Electronic Communications of the EASST
Volume 83 Year 2025

deRSE24 - Selected Contributions of the 4th Conference for Research Software Engineering in Germany

Edited by: Jan Bernoth, Florian Goth, Anna-Lena Lamprecht and Jan Linxweiler

Enhancing Scientific Reproducibility: A Continuous Integration Workflow for High-Performance Computing

Sören Peters, Sven Marcus, Dennis Gläser, Jan Linxweiler

DOI: 10.14279/eceasst.v83.2625

Electronic Communications of the EASST (https://eceasst.org).
Published by **Berlin Universities Publishing**
(https://www.berlin-universities-publishing.de/)

Enhancing Scientific Reproducibility: A Continuous Integration Workflow for High-Performance Computing

Sören Peters[1], **Sven Marcus**[2], **Dennis Gläser**[3], **Jan Linxweiler**[4]

[1] soe.peters@tu-braunschweig.de
Institut for Computational Modeling in Civil Engineering
Technische Universität Braunschweig
[2] sven.marcus@tu-braunschweig.de
University Library
Technische Universität Braunschweig
[3] Institute for Modelling Hydraulic and Environmental Systems
University Stuttgart
[4] j.linxweiler@tu-braunschweig.de
University Library
Technische Universität Braunschweig

Abstract: Scientific software plays an increasingly important role in modern research. Yet, the lack of software development training among researchers combined with limited funding and recognition for software development often results in error-prone and difficult-to-maintain code. Moreover, the often performance-critical nature of scientific software adds additional complexity to the development and testing process as researchers need to work with high-performance computing (HPC) systems. In this paper, we address the issue of integrating and reproducing workflows on an HPC system by introducing a continuous integration (CI) workflow tailored for HPC environments. Our workflow combines three tools to simplify the execution and validation of computational tasks on HPC systems: `Singularity`, `HPC-Rocket`, and `Fieldcompare`. We leverage `Singularity` containers for consistency across computing setups, then utilize `HPC-Rocket` to launch and monitor a simulation on an HPC system from within a CI pipeline, and finally validate the results using `Fieldcompare` to ensure reproducibility.

Keywords: RSE, FAIR, HPC, Singularity, MPI

1 Introduction

The importance of scientific software has grown significantly over the past years, as it has become an essential part of many researchers' daily work [BCH+17]. However, many researchers lack training in software development, which can lead to poorly written code that is difficult to maintain and prone to errors [Mer10]. Furthermore, research often receives limited funding and recognition for software development, which can lead to a lack of investment in quality and testing. Without a proper testing strategy, researchers often resort to a time-consuming manual testing approach that is prone to human error. During manual testing, researchers may inadvertently deviate from their previous test configuration, overlook testing specific aspects of their

workflow, or introduce errors in the test configuration. This leads to uncertainty about whether previous computations can still be reproduced.

Automating scientific workflows can increase the chance of detecting human-introduced programming or execution errors by performing repetitive and time-consuming tasks more accurately and consistently than humans. This can reduce the time and effort required for testing and deployment, and help to ensure that the software meets the required quality standards. Moreover, automation enhances the reproducibility of scientific results by capturing all the necessary steps to generate the results within a script, enabling easy repeatability of workflows.

In scientific research projects that demand significant computational resources, such as high-performance clusters, the need for automation becomes even more crucial. The scale and complexity of these projects often involve numerous repetitive and time-consuming configuration tasks, that are susceptible to mistakes. Therefore, this poses a challenge for researchers regarding reliably performing computations and identifying software defects. There are several workflow automation tools available that can help address these issues, however, most of them have to be executed on the computing cluster directly. To receive continuous feedback about the functionality of the application during development, these workflows should be executed for every change to the application code. However, due to limited user permissions on the High-Performance Computing (HPC) clusters, connecting them to a CI pipeline is often impossible.

In this paper, we focus particularly on automating computationally intensive workflows that require the use of high-performance computers. We will introduce a continuous integration (CI) workflow applicable to simple test cases as well as large-scale simulations to improve the reproducibility of research software in the HPC context. Our workflow uses `Singularity`[1] containers to encapsulate software and its dependencies, ensuring consistency across different computing environments. We also present `HPC-Rocket` [ML23], which we developed in our group to allow computations to be run on an HPC cluster from within CI pipelines, allowing researchers to repeat their experiments on remote machines easily. Finally, we use `Fieldcompare` [GKP+23], a regression testing tool for validating numerical simulation results against trusted reference data, to ensure that the results produced by our software are accurate and reproducible.

