Ref. Ares(2022)6135937 - 05/09/2022

Data driven Computational Mechanics at EXascale

DCoMEX

Data driven Computational Mechanics at EXascale

Work program topic: EuroHPC-01-2019

Type of action: Research and Innovation Action (RIA)

The DCoMEX Framework Prototype (software and report).

DELIVERABLE D.6.2b

Version No 1

http://www.dcomex.eu/

This project has received funding from the European High-Performance Computing Joint Undertaking Joint Undertaking ('the JU'), under Grant Agreement No 956201

D OC U ME N T SU MMA RY I N F ORMA T I ON

Project Title	**Data driven Computational Mechanics at EXascale**
Project Acronym	DCoMEX
Project No:	**956201**
Call Identifier:	EuroHPC-01-2019
Project Start Date	01/04/2021
Related work package	WP 6
Related task(s)	Task 6.3, 6.4, and 6.5
Lead Organisation	ETHZ/CSELab
Submission date	5/09/2022
Re-submission date	
Dissemination Level	PU

Quality Control:

	Who	Affiliation	Date
1	Vissarion Papadopoulos (Coordinator)	NTUA	1/09/2022

Document Change History:

Version	Date	Author (s)	Affiliation	Comment
1.0	28.06.2022	B. Cumming, G. Stavroulakis, T. Christodoulou, A. Fink, L. Amoudruz.	CSCS/ETHZ, CSELab/ETHZ, MGroup/NTUA, MGroup/NTUA	

Contents

1. Introduction

This report provides an overview of the work to date and current status of the DCoMEX Software Framework, the DCoMEX Framework Prototype deliverable, that has been implemented in Work Package 6. The DCoMEX Framework Prototype is a companion deliverable (D6.1a), which provides implementations of the required modules, to serve as a basis for evaluation and further implementation of the final DCoMEX Framework in M24.

Software Development and Availability

The DCoMEX Software Framework is developed in an open source repository on GitHub, github.com/DComEX/dcomex-prototype, using a trunk-based development[1] workflow:

- Developers implement small incremental changes in short-lived branches from the main *trunk* of the repository.
- Developers submit frequent *pull requests* to merge the changes into the *trunk*.
- Each pull request must be reviewed and pass continuous integration tests before it can be merged to the *trunk*.

The individual software modules are developed in separate repositories by the responsible teams, and imported to the Framework using git submodules and NuGet repositories, as appropriate. The software modules in the framework are:

- **Korali**: developed in a single repository on GitHub, github.com/cselab/korali, by ETHZ/CSELab.
- **MSolve**: developed in multiple repositories on GitHub, github.com/mgroupntua, by MGroup/NTUA.
- **AISolve**: Not yet integrated in the framework – integration scheduled for the deliverable “AI-Solve Library Prototype” (D3.1a) in M20 by MGroup/NTUA.

2. Framework

The DCoMEX framework is an integration of software modules, both existing and newly developed for this project, that are integrated with a common API. The modules, developed by separate teams,

[1] https://trunkbaseddevelopment.com/

are based on different software development frameworks and environments. The process of integrating the modules to form the framework prototype, the following steps were taken:

1. For each module: Isolate the components of the module that are required for integration into the framework.
2. For each module: Perform tests to determine requirements required to build and run the module on HPC clusters, with Piz Daint at CSCS as the initial target.
3. For each module: Define a container environment and develop a continuous integration (CI) pipeline that builds and tests the module in the container on Piz Daint
4. Configure a common container environment with compatible versions of all dependencies required to build and run all of the modules.
5. Integrate the modules in the framework in a standalone repository that uses the common container environment to build and deploy.

The following sections will first describe the common methods that were developed in Work Package 6 to build and deploy the framework software (CI/CD and Containers), followed by an overview of steps 1-3 above for each of the modules.

CI/CD and Containerization For HPC Deployment

The framework components – namely MSolve, Korali and AISolve – use different software frameworks with complicated requirements, some of which are not typically deployed on HPC clusters. In order to develop and deploy the framework in a reproducible and robust manner, the first objective of the work package was to gather the requirements of each component, and then find a way to package them for HPC.

