Sodalite

SOftware Defined AppLication Infrastructures managemenT and Engineering

Full release of application and infrastructure performance models

D3.4

HPE

31.10.2021

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825480.

Deliverable data			
Deliverable	D3.4 Full release of application and infrastructure performance models		
Authors	Alfio Lazzaro, HPE Nina Mujkanovic, HPE Tiziano Müller, HPE		
Reviewers	Kamil Tokmakov, USTUTT Jesús Gorroñogoitia, ATOS		
Dissemination level	Public		
History of changes	Name	Change	Date
	Alfio Lazzaro	Initial version created	14.9.2021
	All	First draft	1.10.2021
	All	Results update	22.10.2021
	Alfio Lazzaro (HPE)	Sent for internal review	26.10.2021
	All	Final version	29.10.2021

Acknowledgement

The work described in this document has been conducted within the Research & Innovation action SODALITE (project no. 825480), started in February 2019, and co-funded by the European Commission under the Information and Communication Technologies (ICT) theme of the H2020 framework programme (H2020-ICT-16-2018: Software Technologies)

Executive Summary

Applying optimisations specifically targeting the hardware is crucial to get performance for compute intensive applications. However, it breaks performance portability, such that a user is faced with different optimised implementations of the same application. In this report, we present a procedure to manage optimised applications within containers to specifically address the goal of achieving performance on specific target hardware. This deliverable describes the full release of the application and infrastructure performance models for the static optimisations, ie. optimisations applied before the deployment of the application. In particular, it focuses on the progress of the work with respect to the first-year activity, as reported in the deliverable D3.3 [2]. Additionally, we update on the development of the SODALITE application optimiser called MODAK (Model Optimised Deployment of Applications in Containers), which was already introduced in the deliverable D4.2 [3]. A performance model for the parallel applications is also presented. This model is used within MODAK for predicting the optimal number of parallel processes when running a MPI parallel application for a best usage of the compute resources. Finally, we present results of the optimisations and handling of the optimised containers via MODAK for DL training and MPI-based applications taken from the SODALITE use cases.

For future activity up to the end of the project, we will further extend MODAK functionalities, for example, we plan to introduce an automatic mechanism for efficient data management. Furthermore, we will improve the integration of MODAK with the other SODALITE components. This activity will be reported in the future deliverable D4.3.

Table of Contents

List of Figures

List of Tables

Glossary

Acronym	Explanation
ABI	Application Binary Interface
AI	Artificial Intelligence
API	Application Programming Interface
CNN	Convolution Neural Network
CPU	Central Processing Unit
DL	Deep Learning
DSL	Domain Specific Language
GPU	Graphic Processing Unit
HPC	High Performance Computing
HPCM	HPC Container Maker
IaC	Infrastructure as Code
IDE	Integrated Development Environment
IO	Input/Output
KB	Knowledge Base
ML	Machine Learning
MPI	Message Passing Interface
MODAK	Model Optimised Deployment of Applications in Containers
QE	Quality Expert
SSD	Solid-State Drive

1 Introduction

Software application developers and users are now targeting a wide range of diverse computing platforms, such as on-premises supercomputers and clouds with heterogeneous node architectures. Container technology has grown popular as a bridge between these heterogeneous environments due to ease of use, portability, scalability, and the advent of user-friendly runtimes. Containers provide a straightforward way to share scientific applications and reproduce research on either cloud or High Performance Computing (HPC) systems. Compute intensive applications, such as Deep Learning (DL) training that use HPC systems, have specific requirements for specialised execution environments including computing accelerators, high speed interconnects, and fast memory and storage. Even if containers provide both flexibility and portability, we still need applications to optimally use and benefit from these diverse resources. For example, DL training frameworks require target-specific libraries and drivers to be configured.

In the context of HPC infrastructures, with diverse hardware and software dependencies and libraries, building or selecting an optimised container for deploying DL-based components is crucial. The same concepts apply for Message Passing Interface (MPI) applications, where the applications have to efficiently use the network to get performance and parallel scalability. To address these issues, SODALITE developed a set of application performance models and infrastructure, based on optimised containers. The aim is to take a performance-centric view of the applications and infrastructure and model them to enable performance decisions to be made.

Performance can be optimised before deployment (*static optimisations*) or at runtime (*dynamic optimisations*). This deliverable describes the full version of the application and infrastructure performance models, specifically for the static optimisations. Dynamic optimisations progresses were reported in the deliverable D5.2 [1], published at the end of the second year of the project, and the final description will be reported in the deliverable D5.3 (end of the project). This deliverable focuses on the progress for the static optimisations with respect to what was reported in the initial prototype for application and infrastructure performance models presented in the deliverable D3.3 [2] (first year) and the update of the application optimiser component described in the deliverable D4.2 [3] (second year).

1.1 Deliverable goal

This deliverable describes the full version of the application and infrastructure performance models for the static optimisations. It describes the solution adopted, as well as its application on the SODALITE use-cases. In particular, it focuses on the progress of the work with respect to the first-year activity, as reported in the deliverable D3.3 [2]. Additionally, we update on the development of the SODALITE application optimiser called MODAK (Model Optimised Deployment of Applications in Containers), which was already introduced in the deliverable D4.2 [3].

1.2 Overall objectives of the project

The main project goals along with the WP3 perspective on them are summarised as follows:

- **O1 objective:** *The key objective is to provide code (application), resource (infrastructure) and execution semantic abstractions, injected with infrastructure performance abstractions, to ensure maximum performance of the so-abstracted application and infrastructure when concretized on specific infrastructure. We build the abstractions as extensions of standardised approaches, aiming at both machine and human readability.*

  WP3 perspective: We developed a Domain Specific Language (DSL) for the application optimisation to enable performance decisions to be made before the deployment of an application. The DSL specifies the application optimisations related to the hardware (eg. CPU and GPU specific hardware instructions) and software (eg. optimised libraries). The optimisations are used to build optimised application containers, as documented in this

deliverable, so that they can execute faster with respect to unoptimised baseline containers. The Modelling layer enables the design of the optimisations DSL in the IDE[1].