2 Concept

We automate our workflow with the continuous integration service included with the GitLab platform, GitLab CI. The term continuous integration refers to a development practice where software engineers merge their latest changes into the main code branch in short intervals [DMG07]. In a typical CI configuration, a connected continuous integration service like GitLab CI will pick up the new code version, compile it, and run tests against it. Therefore, continuous integration serves as a feedback mechanism to ensure the correct functionality of the software. The continuous integration service will execute a so-called pipeline, a sequence of tasks or jobs with several

[1] Singularity has been split into a commercial product and a free version named Apptainer. Although not tested specifically, we assume this workflow can be easily ported to Apptainer.

individual steps, forming an automated workflow.

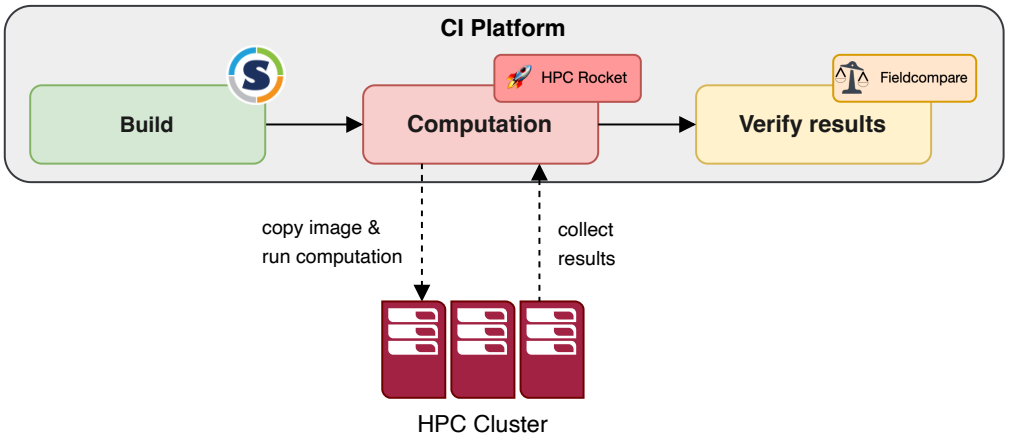

Figure 1: Three main components of the continuous integration pipeline

Figure 1 describes the three main components of our proposed CI workflow. The whole workflow is divided into three successive stages. Our chosen CI Platform, GitLab CI, performs each of these stages in isolated Docker environments. In the first stage, we build a container using `Singularity` that includes all the necessary software and dependencies to run the computation. Containers provide a way to package and distribute software in a portable and reproducible way, making it easier to run the computation on different machines and environments (Section 4). Once the container is built, it can be deployed to a High-Performance Computing cluster using `HPC-Rocket` (Section 5) that will submit the computation as a job to the Slurm scheduling system and monitor its progress. Upon finishing the computation, `HPC-Rocket` collects the produced data and copies it back to the CI pipeline, where it will be compared to a trusted reference data set or experimental measurements using `Fieldcompare` (Section 6).

3 Example Application

In the following, we will outline our proposed workflow using a simplified application written in C++ for demonstration purposes. The program solves the Laplace equation in two dimensions to calculate the heat transfer in a plate. To achieve this, a two-dimensional grid discretizes the domain, and a finite-difference scheme is employed for the discretization of the Laplace equation [ZC08], which yields the following discrete equation to be solved for each point (x,y) in the grid.

$$u_{t+\Delta t}(x,y) = \frac{u_t(x+\Delta x,y)+u_t(x-\Delta x,y)+u_t(x,y+\Delta y)+u_t(x,y-\Delta y)}{4}$$

To solve for the temperature distribution, we need to set boundary conditions, which specify a constant temperature or a temperature gradient at the edges of the material. Once we have set the boundary conditions, we can start iterating over the grid to calculate the temperature by updating every grid point based on the values of its neighbors at the previous time step. This process is repeated until the temperature values converge to a steady state (Figure 2).

