Detecting disaster before it strikes: On the challenges of automated building and testing in HPC environments

Christian Feld, Markus Geimer, Marc-André Hermanns, Pavel Saviankou, Anke Visser, and Bernd Mohr

Abstract Software reliability is one of the cornerstones of any successful user experience. Software needs to build up the users' trust in its fitness for a specific purpose. Software failures undermine this trust and add to user frustration that will ultimately lead to a termination of usage. Even beyond user expectations on the robustness of a software package, today's scientific software is more than a temporary research prototype. It also forms the bedrock for successful scientific research in the future. A well-defined software engineering process that includes automated builds and tests is a key enabler for keeping software reliable in an agile scientific environment and should be of vital interest for any scientific software development team.
While automated builds and deployment as well as systematic software testing have become common practice when developing software in industry, it is rarely used for scientific software, including tools. Potential reasons are that (1) in contrast to computer scientists, domain scientists from other fields usually never get exposed to such techniques during their training, (2) building up the necessary infrastructures is often considered overhead that distracts from the real science, (3) interdisciplinary research teams are still rare, and (4) high-performance computing systems and their programming environments are less standardized, such that published recipes can often not be applied without heavy modification.
In this work, we will present the various challenges we encountered while setting up an automated building and testing infrastructure for the Score-P, Scalasca, and Cube projects. We will outline our current approaches, alternatives that have been considered, and the remaining open issues that still need to be addressed—to further increase the software quality and thus, ultimately improve user experience.

C. Feld · M. Geimer · M.-A. Hermanns · P. Saviankou · A. Visser · B. Mohr
Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, 52425 Jülich, Germany
e-mail: `{c.feld,m.geimer,m.a.hermanns,p.saviankou,a.visser,b.mohr}@fz-juelich.de`

1 Introduction

Software reliability is one of the cornerstones of any successful user experience. Software needs to build up the users' trust in its fitness for a specific purpose for it to be adopted and used in a scientific context. Software failures, at any stage of its use, will add to user frustration and ultimately lead to a termination of usage. Furthermore, most scientific software packages are not only provided to the community to enable scientific exploration, but also form the foundation of research for the developers as well. If software stability is diminished, so is the capability to build reliable prototypes on the available foundation.

With the increasing complexity of modern simulation codes, ensuring high software quality has been on the agenda of the computational science community for many years. Post and Kendall derived lessons learned for the ASCI program at Los Alamos and Lawrence Livermore National Laboratories [40]. Among other factors for successful simulation software engineering, they recommend to use "modern but proven computer science techniques", which means not to mix domain research with computer science research. However, mapping modern software engineering practices to the development of scientific simulation codes has proven difficult in the past, as those practices focus on team-based software development in the software industry, which is often different from scientific code development environments (e.g., the "lone researcher" [20]). As Kelly at al. found, domain researchers outside of computer science may also perceive software engineering practices not as essential to their research but rather as incidental [34] and being in the way of their research progress [33]. Nevertheless, Kelly at al. do stress the importance of strategic software testing to lower the barrier for the introduction and maintenance of software tests in scientific projects [30].

While the adoption of rigorous software testing has not yet found broad adoption among scientific software developers, some development teams do employ techniques and supporting infrastructures already today to a varying degree [16, 21, 26, 38, 39]. However, software testing may not only benefit a single software package at hand, but can also contribute to the assessment of larger, diverse software stacks common on HPC platforms [32]. Still today, such work oftentimes entails the integration of different independent software components to fit all project needs, or the development of new software frameworks to reduce the overhead of maintaining and increasing the quality of a specific research software project.

In this spirit, our work integrates and extends available practices and software components in the context of the constraints given by our own research projects (e.g., time, manpower, experience) and may prove to be adaptable in parts to other scientific software projects. As Figure 1 shows, our software ecosystem consists of different tools and software components with various dependencies between them. Score-P [35] is a highly scalable and easy-to-use instrumentation and measurement infrastructure for profiling, event tracing, and online analysis of HPC applications. It internally uses OPARI2, a source-to-source instrumenter for OpenMP constructs. Scalasca [27, 44] is a set of tools for analyzing event traces and identifying potential performance bottlenecks—in particular those concerning communication and

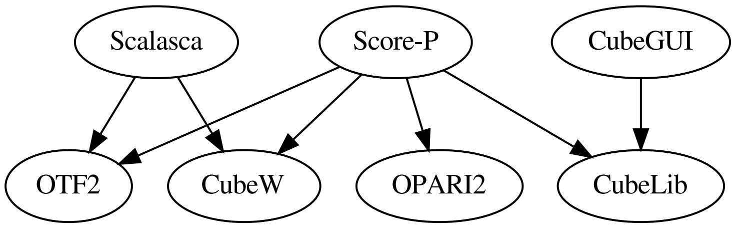

Fig. 1 Our tools and software components and their build dependencies.

synchronization. The Cube components [41] operate on `cubex` files and consist of (1) CubeW – a C library for writing, (2) CubeLib – a C++ library for writing and reading plus a set of command-line tools, and (3) CubeGUI – a graphical performance report explorer. Last but not least, we provide OTF2 [25], a highly scalable and memory efficient event trace data format and library.

It is evident that any severe quality degradation in any of the components above easily affects multiple other components. For example, a defect in the OTF2 component may directly affect Score-P's ability to create new trace measurements and Scalasca's ability to process existing or new trace archives. Furthermore, indirect dependencies may impact the tools as well. For example, if the CubeGUI component experiences a severe regression, exploration of Cube profiles—as generated by Score-P and Scalasca—must fall back on earlier versions of the software. The automated building and testing setup described in this paper is a direct consequence of this interdependence among our software components, in order to spot potential problems early in the development process.

Besides the description of early approaches to automated building and testing for our tools, the contributions of this work include:

- A workflow for using GitLab CI/CD in an HPC environment and for sources hosted in an external Subversion repository.
- An example for building a test suite for integration testing using the JUBE workflow manager.
- An extension of GNU Automake's *Simple Tests* rule to support programming models that require launchers and environment variables to be set.
- Custom TAP printer extensions to the Google Test unit testing framework—including support for MPI tests—for better integration with GNU Automake.

The rest of this paper is organized as follows. Section 2 introduces continuous builds as a quality assurance measure and discusses the history of our automated build setup. It also gives a detailed account of our current implementation, based on the GitLab continuous integration / continuous delivery (CI/CD) framework. Next, Section 3 extends the discussion to automated testing and provides examples of our current testing approaches on various levels. Here, Section 3.3 highlights our systematic integration testing framework for the Scalasca parallel analyzer. Finally, Section 4 concludes this work and provides an outlook on the next steps envisioned for further software quality assurances for our codes.

2 Continuous Builds and Delivery

For any compiled software package, ensuring that the source code can be transformed into an executable can be seen as a very first step toward a positive user experience even before actually using it. Before automated builds were introduced to the Scalasca project, the approach to ensure that the code base compiled and worked on a wide range of HPC machines was a *testathon*, carried out just before releases. Every member of the development team tried to compile the current release candidate on the HPC machines she had access to and collected a few small-scale measurements from simple test codes as a sanity check. Portability bugs showed up just at this late stage in the development process. And since addressing a portability issue with one platform or programming environment might have introduced a new problem with another, team members had to start over multiple times due to the creation of new release candidates. While our experience with this manual approach showed that it was already valuable in order to identify the most serious problems before publishing a release, it was nevertheless quite cumbersome.