The Korali and MSolve components have common requirements, with the most important for HPC deployment being:

- MPI: A standard library for distributed computation, used by Korali to distribute individual simulation instances across a pool of nodes on a HPC systems, and by MSolve to parallelise individual simulation instances.
- CUDA libraries and runtime: used to develop and run software on NVIDIA GPUs.

Additionally, MSolve (see section below on MSolve) is developed in the C# programming language, which is part of the .NET software platform. .NET is not typically used for HPC software deployed on Linux based HPC clusters, with the cross-platform version of .NET being a recent development.

Given the different requirements of the modules, there are two challenges associated with maintaining such an integration:

- The initial process of determining the required frameworks and dependencies, their versions and how to configure them. Together, these will be referred to as the *software stack* that provides all the tools required to develop and deploy the integrated framework. Defining this requires extensive testing and research.
- The ongoing process of maintaining the software stack and change management on HPC systems. Changes are difficult to manage for tightly coupled software stacks. Changes can be both to the HPC system environment (which can break the dependencies of the software stack), and made to the framework by developers (who develop the software locally on their own system).

To overcome those challenges we containerised the software stack, and implemented an automated CI/CD pipeline that builds and tests the software on a target HPC system (Piz Daint at CSCS). Using a containerised environment encapsulates the dependencies and their configuration in the framework repository. Furthermore, it facilitates two types of portability: the first is allowing developers to develop and test in the same environment on their local computer that will be used on the target HPC cluster; and the second is that a container is portable across changes to the underlying HPC system environment by virtue of reducing the number of dependencies to a minimum (MPI ABI compatibility and CUDA driver version for the DCoMEX Framework modules).

Containers and Sarus

Sarus[2] is a container engine for HPC systems that provides a user-friendly way to instantiate feature-rich containers from Docker images achieving native compute performance. It is designed to blend the portability of containers with the unique requirements of HPC installations, such as: native performance from dedicated hardware; improved security due to the multi-tenant nature of the systems; support for network parallel file systems and diskless computing nodes; and compatibility with a workload manager or job scheduler.

Sarus relies on industry standards and open-source software, specifically the OCI specifications[3], that allow Sarus to extend the capabilities of the container runtime through external plugin programs, called *OCI hooks*. Hooks can customize containers to enable specific high-performance features, and

[2] https://products.cscs.ch/sarus/
[3] https://opencontainers.org/

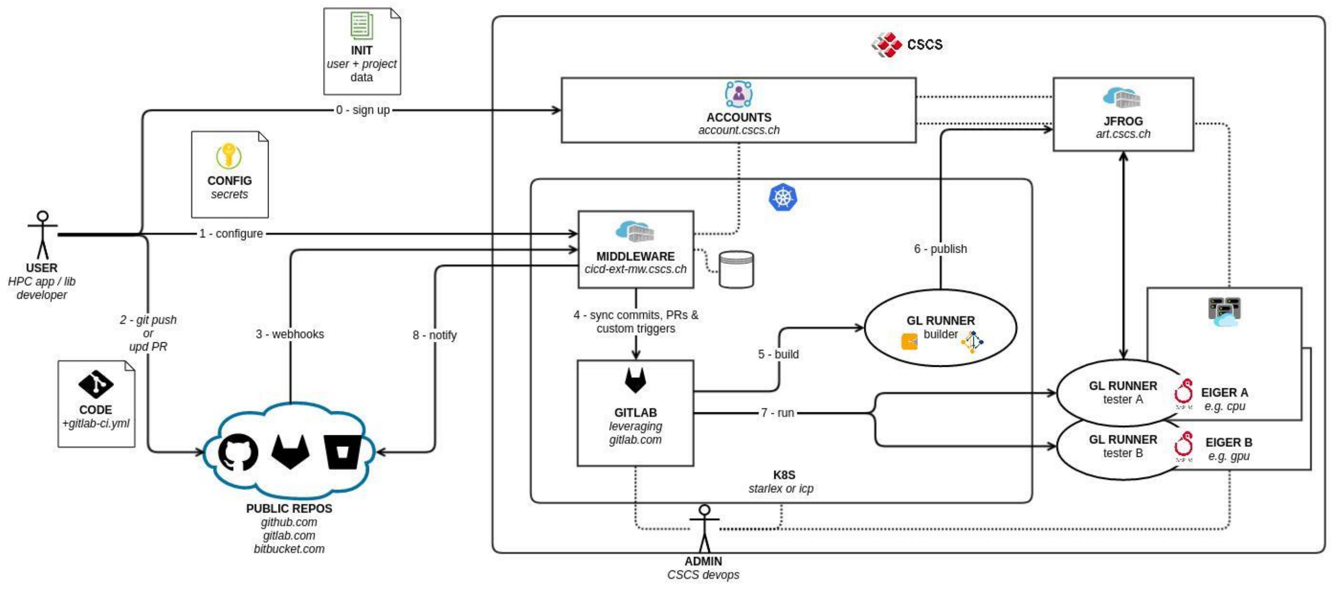
they can be developed independently by third parties to introduce new technologies or improve support for existing ones. For the DCoMEX Framework, the two most important hooks are for MPI and CUDA, which make optimized versions of each library and hardware respectively available to containerised applications.