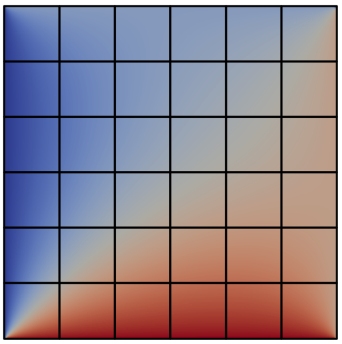

Figure 2: Two-dimensional plate with a steady temperature distribution and constant boundary condition values on the edges.

Parallelization

The finite difference scheme for two-dimensional heat transfer is parallelized by splitting the grid into smaller subdomains and assigning each subdomain to a different processor or thread (Figure 3). Once the grid is split, each processor can independently compute the temperature values in its assigned subdomain using the finite difference scheme. However, communication is required between neighboring processors to compute the values at the boundary between subdomains. This is realized using the Message Passing Interface (MPI).

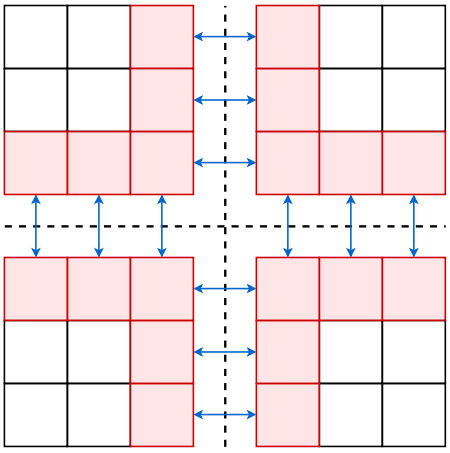

Figure 3: Partitioning of the grid into four sub-grids. Each sub-grid is assigned to a single MPI Process. Information is shared across process boundaries to the cells marked in red.

The following code snippet illustrates the general structure of the application. First, the grid and boundary conditions are initialized. Then, a loop runs until a steady temperature field is reached. Each loop run corresponds to a time step, where the data is exchanged at the edges of the grid using MPI, and the actual calculation of the finite difference scheme happens.

```
initialize_grid()
apply_boundary_conditions()

while(not_steady) {
    exchange_data_using_mpi()
    calculate_grid()
}
```

The computational simulation of heat transfer is a suitable candidate to demonstrate our workflow, as it involves solving a simple differential equation that allows us to leverage the capabilities of HPC clusters due to its parallelizable nature.

4 Virtualisation with `Singularity`

Containers are a virtualization technique allowing the provision of encapsulated runtime environments [Ber14]. We chose the container technology `Singularity` to package our software together with all its dependencies [KG17] [KcB+21]. This allows us to circumvent a common problem with the usage of HPC systems. Namely, the users usually lack permission to install additional software on the HPC system themselves. Containers solve this issue by providing a pre-built environment containing libraries and configurations the user needs to execute their application. Unlike virtual machines, containers share the operating system kernel with the host machine, making them a lightweight solution with less performance overhead. `Singularity`, in particular, was designed specifically for scientific and high-performance computing. It is therefore able to easily utilize HPC-specific software and hardware components like MPI or GPUs.

To create our container, we use a multi-stage build process (Figure 4), with separate build and runtime stages. We install all the necessary dependencies to compile the application in the build stage. The runtime stage only includes the dependencies required to run the software. By excluding the build tools we ensure that the container is as lightweight as possible while still containing all the necessary components for accurate and reliable results.

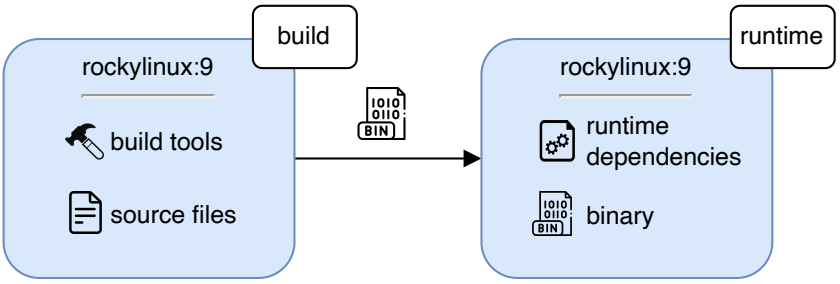

Figure 4: Multistage build process in Singularity

MPI Integration

When it comes to integrating with MPI, two different approaches can be applied (Figure 5) [App]. The first approach is the "Hybrid Model" in which MPI is installed in both the container and the host system. When launched, the host's MPI instance will communicate with the MPI installation inside the container.

