



Software Defined Application Infrastructures management and Engineering

Prototype of application and infrastructure performance models - Final version

D3.3

CRAY

January 2020



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



Deliverable data			
Deliverable	D3.3 Prototype of application and infrastructure performance models - First version		
Authors	Alfio Lazzaro, CRAY Karthee Sivalingam, CRAY Nina Mujkanovic, CRAY Indika Kumara, JADS/UVT Piero Fraternali, POLIMI Rocio Nahime Torres, POLIMI Giovanni Quattrocchi, POLIMI Kamil Tokmakov, USTUTT Ralf Schneider, USTUTT Paul Mundt, ADPT		
Reviewers	Kamil Tokmakov, USTUTT Jesús Gorroñoigoitia, ATOS		
Dissemination level	Public		
History of changes	Name	Change	Date
	Karthee Sivalingam	Initial version created	21.8.2019
	All	First Draft created for review	10.1.2020
	All	First round of reviews completed	20.1.2020
	All	Second Draft created after corrections	24.1.2020
	All	Second round of reviews completed	25.1.2020
	All	Final version created	27.1.2020



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

This deliverable reports on the progress of the SODALITE project in developing a model for application and infrastructure performance. Application performance deployed in an infrastructure can be predicted from the Performance Model of the application and infrastructure. Predicting and improving performance helps not only to reduce application costs but also improve infrastructure utilisation in general. As part of this M12 deliverable, this report describes the prototype model developed and also reports on the baseline performance of use case applications.

Application optimisation can be performed before deployment (static) or at runtime (dynamic). Static application optimisation will enable applications to be optimised based on the optimisation selected by the Application Ops Expert (AOE). Application Optimiser component will map the optimisations to a selected infrastructure target based on the Performance Model and then configure and validate them for the deployment. The application and infrastructure Performance Model for static optimisation is based on a simple statistical model, that is developed from running standard benchmarks and applications on an infrastructure target.

At runtime (dynamically), different available options of deployment will be explored and enacted based on monitoring information and a machine learning model, built by profiling these deployment variants. Also, at runtime, given selected deployment applications can be quickly reconfigured (i.e., vertical scalability) in terms of resources allocated to meet the current demands using a lightweight control theory-based approach.

As part of the deliverable, the initial baseline performance of the demonstrating use case applications are reported. The results from running the benchmarks on an HPC cluster and the resulting application and infrastructure Performance Model are also reported. This model is validated by predicting runtime of a standard HPC application. The Performance Model will be further improved and extended to support heterogenous hardware like GPUs and Cloud based targets before the next deliverable (Full release of application and infrastructure performance models).

As future work, we plan to better integrate the machine learning model that provides deployment refactoring with the control theoretical approach, that provides fine-grained resource allocation, in a hierarchical and comprehensive solution. On the one hand, control theory can request from the machine learning component a new deployment selection when resources are not sufficient to address the current demand. On the other hand, the machine learning solution can change the goals of controllers at runtime when a new deployment model is selected.



Table of Contents

Glossary	7
1 Introduction	8
2 Static Performance Optimisation	8
2.1 Optimisation DSL and recipe	10
2.2 Application Optimiser	11
3 Static Application and Infrastructure performance model	11
3.1 Infrastructure Performance model	12
3.1.1 High-Performance LINPACK benchmark	12
3.1.2 STREAM Benchmark	12
3.1.3 MPI Bandwidth/MPI Latency.....	12
3.1.4 IO Bandwidth	12
3.1.5 PCIe bandwidth.....	12
3.2 Application runtime Model	13
3.3 Application Performance	13
3.3.1 AI Training and Inference	14
3.3.2 Big Data Analytics.....	15
3.3.3 Traditional HPC.....	15
4 Performance Modeling in Refactoring	17
5 Reconfiguration	18
6 Performance results	19
6.1 Performance model on HPC system	20
6.2 Performance model on Cloud test bed	22
6.3 Snow UC	22
6.3.1 Skyline Extraction.....	23
6.3.2 Panorama Alignment.....	25
6.4 Vehicle UC	25
6.4.1. License Plate Recognition	26
6.5 Clinical UC	28
6.5.1 Density Mapping.....	29
6.5.2 Probabilistic Elasticity Mapping	29
6.5.3 Solver	29
7 Concluding Remarks	30
References	30
Appendix 1. HPC Benchmarks Singularity Definition File	32



Table of Figures

Figure 1: Performance Optimisation for applications deployed in a Heterogeneous infrastructure.. 9

Figure 2: Architecture of Optimisation DSL and Optimisation Recipe 10

Figure 3: Architecture of Static Application Optimiser 11

Figure 4: Schema of the Virtual Clinical Trial use case pipeline 16

Figure 5: Control Schema 19

Figure 6: HPC Benchmarks speed-up values on Piz Daint Supercomputer. Speed-up are evaluated with respect to 1 core performance, except for MPI BW where the result on 2 cores is considered. Linear fittings ($y=p_0+p_1 x$) are superimposed for each plot. Note that for the MPI BW we use two linear polynomials to describe the low and high core count behaviours of the speed-up, respectively 21

Figure 7: HPCG Benchmark speed-up values and combined fit..... 22

Figure 8: Initial version of the pipeline as a sub-group of the components of the original one 22

Figure 9: GPU profile of a single epoch of Skyline extraction training 24

Figure 10: Schema of the license plate detection and detection model training pipelines (Vehicle IoT use case) 26

Figure 11: License Plate identification and extraction 26

Figure 12: Schema of the Virtual Clinical Trial use case pipeline 28



Table of Tables

Table 1: Structure of Observation Records	18
Table 2: Skyline extraction training wallclock time profile	23
Table 3: Web component response time.....	24
Table 4: Panorama Alignment response time.....	25
Table 5: License plate recognition times across file sizes	27
Table 6: License Plate Detection Models, data requirements, and current training times	28
Table 7: Execution (wallclock) time over the number of MPI ranks for Probabilistic Mapping	29



Glossary

Acronym	Explanation
AI	Artificial Intelligence
API	Application Programming Interface
ASA	Average Skyline Accuracy
CNN	Convolution Neural Network
CPU	Central Processing Unit
CT	Computer Tomography
DEM	Digital Elevation Model
DNN	Dense Neural Network
DSL	Domain Specific Language
ETL	Extract Transform Load
FPGA	Field- Programmable Gate Array
GPS	Global Positioning System
GPU	Graphic Processing Unit
HPC	High Performance Computing
HPCC	High Performance Computing Challenge
HPCG	High Performance Conjugate Gradient
HPL	High Performance Linpack
IaC	Infrastructure as Code
IO	Input Output
IoT	Internet of Things
ML	Machine Learning
MPI	Message Passing Interface
MT	Model Tree
OCR	Optical Character Recognition
ONNX	Open Neural Network eXchange
PCIe	Peripheral Component Interconnect Express
QoS	Quality of Service
SIMD	Single Instruction Multiple Data
SLA	Service Level Agreement
STL	Standard Template Library
VM	Virtual Machine



1 Introduction

This report describes a prototype for application and infrastructure performance modelling. The prototype aims to take a performance-centric view of the applications and infrastructure and will model them to enable performance decisions to be made. Performance optimisation will be performed before deployment (*static* optimisations) and also after deployment based on refactoring or reconfiguration (*dynamic* optimisations).

SODALITE will provide a model for HPC infrastructure using standard benchmarks and applications based on the features that influence performance. These performance models for infrastructure and application will be used in the *Application Optimiser* (as defined in deliverable D2.1 [1]) component to statically optimise the application and deployment. At runtime, a Machine Learning (ML) based application model will be used for the entire application workflow to dynamically optimise/refactor its deployment model based on performance metrics and monitoring information. We will refer to the static model as *Performance Model* and the latter for refactoring as *ML-based Predictive Model* for clarity. Also, we will explore a model based on control theory for reconfiguration during run time.

For the application Performance Model, the applications will be classified broadly as AI, Big Data and Traditional HPC and performance attributes for classes of application will be defined separately. Applications in demonstrating uses cases will be used to test and validate the model. Further, the hardware infrastructure will be modelled based on HPC benchmarks like HPL₁ and STREAM₂. The HPL benchmark is used to rank the HPC machines in the Top 500₃ list. With the application and infrastructure model, the SODALITE system will be able to make decisions on optimising an application deployment. The model will be further improved based on metrics measured at application run time.

2 Static Performance Optimisation

The performance of an application that is deployed using Infrastructure as Code (IaC) on heterogeneous infrastructure targets is paramount. The performance of an application can be determined by modelling the application and infrastructure. The application modelling extracts the application parameters that influence the performance of an application and the infrastructure modelling will help us extract the performance characteristics of the infrastructure target such as peak performance and memory bandwidth. With the understanding of the Application and Infrastructure, performance optimisation maps the optimal application parameters to the infrastructure target. The application parameters can also be autotuned during run time.

Figure 1 shows the application optimisation requirements in a heterogeneous target. The target infrastructure can have multiple combinations of CPU, GPU or FPGA with different memory hierarchy. The target's File System and Network are also diverse and are usually shared by multiple nodes within the infrastructure. With different schedulers for HPC and Cloud, orchestrating a workflow optimally becomes complex. This diversity in HPC and Cloud based infrastructure targets results in a complex problem of application optimisation. Optimising all applications on a diverse target is out of scope for this project and instead the project will focus on three different application types like AI Training & Inference, Big Data Analytics and Traditional HPC applications like Solver.

¹ The Linpack Benchmark <https://www.top500.org/project/linpack/>

² Sustainable Memory Bandwidth in High Performance Computers <https://www.cs.virginia.edu/stream/>

³ Top 500 supercomputing sites <https://www.top500.org/lists/2019/11/>

This broad spectrum represents a majority of HPC applications and also represents the demonstrating use cases in SODALITE.