- **O2 objective:** *To increase the performance of the deployed software on target platforms through static optimisation, using the infrastructure performance patterns abstractions and through dynamic optimisation, using the predictive deployment refactoring approach, building on the run-time available data from the application and the platform.*
  WP3 perspective: Static optimisations are supported through the MODAK, the SODALITE Application Optimiser introduced in the deliverable D4.2 [3], whose latest developments are discussed in this deliverable.

- **O3 objective:** *To reduce the cost of software development, deployment, management and adaptation or reconfiguration in the modern infrastructures, using abstraction of the typical components (e.g., compute, storage, network) and combining them with non-functional requirements, allowing for an application to target multiple concrete infrastructures.*
  WP3 perspective: The application and hardware model abstraction used to build optimised application containers, together with the application performance model is only based on infrastructure capabilities (accelerators, available runtime environments) and the application itself. It can therefore be shared and reused via MODAK across different infrastructures without rebuilding the containers.

- **O4 objective:** *To address abstractions, technologies, targeted applications, and infrastructures holistically, allowing for flexible, reusable, and long term supported software development stack for modern runtime infrastructures and professional applications.*
  WP3 perspective: The optimisation DSL is an integral part of the Modelling Layer (WP3) of the SODALITE stack and is integrated with the other components developed within the layer, namely the IDE, the Knowledge Base and the Semantic Reasoner. These components interact with components of the other SODALITE layers, namely the IaC (WP4) and Runtime (WP5) layers.

- **O5 objective:** *To use and build on existing solutions, starting with community building or inclusion from day 1.*
  WP3 perspective: All the performance model and MODAK code is open-source. We base our development on other and well-established open-source solutions, more specifically: Pydantic[2] classes are used to model the data objects behind the DSL, FastAPI[3] and Uvicorn[4] to provide an asynchronous REST API and OpenAPI[5] spec and SQLite[6] to store the infrastructure, container, and performance data.

- **O6 objective:** *To demonstrate the developed concepts using relevant professional applications and industries, covering complete software stacks.*
  WP3 perceptive: The static performance optimisation component supports the DL and MPI applications used in the SODALITE use-cases, representative of relevant professional applications and industries.

[1] More details on the IDE can be found in the SODALITE deliverable D3.2.
[2] https://pypi.org/project/pydantic/
[3] https://pypi.org/project/fastapi/
[4] https://pypi.org/project/uvicorn/
[5] https://swagger.io/specification/
[6] https://www.sqlite.org

1.3 Work performed from the beginning of the project

During the first year of the project, we developed an initial prototype of the application and infrastructure performance models using standard benchmarks and applications based on the features that influence performance (see deliverable D3.3 [2]). We considered the raw CPU performance (in terms of floating-point operations), the compute host memory performance (bandwidth), the network performance for connecting compute nodes (bandwidth), and the IO performance when writing to a given file system (number of reads/writes). We did not consider any accelerator performance (such as GPUs). These performance values were used to make an infrastructure performance model for a specific hardware system, based on linear polynomial functions for the scaling part over multiple execution compute nodes. The infrastructure performance model was then used to build a specific application performance model for the multi-node scaling execution of an application running on that hardware. The outcome of the application performance model is the speed-up of execution (time-to-solution metrics) when varying the number of compute nodes.

In the second year of the SODALITE project, we developed the MODAK component to make use of the application performance model, especially for the autoscale component (see section 2). We also tried to extend the application performance model to include accelerator executions, namely GPUs. We found that the initial approach cannot easily be generalised to hybrid hardware with acceleration executions because of the possible asynchronous execution between the CPU and GPU. Furthermore, the initial performance models based on linear polynomials are too approximated for complex application executions. Therefore, in the last year of the SODALITE project we have been developing a new performance model to address all those issues, which is described in section 2.2. The optimisation DSL was finalised (see section 2.1) and MODAK was further extended and integrated with the other SODALITE components, as we describe in section 2.3.

1.4 Progress beyond the state of the art

The activity reported in this deliverable is contributing to the progress beyond the current state of the art by offering a common ecosystem for preparing and running optimised application containers on any system by using a performance model driven approach. The EASEY framework enables not only building application containers for target clusters and MPI libraries, but also manages the deployment, job management, and data staging [4]. While this approach is similar to that of MODAK, it does not model the performance optimisations. Concerning the performance model deduction, Baughman *et al.* [5] used application profiling and historical data gathered on HPC and cloud systems to create application performance models. There are multiple approaches taken in the HPC community to develop a model based on profiling an application, micro-benchmarking primitive components of the application, or simulation of application changes and analytical modelling [6, 7, 8]. Although we use a similar approach, those methods do not consider the possibility to develop optimised containers for the deployment. MODAK also addresses the current challenge in the HPC environment of choosing an appropriate application container, which was built for a given set of hardware capabilities, and assuring the container runs on the correct hardware. Therefore, to the best of our knowledge, our project presents a novel, model-based approach to enable static optimisation of applications within containers for deployment on heterogeneous hardware.

1.5 Structure of the document

This deliverable is structured as follows:

- Section 2 reports on the static performance optimisation, including the description of the performance model, the optimisation DSL, and the MODAK infrastructure.

- Section 3 shows results when using MODAK on two applications, taken from two SODALITE use-cases [9]: a DL training and a mathematical solver, which we have identified as adequate candidates for our optimisations.
- Finally, section 4 presents the conclusions.