The second approach is the "Bind Model" in which the host's MPI installation is mounted into the container. The latter approach is more performant as there is less communication overhead. However, the solution is less portable and, therefore, makes it harder to reproduce results as the container is not executable on its own anymore due to the lack of MPI installation.

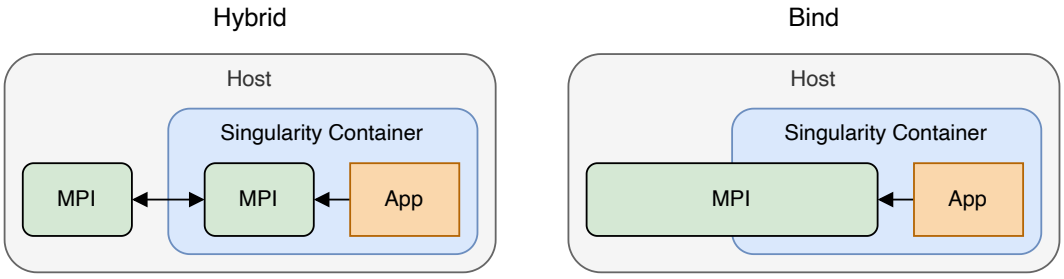

Figure 5: Comparison of the Singularity Hybrid- and Bind-Model

Container definition

In the following, we demonstrate the definition file for a `Singularity` container using the MPI Bind Model. The definition specifies two stages, *build* and *runtime*, with the base image *rockylinux*, which was used due to being the unofficial successor to the discontinued *CentOS* that runs on the HPC cluster used for testing. In the first section, *build*, we use the *%files* section to copy the source files of the example *C++* application to the container file system. The following *%post* section installs the build dependencies and compiles the application in a *build* directory.

```
BootStrap: docker
From: rockylinux:9
Stage: build

%files
    laplace2d/src src
    laplace2d/CMakeLists.txt CMakeLists.txt
%post
    yum update -y                                          && \
    yum group install -y "Development Tools"               && \
    yum install -y mpich mpich-devel cmake                 && \
    source /etc/profile.d/modules.sh && module load mpi && \
    mkdir build && cd build && cmake .. && make
```

In the *runtime* stage, we first copy the *build* directory from the previous stage. The *%post* section then installs the runtime dependencies for the application. Note that we do not install MPI during this step, as we are using the *bind-model* for this container. The *%environment* section adjusts the *PATH* and *LD_LIBRARY_PATH* environment variables so that the mounted MPI from the host system can be found when executing the container. Finally, the *%apprun* section specifies the application to be launched.

```
BootStrap: docker
From: rockylinux:9
Stage: runtime
%files from build
    build build

%post
    yum update -y          && \
    yum install -y gcc-toolset-12 compat-libgfortran-48

%environment
    export MPI_DIR=/cluster/mpi/mpich
    export PATH="$MPI_DIR/bin:$PATH"
    export LD_LIBRARY_PATH="$MPI_DIR/lib:$LD_LIBRARY_PATH"

%apprun laplace
    /build/bin/laplace
```

GitLab CI job

Using the definition file from the previous section, we can define a job that builds the container in a CI pipeline on every commit to the repository. We use a Docker image with `Singularity` installed to run the job. To build the `Singularity` container, we run `Singularity`'s *build* command in the *script* section. The generated container is then stored as an artifact for use in later jobs. Although omitted in this demonstration, it is a good practice to also cache the container to prevent expensive rebuilds.

```
build-singularity-mpich-bind:
  stage: build
  image:
    name: quay.io/singularity/singularity:v3.10.4
    entrypoint: [""]

  artifacts:
    paths: ["mpich-bind.sif"]

  script:
    - singularity build mpich-bind.sif mpich-bind.def
```

5 Connecting to an HPC platform using `HPC-Rocket`

Scientific research often involves the use of complex software tools to analyze data and perform simulations. Such software may require specific hardware and computational resources to run efficiently, making it impractical to execute on regular workstations. High-performance computing clusters provide the necessary computational power to run such simulations. Still, users of these clusters may not have the permissions required to install the software necessary to integrate the cluster with continuous integration service.