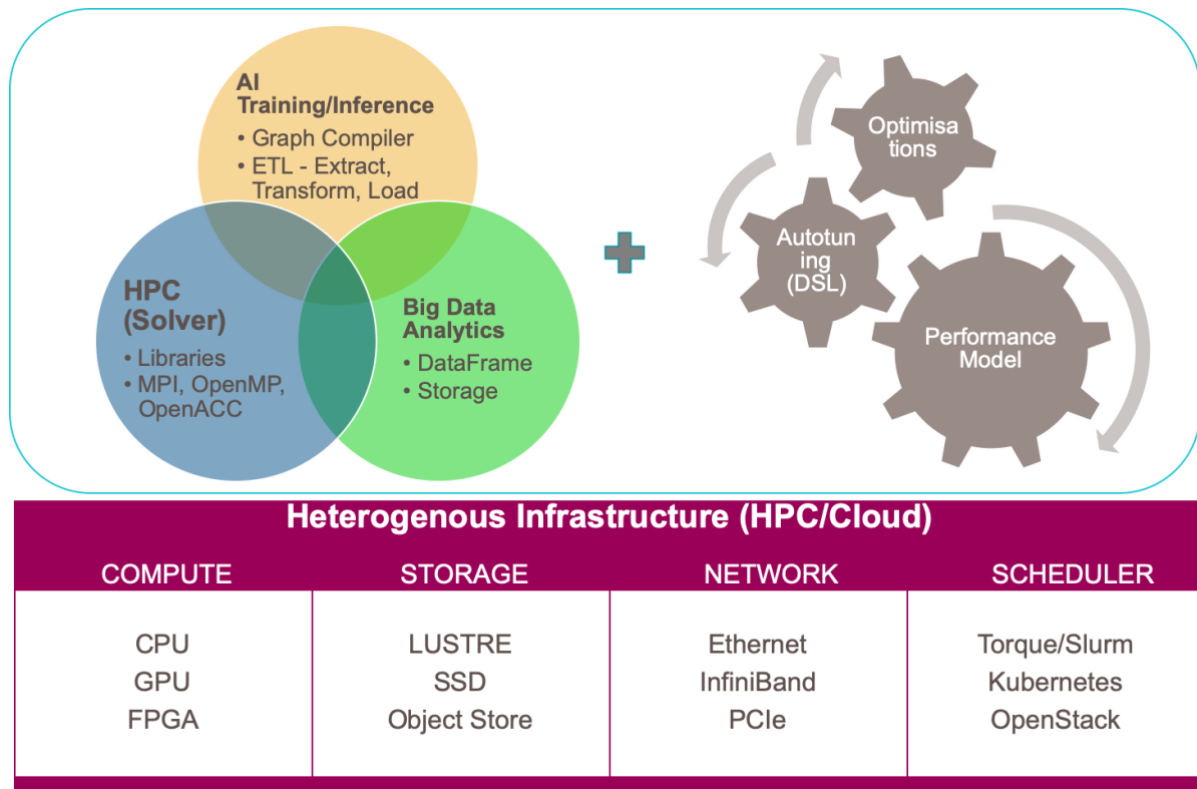


Figure 1: Performance Optimisation for applications deployed in a Heterogeneous infrastructure

The list below shows the mapping of actual applications to application types:

1. AI Training and Inference - Pixelwise Mountain Skyline detection CNN [2] training and PolimiDL[3] inference
2. Big Data Analytics - HiBench Suite⁴ from Intel
3. Traditional HPC - Code Asters based Solver for in-silico clinical trials.

For these applications, the following optimisations will be enabled:

1. Autotuning – Application parameters can be autotuned for performance improvement. We will use the DSL-based autotuner developed as part of the CRESTA EU⁶ project.
2. Multi Architecture support. This will enable applications to efficiently use multiple hardware like CPUs, GPUs or FPGAs. The application will be built for a particular target architecture or use specific target libraries.
3. Specific optimisations for application groups namely, AI training/inference, HPC data Analytics and Traditional HPC application (Solver):
 - a. AI training will be optimised with target-specific libraries and Graph compilers⁷. The Extract, Transform, Load (ETL)⁸ pipeline will be optimised by improving data movement by prefetching, caching and reuse of data.

⁴ <https://github.com/intel-hadoop/HiBench>

⁵ <https://www.code-aster.org>

⁶ <http://www.cresta-project.eu>

⁷ Graph compilers for AI training and inference - <https://www.sodalite.eu/content/graph-compilers-ai-training-and-inference>

⁸ ETL <https://docs.microsoft.com/en-us/azure/architecture/data-guide/relational-data/etl>

- b. Big Data Analytics applications like Apache Spark⁹, DASK¹⁰, NVIDIA's cuDataFrame¹¹ based on DataFrame API will be optimised for target hardware and storage.
 - c. Solver (MPI) application will be optimised by using efficient solver libraries like PETSC¹² and MUMPS¹³ for different targets. HPC Standards MPI¹⁴, OpenMP¹⁵ and OpenACC¹⁶ will be enabled to support performance scaling and portability.
4. Applications will be delivered in an optimised container like Docker¹⁷ or Singularity¹⁸ to ensure portability across different targets.

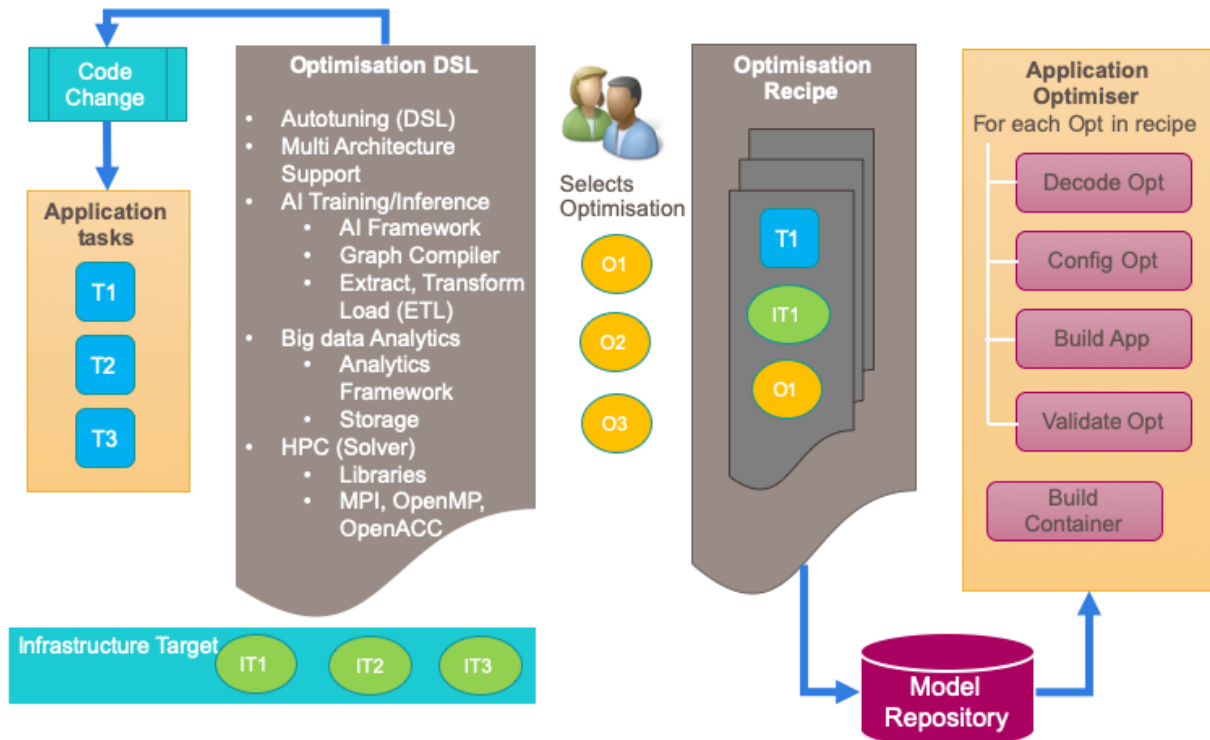


Figure 2: Architecture of Optimisation DSL and Optimisation Recipe

Static application optimisation is achieved in SODALITE using the components described in the following two subsections.

2.1 Optimisation DSL and recipe

Figure 2 shows how the *Optimisation recipe* is built from the *Optimisation DSL*. This DSL contains all optimisation options for a particular application type along with autotuning and multi-architecture

⁹ Apache Spark - Unified Analytics Engine for Big Data <https://spark.apache.org/>

¹⁰ DASK- Scalable Analytics in Python <https://dask.org/>

¹¹ Open GPU Data Science | RAPIDS (<https://github.com/rapidsai/cudf>)

¹² PETSC - Portable Extensible Toolkit for Scientific computation <https://www.mcs.anl.gov/petsc/>

¹³ MUMPS: a parallel sparse direct solver <http://mumps.enseeiht.fr>

¹⁴ Message Passing Interface <https://www.mpi-forum.org/docs/>

¹⁵ OpenMP <https://www.openmp.org/>

¹⁶ OpenAcc <https://www.openacc.org/>

¹⁷ Docker <https://www.docker.com>

¹⁸ Singularity <https://singularity.lbl.gov>

support choices. For example, for AI training workloads, the *Application Ops Expert* (AOE) can select the AI framework to use along with optimisations like Graph compiler and ETL options. This mapping of selected optimisations along with the Infrastructure targets and the Application tasks will be stored as Optimisation Recipe in the *IaC Model Repository*. The *Application Optimiser* component will retrieve the recipe from the repository and will decode, configure, build and validate the optimisations.

2.2 Application Optimiser

The Static *Application Optimiser* optimises application for a given target platform based on the optimisation options selected. Figure 3 shows the architecture of the *Application Optimiser* and its dependencies. The optimiser acts on the *Optimisation Recipe*, which contains the mapping of optimisations to application tasks and Infrastructure targets. This recipe will be retrieved from the *IaC Model repository* which will also host the application and infrastructure *Performance Model*. For each of the optimisations in the *Optimisation Recipe*, the optimisations will be configured and validated. For this, *Application Optimiser* requires the application code to be written in a standard High-level API along with the application inputs and configuration. This enables the optimiser to make performance decisions based on the available target. The optimiser will use the prebuilt optimised containers from the *Image Registry* and modify them to build an optimised container for the application deployment. The Application optimiser will also make changes to runtime/deployment and job scripts for submission to HPC resources.