2.1 History

Therefore, the Scalasca project decided to set up an infrastructure for automated nightly builds about a decade ago. The first nightly build of the Scalasca 1.x code base—which also included the full Cube package—ran on a developer workstation in January 2009. The builds were carried out using a set of home-grown shell scripts triggered by a cron job, which set up the build environment (e.g., by loading required environment modules), checked out the main development branch of the code base from Subversion, ran `configure`, `make`, and `make install`, and finally sent an e-mail to the developer mailing list on failure. Even the very first set of builds already exercised three different compilers (GCC, Intel, Sun), but was based on only a single MPI implementation. Over time, this setup was extended by also including builds with different MPI libraries, additional compilers, and covering 32- and 64-bit environments, as well as builds on the local supercomputers available at the Jülich Supercomputing Centre. Moreover, we enhanced the build scripts to extract compiler warnings from the build log and report them via e-mail.

While this nightly build infrastructure did not provide feedback for every single code revision that had been checked in, it still proved very helpful in identifying portability issues early on. Therefore, we were already convinced that some form of continuous build infrastructure—activated by every commit to our source code repository—was mandatory when starting development of the Score-P, OTF2, and OPARI2 projects. However, it quickly became clear that extending our build scripts to support the new projects would require a significant effort (i.e., basically a rewrite). Thus, the Score-P project partners decided to move away from self-written shell scripts toward a more widespread, community-maintained solution. As Score-P and its companion projects use Trac [24] as a minimalistic web-based project management tool, the Bitten [23] plug-in seemed like a natural choice to implement continuous builds. Subsequently, also Scalasca 2.x and the now fully stand-alone Cube project adopted Bitten to implement their continuous build infrastructure.

Bitten consists of two components: a plug-in for the Trac project environment running on the server side and a Python-based client that needs to be executed on the build machines. The configuration has to be done on both sides. On the server side, *build configurations* are defined. A build configuration listens on commits to specific Subversion paths, defines a build recipe, and a set of *target platforms*. Here, a target platform is a named set of rules against which the *properties* of build clients are matched. In our setup, the target platforms were used to match against environment module and configure options, for example, whether to build static or shared libraries, which compilers and MPI libraries to use, etc. The client side basically consists of a shell script with given properties that is executed regularly (e.g., via a cron job) on the build machines. The script sets up the build environment depending on its properties and then executes the Bitten client. This client connects to the Trac server running the Bitten plug-in providing its properties, and queries whether any commits that satisfy all of the rules associated with the target platform are pending. In this case, matching commits are processed according to the configuration's build recipe.

The Bitten approach to continuous builds has been used over several years for the projects mentioned above. Although the builds were carried out as expected and gave us valuable feedback, we were not fully satisfied with this solution. Building the code base directly from the version control system required that every build client had the build system tools (i.e., specific versions of GNU Autotools [19, 43]; sometimes with custom patches to address the peculiarities of HPC systems) to be installed. Also, the need to maintain the configuration in two places—on the server as target platforms and on the clients as properties—turned out to be tedious and error-prone. For example, every new Subversion branch that should be built continuously required to manually define a new configuration from scratch on the server, as the Bitten plug-in does not provide an option to copy an existing configuration. Moreover, the client configurations on potentially all build machines had to be adapted accordingly.

Over time, we developed the desire to specify dependencies between builds, that is, to create build *pipelines*. On the one hand, this was motivated by the fact that a build failure caused by a syntax error or a missing file rather than a portability issue triggered failure e-mails from every build client, thus polluting the e-mail inboxes of all developers—one couldn't see the wood for the trees. On the other hand, we wanted our tests to be closer to a user's perspective, that is, building from a tarball instead of directly from the version control system. In addition, we wanted to make a successfully built tarball publicly available. Thus, a pipeline that would meet our requirements consists of several stages:

- The first stage builds the code base in a single configuration straight from the repository sources and generates a distribution tarball. Only this initial build requires special development tools, in our case specific and patched versions of

GNU Autotools and Doxygen [7]. Build failures in the generic parts of the code base are already detected during this step and therefore only trigger a single e-mail.

- The next stage performs various builds with different configurations to uncover portability issues, using the generated tarball and the development environments available on the build machines. This corresponds to what a user would experience when building a release.
- A subsequent stage makes the tarball publicly available once all builds report success.
- An additional stage could implement sanity checks between dependent projects, for example, to detect breaking API changes as early as possible. Also, this stage could trigger automated tests to run asynchronously after successful builds.

With the pipeline outlined above, our infrastructure would not only do *continuous builds*, but *continuous delivery* (CD) [31], by ensuring that the latest version of our software can be released as a tarball at any time, even development versions. Please note that our development process is based on feature branches that get integrated into the mainline after review. Thus, our process is currently not based on *continuous integration* (CI) [17]—the practice of merging all developer working copies to a shared mainline several times a day.

As Bitten has no built-in support for pipelines, we experimented with emulating this feature via build attachments and non-trivial shell scripts. However, this setup quickly became more and more complex, and added to the maintenance worries we already had. Moreover, the fact that Bitten had not been actively maintained for the last couple of years also did not increase our trust in this tool.

All in all, we felt a pressing need to find a replacement that lowers the maintenance burden, provides support for build pipelines, and is actively maintained. As a requirement for the replacement, the configuration of the entire infrastructure should allow for easy integration of new clients and new repository branches. This would allow us to easily adjust the number of (HPC) machines that take part in the CD effort. We could also invite users with access to new and exotic systems to become part of our CD infrastructure, just by installing and configuring a client on their side, if they don't have concerns executing our build scripts on their machine. Besides clients running on local servers and login nodes of HPC machines, the possibility of running containerized clients would provide an easy way to improve the coverage of operating systems and software stacks under test. An easy integration of new branches into the CD infrastructure is also considered crucial; setting up CD for a new release branch should be matter of minutes rather than the cause of anxiety.

We did a superficial evaluation of a few available solutions to check whether they would meet our requirements:

- **Jenkins** [11] – The configuration of a Jenkins server requires the use of plug-ins. One can run builds on remote servers using SSH plug-ins. Here the server connects to the client, thus the server's public SSH key needs to be added to the `authorized_keys` file on the client side. From a security point of view this is a little worse than a client connecting to the server. We will address the security related issues in Section 2.3 below. Moreover, all projects hosted by a Jenkins instance by default share the SSH key setup, which is a huge impediment for providing a central Jenkins service also covering projects outside of our ecosystem. At the time of writing, this limitation could only be overcome using a commercial plug-in. Furthermore, the vast number of plug-ins and the available documentation did not guide us to a straightforward configuration of the server, but felt like a time-consuming trial-and-error endeavor.

- **Travis** [14] – Travis CI is a continuous integration service for GitHub [9] projects. Although we envision a transition from Subversion to git for our code base, it is not clear if the code will be hosted on GitHub. Moreover, builds can only be run within containerized environments on provided cloud resources, that is, testing with real HPC environments would not be possible. Therefore, we did not investigate this option any further.
- **GitLab CI/CD** [10] – The CI/CD component integrated into the GitLab platform seemed to best match our requirements. It supports server-side configuration in a single location, build pipelines, and the generation of web pages. Moreover, new build clients can easily be set up by copying a statically linked executable (available for x86, Power, and ARM), and registering the client with the server only once.