2 Static Performance Optimisation

Static performance optimisation targets specific optimisations which are applied before the deployment of an application. This involves porting the application to a target hardware and then manually optimising it. The optimisation process usually involves using target specific optimised libraries, enabling application specific optimisations, and tuning application and library parameters. Parallel scaling requires that these optimisation steps be repeated at increasing scales until efficiency drops below a threshold. Application experts may also modify the code to target specific hardware optimisations. This optimisation process may not be portable to other targets and require repetition when moving to other systems. As such, it is a resource and time intensive effort that requires expertise on application and infrastructure. Most applications are optimised once, and the optimised configuration is reused for subsequent runs. In our setup, we chose to build the applications within containers, therefore the optimisations are applied when building the containers. The Singularity container technology [10] was chosen to provide a portable and reproducible runtime for the application deployment, due to better performance and native support for HPC resources compared to other popular container technologies.

Even though containers offer portability across different targets, optimisations require in-depth knowledge of the system and compatible libraries (MPI, Network) for any particular HPC system. For example, MPI libraries and versions on the host machine and in the container should match when deploying HPC applications for the container to use the hardware-optimised version of MPI available on the host. DL training frameworks require target specific libraries and drivers to be configured. More specifically, binaries can be optimised for a given hardware architecture, for instance for different CPU instruction sets or GPU accelerators (AMD or NVIDIA). Building the containers then becomes an *ad-hoc* process for a specific hardware and application, which implies techniques such as cross-compiling, fat binaries, or runtime binary dispatch. This is a one-to-one process for the application container preparation between application optimisations and target hardware. Hence, for a given application we end up with multiple versions of the container with different optimisations: various combinations of compilers, MPI implementations, Linux distributions, CPU instruction sets, with and without GPUs and GPU type, optimised libraries, etc. Several papers in the literature report that optimised Singularity containers can reach comparable performance to native execution of the applications (see for example [11]).

In the context of our project, the optimised container build is performed by the Quality Expert (QE). The procedure consists of three steps, in order to minimise the execution time:

1. building with given optimisations
2. checking the correctness (for example, to avoid numerical instabilities due to aggressive optimisations) of the optimised containers by running, whenever available, the application test suites
3. benchmarking on the target hardware.

These steps are iterated up to a given performance improvement over the initial baseline represented by the most portable container. In general, the more optimisations are used, targeting the specific hardware, the less portable the container becomes. There exists a wide range of application optimisations, which also depend on the input configuration of the application. However, in most cases there are a few that will heavily influence the performance and scaling of an application. Common optimisations that the QE may consider are:

- Compiler optimisations for the specific hardware, eg. use of the appropriate vector instructions sets (eg. AVX, AVX2, AVX512). Eventually, this requires cross-compiling all the

dependencies instead of using the pre-compiled modules available for example in the Linux distributions.

- Linking with optimised libraries, eg. algebra libraries such as the Intel Math Kernel Library and the Accelerated Linear Algebra (XLA) graph compiler for TensorFlow applications [12].
- Use of the accelerator drivers (CUDA or ROCM) if they are supported by the applications.
- Use of the optimised host libraries, for example for the network communications via MPI. In this case, the host and container MPI implementations must be ABI compatible.

The benchmarking is done by employing representative benchmarks of the final application execution, which can be executed in a limited time and with a limited amount of compute nodes for parallel applications (ideally a single node). Indeed, complex applications can make use of different algorithms that require specific optimisations, depending on the input user configurations, so it is important to use representative benchmarks of real applications usages. The optimisation procedure will fail if a different configuration is used in production. It can be beneficial for the QE to interface with the application domain experts so that existing optimisation knowledge can be reused.

All optimisations are formalised in a DSL configuration (see section 2.1), which is then stored within MODAK and associated with a handle for the optimised container. To handle the proliferation of optimised containers, their definition files are generated via the HPC Container Maker (HPCCM) tool [13]: all Singularity definition files for the optimised containers belonging to an application are generated from a single high level Python recipe. Then the QE prepares the performance models (see section 2.2), which requires running the containers on the specific hardware with different numbers of nodes for parallel applications. Finally, the QE uses MODAK to register the containers with their corresponding input DSL similar to what is described in 4.3.3 of D4.2 [3] and the performance model (see section 2.3). Figure 1 illustrates the procedure for the container's registration. Based on the DSL input configuration, MODAK produces a unique identifier for the container (a combination of name and tag of the container). This name, the optimisation configuration, the definition file used to build the container, and the performance model are registered in a configuration database, while the container is stored in a Singularity image registry.

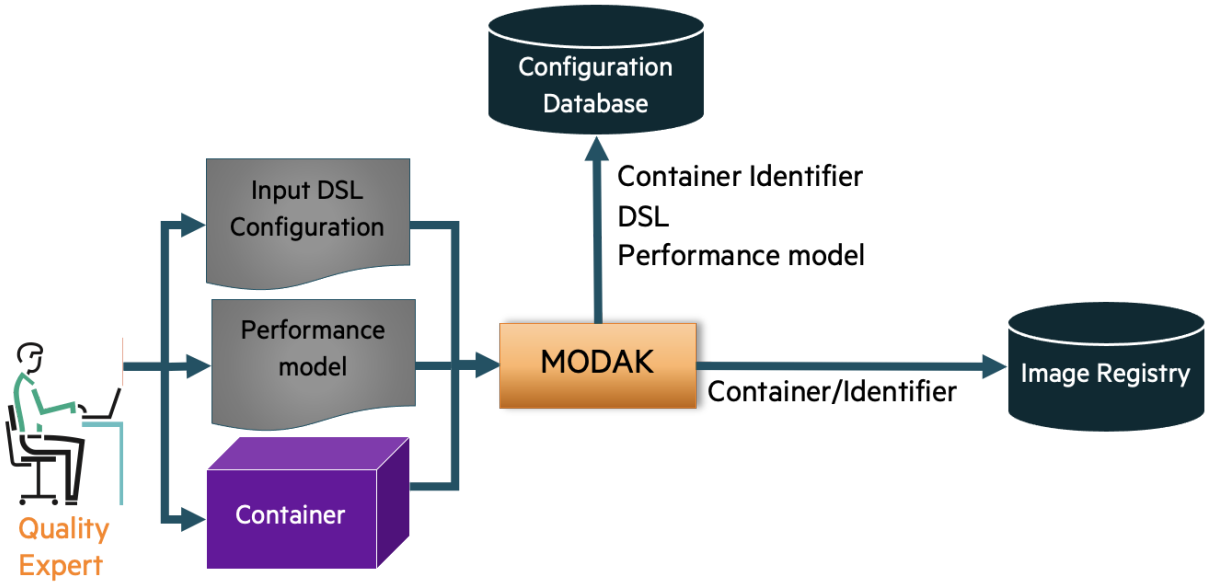

Figure 1. Registration of the optimised containers via MODAK.