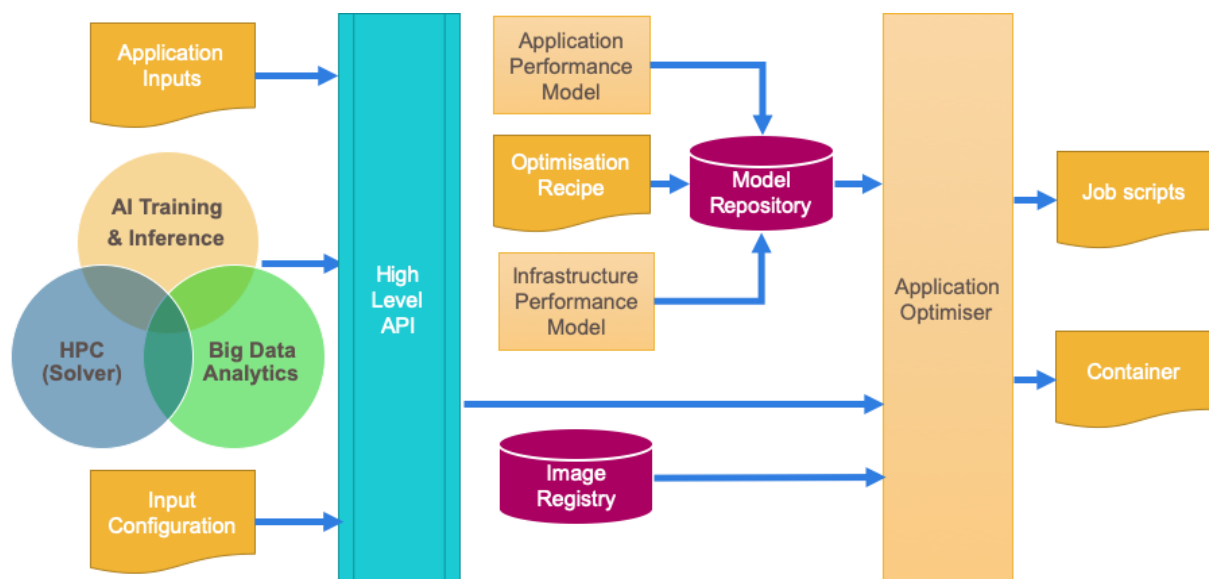


Figure 3: Architecture of Static Application Optimiser

3 Static Application and Infrastructure Performance model

The requirements for the application model will be derived from the demonstrating use cases, whereas the infrastructure model will reflect the test beds. Application/Infrastructure Performance Models help us in predicting the runtime of an application on an appropriate hardware infrastructure. There are multiple approaches taken in the HPC community to develop a model. This may be based on profiling an application, micro benchmarking primitive components of the application or simulation of application changes and analytical modelling. We will use a combination of benchmarking to model infrastructure and then an analytical modelling to model application runtime. We will aim to have a simple model that will help us identify the limiting factors



for performance quickly and understand how the performance changes with application and infrastructure.

3.1 Infrastructure Performance model

A HPC resource can be modelled using the measured performance of benchmarks that capture particular aspects of the resource, for example the bandwidth of the memory. Good examples which we plan to use are HPC benchmarks like HPL, STREAM, MPI Bandwidth/latency, IO Bandwidth. This information along with the number of CPUs/GPUs will characterise an HPC cluster.

3.1.1 High-Performance LINPACK benchmark

The *High-Performance LINPACK* (HPL) benchmark is the standard and popular yardstick used to rank the supercomputers. It involves the solution of a dense system of linear equations and the problem can be scaled to the full size of the machine. This number reflects the *performance of a dedicated system for solving a dense system of linear equations* and is a good indication of the peak performance of the floating-point rate of execution of a machine.

3.1.2 STREAM Benchmark

The *STREAM* benchmark measures the sustainable memory bandwidth in HPC. Historically, computer cores have become faster whereas the memory bandwidth progress is limited. The peak performance that can be achieved for many applications, is limited by the memory bandwidth. The memory bandwidth is measured using simple vector kernels like copy, add, scale and triad. We use triad performance in our model.

3.1.3 MPI Bandwidth/MPI Latency

Network communication is a fundamental part of most parallel applications and the network bandwidth and latency drive the scaling of parallel applications. We will use the *Effective Bandwidth*¹⁹ (b_{eff}) as a single measure to represent the communication network. This benchmark takes into account short and long messages used in real applications to derive this metric. It also uses several communication patterns for the MPI communications (Ping-Pong, Naturally Ordered-Ring, Randomly Ordered-Ring). We use the Randomly Ordered-Ring pattern in our model.

3.1.4 IO Bandwidth

Most HPC applications read and write data to a persistent storage like disks. The speedup of parallel applications at scale depends hugely on the IO bandwidth that can be achieved. We will use *Effective IO bandwidth*²⁰ ($b_{\text{eff_io}}$) as a single measure of IO bandwidth. $b_{\text{eff_io}}$ averages the bandwidth across several access patterns and buffer lengths.

3.1.5 PCIe bandwidth

In heterogeneous architectures, the accelerator is attached to the host CPU via a PCIe. Applications usually transfer data to the accelerator and receive processed data back to the host. This communication impacts the performance of the entire application. We will use the measured PCIe bandwidth as a metric for modelling performance of heterogeneous architectures. For GPUs, the *Babel Stream*²¹ and *CUDA accelerated Linpack benchmarks*²² will be used to help model it.

¹⁹ https://fs.hlrs.de/projects/par/mpi/b_eff/

²⁰ https://fs.hlrs.de/projects/par/mpi/b_eff_io/

²¹ <https://github.com/UoB-HPC/BabelStream>

²² <https://developer.nvidia.com/rdp/assets/cuda-accelerated-linpack-linux64>



3.2 Application runtime Model

An application performance model is used to predict application performance on specific hardware and to determine how this performance scales. Many approaches have been used in the past that combine hardware information along with application profiling data to study their performance. We will use a simple statistical model to predict application performance. This model will be further improved. We will use the application wall clock time as the metric that will be optimised, unless specified otherwise.

An application run time t on a hardware infrastructure can be represented as follows:

$$t = t_{comp} + t_{comm} + t_{mem} + t_{io},$$

where t_{comp} is the time spent on computing, t_{comm} is the communication time, t_{mem} is the time spent in loading the data to memory, and t_{io} is the time spent doing input/output. We can model the application time based on the HPC benchmarks described in the previous section by considering the following associations:

- t_{comp} based on HPL (*HPL*),
- t_{comm} based on b_eff (only bandwidth, *MPI BW*),
- t_{mem} based on STREAM (only bandwidth, *MEM BW*),
- t_{io} based on b_eff_io (*IO BW*).

In the case of heterogeneous architectures, additional parameters for PCIe bandwidth will be added to account for the data communication. We can generalize the equation by considering the dimensionless speed-up factors S , i.e. the ratio of the run time when running on a given number of cores with respect to a reference run time (usually obtained with a single core execution):

$$S = f_{HPL}S_{HPL} + f_{MPI\ BW}S_{MPI\ BW} + f_{MEM\ BW}S_{MEM\ BW} + (1 - f_{HPL} - f_{MPI\ BW} - f_{MEM\ BW})S_{IO\ BW}, (1)$$

where we explicitly include the speed-up factors for each benchmark that are combined with a set of *fractions* f such that by construction the sum of the fractions of all components is equal to 1. Therefore, the speed-ups are measured across different number of cores and hardware configurations and modelled using least square fitting in order to obtain estimated speed-up values \hat{S} . The value of the fractions should indicate which part of the hardware drives or hinders performance. For example, a higher $f_{MPI\ BW}$ denotes high dependency on the communication bandwidth can application can achieve.

Finally, we can estimate the run time of the execution at a given number of cores c

$$t_c = t_r \hat{S}_c \hat{S}_1 / \hat{S}_r, (2)$$

where t_r is the run time measured when executing the application with r cores. An example of application of the model is given in section 6.1.

3.3 Application Performance

There exists a wide range of traditional HPC applications, with different requirements which change rapidly with the increasing AI and Big Data workloads. Our model models the performance of an particular application configuration, and this model will fail if its configuration is changed. Application configuration space is huge, and all the parameters in this space cannot be modelled. For example, there are hundreds of inputs for a DNN training and performance will change with changes to input. Of these hundreds of parameters, there are a few that will heavily influence the performance and scaling of an application.



We will work with the application domain experts to identify these parameters and model them. The section below describes the three different application types that map to demonstrating use cases and also show the parameters that will be modelled.

3.3.1 AI Training and Inference

Skyline Extraction

The *skyline extraction* for Snow use case will drive the AI Training and Inference application model. Fast identification of objects is at the core of Outdoor Augmented reality, and this demands not only high recognition accuracy, but also real time performance with energy and memory efficiency. The *Pixelwise Mountain Skyline detection CNN* [2] network forms the core of *PeakLens* mobile app. This CNN network uses 8940 manually annotated mountain images for training, with images obtained from Flickr and 2000 publicly available webcams. The network evaluates the probability of a pixel belonging to a skyline, which is then post processed to extract the actual skyline.

The CNN model is adapted from the LeNet model, and these pixelwise CNNs have been found successful in edge extraction problems. The model contains 8 layers with a combination of 4 conv, 2 max pool, 1 relu, and a softmax layer. The input to the model is the patches generated from the images, consisting of 300 patches per image. There are usually 100 positive patches - skyline edge detected - and 200 negative patches - no skyline edge detected - per image. The output is a binary classification problem of detecting a skyline or not. The model is trained using the *Caffe framework*²³, and the model output is evaluated using *Average Skyline Accuracy (ASA)*. ASA measures the fraction of image columns where the CNN skyline matches the ground truth. The model is embedded in the PeakLens app, where the initial peak position is calculated using the DEM, GPS, and sensors. After the initial detection, the Skyline extraction is used to correct substantial errors.

PolimiDL Inference

PolimiDL [3] is an application for accelerating DL inference on an embedded system with no impact on accuracy. This is competitive with TFLite²⁴. PolimiDL is a C++ application that supports multi-threading (STL concurrency) and SIMD optimisations with no specific platform support. Optimisations are performed at:

- *Generation-time*: Layer fusion (Depthwise Separable Convolution), precomputing weights, data layout changes to match operations
- *Compile-time*: per layer optimisation like loop unrolling, compilation to shared object, reduce cache miss, tick-tock pipe lining with memory as a dependency
- *Initialisation-time*: Fuse memory buffers in allocation, prioritise memory efficiency over compute speed
- *Configuration time*: Schedule tasks based on profiling and memory requirement
- *Run-time*: Dynamic scheduling to tasks to thread pool

PolimiDL has support only for TensorFlow²⁵ format and does not support GPUs. Evaluation of PolimiDL on different mobile devices show competitive results compared to TFLite.

Most inference optimisations use compression techniques like quantisation (reduce precision), pruning (remove redundant connections), and knowledge distillation (compress and transfer knowledge). Architectures like SqueezeNet also optimise by using small kernels with lesser

²³ <https://caffe2.ai/docs/mobile-integration.html>

²⁴ TensorFlow lite <https://www.tensorflow.org/lite>

²⁵ TensorFlow <https://www.tensorflow.org/>



parameters. Google has developed efficient models for mobile phones called MobileNet, which are based on depthwise separable convolutions and also make multiple trade-offs for performance.

Android Neural Network API and *Metal API* (Apple) also provide access to hardware acceleration in mobile phones. The training for such networks is performed offline using mainstream frameworks on HPC servers. The models are then converted to a mobile framework format before optimising for inference. ONNX²⁶ provides a standard for model definition that enables porting across different frameworks. The optimisations are focussed not only on accuracy and execution time, but also Memory and Energy consumption.

Modelling AI Training and Inference

AI training depends on many parameters and is known particularly for hyper parameter tuning, which helps in optimising the training of Deep Neural network (DNN) models. DNN models are characterised by the number of layers, batch size, and number of parameters in general. We will use these three parameters to study how the application performance model reacts to changes to them. The execution of the training model on the GPU is only part of the overall workload, with the *Extract, Transform, Load* (ETL) part of the workflow also incurring large expenses for Big Data workloads. ETL is usually done by the CPU, after which the data is fed to the GPU. Optimising and tuning the parameters that determine data reuse, prefetching, and caching will be explored.

