 16.7 million computational elements. Strong-scaling results obtained by using nodes consisting of 64 cores are depicted in red. The ideal scaling is represented by the black dashed line. The achieved speed-up in reference to a single node (top) and four nodes (bottom) are shown. The output routines are disabled.

Figure 12 shows the results of strong scaling experiments of Alya on the CTE-AMD system. They are based on the execution times for a single and four nodes, respectively. The simulation case is again the flow around a sphere using 16.7 million computational elements in an unstructured grid. Alya is compiled using either `GCC` or `Intel`. Considering the strong-scaling results based on the single node execution times, a superlinear speed-up of up to four nodes is achieved. This may happen when the workload per core on a single node is high and the communication effort is compared to this small. In contrast, considering the strong-scaling results based on four nodes, Alya achieves a good scaling on up to eight nodes and satisfactory scales up to 16 nodes. Here, the efficiency of Alya on the CTE-AMD system on up to 16 nodes is higher than 76% – not shown for brevity. Moreover, the Alya code compiled with the `Intel` compiler is only slightly faster than the `GCC` variant when 16 nodes are employed. This is particularly interesting since an AMD CPU still slightly benefits from using the `Intel` compiler.

#Nodes / System	1	2	4	8	16	32
CTE-AMD	165.2 /min	73.9 /min	31.1 /min	17.1 /min	10.8 /min	-
CTE-ARM	225.26 /min	114.2 /min	60.12 /min	32.6 /min	19.5 /min	12.1 /min

Table 9: Comparison of Alya run times for a single time step on the CTE-AMD and CTE-ARM systems. The run times are given per node and are multiplied by the number of cores on that specific node.

The run times of Alya for a benchmark with 16.7 million computational elements are quantitatively compared in Table 9 to assess the performance difference between the CTE-AMD and CTE-ARM systems. In both scenarios, `GCC` is used to compile Alya. The run times are multiplied by the number of cores per node on each system (see Sec. 2.3.3 for further explanation). It is observed that the `AMD` processors are between 36% and 93% faster than the `ARM` processors.

4 First analysis of machine learning frameworks

One of the goals of CoE RAISE is to couple the aforementioned CFD / multi-physics codes to novel ML algorithms. Therefore, it is important to assess the scaling behavior of these methods on prototype systems like the DEEP-EST system.

Apart from the Cluster module (`dp-cn`), the DEEP-EST system also consists of an Extreme Scale Booster (ESB) that features 75 nodes, each equipped with one NVIDIA Volta V100 GPGPU and one Intel Xeon Scalable Silver CPU. While the CPU handles the I/O and communication, the GPGPU offers a very high compute throughput. This enables a suited application to Deep Learning (DL) frameworks since Deep Neural Networks (DNNs) require a lot of dense matrix-vector multiplications. Moreover, other ML methods show benefits for GPGPUs as well.

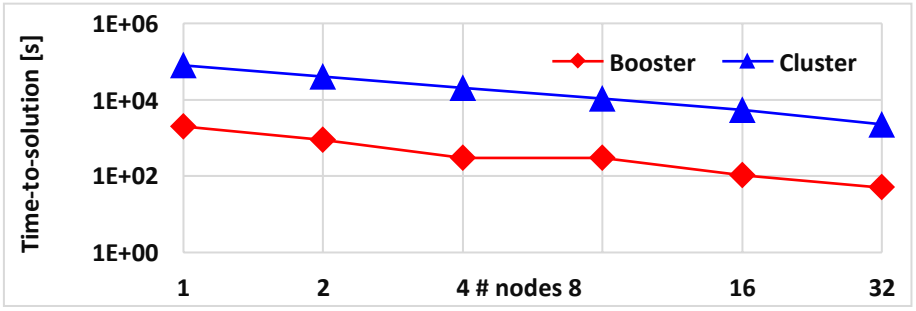

Figure 13: NextDBSCAN time-to-solution on the DEEP-EST Cluster (`dp-cn`) and ESB (`dp-esb`) modules. Mind the logarithmic y-axis.