A user that uses the SODALITE framework provides the input DSL for the optimisations, which is passed to MODAK, as shown in Figure 2. MODAK exposes a high-level application API for the two

types of applications supported: DL training and inference and MPI-parallelised applications. In summary, MODAK requires the following inputs for the user deployment applications:

- Job submission options for batch schedulers like SLURM and TORQUE (but in a scheduler-independent fashion), if any (the fallback is to run without a batch scheduler)
- Application configuration such as application type (see above), run and build commands
- Optimisation DSL with the specification of the target hardware, software libraries, and optimisations to encode. Some of this information (e.g., the target hardware) is automatically generated during the workflow deployment preparation based on the target infrastructure (which has to be registered beforehand).

After providing the inputs, MODAK searches in the configuration database for an existing container matching the optimisation input request. Note that the selection of the optimised container is based on a best match logic, ie. MODAK tries to find a working container which best matches with the input requests. For example, if a user requests an application container with GPU support, which is not available in the image registry, MODAK will fall-back to the CPU version. In other words, optimisations are applied incrementally, providing a portable baseline container version. Then, if the container is found, MODAK uses the corresponding performance model output to produce a job script for the execution batch submission and returns the link to download the optimised container from the image registry. In this respect, MODAK acts as a container manager, such that users do not need to search the optimised application container for their hardware among possibly hundreds of available container versions, letting MODAK do the selection.

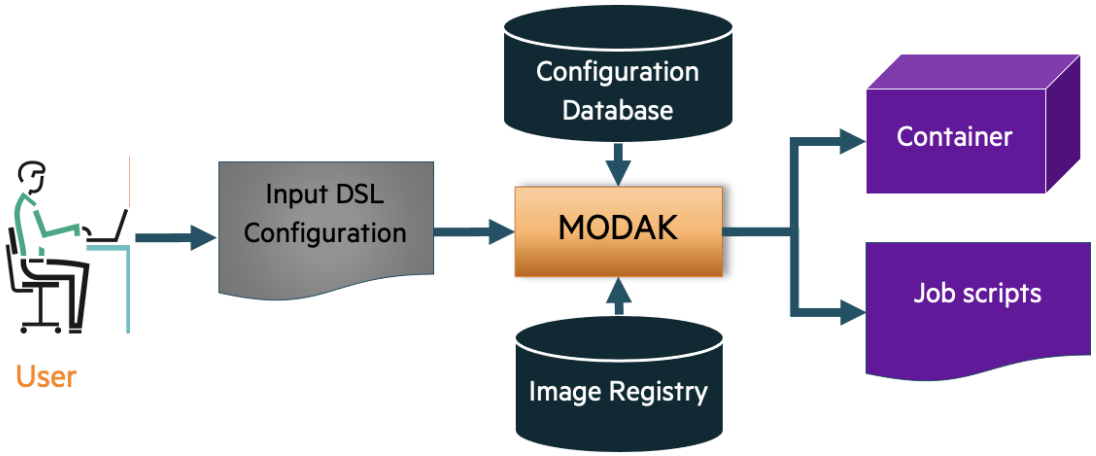

Figure 2. Selection of the optimised containers via MODAK.

2.1 Optimisation DSL

The optimisation DSL contains the user specified input used by MODAK to recognize optimised application containers. The SODALITE IDE and the other components developed within WP3 are used to fill the DSL, whose schema definition is shown in appendix A.

During the last year of the project, the DSL was finalised and implemented in MODAK via Pydantic classes used to model the data objects behind the DSL. From there, both OpenAPI and JSON schema and then further documentation can be generated.

The DSL follows in principle the layered structure of the MODAK application itself (Mapper, Enforcer, etc) in terms of which attributes are mandatory, but also contains more concrete information about the runtime environment like the number of processors requested. To avoid maintaining application state and not tying the data within MODAK to explicit workflow steps within SODALITE, the DSL also has to provide additional application specific configuration and

hence contains explicit support for different ML frameworks (such as PyTorch, TensorFlow) and programming environments, down to application specific notions like AI training vs inference.

With that, the DSL supports the DL training and MPI applications, which are the two compute intensive workloads of the SODALITE use-cases (see section 3). Specific attributes are provided to optimise the DL TensorFlow application, namely for the XLA graph compiler. Finally, other attributes are provided to request GPU accelerators, the deployment options, and eventual data to be used to execute the application.

The following table shows the current definition of the MODAK DSL (limited to the frameworks used in the SODALITE use cases). This DSL is used to communicate with MODAK through a HTTP REST API on different endpoints. Each request shall be done by sending a compliant JSON document within a POST request to obtain a partially augmented document in the answer body as follows:

- `/optimise` to get `job.job_script` and `job.build_script` attributes filled
- `/get_image` to get `job.application.container_runtime` attribute filled
- `/get_build` to get `job.build_script` attribute filled
- `/get_optimisations` to get `job.application.container_runtime` and `job.job_content` attributes filled

Note: whenever `job.*_script` is returned, the corresponding `job.*_content` attribute may be filled instead with the full content of the generated script.