3.3.2 Big Data Analytics

The HiBench Suite²⁷ from Intel will be used as a driving use case for Big Data Analytics. This benchmark contains nineteen different workloads that represent the spectrum of different Big data applications. These benchmarks are supported on multiple Big data frameworks like Spark, Hadoop²⁸, Flink²⁹, Kafka,³⁰ and Storm³¹. Pandas³² is another famous framework used widely. Pandas, Apache Spark, and NVIDIA support dataframe like API for relational data analytics.

Big Data Analytics performance depends mainly on efficient data movement and reuse. Tools like Apache Spark gained popularity by keeping the data in memory instead of writing to files. Optimisation and tuning of parameters that determine memory and storage usage, along with data reuse, prefetching, and caching will be explored. For the performance studies, Big data benchmark will be used to measure and study the response time for a handful of relational queries.

3.3.3 Traditional HPC

Virtual Clinical Trials

The use case of in-silico clinical studies for spine surgery aims at the development of a simulation process chain that supports clinical in-silico studies of bone-implant systems in neurosurgery, orthopaedics and osteosynthesis.

²⁶ <https://onnx.ai>

²⁷ <https://github.com/intel-hadoop/HiBench>

²⁸ Apache Hadoop - <https://hadoop.apache.org>

²⁹ Apache Flink — Stateful Computations over Data Streams <https://flink.apache.org>

³⁰ Apache Kafka- a distributed streaming platform <https://kafka.apache.org>

³¹ Apache Storm <https://storm.apache.org>

³² Pandas- a data analysis library - <https://pandas.pydata.org>

A simulation process chain is used to analyse and evaluate screw-rod fixation systems for instrumented mono- and bisegmental fusion of the lumbar spine. An essential characteristic is that the data involved are not only passed on from one component to another, but that various recombination and multiple uses of the data in different components take place. The layout of the complete process pipeline with its data flow is shown in Figure 4.

The process starts with *clinical CT-image data* which in the first place have to be treated by *image Processing and Filtering* techniques to enhance their quality. This has to be done mainly due to that fact that classically clinical imaging data are meant to be analysed by the human eye and not by image processing or numerical algorithms.

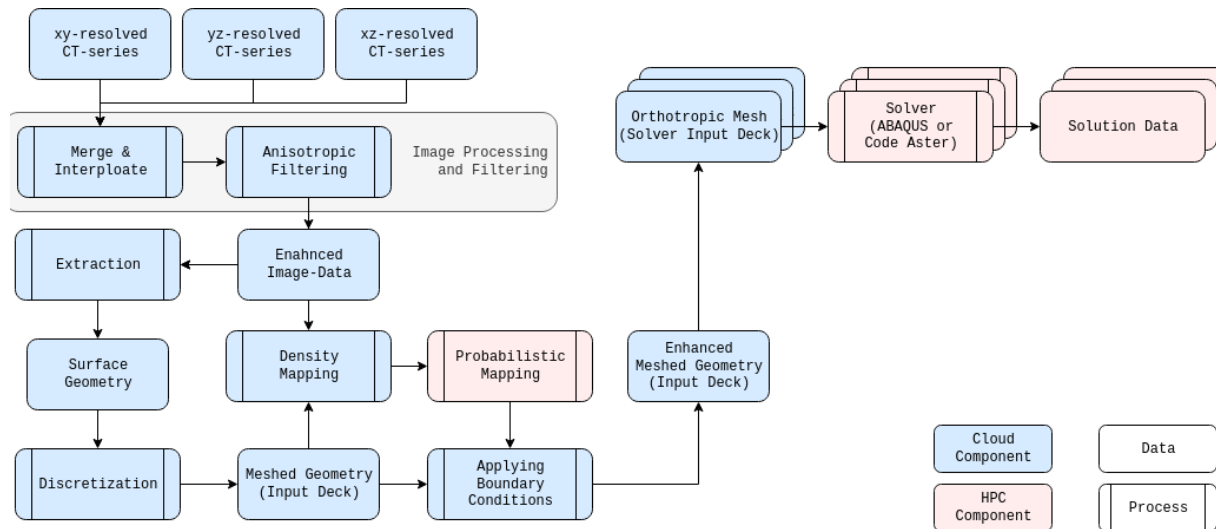


Figure 4: Schema of the Virtual Clinical Trial use case pipeline

After image processing in a *Geometry Extraction* step, the surface geometry of the bone structures to be simulated is extracted from the *Enhanced Image-Data*. The surface geometry is then passed on to a *Discretization* component which reconstructs the triangularization of the surface to ensure proper element quality and then applies a volume meshing algorithm to produce a three-dimensional simulation mesh of the bone structures. This mesh has to be further enhanced by additional soft tissue structures like the intervertebral discs, tendons and other cartilage structures. This is done based on templates in the *Applying Boundary Conditions* component. Additionally, supports and boundary conditions are applied in this step.

Another essential part of the simulation model is the usage of an inhomogeneous, patient specific material model. The basis for this model is a mapping between the bone density given by the voxel mesh of the clinical imaging data and the three-dimensional unstructured simulation mesh of the bone structures. This mapping is done in the *Density Mapping* component. After the mapping, each element in the simulation mesh holds a density value. Since one aspect of the simulation process chain is the quantification of the uncertainty that arises once the mapped density data are used to generate orthotropic material data by parametrized transfer functions, the mapped density data are passed on to the *Probabilistic Mapping* component. In this component a probabilistic programming approach is used to determine the boundaries for the 95% highest density interval, as well as the mode. This means by the data produced in this step, the deterministic model can be transformed into three models representing the uncertainty of the material distribution.



The integration of the results of the *Probabilistic Mapping* component into the simulation model is also done by the *Applying Boundary Conditions* component, which finally produces three *Enhanced Meshed Geometries*. These so-called input decks are processed by the *Solver* component which is based on *Code_Aster* a Finite-Element-Method simulation software package. The resulting displacements, stress and strain fields of the bone structures are placed in three separate result files that can be processed by visualization or data analytics methodologies.

Modelling Traditional HPC

Traditional HPC applications are built on top of standards like MPI for Message passing, OpenMP for thread-based parallelism, and OpenACC for GPU acceleration. Applications can also use libraries that parallelise and run on different hardware. Performance depends on how the application is configured to run. The number of MPI ranks, CPU threads, GPU threads, and possibly numerous application parameters may be relevant. For Solvers, performance will be modelled based on the size of the matrices and the solver methods in general.

4 Performance Modeling in Refactoring

In this section, we present the preliminary results of the performance modeling for deployment refactoring. In the first prototype of the performance models, we focus only on the cloud applications.

The individual components of an application can be deployed in different ways using different resources (e.g., a small VM and a large VM) and deployment patterns (single node, cluster with load balancer, with or without cache, with or without firewall). We call these deployment possibilities (application component) deployment options. A valid selection of deployment options results in a valid deployment model variant for the application. The deployment refactoring requires a model that can estimate the impacts of a given deployment option selection on the QoS (Quality of Service) metrics such as latency and cost, under different contexts such as different workloads. We build a machine learning based predictive model for this purpose. At design time, we profile the deployment variants to collect the data required to build the machine learning model. At run time, we use the monitoring of the running application to collect the data and to update the learned model as necessary. The predictive model enables the deployment refactorer to predict any potential violations of the application goals, and consequently to find alternative deployment model variants.

We assume that the *App Ops Expert* has modeled the allowed set of deployment model variants in terms of (application component) deployment options and their dependencies. The *App Ops Expert* can use an existing variability modeling technique and tool, such as feature modeling [4].

At design time, we deploy deployment model variants and profile them to collect the performance metrics. Our initial focus is on the execution time/latency. The nature of the collected data is illustrated in Table 1. Each deployment model variant consists of a subset of the application component deployment options (D1, D2, ...). In the table, 0 means that the corresponding deployment option is not used by the deployment model variant, whereas 1 means the opposite. The performance metrics are obtained for different workloads. The collected data are used for learning the functions that estimate the impacts (increase or decrease of the metric) of the deployment options on performance metrics. The collected data are also used to test the accuracy of the learned functions.



Application Component Deployment Options						Performance Metrics		
D1	D2	D3	D4	D5	..	PM1	PM2	..
0	1	1	1	0	..	x1	x2	..
1	0	0	1	1	..	y1	y2	..
1	1	1	0	0	...	z1	z2	..
..

Table 1: Structure of Observation Records

A given deployment model variant can behave differently under different workloads (generally, in different contexts). In that case, the result of learning would be a set of equations that estimate the impact of deployment option selection in different contexts. For example, the equations that can calculate the impacts of deployment option selection on the performance metric PM1 under three different workloads would be:

$$PM1 = \begin{cases} a1D1 - b1D2 + c1D3 & w \leq x1 \\ a2D1 - b2D2 + c2D3 & x1 < w \leq x2 \\ a3D1 + b3D3 + c3D1D3 & w > x2 \end{cases} \quad (3)$$

Each deployment option is assigned a coefficient that is effective only when the deployment option is selected (i.e., it is set to “1”). For example, the expected value of PM1 under the workload range ($x1 < w \leq x2$) for a deployment selection, where only D1 and D2 are selected can be calculated as follows:

$$PM1 = a2 * 1 - b2 * 1 + c2 * 0 = a2 - b2$$

Our approach is inspired by the works on data-driven approach to self-adaptive systems [5-6] and data-driven approaches to predicting the performance of cloud applications [7-8]. While the approach is not tied to a particular learning algorithm, for the first prototype, we are experimenting with M5 model tree algorithm (as in [5]) and multiple linear regression model.

5 Reconfiguration

Elasticity is the capability of a system to adapt to workload changes by provisioning or de-provisioning resources automatically such that at each point in time the available resources match the current demand as closely as possible [9].

Several elastic systems for cloud computing have been proposed both in academia and industry. Traditionally, solutions usually lack of speed in the adaptation process that relies on slow-to-boot virtual machines as resources that need to scale, or because they optimize resource allocation by exploiting heavy-weighted combinatorial formulations. Moreover, those systems use a coarse granularity of allocation that heavily depends on the pre-configured virtual machines available from the cloud provider [10, 11, 12].

For these reasons, we present a solution that exploits two lightweight enablers: containers and control-theory. Containers can be reconfigured in milliseconds (vs minutes of VMs) [13], while control-theoretical planners can compute next allocation in constant time.