Considering the results of this quick evaluation, we decided to replace the Bitten-based continuous builds by GitLab CI/CD, starting with the Cube project. The fact that GitLab allowed self-hosting of projects already stirred interest for it as a replacement for the multi-project Trac server run by our institute. This has certainly influenced our decision to investigate its capabilities early on. While migrating the Cube project, it quickly turned out that GitLab CI/CD was as good match for our requirements. We then moved to GitLab CI/CD also for our remaining projects (Score-P, Scalasca 2.x, OTF2, and OPARI2).

2.2 GitLab CI/CD

For every project to be continuously delivered[1]—that is, Score-P, Scalasca 2.x, Cube, OTF2, and OPARI2—we created a corresponding GitLab project, each providing an associated git repository. However, as mentioned before, all the components listed above are currently still hosted in Subversion repositories, and migrating everything to git was not an option within the available time frame. Thus, the GitLab project is currently only used to provide the CD functionality, and the git repository to store the configuration of the CD system. Yet, this required us to find a way to trigger GitLab CI/CD actions from Subversion commits. Our solution involves three parties: the Subversion repository, an intermediary GitLab project, and the GitLab CI/CD project responsible for the continuous builds.

[1] From the perspective of setting up and configuring the infrastructure, there is no real distinction between continuous integration and continuous delivery.

On the Subversion side, a post-commit hook collects information about each commit, for example, path, revision, author, commit message, etc. Using GitLab's REST API, this data is passed to the CI/CD pipeline of the intermediary GitLab project, whose sole purpose is to match the Subversion commit's path with a branch in the GitLab CI/CD project.[2] In case of a match, the build recipe of the intermediary GitLab project commits the Subversion data it received from the post-commit hook to a file in the matching GitLab CI/CD branch, thereby providing all information necessary to access the correct Subversion path and revision from within build jobs of this GitLab CI/CD branch. Only then the build pipeline for the matching branch is triggered explicitly from the intermediary project, again using a REST API call. While the functionality of the intermediary GitLab project's CD recipe could also be included in the Subversion post-commit hook, this approach decouples the trigger from the actual branch mapping. This allows for convenient changes using a git repository without the need to update the post-commit hook on the Subversion server.

With GitLab CI/CD, the entire configuration is specified in a single file named `.gitlab-ci.yml` that resides inside a git branch. This file defines a *pipeline*, which by default is triggered by a new commit to this branch. A pipeline consists of an ordered set of *stages*. Each stage comprises one or more *jobs*, with each job defining an independent set of commands. A stage is started only after all jobs of the previous stage have finished. Individual jobs are executed by *runners* (e.g., build clients), which have to be registered once with the GitLab server. Runners can either be specific to a single project or shared among multiple projects. Optionally, a list of *tags* (arbitrary keywords) can be associated with a runner at registration time. This way it can be restricted to only execute jobs that exclusively list matching tags in their job description. Communication between stages beyond success and failure, which is provided by default, can be achieved via per-job *artifact* files. Artifact files of jobs are automatically available to all jobs of subsequent stages.

In our current implementation, the CI/CD pipeline consists of the following five stages: (1) creating a distribution tarball, (2) configuring, building, and testing the tarball with different programming environments, (3) evaluating the build results and sending out e-mail notifications if necessary, (4) preparing tarball delivery, and (5) generating web pages.

The initial, single-job *create_tarball* stage first checks out the corresponding project's source code from Subversion, leveraging the commit information provided by the intermediary GitLab project as outlined above. It then configures and builds the sources in a single configuration (currently GCC/Open MPI), generates the user documentation, and then creates a self-contained distribution tarball. Finally, this tarball is uploaded to the GitLab server as a job artifact. As this is the only stage working with a checkout from a version control system, special development tools such as GNU Autotools or Doxygen are only required during this step.

Since job artifacts like the distribution tarball are readily available in subsequent stages, the follow-up *test_tarball* stage can use it in the same way a user would

[2] For example, all commits to Subversion `trunk` matches git `trunk`, and commits to Subversion `branches/RB-4.0` matches git `RB-4.0`.

build and install our released versions. This stage comprises multiple jobs, each testing the tarball with a specific configuration or on a particular HPC platform. Each job executes the `configure` script with appropriate options, runs `make` and `make install`, and triggers a number of automated tests (see Sections 3.1 and 3.2). We currently cover configurations using different compilers (GCC, PGI, Intel, Cray, IBM XL, Fujitsu), diverse MPI implementations (Open MPI, SGI MPT, MVAPICH2, Intel MPI, Cray MPI, Fujitsu MPI), multiple architectures (x86, Power, SPARC, ARM), HPC-specific programming environments (Cray XC, K computer), and several configuration options (e.g., with internal/external subcomponents, whether to build shared or static libraries, or with PAPI, CUDA, or OpenCL support enabled).

The *create_tarball* and *test_tarball* stages also upload additional build artifacts to the GitLab server, for example, the build log in case of failure. The subsequent *evaluate* stage analyzes these artifacts from the previous stages and, in case of an error, determines which message to send to which audience via e-mail. Here, we distinguish four different error cases: (1) *create_tarball* took too long or did not start, (2) *create_tarball* failed, (3) one or more *test_tarball* jobs took too long or did not start, and (4) one or more *test_tarball* jobs failed. Examples for (3) are jobs that are supposed to run on remote machines where the machine is in maintenance or not accessible due to other reasons. These jobs are marked *allowed_to_fail*, which allows for subsequent stages to continue. Nevertheless, we are able to detect these jobs since they do not provide a specific artifact file created by successful/failing jobs. The error cases (1), (2), and (3) are communicated to the commit author and the CI/CD maintainer only, as they should be able to figure out what went wrong. This way we refrain from bothering the entire developer community. In contrast, case (4)—a real build failure—is communicated to a wider audience with the hope in mind that the community helps to fix the issue. Note that there is only a single failure e-mail sent out per pipeline invocation, summarizing all build failures and providing links to the individual build logs for further investigation.

In absence of a real build failure, the *prepare_delivery* stage is responsible for copying the distribution tarball, the generated documentation, and related meta-data to a shared directory. This directory collects the artifacts from the last *N* pipeline invocations for multiple branches.

The last stage, *generate_pages*, operates on this shared directory and creates a simple website that provides the tarballs and corresponding documentation for all successful builds it will find in the directory. This website is published via GitLab CI/CD's built-in feature *pages* and is publicly accessible.[3] This way bug fixes and experimental features are available to the interested audience soon after they have been committed and in a completely automated process.

With the entire configuration specified in a single file stored in a git repository, bringing a new branch under CD is now trivial: we just need to create a new branch from an existing one that is already under CD. As the configuration file lives inside the git branch, it is copied and will carry out jobs for the new branch as soon as

[3] See, e.g., `http://scorepci.pages.jsc.fz-juelich.de/scorep-pipelines/`

the CI/CD pipeline is triggered. However, care needs to be taken not to hardcode any paths (e.g., the installation prefix) or variables into the configuration file as this might lead to data races when pipelines from different branches run in parallel. To prevent this, GitLab CI/CD offers a set of predefined branch-unique variables that can be used inside the configuration file to make it race-free. Using these variables, a strict branch naming scheme, and GitLab CI/CD's feature to restrict jobs to matching branch names, we were able to create a single `.gitlab-ci.yml` file and associated shell scripts that can be used for any newly created branch.