CI/CD Service at CSCS

CSCS is developing a CI/CD service for software developed using public Git VCS services – GitHub, GitLab and BitBucket – for better testing of scientific software on target hardware systems and for direct deployment of tested software for CSCS users. The DCoMEX Framework is an ideal target use case for the system, and has been used to co-design the service (as part of Task 6.4 Testing and Benchmarking).

Adding a webhook in the public repository pointing to the middleware server and an entrypoint YAML file, typically *.gitlab-ci.yml,* enables CI/CD at CSCS. The webhook will inform the middleware about any event in the public repository and start CI/CD jobs for every event that involves testing newly pushed code. The public repository code is mirrored to a gitlab instance, such that the gitlab runners deployed on CSCS' systems can interact with the code repository. The YAML recipe describes how the software is built and defines the test cases. Since we are leveraging gitlab as the orchestrator the documentation to write such CI/CD recipes can be found at gitlab.com. A CI/CD pipeline is fully defined by the owner of the public repository and consists of many user-defined stages. Each stage consists of many jobs and each job is a set of user-defined commands. Jobs belonging to the same stage are running in parallel, while a successor stage has an implicit dependency on its predecessor stage. Each job defines on which gitlab runner it should be mapped to, where a gitlab runner has different capabilities. The two most prominent gitlab runners at CSCS are:

1. Docker image builder: This runner builds a new docker image and pushes the result to a Docker repository (JFrog).
2. Sarus test runner: This runner pulls images from JFrog and runs tests inside containers on multiple MPI ranks, i.e. it allows testing ones distributed computations. The number of nodes and tasks per node are fully defined by the user and no restrictions are imposed by the CI/CD service.



MSolve

The test application of MSolve utilizes modules imported from the official MSolve GitHub Organisation (https://github.com/mgroupntua/) as binary files. These provide the essential libraries regarding the description of a physical problem, spatial and temporal discretization, linear algebra definitions, as well as “Environments”, such as MSolve’s MPI abstraction.

This MSolve application is running inside a Docker container, configured to facilitate the integration to Piz-Daint. The container is based on the cuda:11.2.0-cudnn8-devel-ubuntu20.04 image, providing the basic infrastructure consisting of Ubuntu 20.04 with CUDA support. Aditionally, MPICH is used as the MPI implementation of choice, allowing for full integration with slurm and Piz Daint. Last part of this architecture is the .Net 6.0 SDK, used to build the application natively.

For the MPI interface we are using the MPI.net library, typically included as a nuget package, that wraps the MPI API for use in .Net applications. The ILGPU library is also included as a nuget package, used as a JIT compiler for GPU-powered computations from our .NET app, providing an efficient interface with the underlying CUDA infrastructure. These libraries are downloaded using NuGet[4], a .NET service similar to apt that downloads and installs .NET libraries.