To address this challenge, we develop `HPC-Rocket` [ML23], a command-line application written in Python. `HPC-Rocket` aims to bridge the gap between CI services and HPC clusters, allowing users to execute large-scale simulations on remote clusters without the need for extensive permissions. Being a simple command-line application, `HPC-Rocket` can easily be used in any continuous integration environment, therefore making it more portable than other solutions that integrate directly with the CI platform (e.g. Jacamar CI [Exa]).

`HPC-Rocket` connects to a remote cluster via SSH and copies the files necessary for the software execution. It then executes a specified job script via the Slurm scheduling system, which is widely used in HPC clusters. Finally, the result data can also be collected and transferred back to the machine that is executing the CI pipeline for further investigation. Figure 6 shows an activity diagram of the `HPC-Rocket` workflow.

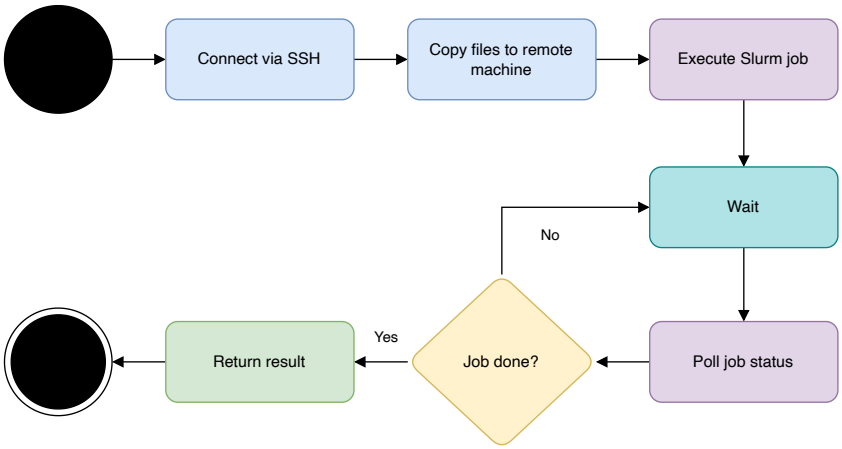

Figure 6: Activity diagram of `HPC-Rocket`

`HPC-Rocket` can be easily configured using a specific file written in the YAML file format that is also commonly used for the definition of CI pipelines. The configuration file contains the address and credentials in the form of an SSH key or password for the target machine. Moreover, `HPC-Rocket` supports the usage of environment variables, therefore allowing easy integration with secret stores provided by the respective CI services. In case the remote cluster is only accessible from a specific network `HPC-Rocket` is also capable of tunneling SSH commands

through multiple proxy jumps. Additional sections describe file copying, collection, and cleaning instructions. The final setting specifies which file should be passed to the Slurm scheduling system. A configuration that works with the container produced with the GitLab CI job in section 4 is presented in the following listing:

```
host: $REMOTE_HOST
user: $REMOTE_USER
password: $REMOTE_PASSWORD

copy:
  - from: laplace-mpich-bind.job
    to: laplace2d-mpich-bind/laplace.job

  - from: mpich-bind.sif
    to: laplace2d-mpich-bind/mpich-bind.sif

collect:
  - from: laplace2d-mpich-bind/results/*
    to: results

  - from: laplace2d-mpich-bind/laplace.out
    to: results

sbatch: laplace2d-mpich-bind/laplace.job
```

GitLab CI job

To run the configuration given in the previous section, we define a new CI job using a Docker image with Python installed. Since it requires the previously produced `Singularity` container, it depends on the CI job defined in section 4 as can be seen in the *needs* section of the job.

Initially, `HPC-Rocket` is installed in the Docker container running in the CI pipeline using Python's package manager *pip*. There is no need to install any software on the HPC cluster itself.