In our setup, the GitLab CI/CD runners execute on the build machines (local servers dedicated for CD and testing as well as login nodes of multiple HPC systems we have access to) and identify themselves via tags. They regularly connect to the GitLab server to check whether there are any jobs waiting to be executed that match their tags. If this is the case, the git repository is cloned on the client side and the job is executed according to the job definition contained in `.gitlab-ci.yml`. For our projects, runners are started in the default *run* mode, creating a process that is supposed to run forever. However, this is problematic on HPC systems as there might be a policy in place to kill long-running login-node processes after a certain time period, usually without prior notice. As we want the runners to be operating continuously, we have implemented a kill-and-restart mechanism that preempts the system's kill. This mechanism is under our control and works as follows: when starting the runner, we schedule a `QUIT` signal after a given time period by prefixing the runner command with `timeout -s QUIT <period>`. If a runner receives this signal and is currently processing a job, it first will finish this job and then exit. Otherwise, it will exit immediately. To keep the runner operational, a cron job on the HPC machine or on one of our local test systems will regularly restart the runner, again scheduling the `QUIT` signal. This regular restart will be a no-operation if the runner is still alive.

With GitLab CI/CD and the Subversion-to-GitLab trigger mechanism, we now have a robust, low maintenance, and extensible continuous delivery infrastructure that is a vast improvement over the previous Bitten-based approach.

The source code of the Score-P and Scalasca projects already include dependent projects like OTF2, OPARI2, CubeW, or CubeLib. When building these parent projects, the subproject code is built as well, at least during the *create_tarball* stage. However, this does not hold for the—with regards to source code—independent Cube components that do not utilize the dependency management mentioned above. These components, except for CubeGUI, work directly on `cubex` files. As changes in one component might affect the ability to read and/or write `cubex` files in other components, care must be taken to keep them synchronized. This is ensured by an additional GitLab CI/CD project that is triggered from Subversion commits to any of the Cube components, also using the intermediary GitLab CI/CD project for branch matching described above. This additional GitLab CI/CD project starts with building all Cube components. On success, CubeW is used to generate `cubex` files with given contents. These files are passed to CubeLib, where they are read and the validity of the contents is verified. This way, we are able to detect early on when the components get out of sync.[4]

2.3 Open Issues

We need to mention that all build clients in both our previous Trac/Bitten and the current GitLab CI/CD setup run with user permissions. This is a potential security risk as all users with permission to commit to a Subversion repository may trigger a build pipeline that executes code on a system they might not have access to. The code that is executed are configure scripts and Makefile targets. Adding malicious code to these files is possible for every developer with commit rights. In our projects, however, only trusted developers have the right to commit and are therefore able to trigger builds. Moreover, new features undergo a thorough code review process so that malicious code is likely to be detected.

Compared to a manual build by a user, either using a tarball or sources checked out from a version control system's repository, we do not see a significant difference to automated builds with regards to security. Although a manual build would allow for an in-depth examination of the entire code base before executing a build, this is not feasible in general.

Another attack vector would be a malicious change to the `.gitlab-ci.yml` configuration file and associated helper scripts. In our case, write access to these files is only granted to a subset of the trusted developers with Subversion commit rights. We consider this setup *safe enough* for our purposes and the security concerns of HPC sites.

In contrast to GitLab CI/CD, Jenkins jobs are not initiated by a polling client, but a pushing Jenkins server. With access to the server it might be possible to log into the remote build machines, making it slightly easier to perform harmful activities.

As already mentioned, we plan to migrate our code base from Subversion to git, potentially hosted on a publicly accessible platform. Here, we might get merge requests from *untrusted* developers. To deal with such requests, we envision a three-step CD approach. In the first step we would run the *create_tarball* stage and some of the *test_tarball* jobs in a Docker-containerized environment [6]. On success, the second step would comprise a manual code review process. CD jobs on the HPC systems will be triggered only if this review has a positive outcome.

Unfortunately we cannot move all CD jobs into containers, as our target systems are real HPC machines with their non-standard setups and programming environments. To the best of our knowledge, containers for these site-specific setups do not exist, and it is beyond our abilities to create such container images ourselves. Thus, it would be of great help if HPC sites acknowledge the need for continuous building and testing, and make some dedicated resources available for this purpose, as well as assist in creating containerized environments of their respective setups.

[4] Instead of using an additional GitLab CI/CD project, the Cube component's individual GitLab CI/CD projects could trigger each other using GitLab CI/CD's REST API.

3 Automated Testing

While continuous builds are vital to get rapid feedback on whether a code base compiles with different programming environments and configurations, they do not ensure that the code actually works as expected. For this, tests have to be executed on various levels—ideally also in an automated fashion.

For automated testing we need to differentiate between tests that are executed on login nodes and tests that are supposed to run on compute nodes. While the former can be run directly in the build environment—if not cross-compiled—the latter pose a challenge as execution of compute-node tests might need a special and non-standardized environment and startup procedure like a job submission system. In Section 3.3 we present our approach to tackle this challenge with regards to the Scalasca project. The following two sections will describe how we approach tests that do not require a job submission system. As the build systems of our tools are exclusively based on GNU Autotools, we use the standard targets `make check` and `make installcheck` to execute these tests.

3.1 make check

The Scalasca project started its journey toward more rigorous and systematic testing in 2012. As an initial step, we surveyed a number of C++ unit testing frameworks with respect to their documentation, ease of use, feature sets, and extensibility. In particular, we evaluated the following unit testing frameworks: CppUnit [3], CppUnitLite [4], CxxTest [5], UnitTest++ [15], FRUCTOSE [8], CATCH[5], Boost.Test [1], and Google Test [29]. After weighing the strengths and weaknesses of the various solutions, we opted for Google Test to implement unit tests for Cube and Scalasca, as it seemed to best fit our needs. (Note that a re-evaluation would be necessary when starting a new project today, as the capabilities of the frameworks have evolved over the past years.)

All our unit tests are part of the main code base and triggered using the standard GNU Autotools `make check` target. However, the default test result report generated by Google Test is quite verbose and does not integrate well with the GNU Autotools build system. Therefore, we have developed a TAP [42] result printer extension, which is able to communicate the outcome of every single unit test to the Automake test harness rather than indicating success/failure on the granularity of test executables. In addition, details on failed tests (actual vs. expected outcome) are included in the test log as TAP comments. While TAP test support is not yet a first-class citizen in Automake, it only requires a one-off manual setup.

We also enhanced Google Test to support MPI-parallel tests. In this mode, the TAP result printer extension first collates the test results from each rank, and then prints the combined overall test result from rank 0. Such parallel tests are only run

[5] Original website no longer accessible; see [2] for the follow-up project.

when enabled during `configure` using the `--enable-backend-test-runs` option, as it is often not allowed to execute parallel jobs on the login nodes of HPC systems.[6] We also use them sparingly (e.g., to test a communication abstraction layer on top of MPI), as parallel tests are expensive. In addition to the enhancements outlined above, a minimal patch was required to make Google Test compile with the Fujitsu compilers on K computer.