Given initial, static, allocations, a set of SLAs and applications deployed within containers, resources must be re-allocated at runtime in order to meet unforeseen peaks of workload, divergences from the expected performance and failures. For this reason, we equipped each container deployed in the nodes with a dedicated control-theoretical planner that produces a continuous, fast and fast-grained re-configuration of it by changing its allocated resources (e.g., CPU-time). This methodology was already applied to microservices [14] and big-data applications [15]. Herein, we describe how we extended our work to support machine-learning applications (in inference mode) that can exploit not only CPU cores but also GPUs (e.g., TensorFlow applications). The current implementation of this approach works within the Cloud; in the future we will investigate the possibility to apply the same in HPC.

Since the GPU is the faster and more optimized resource for ML applications, in our approach, firstly we let that the workload saturates the GPU and then, if needed, a CPU allocation is enacted in the case of possible SLA violations. Assuming that multiple deployed applications can share the same resources (CPUs and GPUs) we adopted the following control schema.

GPUs are controlled through an event-based heuristic that employs a priority queue to dispatch requests from chosen applications to the available GPUs. Every time a GPUs completes the processing of a request, it pops a new one from the queue. The prioritization mechanism could be configured by the users; by default, applications with more requests in the queue are preferred among the others.

We define the expected speed of answer v_{SLA} of the system as the inverse of the SLA defined as a constraint over the response time. At every control step the speed of answer provided by the GPUs v_{GPU} is computed. As reported in the control schema in Figure 5, if the difference between v_{SLA} and v_{GPU} is greater than 0, this difference becomes the set point v_{oCPU} of the CPU controller C_{CPU} .

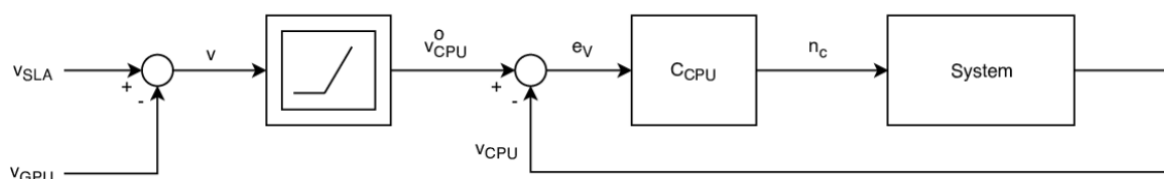


Figure 5: Control Schema

C_{CPU} is implemented as a PID controller [16] with the goal of reducing the error e_v computed as the difference between v_{oCPU} and the actual CPU speed v_{CPU} . To do that it computes the next CPU allocation n_c for the target container which is enacted using Docker commands.

6 Performance results

This section illustrates how the prototype Infrastructure and Application Performance Model works in an HPC cluster and also reports on the measured baseline performance of use case applications.



6.1 Performance model on HPC system

We will use the concepts described in section 3.1 to build a performance model. The HPC benchmarks used to characterize the infrastructure performance, are included in a Singularity container. The use of a Singularity container will enable portability across HPC and Cloud platforms with almost no loss in performance. The HPL, STREAM and b_eff benchmark were taken from the HPCC benchmark suite (v1.5)³³, while b_eff_io is built from the official source. The container image is based on Debian Buster, where we also install MPICH v3.3.1. The exact definition can be found in [Appendix 1](#).

The first iteration of the HPC testbed hosted in USTUTT is intended for experimentation with the First Prototype of SODALITE platform and running initial version of the demonstrating use cases, and not intended for performance. The testbed has a small scale (only two nodes are available), as described in the deliverable D6.2. Hence the initial benchmarks run on the testbed resulted in the limited Performance Model, which could not predict the performance scaling of an HPC applications accurately. Therefore, we used the Cray XC40 Piz Daint supercomputer at the Swiss National Supercomputing Centre (CSCS) to build and validate the performance model. Each node of the system is equipped with a CPU Intel Xeon E5-2695 v4 @ 2.10GHz (2 x 18 cores, 64GB DRAM). The system features a full Cray's Aries network and a LUSTRE filesystem. We use the following versions on Piz Daint :

- singularity/3.4.2
- cray-mpich-abi/7.7.10³⁴

where cray-mpich-abi is the cray build of the mpich-abi compatibility library.

Figure 6 shows the HPC Benchmarks speed-up values on Piz Daint Supercomputer. The speed-up values for each HPC benchmark are evaluated with respect to 1 core performance, except for MPI BW where the result on 2 cores is considered as reference (clearly, we need at least two cores for the MPI communication). We use a single MPI rank per core. We also report the results of the linear fittings $y = p_0 + p_1x$, superimposed for each plot. Note that for the MPI BW we use two linear polynomials to describe the low and high core count behaviours of the speed-up, respectively. For low core-counts, the communication performance is dominated by the fast intra-node communications, therefore it gets worse when we increase the number of involved nodes. This effect is compensated by the reduction of the message size exchanged between the MPI processors; therefore, the overall performance of the communication becomes better with high core-counts.

The official suggested values in *HPCC benchmark suite* are used to run the benchmarks. Then, we use these fits to model the *HPCG benchmark*³⁵ application. HPCG is a high-performance application that implements preconditioned conjugate gradient (PCG) method with a local symmetric Gauss-Seidel preconditioner. This application represents most common computational kernels and data access patterns found in scientific HPC applications. The code is written in C++ and parallelised using MPI and OpenMP.

Figure 7 shows the speed-up of the HPCG application and the results of the combined fit (see section 3.2, formula (1), for a description of the model). As expected for the HPCG benchmark, the performance is dominated by the MEM BW part compared to HPL and MPI BW. We can expect improved application performance if the infrastructure memory bandwidth improves. The I/O dependency is zero as the application does not do any I/O. Finally, we can use this model to predict the performance scaling (formula (2) in section 3.2). For example, at 720 cores (20 nodes), model

³³ <https://icl.utk.edu/hpcc/>

³⁴ MPICH ABI Compatibility Initiative <https://www.mpich.org/abi>

³⁵ <https://www.hpcg-benchmark.org/>



predicts application performance of (338.333 ± 42.746) GFLOP/s, which is well in agreement with the measured value of 335.378 GFLOP/s.

The aforementioned approach and analysis will be applied to derive the Performance Model of the SODALITE HPC testbed, when more compute nodes are introduced in the testbed.

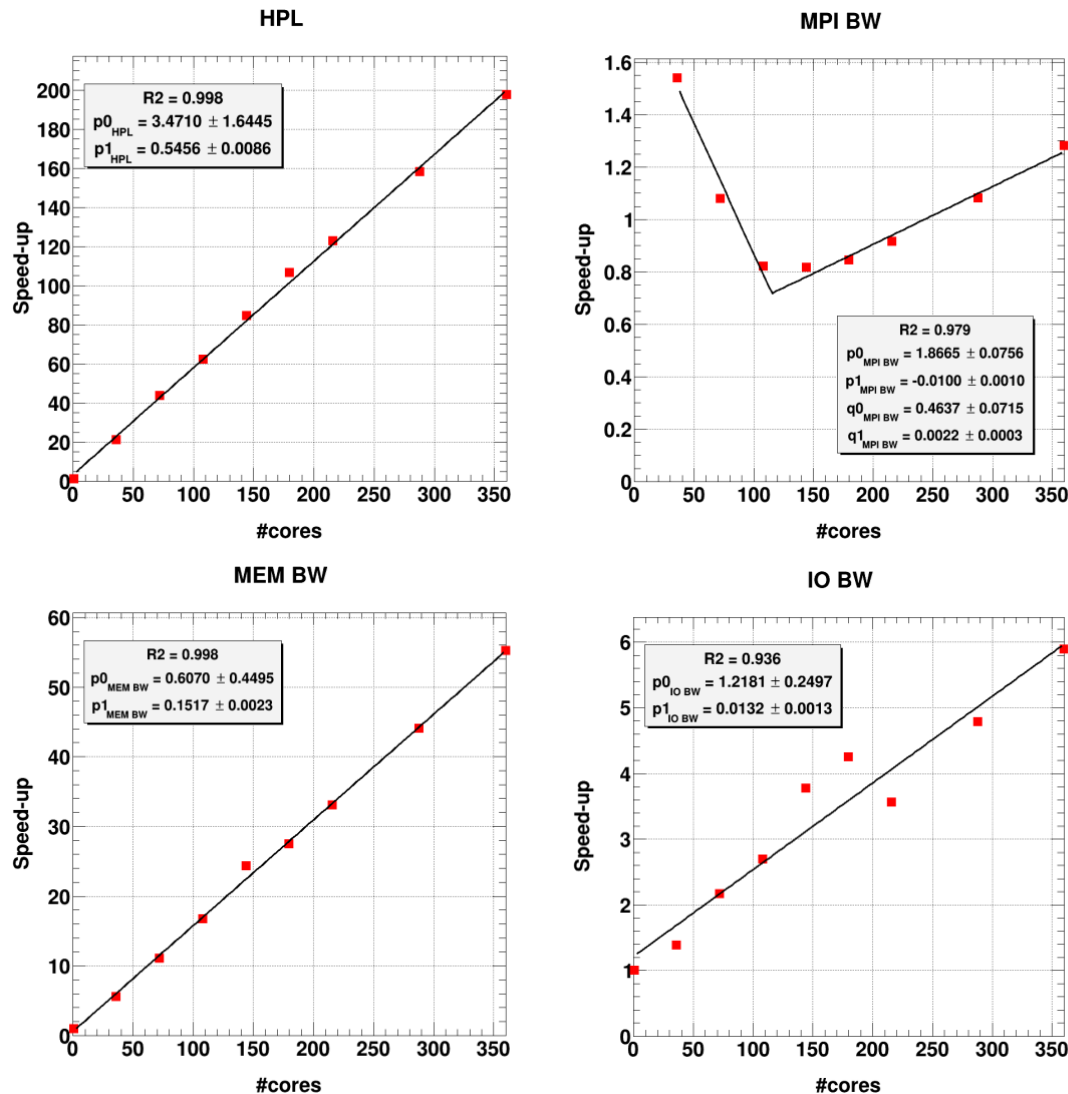


Figure 6: HPC Benchmarks speed-up values on Piz Daint Supercomputer. Speed-up are evaluated with respect to 1 core performance, except for MPI BW where the result on 2 cores is considered. Linear fittings ($y=p_0+p_1x$) are superimposed for each plot. Note that for the MPI BW we use two linear polynomials to describe the low and high core count behaviours of the speed-up, respectively

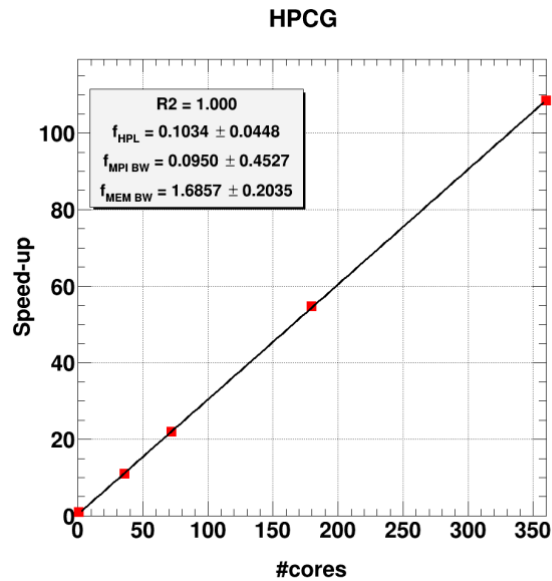


Figure 7: HPCG Benchmark speed-up values and combined fit

6.2 Performance model on Cloud test bed

One of the main differences between HPC and Cloud is the way the hardware is shared. In HPC, the compute and memory are dedicated, whereas the network and storage are shared by applications. In Cloud, compute, memory, network and storage are virtualized and are usually shared by applications. This causes high variance in application performance. This high variance should be modeled for the Cloud. For the cloud, the performance model described in section 3.1 and illustrated in section 6.1 will be modified to include performance variation. This will be developed after M12 as part of the full application and Infrastructure performance model.