2.2 The performance model

Scaling applications to more nodes improves the performance of most MPI parallel applications. The parallel speedup and scaling efficiency is defined as follows:

$$\text{Parallel Speedup} = \frac{T_{ref}}{T_{parallel}},$$

$$\text{Efficiency} = \frac{n_{ref} T_{ref}}{n\, T_{parallel}},$$

where T_{ref} and $T_{parallel}$ correspond to the runtime on a reference number of nodes n_{ref} (usually a single node), and the runtime on n nodes, respectively. We developed a performance model that aims to model the efficiency of an application. Once the optimised container has been built, the QE can use the representative benchmarks used during the optimisation to check how the performance scales versus the number of nodes, so that the parallel efficiency can be extracted. This includes the execution of the GPUs, if available. In order to make better measurements, timings are taken multiple times and the averages are used. Then the points are fitted with Amdahl's law [14]:

$$\text{Model\_Efficiency} = \frac{n_{ref}}{n(1-F) + F}.$$

The fit is based on a maximum likelihood fitting procedure, where F is the free parameter. Errors are propagated via a Monte Carlo technique. An example of the performance model extraction is given in section 3.2.

While we aim to achieve higher speedups as we increase nodes, poor efficiency denotes higher overheads and higher costs. Applications are usually scaled until the efficiency drops below a certain percentage (the latest when the efficiency becomes negative). In MODAK, the autoscale component uses the performance model to predict the efficiency of an application on n nodes and automatically increase the number of nodes of the deployment, if requested by the user, to reach a given efficiency value. Furthermore, additional constraints may be added in this optimisation step, like favoring specific numbers of processors (usually square numbers or power of 2).

2.3 MODAK infrastructure

MODAK has already been described in deliverable D4.2 [3]. Here, we briefly give an overview of its components:

- **Mapper**: maps application deployment to an optimised container based on the user specified input DSL (see section 2.1) based on the information provided previously by the QE. The configuration database is implemented with SQLite, to accomodate for expected high read to write ratio and ease of deployment.
- **Enforcer**: sets the pre- and post- requisites for running a container. For example, enabling graph compiler-based optimizations in a DL framework requires environment settings to be modified. For MPI-based applications, there are many environment settings that change the way message passing is optimised based on message size and communication pattern.
- **Autotune**: allows users to run a user-defined script for autotuning the applications before they are deployed. This allows it to integrate further optimisations beyond the current scope of MODAK.
- **Autoscale**: uses the performance model to scale MPI parallel applications to use more nodes. Applications are usually scaled until the parallel efficiency drops below a certain percentage defined by the user (see section 2.2).

Except for the autotune, which will be implemented in the last months of the project within the WP4 activity and reported in the deliverable D4.3 by the end of the project, all other components were implemented. In particular, during the last year of the project we have implemented an autotune part and finalised the mapper and enforcer. A Singularity registry has been developed within WP6 and it will be documented in the deliverable D6.4 by the end of the project.

3 Performance results

In this section, we report results for the application optimisation procedure, the performance model extraction, and the MODAK use on two applications. We have identified two applications that are compute intensive workloads of the SODALITE use-cases [9]:

- a DL training part of the Snow UC
- an MPI parallel mathematical optimisation procedure part of the Clinical UC

Both examples have been identified as adequate candidates for our optimisations and they represent examples of relevant professional applications.

The HPC testbed hosted in USTUTT [15] was used during the tests. It consists of a front-end node running Torque, and 5 compute nodes, each hosting an Nvidia GeForce GTX 1080 Ti GPU, a dual-socket Intel(R) Xeon(R) CPU E5-2630 v4 processor (20 cores in total), and 128GB of main memory. Two MPI implementations are available: OpenMPI v3.1.3 and MPICH v3.3.1. Singularity is version 3.8.1.

3.1 Snow UC

We have optimised the skyline extraction component for the Snow use case. The goal of this component is to obtain the landscape skyline of a photograph via a DL classification method run in TensorFlow. The dataset used for the training consists of 8,856 images with skyline annotations, from which 80% is used for training and validation and the remaining 20% for testing. The component was initially trained using TensorFlow 1.11. The training was performed with a baseline container taken from DockerHub (tensorflow/tensorflow:1.11.0-gpu-py3) and converged within approximately 7.2 hours on one GPU node of the HPC testbed (using single core execution). The training executed until convergence was achieved and early stopping initiated at epoch 20. A detailed profiling of the application execution is reported in section 6.3.1.1 of the deliverable D3.3 [2].

The first step of the optimisation process was the porting of the Python training code for TensorFlow 2.2, as it has been optimised by the developers to outperform the outdated TensorFlow 1.11. Therefore, we built an optimised Singularity container with TensorFlow 2.2. As a sanity check, we performed a run until convergence which finished within 8056s (approximately 2.3 hours) and 20 epochs.

As training times converge across epochs within 2-3 epochs, we trained the skyline extractor for 5 epochs across every further optimisation we considered. For the initial Singularity container with TensorFlow 2.2, that took 3473s, of which 872s constitute training time, while the rest includes data batching time. This is a well-known bottleneck for DL applications involving massive datasets. To account for this, we optimised the Python code to perform batch dataset prefetching to the GPU via the TensorFlow Data API. This shortens the execution pipeline by performing training and data input concurrently. The training time thus improved to 2181s, of which 514s constitute training time.

As a final optimisation, we optimised the data movement by staging the dataset on an SSD attached to the GPU node. The dataset was moved to the SSD, and the dataset directory passed to Singularity via file binding. This optimisation improved the training time to 424s, of which 236s constitute training time. This yields an 8.2x speedup improvement over the initial TensorFlow 2.2 run. We tested additional optimisations such as using XLA and various combinations of SSD, GPU prefetching, and XLA, but these did not yield significant improvements.

Finally, we executed the optimised container up to convergence. It takes 2042s with 21 epochs. Overall, this is a 12.7x speedup. The test accuracy value is compatible with the baseline container result. A summary of the execution times is reported in Table 1.