The Score-P project took a different approach to tune the `make check` rule to fit its specific testing needs. As in the Scalasca project, tests that are supposed to run on compute nodes need to be explicitly enabled during `configure`. Besides that, Score-P comes with login-node tests that are always executed. We implement these tests using the standard Automake *Simple Tests* rule. While the login-node tests are purely serial programs, the compute-node tests consist of serial, OpenMP, MPI, MPI+OpenMP, and SHMEM programs. While serial compute-node programs do not need any special treatment, OpenMP programs require at least a reasonable value for `OMP_NUM_THREADS`, and MPI and SHMEM programs are started via `mpiexec` and `oshrun` launchers, also requiring a reasonable value for the number of ranks to be used. As the different startup mechanisms and specific environment settings could not be modeled by the standard Automake *Simple Tests* rule, we decided to slightly modify the existing rule for each programming model and combinations thereof. That is, we copied the default `make check` related rules together with their associated variables. This results in quite some code duplication[7], but is a straightforward and extensible way of implementing the required functionality. After copying, we made the duplicate target and variable names unique by adding programming model specific postfixes. As a next step, we modified minor portions of the code in order to set the required environment variables and to introduce standard program launchers. Finally, we use the standard `check-local` rule to trigger the new programming model specific tests. The mechanism described above allows us to stay entirely within the GNU Autotools universe and to use Automake's *Simple Tests* framework for arbitrary programming models. We need to stress that this extension of the `make check` rule is not supposed to be used with job submission systems with their asynchronous nature, but with standard, blocking launchers like `mpiexec` and `oshrun`.

The aforementioned CubeLib component also provides `make check` targets. As all Cube components work on `cubex` files, it is an obvious approach to try to compare generated files against a reference. However, this is not easily possible as `cubex` files are regular `tar` archives. These `tar` archives bundle the experiment meta-data with multiple data files. Since the experiment meta-data usually also includes creator version information, and `tar` archives store file creation timestamps, a simple file comparison against a reference solution using, for example, the `cmp` command is not feasible.[8]

[6] Note that all tests are built unconditionally during `make check`, and thus can be run on a compute node afterwards outside of the build system.

[7] Note that the copies might need to be updated for every new version of GNU Automake.

[8] The deprecated Cube v3 file format is a pure XML format and can be compared using `cmp`.

The CubeLib component is able to write and read `cubex` files and comes with a set of file manipulation tools. The test workflow here is as follows: we write `cubex` files using the writer API. The generated file is then processed by a CubeLib tool, using the reader API. The resulting file is compared against a reference using a special `cube_cmp` tool which overcomes the problems of comparing Cube's `tar` archives mentioned above.

3.2 make installcheck

The `installcheck` rule is used by the Score-P, OTF2, OPARI2, and Cube projects. It is supposed to work on an installed package as a user would see it. Amongst others, we use this target to verify that our installed header files are self-contained. To test this we created minimal test programs that only include individual header files of our package installations and check whether these programs compile. In addition, we have implemented a huge number of Score-P link tests which test one of the central components of the Score-P package, the `scorep` instrumentation command. It is used as a compiler prefix in order to add instrumentation hooks and to link additional libraries to the executable being built. This command comes with lots of options, libraries to be added, and libraries to be wrapped at link time. We need to ensure that every valid combination of options leads to a successfully linked application. We do this by building `scorep`-instrumented example programs covering nearly the entire valid option space[9] and inspecting them afterwards to verify that they were linked against the expected libraries. The run-time behavior of these instrumented programs is not tested here. For this we would need to execute the programs on compute nodes with their non-standard way of submitting jobs. Besides that, the nature of the test programs was not chosen to test for specific Score-P internals, but to provide a way of testing the compile- and link-time behavior of the `scorep` command. Furthermore, other Score-P login-node executables are accompanied by a few tests to check if their basic functionality works as expected. In case of the OPARI2 project, `make installcheck` just tests the basic workflow of the OPARI2 source-to-source instrumenter and checks if an instrumented *hello world* program can be compiled.

Cube components come with associated `cube*-config` tools. These tools provide compile and link flags a user needs to apply when building and linking against one of the Cube components. To ensure that these config tools provide the correct flags and paths, we reuse a subset of the `make check` tests also during `make installcheck`. Here, we are no longer interested in the functionality of the internals but whether a user could build an application linked against a Cube component. For that we replace the `make check` build instructions—provided by the Cube component's build system—with the ones provided by the already installed config tool.

[9] We choose this time- and disk-space-consuming *brute force* approach in a early stage of the Score-P project as it was the easiest to implement in a period of high code change rate.

The same approach is used to test the CubeGUI plug-in API. Here we build a *hello world* plug-in that exemplifies the entire API. Furthermore, the CubeGUI package does not provide any additional tests, in particular no tests for the graphical user interface itself, as these are difficult to automate.

3.3 Scalasca Testing Framework

As already outlined in Section 2, system testing of the Scalasca Trace Tools package traditionally has been done manually in a rather ad-hoc fashion before publishing a new release. In addition, it has been continuously tested implicitly through our day-to-day work in applying the toolset to analyze the performance of application codes from collaborating users, for example in the context of the EU Centre of Excellence "Performance Optimisation and Productivity" [13]. While this kind of manual testing has proven beneficial, it is not only laborious and time-consuming, but usually also only covers the core functionality, and therefore a small fraction of all possible code paths. Moreover, it suffers from non-deterministic test inputs, for example, due to using applications with different characteristics, run-to-run variation in measurements, or effects induced by the platform on which the testing is carried out. This makes it hard—if not impossible—to verify the correctness of the results.

To overcome this situation and to allow for a more systematic system testing of the Scalasca Trace Tools package, we developed the Scalasca Test Suite on top of the JUBE Benchmarking Environment [36]. JUBE is a script-based framework that was originally designed to automate the building and execution of application codes for system benchmarking, including input parameter sweeps, submitting jobs to batch queues, and extracting values from job outputs using regular expressions to assemble result overview tables. With JUBE v2, however, it has evolved into a generic workflow management and run control framework that can be flexibly configured also for other tasks, using XML files. JUBE v2 is written in Python and available for download [12] under the GNU GPLv3 open-source license.

In case of the Scalasca Test Suite, we leverage JUBE to automate a testing workflow which applies the most widely used commands of the Scalasca Trace Tools package on well-defined input data sets, and then compares the generated output against a "gold standard" reference result. Commands that are supposed to be run in parallel on one or more compute nodes of an HPC system (e.g., Scalasca's parallel event trace analysis) are tested by submitting corresponding batch jobs, while serial tools that are usually run on login nodes (e.g., analysis report post-processing) are executed directly. Each test run is carried out in a unique working directory automatically created by JUBE, and consists of the following steps:

- **prereq** – This initial step checks whether all required commands are available in `$PATH`, to abort early in case the testing environment is not set up correctly.
- **fetch** – This step copies the input experiment archives (i.e., event traces) and reference results, both stored as compressed `tar` files, from a data storage server to a local cache directory which is shared between test runs. To only transfer

new/updated archives, we leverage the `rsync` file-copying tool which uses an efficient delta-transfer algorithm. The connection to the data storage server is via SSH, with non-interactive operation being achieved using an SSH authentication agent. The list of files that need to be considered in the data transfer is generated upfront and passed to a single `rsync` call, thus avoiding being banned by the data storage server due to trying to open too many connections in a short period of time.