In the next subsections, we describe the demonstrating use cases and report on their measured baseline performance on the testbed.

6.3 Snow UC

Figure 8 shows the six main components of the Snow UC image processing pipeline as planned for the first version. This will be extended further in future releases.

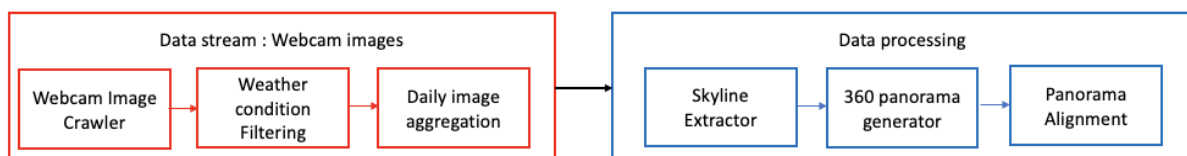


Figure 8: Initial version of the pipeline as a sub-group of the components of the original one

Two of these components are already been executing in the HPC and Cloud testbeds, while the others are still being deployed and not ready to measure the baselines. In this section we will provide a first metric of performance for the Skyline extraction and the Panorama Alignment components. For the components that were deployed in the Cloud testbed the employed VMs have the following configuration (when not otherwise specified): 2vCPUs, 4GB RAM and 40GB Disk space, whereas for the HPC component (only model training of the Skyline Extractor), only one compute node with 20 cores CPU, 128GB RAM and one GPU was used.



6.3.1 Skyline Extraction

The goal of this component is to obtain the landscape skyline of a photograph, i.e., the set of all the points that represent the boundary between the terrain slopes and the sky. For this purpose, every pixel of the input image is fed to a binary classifier, and only positive edges are retained. The training and validation of the classifier is done using a dataset of mountain images, where for each one exists an annotation containing the skyline present on it. Once the classifier has been trained it is used on a web service whose goal is to take an image as input, execute the model, perform some image post-processing, and return the skyline mask, which will be later used to perform the panorama alignment.

6.3.1.1 Model training

For the Skyline Extraction model training, the dataset consisted of 8,856 images annotated with skyline, from which 80% was used for training and validation and the remaining 20% for testing. The component was trained using a modified version of a well-known computer vision model known as LeNet [17], using TensorFlow 1.11.

The complete training of the Skyline Extraction model takes approximately 7.2 hours in one GPU node of the HPC testbed. The training executes multiple epochs and finishes in epoch 20 due to early stopping with a test accuracy of 0.955485. Profiling the complete run and studying the performance is complex, so instead, a single epoch is profiled and studied to understand the performance,

For a single epoch, the application took 1200 seconds and this time can be further classified as shown below in Table 2.

Application component	Subcomponent	Time in seconds
Initialisation time		12
Training time	Get next training batch	896
	Calculate loss	18
	Training step	29
Validation time		233

Table 2: Skyline extraction training wallclock time profile

The application spends the bulk of the time getting the next batch of data for training. Using the GPU profiler and profiling a single epoch shows that only 32 seconds are spent in the GPU (runtime of ~1200 seconds), which corresponds to only 3% of the total time. Figure 9 shows the GPU profile of Skyline extraction training (single epoch). Kernel *dgrad_engine* calculates the gradients for back propagation. *CUDA_memcpy_HtoD* represents the time spent in copying data from host to device. *sgemm* and *relu* represents single precision matrix multiplication and *REctifier Linear Unit* kernels. The full training and GPU kernel profiles clearly indicate data movement as a bottleneck for performance.

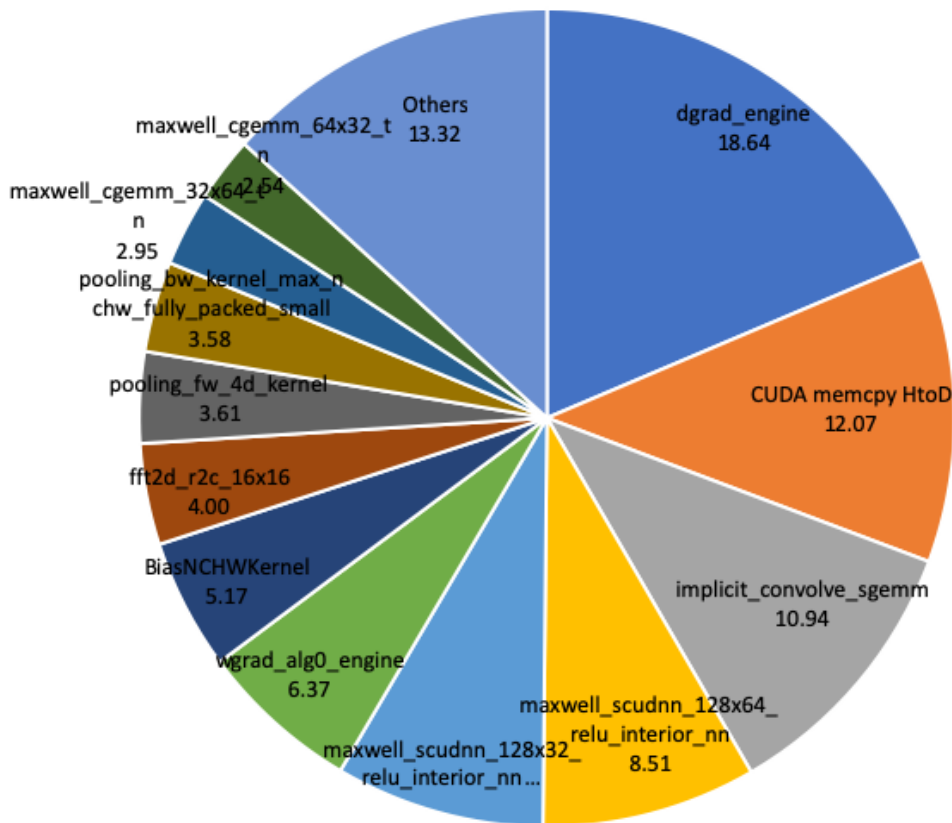


Figure 9: GPU profile of a single epoch of Skyline extraction training

6.3.1.2 Web component

The service to make the skyline extraction available was developed in Java with the Spring framework, making use also of the OpenCV library for the image processing. To measure its processing and response time a small set of 20 images was created. Such images have on average a size of 1MB. We measure the time that it takes to send the request and write the result in the file system. In this case, the result is a .jpg containing the skyline mask.

	Total Time	Time Namelookup	Time to Connect	Time Start Transfer
Average	1.53s	31.9µs	66.94ms	0.14s

Table 3: Web component response time

On average, the component takes 1.5 seconds to process an image of 1MB. This time can be split into different parts of the process. *Time Namelookup* is the time, in seconds, it took from the start until the name resolving was completed. *Time to connect* is the time, in seconds, it took from the start until the TCP connection to the remote host (or proxy) was completed. Finally, we have the *Time Start Transfer*, which is the time, in seconds, it took from the start until the first byte was just about to be transferred. This includes all pre-transfer commands and negotiations that are specific to the particular protocol(s) involved and also the time the client needed to calculate the result. In this case, this time is about 0.14 seconds.



If we detract the time the client needed to prepare the request (0.2s) from the total time it took to send, process the request and write the results, we get that the component takes on average 1.3 seconds.

6.3.2 Panorama Alignment

The alignment can be seen as the search for the correct overlap between two cylinders (assuming the zero tilt of the photograph): one containing the 360° panorama of the terrain at a given location and the other one containing the photo, suitably scaled. It is useful to obtain the Mountain Mask that will be the input to the successive components to calculate the Snow Index.

The service to make the panorama extraction was developed in Java with the Spring framework, making use also of the OpenCV library for the image processing. To measure its processing and response time a small set of 20 images was created. Such images have in average a size of 1MB.

We measure the time that it takes to send the request and write the result in the file system. In this case, the result is a json file containing the mountain mask (along with other attributes) that will be later used to compute the snow index.


	Total Time	Time Namelookup	Time to Connect	Time Start Transfer
Average	13.25s	37.15µs	66.56ms	0.14s

Table 4: Panorama Alignment response time

On average, the component takes 13.2 seconds to process an image of 1MB. This time can be split into different parts of the process. *Time Namelookup* the time, in seconds, it took from the start until the name resolving was completed. *Time to connect* is the time, in seconds, it took from the start until the TCP connect to the remote host (or proxy) was completed.

Finally, we have the *Time Start Transfer*, which is the time, in seconds, it took from the start until the first byte was just about to be transferred. This includes all pre-transfer commands and negotiations that are specific to the particular protocol(s) involved and also the time the client needed to calculate the result. In this case, the time is about 0.14 seconds.

If we detract the time the client needed to prepare the request (0.2s) from the total time it took to send, process the request and write the results, we get that the component takes on average 1.3 seconds.

 *This component invokes the 360 Panorama generation. At the moment and for testing purposes, it is invoking a POLIMI instance of the panorama generator. I think it would be more accurate to actually measure the performance using the SODALITE testbed instance of the 360 panorama.*

6.4 Vehicle UC

Within the Vehicle IoT use case, individuals may, at various times, submit license plate images for recognition. These purposes include the initial registration of the vehicle with the mobile app, capturing evidence to support insurance claims preparation (such as in the case of a collision), or, within a fleet or car sharing scenario, for accessing a vehicle that has been allocated to the individual to use. In order to enable these use cases, the accuracy of the detection model is of fundamental importance, and, therefore, is something that must be iteratively improved over time.