[4] https://www.nuget.org/

Development and testing of the aforementioned infrastructure and application was conducted in a GitHub repository[5]. In this repo, CI pipelines were set up through gitlab, targeting the master branch and continuously testing the containerized application and its interface with Piz-Daint, end-to-end.

The end result in the msolve-deploy-poc repository was an example .NET application, running on a docker container with CUDA and MPI support, employing the underlying Piz-Daint infrastructure. The application was able to run on a Piz-Daint, slurm powered, MPI environment, using distributed GPU and CPU nodes. The creation of this containerized, concrete, but simple infrastructure, allowed for easy scalability, portability, as well as integration - as presented in the Koralli integration.

Korali

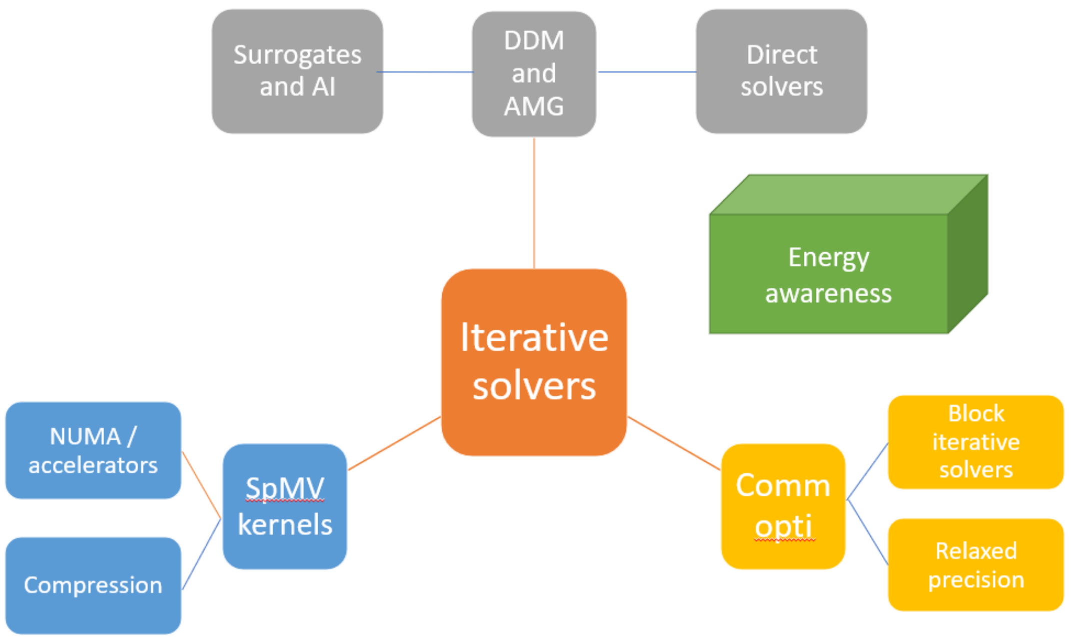
The DCoMEX framework uses Korali v3.0.1 as the engine for Bayesian inference and optimization. Due to the large-scale nature of the problems to solve, the MPI conduit of Korali is used, allowing to run distributed simulations in parallel during sampling and optimization.

Korali is containerized in a Docker image allowing it to run distributed jobs on Piz Daint. The container includes the necessary build systems to compile Korali (meson, ninja) and the required libraries (e.g. pybind11). The framework is automatically tested through a CI/CD pipeline, triggered at every modification of the code database. The CI/CD pipeline consists of a series of unit tests and validation tests for all sampling and optimization solvers implemented in Korali.

AISolve

One of the key and novel modules of DCoMEX will be AISolve, an iterative solver library for the optimized solution of computational mechanics problems in HPC environments.

[5] https://github.com/DComEX/msolve-deploy-poc



The AISolve ecosystem

AISolve will be implemented as an MSolve solver module that adheres to the architecture and corresponding class hierarchy and interfaces of *MSolve.Core*[6]. It will be based upon the *Solvers Module*[7] for all things pertaining to iterative solvers and existing preconditioners, and rely upon the distribution abstraction of the *Environments Module*[8].