- **extract** – This step extracts the input experiment archive and reference result `tar` files into the per-test working directories.
- **scout** – During this step, Scalasca's parallel event trace analyzer `scout` is run on the input experiment archives. For multi-process experiments (e.g., from MPI codes), the input trace data is pre-processed by applying the timestamp correction algorithm based on the controlled logical clock (clc) [18]. If the analysis completes successfully, the generated analysis report is compared to a reference result.
- **remap** – This step depends on the successful completion of the previous *scout* step. It executes the `scalasca -examine` command to post-process the generated trace analysis report. If successful, the post-processed report is compared to a reference result.
- **clc** (multi-process experiments only) – This step runs the stand-alone timestamp correction tool on the input experiment archives. This parallel tool uses the same controlled logical clock algorithm as the trace analyzer, but rewrites the processed trace data into a new experiment archive. As storing reference traces for comparison is quite expensive in terms of disk space, the execution of this tool—if successful—is followed by another run of the event trace analyzer with the timestamp correction turned off. The resulting analysis report is then again compared to a reference solution.
- **analyze** – This step parses the `stdout` and `stderr` outputs of the previous steps to determine the number of successful/failed tests, and to generate an overview result table.

The individual test cases (i.e., input experiment archives) are currently structured along three orthogonal dimensions, which form a corresponding JUBE parameter sweep space:

- Event trace format
- Programming model
- Test set

Our current set of more than 180 test cases already covers experiments using the two supported event trace formats OTF2 and EPILOG (legacy Scalasca 1.x trace format), traces collected from serial codes as well as parallel codes using OpenMP, MPI, or MPI+OpenMP in combination, and three different test sets: *benchmark*, *feature*, and *regression*. The *benchmark* test set includes trace measurements from various well-known benchmarks (e.g., NAS Parallel Benchmarks [37], Barcelona OpenMP Tasks Suite [22], Sweep3D). The *feature* test set, on the other hand, consists of trace measurements from small, carefully hand-coded tests, each focussing on a particular aspect (see Section 3.4). Finally, the *regression* test set covers event traces related to tickets in our issue tracker. Leveraging traces in the test suite which have triggered defects that were subsequently fixed ensures that those defects are not accidentally re-introduced into our code base. Note that additional parameter values (e.g., to add a new test set) can easily be supported by our test suite; only adding new programming models (e.g., POSIX threads) requires straightforward enhancements of the JUBE configuration, as this usually impacts the way in which the (parallel) tools have to be launched.

Instead of handling input experiment archives as an additional JUBE parameter, the testing steps outlined above are only triggered for each *(format, model, test set)* triple, and then process all corresponding input experiment archives in one go. Otherwise, an excessive number of batch jobs would be submitted for each test run. Moreover, many test cases—especially in the *feature* test set—are quite small, leading to tests that execute very quickly. Thus, the batch queue management and job startup overhead would significantly impede testing turnaround times. Each step therefore evaluates two text files listing the names of the experiments—one per line—that shall be considered for the current parameter triple: one file lists all input archives for which the tests are supposed to pass successfully, while a second file lists the experiments for which testing is expected to fail (e.g., to check for proper error handling). For improved readability, additional structuring, and documentation purposes, both files may include empty lines as well as shell-style comments. The steps then iterate over the list of experiment archives and execute the corresponding test for each input data set. Since the tested tool may potentially run into a deadlock with certain input experiments due to some programming error, each test execution is wrapped with the `timeout` command—to be killed after a (globally) configurable period of time—and thus ensures overall progress of a test run.

After a test batch job has completed, result verification is performed serially (on the login node) based on the resulting `cubex` files. As mentioned before, `cubex` files are regular `tar` archives which cannot be compared against a reference solution using `cmp`. Instead, we use a combination of multiple Cube command-line tools (`cube_info`, `cube_calltree`, and `cube_dump`) to first extract and compare the list of metrics, the calltree, and the topology information, respectively. Then, the actual data is compared using `cube_test`, a special tool that can be configured to compare metrics either exactly by value, or whether the values are within provided absolute or relative error bounds.

Although the Scalasca Test Suite operates on fixed pre-recorded traces, the Scalasca event trace analyzer still produces non-deterministic results for several metrics. For historic reasons, its internal trace representation uses double-precision floating-point numbers for event timestamps (seconds since the begin of measurement). Therefore, the integer event timestamps used by the OTF2 trace format are converted on-the-fly to a corresponding floating-point timestamp while reading in the trace data. During the analysis phase, timestamp/duration data from multiple threads or processes is then aggregated (e.g., via MPI collective operations or shared variables protected by OpenMP critical regions) to calculate several metrics. As the evaluation order of those aggregations cannot be enforced, their results are subject to

```
$ ./testsuite.sh
Executing Scalasca Test Suite (this can take a while...)

OVERALL:
#tests | #pass | #fail | #xfail | #xpass | #error | #miss
-------+-------+-------+--------+--------+--------+------
   189 |   182 |       |        |        |      4 |     3
```

Fig. 2 Scalasca Test Suite: Example overview result table. The columns list the overall number of tests (`#tests`), the number of tests that expectedly passed result verification (`#pass`), failed result verification (`#fail`), expectedly failed (`#xfail`), were expected to fail but passed (`#xpass`), failed with a non-zero exit code, crashed, or were killed due to timeout (`#error`), or where the required input data or a reference solution was missing (`#miss`), respectively.

run-to-run variation and therefore prohibit an exact bitwise comparison. However, the results are still within small error bounds and can be verified using a "fuzzy compare" with the aforementioned `cube_test` tool. This issue could be fixed by consistently using only integer values for timestamps and durations, however, this requires a major code refactoring that affects almost the entire code base, and thus should not be done without having proper tests in place.

From a user's perspective, the main entry point for the Scalasca Test Suite is the `testsuite.sh` shell script, which automates the execution of various JUBE commands to perform all preparatory steps, submit the test batch jobs, wait for their completion, trigger the result verification, and generate an overview result table (see Figure 2). Test runs can be configured via a central configuration file, `config.xml`, for example, to restrict the parameter space (e.g., only run tests for MPI experiments in the *feature* test set) or to skip particular test steps (e.g., the *clc* step) using JUBE's *tag* feature. This can be useful to focus the testing effort on a particular area during development, and thus improve turnaround times. In case of failing tests, a more detailed report can be queried (Figure 3). The step name and number as well as the parameter values then uniquely identify the corresponding JUBE working directory. For example, all input experiment archives, job scripts and outputs, and results from step 5 of the example run shown in Figure 3 can be found in the directory `000005_clc_otf2_mpi_feature/work` for further analysis.

3.4 Systematic test cases

As outlined in the previous section, the *feature* test set of the Scalasca Test Suite consists of trace measurements from small test codes that each focus on a particular aspect. The main advantage of such targeted test codes compared to benchmarks, mini-apps, or full-featured applications is that they are simple enough to reason about the expected result, and thus allow for verifying correct behavior.