6.4.1. License Plate Recognition

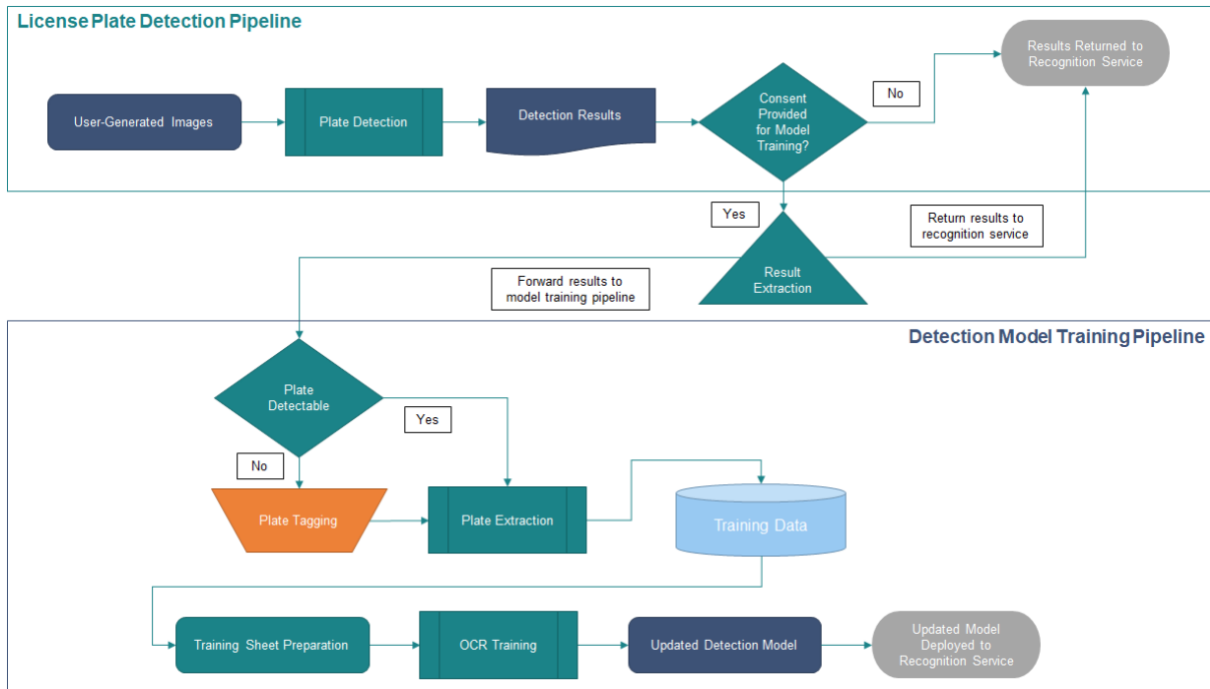


Figure 10: Schema of the license plate detection and detection model training pipelines (Vehicle IoT use case)

License Plate Recognition itself is carried out via a dedicated microservice, which is backed by an ML-based plate detection pipeline using a pre-trained model. The process for this, and the training of the detection model are outlined in Figure 10.



Figure 11: License Plate identification and extraction

In the case of the license plate recognition service, an image of the plate is sent to an OpenALPR-backed³⁶ microservice via a REST API, which, in turn, makes two passes across the image. The first pass looks to detect a license plate within the image (using OpenCV³⁷), which is then extracted and processed by Tesseract OCR³⁸ in order to return the detected textual results. This process is exemplified in Figure 11.

³⁶ Automatic License Plate Recognition <https://www.openalpr.com>

³⁷ <https://opencv.org>

³⁸ <https://github.com/tesseract-ocr/tesseract>



6.4.1.1. License Plate Detection Pipeline

Using a trained model, results are grouped into ‘detectable’, ‘partially detectable’, and ‘undetectable’ images. *Detectable images* are ones where the text and the image match, with a reasonably high confidence level (> 90%). *Partially detectable images* are ones where a plate has been identified and parts of the number string have been correctly identified, but others not (this is often as a result of plate orientation, such as when an image is taken from an angle, rather than head-on) - confidence levels may be closer to 50-75%. *Undetectable images* are ones where the service is unable to identify a license plate in the image at all, failing to extract the plate and make it beyond the first pass detection.

In Table 5 below, baseline measurements have been obtained on a quad-core Intel Xeon E3-1275 v5 server with 64GB of RAM. These are split between the *Detection Time* (the time spent identifying the existence of a license plate and extracting a cropped section for OCR processing), *Processing Time* (the time spent performing character recognition on the extracted plate and identifying the plate number string), *Transfer Time* (the time spent in transmitting the source image to the service for analysis), and the *Total Time* spent on each of these activities combined.

	Image Size	Total Time	Detection Time	Processing Time	Transfer Time (Local)
Detectable Images	300kB	125ms	83.75ms	17.05ms	24.20ms
		126ms	83.61ms	16.37ms	26.02ms
	1MB	334ms	267.09ms	18.42ms	48.49ms
	1.7MB	420ms	277.42ms	28.01ms	114.57ms
	3.4MB	476ms	260.40ms	26.74ms	188.86ms
	11MB	1,182ms	700.14ms	24.67ms	457.20ms
Partially Detectable Images	100kB	189ms	153.99ms	16.85ms	18.16ms
Undetectable Images	200kB	222ms	207.62ms	-	14.38ms
	300kB	248ms	224.54ms	-	23.46ms
	2.2MB	546ms	217.38ms	-	328.62ms

Table 5: License plate recognition times across file sizes

From Table 5, we can see that most of the time is (as expected) in transferring the image to the service (especially in cases of larger file sizes) and in identifying and extracting the plate from the image (*Detection Time*). Once the plate has been extracted, the amount of time to decode the plate



string (*Processing Time*) does not vary significantly. The amount of time spent on detection similarly does not fluctuate significantly, regardless of whether the plate is detectable in whole, in part, or not at all.

6.4.1.2. Detection Model Training Pipeline

Plates that are partially detectable are identified as candidates for improving the OCR detection model and are saved off for subsequent retraining (on an opt-in basis, with the end-user’s explicit consent) and subsequent re-deployment. In order to produce a new Tesseract OCR training sheet, at least 200+ new images should be included, requiring an average processing time of 16-20 hours (on a quad-core Intel Xeon E3-1275 v5 server with 64GB of RAM), depending upon image complexity.

Plates that are undetectable highlight a larger problem or shortcoming with the first-pass detector, as no plate can be identified within the image. In this case, a much larger range of images and processing times are required. For best results, at least 3000+ new images should be included, requiring processing time of anywhere from 40-60 hours, depending upon image complexity.

Component	Images Required	Approximate Training Time
OCR Training (Plate Extraction)	200+	16-24 hours
Plate Detector	3000+	40-60 hours

Table 6: License Plate Detection Models, data requirements, and current training times

As neither of these cases is frequently run (there is no online learning carried out), processing can be batched - lending itself well to the increased computational capabilities of HPC systems. Plate tagging, however, remains a manual process, and effectively serializes this part of the workload.

6.5 Clinical UC

Deliverable D6.2 "Initial Implementation and Evaluation of the SODALITE Platform and Use Cases" (Section 4.2) describes the prototype implementation of the Clinical UC, the components of which are presented in Figure 12. Due to the higher complexity of the reconstruction and discretization steps, the initial version of the segmentation, geometry extraction and discretization steps were performed manually, therefore the simulation process chain starts from the Density Mapping component.

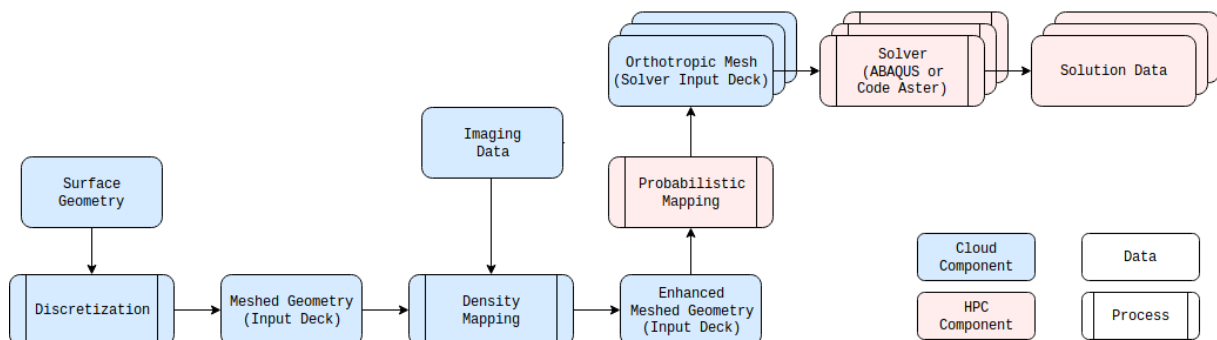


Figure 12: Schema of the Virtual Clinical Trial use case pipeline



The components that were deployed and run on the HPC testbed are Density Mapping, Probabilistic Mapping and Code_Aster Solver. The components were encapsulated into the Singularity (v3.4.1) containers and based on the ubuntu:18.04 image. All measurements were made in the frontend node, which has 20 compute cores, and currently only application wallclocktime is measured. The following subsections report on the configuration details and measurements for the deployed components.

6.5.1 Density Mapping

This component receives three inputs decks with meshed geometries as well as three CT data, and then maps density values into each element inside the volume mesh, producing modified input decks that are used to generate orthotropic material data.

The implementation of *Density Mapping* is done in Fortran. It is a sequential process and not parallelizable, therefore the configuration cannot be changed for further improvement. Running it on the testbed took 122.4 seconds on average.

6.5.2 Probabilistic Elasticity Mapping

Probabilistic Mapping receives the output from *Density Mapping* and computes a probability distribution for the elasticity for each element, extracting the low and high bounds and the mode of the density interval. It then produces three solver input decks (low, mode, high) for each input deck.

Probabilistic Mapping was implemented in Python and uses mpi4py³⁹ Python library for MPI parallelization. We ran the Probabilistic Mapping component over various number of ranks utilizing MPI (starting from single rank and exponentially doubling the number of ranks) with the following parameters: number of samples is 10000 and number of datapoints of the volume datasets is 8192. The Execution (wallclock) time over the number of MPI ranks for Probabilistic Mapping is shown in **Error! Reference source not found..**

Number of MPI ranks	1	2	4	8	16	32	64
Wallclocktime, s	148.66	76.16	43.22	27.73	21.42	46.84	62.60

Table 7: Execution (wallclock) time over the number of MPI ranks for Probabilistic Mapping

The results show that with the increase of ranks number, the performance improves (i.e. decrease of execution time) until 16 ranks; after that the performance is getting worse. The reason is the CPU bound, since the number of ranks exceeds the number of cores on the frontend node, thus the execution of the probabilistic mapping should be distributed over multiple physical nodes to further gain a speedup.