The AISolve module is under active development in Work Package 3, with the prototype due as a deliverable in M20, five months after the publication of this framework prototype. As AISolve will be another module MSolve library, it will be integrated framework using the same procedure as the modules that have been integrated to date.

Integration

Once the individual framework components, Korali and MSolve, had been tested on Piz Daint, the collaborators started work on the integration of the Framework Prototype. This can

- Create a single containerised environment that provides all of the build and runtime requirements of the framework modules. This is defined in a Dockerfile in the repository, so that it can be updated as needed alongside changes to the framework: https://github.com/DComEX/dcomex-prototype/blob/master/ci/Dockerfile_base

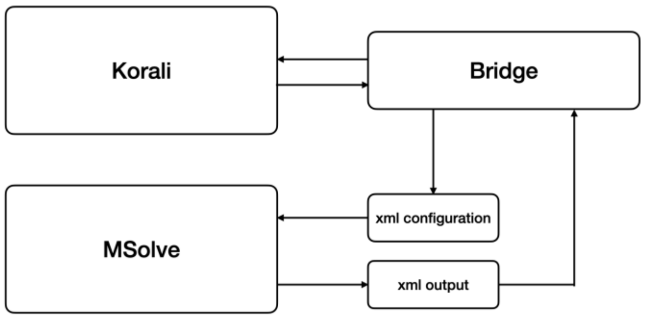
[6] https://github.com/mgroupntua/MSolve.Core
[7] https://github.com/mgroupntua/Solvers
[8] https://github.com/mgroupntua/Environments

- Create a publicly available GitHub repository that pulls in the modules from their respective repositories or package managers: https://github.com/DComEX/dcomex-prototype
- Implement the integration wrapper in Python that handles interfacing MSolve and Korali
- Implement validation test cases that perform inference in Korali for physical models implemented in MSolve to demonstrate the integration.
- Configure a CI/CD pipeline that automates the process of building and validating the framework by running the validation tests on Piz Daint.

Coupling

The coupling between MSolve and Korali is implemented through files. During a Korali experiment, Korali produces a configuration file (in xml format) that describes the problem run by MSolve. Korali then launches a MSolve subprocess (msolve/MSolveApp/ISAAR.MSolve.MSolve4Korali/) with this configuration file. In turn, MSolve produces an output file in xml format that is parsed by Korali. The parsed information is used to update the state of the Korali experiment. This procedure is then repeated up until convergence of the sampling or optimization experiment.

The communication through files is implemented through a python library (bridge) that produces configuration files for MSolve and simplifies parsing the MSolve output.



Integration workflow: the bridge module converts inputs and outputs from the MSolve models, in XML format, to the Korali API.

Validation

As a prototype, two tests were implemented and included in the CI/CD pipeline[9]. The first test consists of evaluating the model implemented by Msolve for a series of parameters. The second test is an inference experiment. It aims to infer the position of a heat source on a two dimensional plate,

[9] For an example of the output of running the pipeline after a change to the repository: https://gitlab.com/cscs-ci/ci-testing/webhook-ci/mirrors/5536596473148359/7418444580822510/-/pipelines/569646257

given temperature measurements at known positions. Korali produces samples through the TMCMC algorithm, where each sample corresponds to a MSolve simulation solving the heat equation for a given heat source position. The temperature at the measurement locations are then parsed from the MSolve output file and compared against the data.

3. Future Work

The DCoMEX Software Framework Prototype presented here will continue to be developed, for release as the full framework in M24 (April 2023). The main objectives of the remaining work are:

- Integrate and test the AISolve module when the first prototype is delivered in M20.
- Implement a series of more complex models with use cases that will demonstrate the usage of the DCoMEX framework. Such use cases will include models pertaining to both structural/material problems and bioengineering problems, as described in WP7 of the DCoMEX project.
- Develop the CI/CD Framework to build, test and *deploy* the software on multiple target HPC hardware architectures, so that correct and up-to-date versions are available directly to users on the CSCS HPC infrastructure.
- Benchmark and optimize the MSolve, Korali and AISolve components for the target HPC hardware architectures.
- Write user documentation, to be used by other work packages to implement the science use cases.