One example of such feature test codes is a set of programs covering all blocking MPI collective operations. Building upon the ideas of the APART Test Suite [28],

```
$ ./testsuite.sh -d
OVERALL:
#tests | #pass | #fail | #xfail | #xpass | #error | #miss
-------+-------+-------+--------+--------+--------+------
   189 |   182 |       |        |        |      4 |     3

SCOUT:
step | format | model | testset | #tests | #pass |       | #error | #miss
-----+--------+-------+---------+--------+-------+- ... -+--------+------
   2 |   otf2 |   mpi | feature |     40 |    40 |       |        |
   3 |   epik |   mpi | feature |     23 |    23 |       |        |

REMAP:
step | format | model | testset | #tests | #pass |       | #error | #miss
-----+--------+-------+---------+--------+-------+- ... -+--------+------
   4 |   otf2 |   mpi | feature |     40 |    40 |       |        |
   6 |   epik |   mpi | feature |     23 |    23 |       |        |

CLC:
step | format | model | testset | #tests | #pass |       | #error | #miss
-----+--------+-------+---------+--------+-------+- ... -+--------+------
   5 |   otf2 |   mpi | feature |     40 |    36 |       |      4 |
   7 |   epik |   mpi | feature |     23 |    20 |       |        |     3
```

Fig. 3 Scalasca Test Suite: Example of a detailed result table of a test run limited to MPI tests of the *feature* test set. Due to space restrictions, the (empty) `#fail`, `#xfail` and `#xpass` columns have been omitted from the per-step result tables.

each individual test program exercises the corresponding operation on different communicators, like `MPI_COMM_WORLD`, `MPI_COMM_SELF`, and communicators comprising all odd/even ranks and the upper/lower half of ranks, respectively, as well as with different payloads. Moreover, we use pseudo-computational routines (i.e., functions that busy-wait for a specified period of time) to induce imbalances, thus constructing event sequences exhibiting a particular wait state detected by the Scalasca event trace analyzer in a controlled fashion. As a sanity check, each test program also includes at least one situation in which the analyzer should not detect any (significant) wait state.[10] Each specific situation to be tested is wrapped inside a unique function, which leads to distinct call paths in the analysis report and thus allows easy identification. For all of these test cases, we verified that the key metrics calculated by the Scalasca trace analyzer match the expected results, and created tickets for further investigation in our issue tracker when broken or suspicious behavior was encountered. Although the programming language in which the tests are written is irrelevant for a trace analysis operating on an abstract event model, we implemented them in both C and Fortran, as we anticipate that the test codes will also be useful for testing Score-P—in particular its MPI adapter—on a regular basis. For this reason, we also created additional variants using magic constants (e.g., `MPI_IN_PLACE`) that require special handling during measurement.

[10] For some calls, for example N-to-N collectives such as `MPI_Allreduce`, it is impossible to construct a test that does not exhibit any wait state. However, the detected wait state will be very small if the preceding computation is well-balanced, and thus can be distinguished from a "real" wait state.

In addition, we also developed a configurable trace-rewriting tool—with the intention to create even more test cases based on trace measurements collected from the test codes outlined above. This tool allows simple operations such as modifying event timestamps or dropping individual events. We currently use it to inject, for example, artificial clock condition violations into event traces to test Scalasca's timestamp correction algorithm, or other artifacts to test proper error handling.

Obviously, writing good test cases is a non-trivial undertaking that requires quite a bit of thought. However, we consider them a well-spent effort that pays off in the long run. For example, these systematic tests already helped to uncover a number of issues in both the Score-P and Scalasca Trace Tools packages that would have been very hard to spot with real applications. Moreover, the collected trace measurements used in the *feature* test set of the Scalasca Test Suite have proven worthwhile as a "safety net" during various larger refactorings in the Scalasca code base.

3.5 Open Issues

With many scientific projects, the initial focus of development usually is on making quick progress rather than on writing code that is also covered by tests—and our projects were no exception. That is, most of our tests that exist today have been written after-the-fact and lots of legacy code is still untested. However, retroactively adding tests for entire code bases that have grown for a decade or more is prohibitive. For example, the MPI 3.1 standard already defines more than 380 functions for which the current Score-P 4.1 is providing wrappers that would require appropriate tests to be written—not to mention SHMEM, CUDA, OpenCL, etc. Thus, we strive to add tests for newly written or refactored code (following the so-called "boy scout rule"), thereby slowly increasing our test coverage.

As mentioned in the previous section, we also envision that the systematic test cases could be used for regular and automated Score-P testing. For this purpose, a JUBE-based framework similar to the Scalasca Test Suite needs to be implemented. While various parts of the existing JUBE configuration could likely be reused, and building, instrumenting, and running test codes with JUBE is straightforward, result verification is much more challenging than in the Scalasca case using fixed input data sets. Whereas most counter values such as the number of bytes sent/received by message passing calls can be compared exactly, time measurements may vary considerably between runs. Also, different compilers (and even compiler versions) may use different name mangling schemes or inlining strategies, thus leading to variations in the experiment meta-data, in particular the definitions of source code regions and the application's call tree. Moreover, measurements from task-based programs are inherently non-deterministic. Although we do not have a good answer for how to address these issues at this point, both profile and trace measurement results could nevertheless be subject to various sanity checks. For example, profile measurements should only include non-negative metric values and generate self-contained files that can be read by the Cube library, and event traces should use consistent event sequences such as correct nesting of ENTER/LEAVE events. This would at least provide a basic level of confidence in that code changes do not break the ability to collect measurements, and thus still renders such a test suite to be beneficial.

While the `check` and `installcheck` Makefile targets outlined in Sections 3.1 and 3.2 are already triggered by our GitLab CI/CD setup, the Scalasca Test Suite still has to be run manually. Instead, it would be desirable to run it automatically on a regular basis, for example, as a scheduled pipeline once per night or on each weekend—depending on the average code-change frequency of the project—using the last successful build of the main development branch. Likewise, this also applies to tests for other projects that are not integrated into the build process, such as the yet-to-be-written Score-P Test Suite mentioned above.

4 Conclusion and Outlook

In this article, we have presented an overview of the evolution of our approaches regarding continuous builds and delivery as well as automated testing in the context of the Score-P, Scalasca, Cube, OTF2, and OPARI2 projects. We have described the main challenges we encountered along the way, outlined our current solutions, and discussed issues that still need to be addressed. The automated approaches have proven beneficial to identify a multitude of functional and portability issues early on, way before our software packages were made available to our user community. Although our implementations are clearly geared toward our needs for testing performance analysis tools and the underlying libraries and components, we believe that the general approaches are also applicable for testing other HPC-related software, such as scientific codes.

In the future, we plan to address the open issues outlined in Sections 2.3 and 3.5. As a short-term goal, we will work toward extending our GitLab CI/CD setup to automatically trigger the execution of the Scalasca Test Suite in regular intervals using the latest successfully built Scalasca package. In addition, we plan to explore the use of containerized build environments, a prerequisite for dealing with the security implications of the envisioned migration of our source codes to (potentially public) git repositories and contributions from external, untrusted sources. In the medium term, we plan to implement a test suite for Score-P to carry out functional tests, similar in spirit to the Scalasca Test Suite. As a continuous and long-term effort, we will of course also develop new systematic test cases to increase the coverage of our testing.