Container configuration	Execution time (seconds)
Baseline container taken from DockerHub (tensorflow/tensorflow:1.11.0-gpu-py3)	25920
Optimised Singularity container: TensorFlow 2.2, batch dataset prefetching, dataset staging on SSD	2042

Table 1. Execution time of the SnowUC DL baseline and optimised containers used for the training up to convergence of the model.

MODAK can be used to automate the process of choosing an optimal container, thus returning the best possible container, in this case one that stages the dataset to an SSD. MODAK stores and retrieves the optimised container to and from the SODALITE registry by means of the following DSL input:

```
{
  "job": {
    "target": {
        "job_scheduler_type": "torque"
    },
    "job_options": {
      "job_name": "skyline-extraction-training",
      "node_count": 1,
      "request_gpus": 1,
      "request_specific_nodes": "ssd"
    },
```

```
    "application": {
      "app_tag": "skyline-extraction-training",
      "app_type": "python",
      "executable": "python3 peaklens-original-training_new.py"
    },
    "optimisation": {
      "enable_opt_build": true,
      "app_type":"ai_training",
      "opt_build": {
        "cpu_type": "x86",
        "acc_type": "nvidia"
      },
      "app_type-ai_training": {
        "config": {
          "ai_framework": "tensorflow"
        },
        "ai_framework-tensorflow": {
          "version": "2.2.1",
          "xla": true
        }
      }
    }
  }
}
```

Based on the above DSL input, MODAK will provide the link to the optimised container and the following submission script for Torque:

```
#PBS -S /bin/bash
## START OF HEADER ##
#PBS -N skyline-extraction-training
#PBS -l nodes=1:ppn=1:gpus=1
#PBS -l nodes=ssd
#PBS -o job.out
#PBS -j oe
## END OF HEADER ##

cd "${PBS_O_WORKDIR}"
export PATH="${PBS_O_WORKDIR}:${PATH}"
## MODAK: START OF OPT:XLA ##
mkdir xla_dump
export TF_XLA_FLAGS="--tf_xla_auto_jit=2 --tf_xla_cpu_global_jit"
export XLA_FLAGS="--xla_dump_to=xla_dump/generated"
## MODAK: END OF OPT:XLA ##
```

```
singularity exec --nv "$SINGULARITY_DIR/tensorflow_2.2.1-gpu.sif" python3
peaklens-original-training_new.py
```

The $SINGULARITY_DIR is the path where MODAK assumes the orchestration framework downloads the container to. For the sake of completeness, we also report the MODAK response that will contain the same DSL with some of the defaults filled and additional attributes added as described in section 2.1:


```
{
  "job": {
    "job_options": {
      "job_name": "skyline-extraction-training",
      "wall_time_limit": null,
      "node_count": 1,
      "request_gpus": 1,
      "core_count": null,
      "process_count_per_node": 1,
      "standard_output_file": "job.out",
      "standard_error_file": null,
      "combine_stdout_stderr": true,
      "request_event_notification": null,
      "email_address": null,
      "copy_environment": null,
      "copy_environment_variable": null,
      "request_specific_nodes": "ssd"
    },
    "target": {
      "job_scheduler_type": "torque",
      "name": null
    },
    "application": {
      "app_tag": "skyline-extraction-training",
      "app_type": "python",
      "executable": "python3 peaklens-original-training_new.py",
      "arguments": null,
      "container_runtime": "library://tensorflow_2.2.1-gpu",
      "mpi_ranks": 1,
      "threads": 1,
      "build": null
    },
    "optimisation": {
      "enable_opt_build": true,
      "enable_autotuning": false,
      "app_type": "ai_training",
      "opt_build": {
        "cpu_type": "x86",
        "acc_type": "nvidia"
```

```
      },
      "app_type_hpc": null,
      "app_type_ai_training": {
        "config": {
          "ai_framework": "tensorflow"
        },
        "data": {},
        "ai_framework_tensorflow": {
          "version": "2.2.1",
          "xla": true
        }
      },
      "autotuning": null
    },
    "job_script": "output/skyline-extraction-training_20211026072252.sh",
    "build_script":
"output/skyline-extraction-training_build_20211026072252",
    "job_content": null,
    "build_content": null
  }
}
```


3.2 Clinical UC

In the Clinical use case, the part that we optimise is the Code-Aster Solver component. This component uses finite element methods to compute a solution which shows the strain and stress distribution within the simulated structures, as well as the displacement field for the simulation of two human vertebrae.

The Code-Aster [16] version used in the test is v14.4.0. It is parallelised via MPI. It requires multiple dependencies including numpy, OpenBLAS, SCALAPACK, HDF5, MED, METIS, PARMETIS, TFEL, HOMARD, SCOTCH, MUMPS, PETSc [17]. The baseline container is taken from DockerHub (codeastersolver/codeaster-mpi:14.4.0), which is based on OpenMPI implementation and without any GPU support. Executing the code on a single MPI rank based on the mumps library takes about 970s, which is the reference for the optimisations. As a first step for the optimisation, we built an optimised Singularity container where we specified for the Code-Aster library dependencies the compiler flags to enable specific hardware optimisations, ie. the GCC compiler flags `-mtune=broadwell -march=broadwell -mavx2 -mfma`. This container execution took 783s. Then, as suggested by the application domain experts of the Clinical UC, we tried to use the PETSc library with METIS domain distribution, which further lowers the execution time to 133s. We considered that as the final optimised container (7.3x faster than the baseline). A summary of the execution times is reported in Table 2. For future development, we will investigate the possibility of building a container with GPU support for the PETSc library.

Container configuration	Execution time (seconds)
Baseline container taken from DockerHub (codeastersolver/codeaster-mpi:14.4.0), MUMPS library	970
Optimised Singularity container: compiler specific hardware optimisations, PETSc library with METIS domain distribution	133

Table 2. Execution time of the ClinicalUC Code-Aster baseline and optimised containers.