6.5.3 Solver

Receiving input decks from the *Probabilistic Mapping*, the Solver component uses finite element methods to compute a solution for lower and upper bounds and the mode of the highest density intervals, producing a solution file for each deck. The Solver is based on *Code_Aster* software package, which in turn is written in Fortran and can be parallelized with MPI. Despite parallelization

³⁹ <https://mpi4py.readthedocs.io/en/stable/>



support, the prototype version of this component was run on a single thread, resulting in average execution time of 789.17 seconds.

7 Concluding Remarks

Application performance depends on multiple factors and modelling the performance will help resource experts to better provision the infrastructure and also get maximum performance for their application. Applications can be optimised before deployment (static) or at runtime (dynamic). In this report, we presented a prototype of Infrastructure and application *Performance Model* for static performance optimisation. We can use this model to predict HPC application performance in a HPC or a Cloud based infrastructure. We have shown how a benchmarking-based model can be used to accurately predict performance scaling of a standard HPC application. This model will be further adapted to model cloud-based infrastructure and improved to also model accelerator-based hardware like GPUs.

Moreover, we introduced an approach for the runtime re-configuration of resources for machine learning applications deployed in the Cloud that can be executed using both CPU and GPUs. The solution is an extension of TensorFlow based on lightweight control theory planners that exploits the speed of containers in being re-configured to perform fast and fine-grained vertical scaling of resources. This allows us to efficiently fulfil the quality of service requirements over the response time.

The deployment refactoring requires the ability to select a particular set of deployment options as well as to change the current deployment option selection in order to maintain the performance goals of the application under changing workloads. To support this decision making, a model is trained to estimate the impacts of a deployment option selection on the performance metrics. This model will be used to compare and select alternative deployment variants at runtime, and to guide the modifications to the deployment model from the dynamically discovered deployment options.

References

1. SODALITE Consortium, Requirements, KPIs, evaluation plan and architecture - First version, Technical deliverable 2.1, 2019.
2. D. Frajberg, P. Fraternali, and R. N. Torres, "Convolutional neural network for pixel-wise skyline detection," in International Conference on Artificial Neural Networks. Springer, 2017, pp. 12–20.
3. D. Frajberg, C. Bernaschina, C. Marone, and P. Fraternali, "Accelerating deep learning inference on mobile systems," in International Conference on AI and Mobile Services. Springer, 2019, pp. 118–134.
4. Berger, Thorsten, et al. "A study of variability models and languages in the systems software domain." IEEE Transactions on Software Engineering 39.12 (2013): 1611-1640.
5. Esfahani, Naeem, Ahmed Elkhodary, and Sam Malek. "A learning-based framework for engineering feature-oriented self-adaptive software systems." IEEE transactions on software engineering 39.11 (2013): 1467-1493.
6. Guo, Jianmei, et al. "Variability-aware performance prediction: A statistical learning approach." 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2013.



7. Yadwadkar, Neeraja J., et al. "Selecting the best vm across multiple public clouds: A data-driven performance modeling approach." Proceedings of the 2017 Symposium on Cloud Computing. ACM, 2017.
8. Baughman, Matt, et al. "Profiling and Predicting Application Performance on the Cloud." UCC (IEEE/ACM International Conference on Utility and Cloud Computing). 2018.
9. Herbst, Nikolas Roman, Samuel Kounev and Ralf H. Reussner. "Elasticity in Cloud Computing: What It Is, and What It Is Not." ICAC (2013). 9
10. Nguyen, Hiep, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah and John Wilkes. "AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service." ICAC (2013). 10
11. Xiao, Zhen, Weijia Song and Qi Chen. "Dynamic Resource Allocation Using Virtual Machines for Cloud Computing Environment." IEEE Transactions on Parallel and Distributed Systems 24 (2013): 1107-1117.
12. AWS Autoscaling, <https://aws.amazon.com/autoscaling/>
13. Baresi, Luciano and Giovanni Quattrocchi. "Towards Vertically Scalable Spark Applications." Euro-Par Workshops (2018).
14. L. Baresi, S. Guinea, A. Leva, and G. Quattrocchi, "A discrete-time feedback controller for containerized cloud applications". FSE 2016.
15. L. Baresi, A. Leva and G. Quattrocchi, "Fine-grained Dynamic Resource Allocation for Big-Data Applications," in IEEE Transactions on Software Engineering. 2019
16. Åström, Karl Johan. "PID Controllers: Theory, Design, and Tuning." (1995).
17. Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998.



Appendix 1. HPC Benchmarks Singularity Definition File

Bootstrap: docker
From: debian:buster

```
%help
    This is the container used to run SODALITE (https://www.sodalite.eu/)
    benchmarks for performance modelling (WP3).

%labels
    Author cerl@cray.com -- Cray EMEA Research Lab
    Version v0.1.0

%environment
    export BINDIR=/workdir/bin

    # Taken MPI parameters
    export MYRANK=${${BINDIR}/mpirank | cut -d':' -f 1}
    export NPROCS=${${BINDIR}/mpirank | cut -d':' -f 2}

    # Ignore inputs if multiple ranks are recognized
    if test ${NPROCS} -eq 1; then
        if test ! -z ${NP+x}; then
            export NPROCS=${NP}
            export RUNCMD="mpiexec -np ${NPROCS}"
        fi
    fi

%runscript
    timestamp=${timestamp:-$(date '+%Y%m%d%H%M')}
    filelabel=${filelabel:-"ranks${NPROCS}"}
    maindir=${PWD}
    rundir=${rundir:-${maindir}}
    if test ${MYRANK} -eq 0; then
        echo "NPROCS:" ${NPROCS}
        echo "Label file:" ${filelabel}
        echo "Rundir:" ${rundir}
    fi

    cd ${rundir}

    case "${APP}" in
        hpcc)
            if test ${MYRANK} -eq 0; then
                # Delete previous HPCC output
                rm -f hpccoutf.txt
                echo "Run:" ${RUNCMD} ${BINDIR}/hpcc
                test -e $rundir/hpccinf.txt || \
                    cp ${maindir}/hpccinf.txt ${rundir}/hpccinf.txt
            fi
            ${RUNCMD} ${BINDIR}/hpcc
            if test ${MYRANK} -eq 0; then
                # Rename HPCC output
                if test -f hpccoutf.txt; then
                    mv hpccoutf.txt \
                        ${maindir}/results/${filelabel}_${timestamp}_hpccoutf.txt
                else
```



```
        echo "Exit with error!"
        exit 1
    fi
fi
;;
beffio)
TOTMEM=`free -m | grep Mem | awk '{print $2}'`
COREMEM=$(( TOTMEM / $(getconf _NPROCESSORS_ONLN) ))
BEFFFLAGS=${BEFFFLAGS:-"${maindir}/beffio_flags.txt"}
BEFFIOCMD="-MB ${COREMEM} -MT ${TOTMEM} \
$(cat ${BEFFFLAGS}) \
-N ${NPROCS} -f ${filelabel}_${timestamp}_b_eff_io"
if test ${MYRANK} -eq 0; then
    echo "Run:" ${RUNCMD} ${BINDIR}/b_eff_io ${BEFFIOCMD}
fi
${RUNCMD} ${BINDIR}/b_eff_io ${BEFFIOCMD}
if test ${MYRANK} -eq 0; then
    if test -f ${filelabel}_${timestamp}_b_eff_io.prot \
        -a -f ${filelabel}_${timestamp}_b_eff_io.sum; then
        mv ${filelabel}_${timestamp}_b_eff_io.prot \
            ${filelabel}_${timestamp}_b_eff_io.sum ${maindir}/results
    else
        echo "Exit with error!"
        exit 1
    fi
fi
fi
;;
*)
if test ${MYRANK} -eq 0; then
    echo "Specify an app with APP=<app>, where <app> can be 'hpc' or
'beffio'!"
fi
;;
esac

cd ${maindir}

%files
# Copy ARCH file to compile HPCC
# Based on https://github.com/hpc-uk/archer-benchmarks/blob/master/synth/HPCC/Make\_arch/Athena/Make.athena\_gcc
# Create workdir
build/Make.Linux /workdir/

# Copy C program to check MPI rank
build/mpirank.c /workdir

%post
#
# Binary directory
#
```



```
export BINDIR=/workdir/bin
mkdir -p ${BINDIR}

#
# First install default packages and clean caches
#
apt-get update && apt-get -y upgrade --no-install-recommends
apt-get -y install --no-install-recommends \
    build-essential wget less gfortran procs \
    libopenblas-dev

apt-get clean && rm -rf /var/lib/apt/lists/* # do not forget to clean!

#
# Move inside build directory
#
cd /workdir

#
# Install MPICH
#
export MPICH_VERSION=3.3.1
wget -q \ http://www.mpich.org/static/downloads/${MPICH_VERSION}/mpich-
${MPICH_VERSION}.tar.gz
tar xf mpich-${MPICH_VERSION}.tar.gz && rm mpich-${MPICH_VERSION}.tar.gz
cd mpich-${MPICH_VERSION}
./configure --prefix=/usr/local --disable-static --disable-rpath \
    --disable-wrapper-rpath \
    --mandir=/usr/share/man --enable-fast=all,O3
make -j$(getconf _NPROCESSORS_ONLN) install
ldconfig
cd .. && rm -rf mpich-${MPICH_VERSION}

#
# Install HPCC (https://icl.cs.utk.edu/hpcc/)
#
export HPCC_VERSION=1.5.0
wget \
    http://icl.cs.utk.edu/projectsfiles/hpcc/download/hpcc-${HPCC_VERSION}.tar.gz
tar xf hpcc-${HPCC_VERSION}.tar.gz && rm hpcc-${HPCC_VERSION}.tar.gz
cd hpcc-${HPCC_VERSION}
# Move Arch file
mv ../Make.Linux hpl/
make arch=Linux
cp /workdir/hpcc-${HPCC_VERSION}/hpcc ${BINDIR}
cd .. && rm -rf hpcc-${HPCC_VERSION}

#
# Install b_eff_io (https://fs.hlr.de/projects/par/mpi//b_eff_io/)
#
export BEFFIO_VERSION=2.1
wget --no-check-certificat \
```



```
https://fs.hlrs.de/projects/par/mpi//b_eff_io/b_eff_io_v${BEFFIO_VERSION}.tar.gz
tar xf b_eff_io_v${BEFFIO_VERSION}.tar.gz && \
  rm b_eff_io_v${BEFFIO_VERSION}.tar.gz
cd b_eff_io
mpicc -o ${BINDIR}/b_eff_io b_eff_io.c -lm
cp -r eps ${BINDIR}/b_eff_io_eps
cd .. && rm -rf b_eff_io

#
# Compile program to get MPI rank
#
mpicc /workdir/mpirank.c -o ${BINDIR}/mpirank
```