References

1. Boost C++ libraries. `https://www.boost.org/`. Last access: 2018-08-14.
2. CATCH2. `https://github.com/catchorg/Catch2`. Last access: 2018-08-14.
3. CppUnit – C++ port of JUnit. `https://sourceforge.net/projects/cppunit/`. Last access: 2018-08-14.
4. CppUnitLite. `http://wiki.c2.com/?CppUnitLite`. Last access: 2018-12-05.
5. CxxTest. `https://cxxtest.com/`. Last access: 2018-08-14.
6. Docker. `https://www.docker.com/`. Last access: 2018-09-06.
7. Doxygen – Generate documentation from source code. `http://www.doxygen.nl/`. Last access: 2018-11-28.
8. FRUCTOSE. `https://sourceforge.net/projects/fructose/`. Last access: 2018-08-14.
9. GitHub. `https://github.com/`. Last access: 2018-11-26.
10. GitLab Continuous Integration and Delivery. `https://about.gitlab.com/product/continuous-integration/`. Last access: 2018-11-26.
11. Jenkins. `https://jenkins.io/`. Last access: 2018-11-26.
12. JUBE Benchmarking Environment website. `http://www.fz-juelich.de/jsc/jube/`. Last access: 2018-11-23.
13. Performance Optimisation and Productivity: A Centre of Excellence in HPC. `https://pop-coe.eu/`. Last access: 2018-12-07.
14. Travis CI. `https://travis-ci.org/`. Last access: 2018-11-26.
15. UnitTest++. `https://github.com/unittest-cpp/unittest-cpp/`. Last access: 2018-08-14.
16. Mark J. Abraham, Adrien S. J. Melquiond, Emiliano Ippoliti, Vytautas Gapsys, Berk Hess, Mikael Trellet, Joo P. G. L. M. Rodrigues, Erwin Laure, Rossen Apostolov, Bert L. de Groot, Alexandre M. J. J. Bonvin, and Erik Lindahl. BioExcel Whitepaper on Scientific Software Development, March 2018. `https://doi.org/10.5281/zenodo.1194634`.
17. Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.
18. Daniel Becker, Markus Geimer, Rolf Rabenseifner, and Felix Wolf. Extending the scope of the controlled logical clock. *Cluster Computing*, 16(1):171–189, March 2013.
19. John Calcote. *Autotools: A Practioner's Guide to GNU Autoconf, Automake, and Libtool*. No Starch Press, San Francisco, CA, USA, 1st edition, 2010.
20. Jeff Carver. *ICSE Workshop on Software Engineering for Computational Science and Engineering (SECSE 2009)*. IEEE Computer Society.
21. A. Dubey, K. Antypas, A. Calder, B. Fryxell, D. Lamb, P. Ricker, L. Reid, K. Riley, R. Rosner, A. Siegel, F. Timmes, N. Vladimirova, and K. Weide. The software development process of FLASH, a multiphysics simulation code. In *2013 5th International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE)*, pages 1–8, May 2013.
22. Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.
23. Edgewall Software. Bitten – A continuous integration plugin for Trac. `https://bitten.edgewall.org/`. Last access: 2018-08-14.
24. Edgewall Software. trac – Integrated SCM & Project Management. `https://trac.edgewall.org/`. Last access: 2018-08-14.
25. Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. Open Trace Format 2 - The next generation of scalable trace formats and support libraries. In *Proc. of the Intl. Conference on Parallel Computing (ParCo), Ghent, Belgium, August 30 – September 2 2011*, volume 22 of *Advances in Parallel Computing*, pages 481–490. IOS Press, 2012.

26. FLEUR Developers. FLEUR GitLab pipelines.
`https://iffgit.fz-juelich.de/fleur/fleur/pipelines`.
Last access: 2018-11-29.
27. Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The SCALASCA performance toolset architecture. In *International Workshop on Scalable Tools for High-End Computing (STHEC), Kos, Greece*, pages 51–65, June 2008.
28. Michael Gerndt, Bernd Mohr, and Jesper Larsson Träff. A test suite for parallel performance analysis tools. *Concurrency and Computation: Practice and Experience*, 19(11):1465–1480, August 2007.
29. Google, Inc. Google Test. `https://github.com/google/googletest`.
Last access: 2018-08-08.
30. Daniel Hook and Diane Kelly. Testing for trustworthiness in scientific software. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, SECSE '09, pages 59–64, Washington, DC, USA, 2009. IEEE Computer Society.
31. Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010.
32. Vasileios Karakasis, Victor Holanda Rusu, Andreas Jocksch, Jean-Guillaume Piccinali, and Guilherme Peretti-Pezzi. A regression framework for checking the health of large HPC systems. In *Proceedings of the Cray User Group Conference*, 2017.
33. Diane Kelly and Rebecca Sanders. Assessing the Quality of Scientific Software. In *First International Workshop on Software Engineering for Computational Science and Engineering, Leipzig, Germany*, May 2008.
34. Diane Kelly, Spencer Smith, and Nicholas Meng. Software engineering for scientists. *Computing in Science and Engineering*, 13(5):7–11, 2011.
35. Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P – A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. of the 5th Int'l Workshop on Parallel Tools for High Performance Computing, September 2011, Dresden*, pages 79–91. Springer, September 2012.
36. Sebastian Lührs, Daniel Rohe, Alexander Schnurpfeil, Kay Thust, and Wolfgang Frings. Flexible and Generic Workflow Management. In *Parallel Computing: On the Road to Exascale*, volume 27 of *Advances in Parallel Computing*, pages 431–438, Amsterdam, Sep 2016. IOS Press.
37. NASA Advanced Supercomputing Division. NAS Parallel Benchmarks.
`https://www.nas.nasa.gov/publications/npb.html`.
Last access: 2018-11-25.
38. NEST Initiative. NEST developer space: Continuous integration.
`http://nest.github.io/nest-simulator/continuous_integration`.
Last access: 2018-08-08.
39. Szilárd Páll, Mark James Abraham, Carsten Kutzner, Berk Hess, and Erik Lindahl. Tackling Exascale software challenges in molecular dynamics simulations with GROMACS. In *Solving Software Challenges for Exascale*, volume 8759 of *LNCS*, pages 3–27. Springer, 2015.
40. D. E. Post and R. P. Kendall. Software Project Management and Quality Engineering Practices for Complex, Coupled Multiphysics, Massively Parallel Computational Simulations: Lessons Learned From ASCI. *The International Journal of High Performance Computing Applications*, 18(4):399–416, November 2004.
41. Pavel Saviankou, Michael Knobloch, Anke Visser, and Bernd Mohr. Cube v4: From performance report explorer to performance analysis tool. In *Proceedings of the International Conference on Computational Science, ICCS 2015, Computational Science at the Gates of Nature, Reykjavík, Iceland, 1-3 June, 2015*, pages 1343–1352, 2015.
42. Michael G. Schwern and Andy Lester. Test Anything Protocol.
`https://testanything.org/`. Last access: 2018-08-08.

43. Gary V. Vaughan, Ben Elliston, Tom Tromey, and Ian Lance Taylor. *The Goat Book*. New Riders, 2000.
44. Ilya Zhukov, Christian Feld, Markus Geimer, Michael Knobloch, Bernd Mohr, and Pavel Saviankou. Scalasca v2: Back to the future. In *Proc. of Tools for High Performance Computing 2014*, pages 1–24. Springer, 2015.