The *script* section then executes `HPC-Rocket` with the given configuration. Finally, the results produced by the simulation will be copied from the HPC cluster back to the CI pipeline and uploaded as an artifact to be verified in the next CI job using the software `Fieldcompare` as described in the next section.

```yaml
run-hpc-cluster-mpich-bind:
  image: python:3.10

  stage: simulation

  needs:
    - build-singularity-mpich-bind

  before_script:
    - pip install hpc-rocket==0.4.0

  script:
    - hpc-rocket launch rocket-mpich-bind.yml

  artifacts:
    paths:
      - results/
```

6 Regression-testing with `Fieldcompare`

Testing is a fundamental part of software quality assurance, and a comprehensive test suite is crucial to verify a software's *correctness*. By testing all parts of a system and their interoperability, the developers make sure the software behaves in correspondence with their intentions. Research software, particularly those designed for numerical simulations, is frequently used for the analysis and exploration of physical systems. This makes testing more difficult due to the lack of a known expected behavior to test against.

However, with *regression-testing* one may ensure that changes to the code do not introduce any undetected and unintentional changes to the software's behavior (bugs). In this technique, the results produced by the software are compared against stored reference results, for instance, from computations with an earlier version of the software. If significant deviations are detected between the results, the test suite is considered to have failed, thereby notifying the developers that a particular code modification led to a change in behavior. What a *significant* deviation is has to be decided for each test individually, and tolerances have to be chosen such that physically relevant deviations are detected while avoiding false positives from machine precision issues.

In this workflow, we employ `Fieldcompare` [GKP+23], a regression-testing tool supporting multiple *VTK* and several other mesh based formats, to detect deviations between simulation results produced in the CI pipeline and stored reference results (see Figure 7).

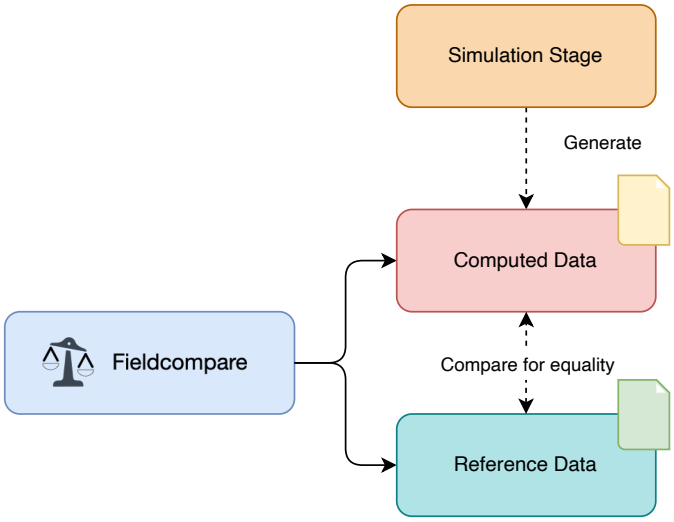

Figure 7: Compare computed data from a simulation stage with pre-computed reference data utilizing `Fieldcompare`

`Fieldcompare` offers several commandline options to control relative and absolute tolerances to determine whether deviations in results are considered significant. The default behavior for allowed deviations is rather strict but suffices for the demonstrated example workflow. The CI configuration for the *regression-test* job is given in the following listing:

```
regression-test:
  image: python:3.10

  stage: test

  needs: ["run-hpc-cluster-mpich-bind"]

  before_script:
    - pip install "fieldcompare[all]"

  script:
    - |
      fieldcompare file results/TemperatureField.avs \
                        reference_data/TemperatureField.avs
```

7 Workflow implementation for a regression test suite

In the presented sections Virtualisation with `Singularity`, Connecting to an HPC platform using `HPC-Rocket` and Regression-testing with `Fieldcompare` we described the single components of our workflow and how the CI jobs can be composed. In this section, we describe how to dynamically implement a complete regression test suite for our sample application (see section 3) that can easily be extended with new test cases. This approach is optional and the steps introduced in the previous chapters can of course be used by themselves. While this dynamic generation of additional CI jobs adds more complexity to the pipeline setup, it allows developers less familiar with CI pipeline technologies to add new test cases without modifying the CI configuration.