Figure 13 shows the strong-scaling performance of the clustering algorithm NextDBSCAN[36] (an optimized version of DBSCAN[37]) conducted on the Cluster `dp-cn` and ESB `dp-esb` partitions of the DEEP-EST system. This case applies the algorithm on large LiDAR point-cloud datasets[38,39]. It should be noted that these results are taken from previous work found in this reference[40]. The results in Figure 13 show that the GPGPU-based Booster is a factor of 50 faster than the CPU-based Cluster. This undermines the importance of using GPGPUs for achieving good performance with ML methods.

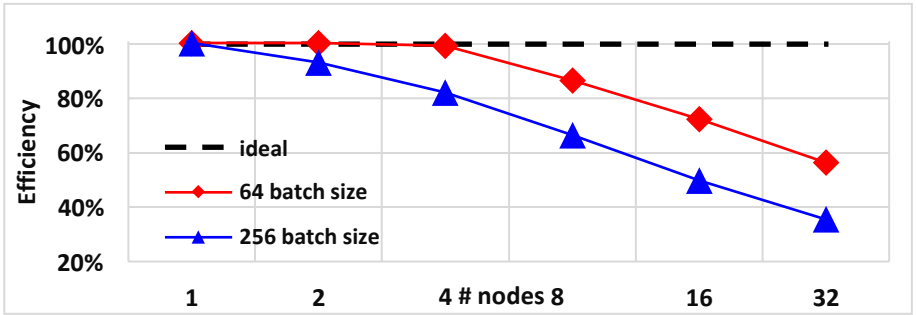

Figure 14: Parallel efficiency of a DL benchmark with satellite image data on the DEEP-EST Extreme Scale Booster using different batch sizes. The ideal efficiency is represented by the black dashed line.

Figure 14 shows the parallel efficiency of a benchmark for training a deep convolutional neural network (CNN) on satellite image data[41] on the DEEP-EST system's `dp-esb` partition. The

[36] NextDBSCAN https://github.com/ernire/nextdbscan-exa
[37] DBSCAN http://www2.cs.uh.edu/~ceick/7363/Papers/dbscan.pdf
[38] https://downloads.pdok.nl/ahn3-downloadpage
[39] https://b2share.eudat.eu/records/7f0c22ba9a5a44ca83cdf4fb304ce44e
[40] DEEP-EST Deliverable 1.5: Final Report in application experience
[41] https://scihub.copernicus.eu/

results of two different batch sizes are given to show the influence on the efficiency. Since several tasks in CoE RAISE, e.g., Task 4.2 on seismic imaging deal with image data, this benchmark is meant to provide a significant indication of the expected performance. The authors used the `Keras`[42] and `TensorFlow`[43] modules on the DEEP-EST system to build the neural network and `Horovod`[44] for distributed training on the GPGPUs.

The results in Figure 14 show high efficiencies on up to four nodes when the batch size is large. The efficiency drops below 60% when 32 nodes are used. Using a smaller batch size negatively affects the efficiencies. Other hyperparameters have no significant effect on the scalability, indicating that adapting the training batch size to the specific hardware is important for optimizing performance.

It is duly noted that these benchmarks are performed only on the DEEP-EST system. For future activities, rigorous investigations on the scaling of ML frameworks are planned on various systems such as the JUAWEI, CTE-ARM, and CTE-AMD systems.

[42] `Keras` https://keras.io/
[43] `TensorFlow` https://www.tensorflow.org/
[44] `Horovod` https://github.com/horovod/horovod

5 Summary and outlook

This document reported on the initial support activities that took place within the scope of WP2 for a period of the past six months. Cases relevant to the use cases of WP3 and WP4 were considered.