For the performance model extraction, we built two optimised containers with OpenMPI and MPICH, respectively. We made sure those containers were compatible with the MPI implementations available on the host, so that we could access the network optimised libraries. We extracted the scalability performance model for both containers. Runs were executed on 1, 2, 4, and 6 MPI ranks. Then, we tested the performance estimations for a different number of MPI ranks (10, 14). The results are shown in Figure 3. We found good agreement with the efficiency measured with the application execution. Those models were stored in the MODAK internal database.

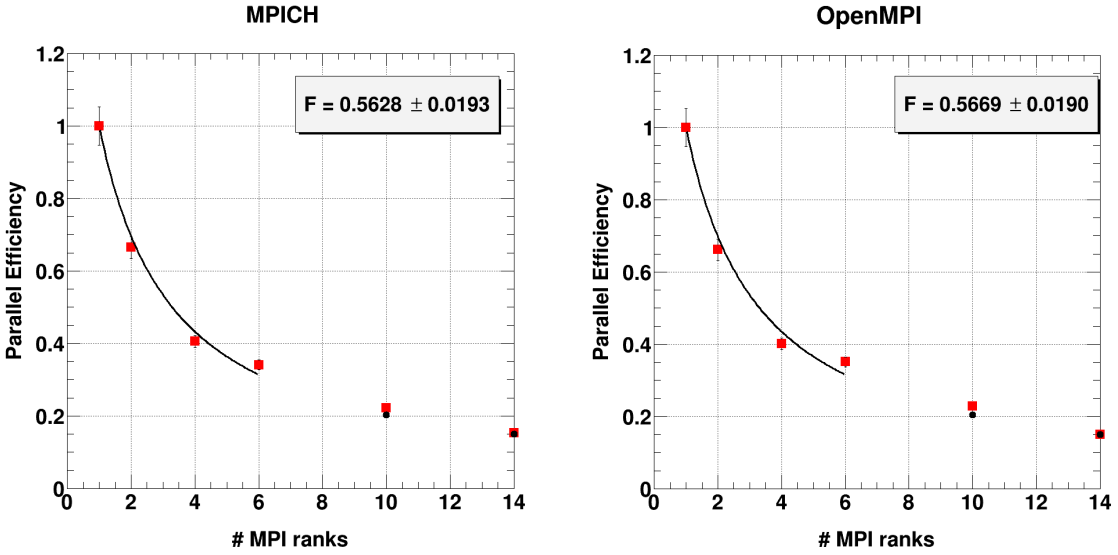

Figure 3. Results on the performance model extraction and verification for the Code-Aster test linked with the two MPI implementations (MPICH on the left plot, OpenMPI on the right plot): the square red points are measured data, the black line is the performance model fit, the circle black points are estimated values by using the performance model.

Finally, the following DSL was used to store and retrieve the optimised container with MPICH support to and from the SODALITE registry, executing it with 4 MPI ranks:

```
{
  "job":{
    "target": {
      "job_scheduler_type": "torque"
    },
    "job_options": {
      "job_name": "solver",
      "node_count": 1,
```

```
      "process_count_per_node": 20
    },
    "application": {
      "app_tag": "solver_clinicalUC",
      "app_type": "hpc",
      "executable": "${ASTER_ROOT}/14.4/bin/aster ",
      "arguments": "${ASTER_ROOT}/14.4/lib/aster/Execution/E_SUPERV.py
-commandes ${ASTER_INPUT} --memjeveux=8192.0 --tpmax=3600",
      "mpi_ranks": 4,
      "threads": 1
    },
    "optimisation": {
      "enable_opt_build": true,
      "app_type":"hpc",
      "opt_build": {
        "cpu_type": "x86"
      },
      "app_type-hpc": {
        "config":{
          "parallelisation":"mpi"
        },
        "parallelisation-mpi": {
          "library": "mpich",
          "version": "3.3.1"
        }
      }
    }
  }
}
```

Similarly, it is possible to consider the OpenMPI container implementation. Optionally, users can use the performance model to set the number of MPI ranks to be used to reach a given parallel efficiency by setting the autoscale DSL configuration.

Based on the above DSL input, MODAK will provide the link to the optimised container and the following submission script for Torque:

```
#PBS -S /bin/bash
## START OF HEADER ##
#PBS -N solver
#PBS -l nodes=1:ppn=20
#PBS -o job.out
#PBS -j oe
## END OF HEADER ##

cd "${PBS_O_WORKDIR}"
export PATH="${PBS_O_WORKDIR}:${PATH}"
```

```
export OMP_NUM_THREADS=1
export ASTER_ROOT=/usr/local/workdir/aster

mpirun -np 4 singularity exec \
  "$SINGULARITY_DIR/code_aster_14.4.0_mpich_broadwell.sif" \
    ${ASTER_ROOT}/14.4/bin/aster \
    ${ASTER_ROOT}/14.4/lib/aster/Execution/E_SUPERV.py \
      -commandes ${ASTER_INPUT} --memjeveux=8192.0 --tpmax=3600
```

The $SINGULARITY_DIR is the path where MODAK assumes the orchestration framework downloads the container to. The $ASTER_INPUT points to the input configuration file to run Code-Aster.

4 Concluding Remarks

Applying optimisations specifically targeting the hardware is crucial to get performance for compute intensive applications. However, it breaks performance portability, such that a user is faced with different optimised implementations of the same application. In this report, we presented a procedure to manage optimised applications within containers to specifically address the goal of achieving performance on specific target hardware. We presented the full release of the application and infrastructure performance models integrated in the SODALITE MODAK component. A performance model for the parallel applications was also presented. We can use this model to predict application performance scaled to execute on multiple nodes. Furthermore, we explained the MODAK DSL to select the optimised application containers and how they can run on batch systems. The work reported is relevant to the T3.3 “Application and Infrastructure Performance Optimisation Modelling”. Results of the optimisations and handling of the optimised containers via MODAK were also presented for DL training and MPI-based applications taken from the SODALITE use cases. We found that the use of optimised containers and setup gives up to a 13x speed-up in performance. Furthermore, we tested the possibility to auto-scale the MPI-based application execution by means of the performance model.