To run our regression tests, we implemented the Python module `jobgeneration` to dynamically generate a new CI workflow. The Python module will automatically pick up any file in a *test* directory ending in *-test*. Using a template file, new GitLab CI jobs with `HPC-Rocket` and `Fieldcompare` instructions will be generated for each of the collected test files. The first job *create-test-ci* is responsible for creating the YAML file `regression-test-ci.yml` for the new CI pipeline by executing the *jobgeneration* module and will upload the result as an artifact.

```yaml
create-test-ci:
  image: python:3.10

  stage: test

  before_script:
    - pip install -r jobgeneration/requirements.txt

  script:
    - python3 -m jobgeneration test

  artifacts:
    paths:
      - generated/
```

A second job, *trigger-test-ci*, downloads the CI pipeline file generated by the previous job and uses it to launch a child pipeline for the regression tests.

```yaml
trigger-test-ci:
  stage: test
  needs:
    - create-test-ci

  trigger:
    strategy: depend
    include:
      - artifact: generated/tests-ci.yml
        job: create-test-ci

  variables:
    PARENT_PIPELINE_ID: $CI_PIPELINE_ID
```

The creation of the new CI pipeline file is realized by evaluating a jinja template. For each test scenario, two jobs are created. The first job runs the actual test scenario using `HPC-Rocket` while the second job performs the verification with `Fieldcompare`. A shortened version of the template is shown in the listing below:

```
{% for test_case in test_cases %}
run-{{ test_case }}:
    stage: simulation

    script:
    - hpc-rocket launch --watch tests/rocket.yml

    # ...

verify-{{ test_case }}:
    stage: verify
    needs: ["run-{{ test_case }}"]

    script:
    - |
      fieldcompare file results/TemperatureField.avs \
                        reference_data/{{ test_case }}.avs

    # ...
{% endfor %}
```

Adding a new test requires the addition of a Slurm file to the *tests* subfolder. The Python *jobgeneration* script simplifies the process further by automatically scanning through all the Slurm scripts and generating two new CI jobs to execute `HPC-Rocket` and `Fieldcompare` as described above.

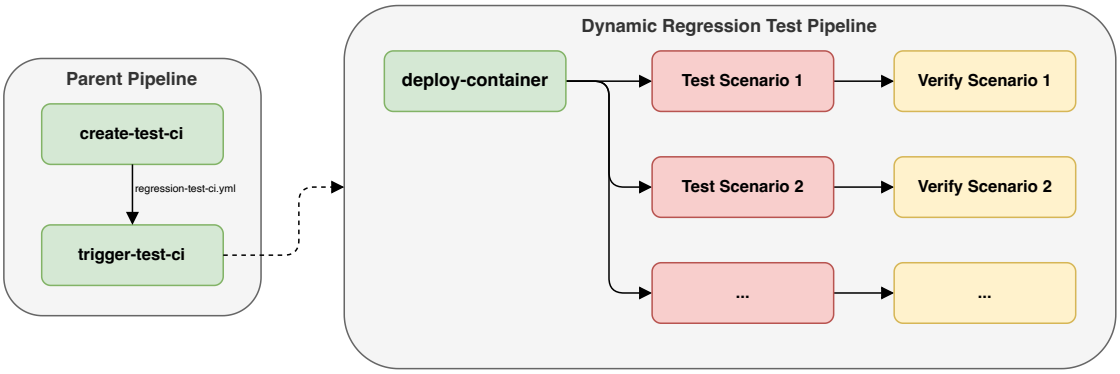

Figure 8: Dynamically generated child CI pipeline

8 Conclusion

The role of software in scientific progress has increased significantly during the past decades. Nevertheless, researchers still lack knowledge of software development and are confronted with additional complexity when dealing with high-performance computing environments. Therefore, they often face difficulties in simultaneously addressing the reproducibility and performance needs of computational tasks.

Our paper presents an easily repeatable workflow leveraging Continuous Integration pipelines to address the challenges of consistently running scientific computations in high-performance computing environments. We have established a comprehensive and reusable solution by incorporating `Singularity` to build containers with a reproducible environment, `HPC-Rocket` to execute computations on HPC clusters directly from CI pipelines, and `Fieldcompare` for robust regression testing against trusted reference data.

Through the integration of automation and containerization, our workflow ensures consistent and reliable execution of scientific tasks, enhancing reproducibility and reducing the likelihood of human-induced errors.