Initially, the multiphysics simulation code from RWTH, namely m-AIA, was compiled and run on the prototype systems DEEP-EST and JUAWEI at FZJ, where detailed instructions were provided. The required software libraries were identified and the scaling performance of m-AIA on these systems was analyzed. The code was able to show exceptional scaling performance on these prototype systems. For large cases with many computational elements, the scaling performance of m-AIA yielded better performance for the same case with fewer computational elements. The same exceptional scaling performance was obtained for different compilers and CPU architectures.

The tests continued for Alya, a multiphysics simulation code developed at BSC. Alya was compiled and run on four prototype systems, i.e., on the DEEP-EST and JUAWEI at FZJ, as well as on the CTE-AMD and CTE-ARM systems at BSC. Similar to m-AIA, the scaling performance of Alya was tested using various compilers and CPU architectures. The results of Alya showed an almost perfect scaling performance, even when more than half of the cores on these prototype systems were used.

Finally, an overview of the ML capabilities of the DEEP-EST system were analyzed, noting that further research on this topic is currently ongoing. The importance of the GPGPU-based parallelization and the size of a batch for CNN training was emphasized. It was evident that the ML framework highly benefits from using GPGPUs rather than CPUs.

It was demonstrated that the simulation frameworks relevant to WP3 and WP4 can benefit from using the prototype systems mentioned in this document. The efficiencies of these frameworks utilizing `ARM`-based CPUs are still under investigation and the presented results are still preliminary. That is, further investigations are necessary to achieve similar scaling performances on `ARM`-based systems with respect to scaling and time to solution under a potential reduction of the required energy. Furthermore, the scaling performance of ML frameworks is to be tested on these prototype systems. In the future, the multiphysics simulation codes and ML frameworks are planned to be executed simultaneously to implement complex intertwined simulation- and AI-workflows at Exascale. This can only be realized through the development and extension of individual codes using various prototype components of systems as investigated here.

References

[1] Lintermann, A., Meinke, M., & Schröder, W. (2020). Zonal Flow Solver (ZFS): a highly efficient multi-physics simulation framework. International Journal of Computational Fluid Dynamics, 34(7–8), 458–485. https://doi.org/10.1080/10618562.2020.1742328

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

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

List of Acronyms and Abbreviations

AI	Artificial Intelligence
AIA	Aerodynamic Institute Aachen (Institute of Aerodynamics and Chair of Fluid Mechanics)
ARM	Advanced RISC Machines
BSC	Barcelona Supercomputing Center
CNN	Convolutional Neural Network
CFD	Computational Fluid Dyanmics
CI/CD	Continuous Integration / Continuous Delivery
CTE	Cluster de Tecnologies Emergents
DEEP-EST	Dynamical Exascale Entry Platform - Prototype System
DL	Deep Learning
DLB	Dynamic Load Balancing
DNN	Deep Neural Network
EoCoE-II	Energy Oriented Center of Excellence II
ESB	Extreme Scale Booster
FEFS	Fujitsu Exabyte File System
FFTW	Fastest Fourier Transform in the West
FPGA	Field Programmable Gate Array
FZJ	Forschungszentrum Jülich
GCC	GNU Compiler Collection
GPFS	General Parallel File System
GPGPU	General Purpose Graphics Processing Unit
HDF5	Hierarchical Data Format version 5
HLRS	High-Performance Computing Center Stuttgart
HPC	High-Performance Computing
I/O	input/output
JURECA	Jülich Research on Exascale Cluster Architectures
JUWELS	Jülich Wizard for European Leadership Science
KNL	Knight’s Landing
m-AIA	multiphysics-AIA
ML	Machine Learning
MSA	Modular Supercomputing Architecture
MPI	Message Passing Interface
NAM	Network Attached Memory
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
RISC	Reduced Instruction Set Computer
RWTH	RWTH Aachen University
TBL	Turbulent Boundary Layer
TGV	Taylor-Green Vortex
UCX	Unified Communication X
UEABS	Unified European Applications Benchmark Suite
WP	Work Package