References

1. SODALITE Consortium, Application deployment and dynamic runtime - Intermediate version, Technical deliverable 5.2, 2021.
2. SODALITE Consortium, Prototype of application and infrastructure performance models, Technical deliverable 3.3, 2020.
3. SODALITE Consortium, IaC Management - intermediate version, Technical deliverable 4.2, 2021.
4. Maximilian Höb and Dieter Kranzlmüller, Enabling EASEY deployment of containerized applications for future HPC systems, 2020, arXiv:2004.13373 [cs.DC].
5. Matt Baughman, Ryan Chard, Logan T. Ward, Jason Pitt, Kyle Chard, and Ian T. Foster, Profiling and Predicting Application Performance on the Cloud, 2018, IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC), 2018, pp. 21-30, doi: 10.1109/UCC.2018.00011.
6. Alexandru Calotoiu, Marcin Copik, Torsten Hoefler, Marcus Ritter, Sergei Shudler ,and Felix Wolf, ExtraPeak: Advanced Automatic Performance Modeling for HPC Applications, 2020, In Software for Exascale Computing-SPPEXA 2016-2019. Springer, Cham, 453–482.
7. Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha, A framework for performance modeling and prediction, 2002, In SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing. IEEE, 21–21.
8. Allan Snavely, Xiaofeng Gao, Cynthia Lee, Laura Carrington, Nicole Wolter, Jesus Labarta, Judit Gimenez, and Philip Jones, Performance modeling of HPC applications, 2004, In Advances in Parallel Computing. Vol. 13. Elsevier, 777–784.
9. SODALITE Consortium, SODALITE platform and use cases implementation plan, Technical deliverable 6.1, 2019.
10. Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer, Singularity: Scientific containers for mobility of compute, 2017, PLoS ONE 12(5): e0177459. https://doi.org/10.1371/journal.pone.0177459.
11. Alfred Torrez, Timothy Randles and Reid Priedhorsky, HPC Container Runtimes have Minimal or No Performance Impact, 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), Denver, CO, USA, 2019, pp. 37-42, doi: 10.1109/CANOPIE-HPC49598.2019.00010.
12. Chris Leary and Todd Wang. XLA: TensorFlow, compiled, 2017, TensorFlow Dev Summit.
13. Scott McMillan, Making containers easier with HPC container maker, 2018, In Proceedings of the SIGHPC Systems Professionals Workshop (HPCSYSPROS 2018), Dallas, TX, USA.
14. Gene M. Amdahl, Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities, 1967, AFIPS Conference Proceedings (30): 483–485. doi:10.1145/1465482.1465560.
15. SODALITE Consortium, Initial implementation and evaluation of the SODALITE platform and use cases, Technical deliverable 6.2, 2020.
16. Code-Aster, https://www.code-aster.org, 2021.
17. Code-Aster dependencies description at https://www.code-aster.org/spip.php?article275, 2021.

Appendix A - MODAK DSL Schema Definition

MODAK DSL schema definition with a brief description of the attributes. Mandatory attributes are marked with a *.

job*

The toplevel Job object

job_options*

Options to pass to the queueing system

job_name: string default: job
wall_time_limit: number
node_count: integer > 0
request_gpus: integer > 0
core_count: integer > 0 *core count to use when running this job. Passed to the queueing system.*
process_count_per_node: integer > 0 default: 1
standard_output_file: string default: job.out
standard_error_file: string
combine_stdout_stderr: boolean default: True
request_event_notification: string
email_address: string
copy_environment: boolean
copy_environment_variable: string
request_specific_nodes: string

target

Description of the target where this application is going to run.
If nothing is specified only a Unix shell environment will be assumed.

job_scheduler_type: string *The queuing system to target if the infrastructure name is not specified*
name: string *The target infrastructure*

application*

app_tag: string

app_type: string
this applications type

executable*: string

arguments: string

container_runtime: string
Will be filled/overwritten in the response if an optimised container was found.

mpi_ranks: integer > 0
Number of MPI ranks to use when running the application as part of a job. Passed to mpirun or srun.
default: 1

threads: integer > 0
Number of OpenMP threads to use when running the application as part of a job. Set before mpirun or srun.
default: 1

build
Build information in case on-site rebuilding of the application is desired and possible

src*: string
Source URL for the application

build_command*: string
commands (shell script) to build the application, use {{BUILD_PARALLELISM}} to obtain number of parallel build jobs

build_parallelism: integer > 0
Number of parallel build jobs
default: 1

optimisation

enable_opt_build*: boolean

enable_autotuning: boolean
default: False

app_type*: string
Application type

opt_build

cpu_type*: string
The CPU to optimise the build for

acc_type: string
The accelerator to optimise the build for

app_type-hpc
MPI specific configuration for optimisation

config*

parallelisation*: string
Parallelisation used in this HPC application

data
Application specific data

parallelisation-mpi*

library*: string

version*: string

app_type-ai_training
DL training application

config*

ai_framework*: string
An enumeration.

data
Application specific data

ai_framework-tensorflow*

version*: string

xla*: boolean

autotuning

tuner*: string

input*: string

autoscale

efficiency: value>0
Efficiency value used in the performance model to retrieve the number of MPI ranks.

max_mpi_ranks: integer > 0
Number of maximum MPI ranks to use when running the application as part of a job.
default: same as mpi_ranks

job_script: string
A link to the job script generated for the request

build_script: string
The content of the build script generated for the request

job_content: string
The content of the job script generated for the request

build_content: string
The content of the build script generated for the request