The workflow has already been incorporated into the advanced research code VirtualFluids. This software serves as a Computational Fluid Dynamics solver and is specifically designed to address a wide range of complex flow problems. These include turbulent, multiphase, and multi-component flows, as well as multi-field problems like Fluid-Structure Interaction [KGK18]. Due to the computationally intensive nature of the numerical method, VirtualFluids has been devel-

oped to shift the costly calculations to both multi-CPU architectures and multiple GPGPU accelerators. This workflow was employed to execute a set of regression tests on a local HPC cluster to validate the consistent behavior of the VirtualFluids software. A current implementation of this workflow can be found in the newest software publication of VirtualFluids [KSG+23].

Software and Data Availability

The software and code examples shown in the paper are developed under the MIT license and are available on GitHub at https://github.com/TUBS-Suresoft/suresoft-hpc-workflow. All releases of the software are published at Zenodo https://doi.org/10.5281/zenodo.7568959. This paper is based on Version 0.1.0 of this software publication.

Acknowledgment

The work has been done as part of the SURESOFT project [Tec22] funded by the German Research Foundation (DFG) as part of the "e-Research Technologies" funding program under grants: EG 404/1-1, JA 2329/7-1, KA 3171/12-1, KU 2333/17-1, LA 1403/12-1, LI 2970/1-1 and STU 530/6-1.

Bibliography

[App] Apptainer project. Apptainer and MPI applications. Accessed: 2023-03-14. https://apptainer.org/docs/user/latest/mpi.html

[BCH+17] A. Brett, M. Croucher, R. Haines, S. Hettrick, J. Hetherington, M. Stillwell, C. Wyatt. Research Software Engineers: State of the Nation Report 2017. Apr. 2017. doi:10.5281/zenodo.495360 https://doi.org/10.5281/zenodo.495360

[Ber14] D. Bernstein. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing* 1(3):81–84, 2014. doi:10.1109/MCC.2014.51

[DMG07] P. Duvall, S. M. Matyas, A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Signature Series. Addison-Wesley, Upper Saddle River, NJ, 2007. http://my.safaribooksonline.com/9780321336385

[Exa] Exascale Computing Project. Jacamar CI. https://gitlab.com/ecp-ci/jacamar-ci

[GKP+23] D. Gläser, T. Koch, S. Peters, S. Marcus, B. Flemisch. fieldcompare: A Python package for regression testing simulation results. *Journal of Open Source Software* 8(81):4905, 2023.

doi:10.21105/joss.04905
https://gitlab.com/dglaeser/fieldcompare

[KcB+21] G. M. Kurtzer, cclerget, M. Bauer, I. Kaneshiro, D. Trudgian, D. Godlove. hpcng/singularity: Singularity 3.7.3. Apr. 2021.
doi:10.5281/zenodo.4667718
https://doi.org/10.5281/zenodo.4667718

[KGK18] K. Kutscher, M. Geier, M. Krafczyk. Multiscale simulation of turbulent flow interacting with porous media based on a massively parallel implementation of the cumulant lattice Boltzmann method. *Computers & Fluids*, 2018.
doi:10.1016/j.compfluid.2018.02.009

[KSG+23] K. Kutscher, M. Schönherr, M. Geier, M. Krafczyk, H. Alihussein, J. Linxweiler, S. Peters, A. Wellmann, H. Safari, S. Marcus, H. Asmuth, H. Korb. VirtualFluids. Dec. 2023.
doi:10.5281/zenodo.10283049
https://doi.org/10.5281/zenodo.10283049

[KG17] B. M. Kurtzer GM, Sochat V. Singularity: Scientific containers for mobility of compute. *PLoS ONE* 5(12), 2017.
doi:10.1371/journal.pone.0177459
https://doi.org/10.1371/journal.pone.0177459

[Mer10] Z. Merali. Computational science: Error, why scientific programming does not compute. *Nature* 467(7317):775–777, Oct. 2010.
doi:10.1038/467775a

[ML23] S. Marcus, J. Linxweiler. hpc-rocket. Jan. 2023.
doi:10.5281/zenodo.7545130
https://github.com/SvenMarcus/hpc-rocket

[Tec22] *SURESOFT: Towards Sustainable Research Software*. Technische Universität, Braunschweig, 2022.
doi:10.24355/dbbs.084-202210121528-0
https://leopard.tu-braunschweig.de/receive/dbbs_mods_00071451

[ZC08] D. G. Zill, M. R. Cullen. *Differential Equations with Boundary-Value Problems*. Volume 7. Cengage Learning, 2008.