Automated File System Correctness and Performance Regression Tests

— Bachelorarbeit —

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Anna Fuchs
E-Mail-Adresse:	0fuchs@informatik.uni-hamburg.de
Matrikelnummer:	6208669
Studiengang:	Software-System-Entwicklung
Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Prof. Dr. Norbert Ritter
Betreuer:	Michael Kuhn

Hamburg, den 23.09.2013

Abstract

To successfully manage big software projects with lots of developers involved, every developing step should be continuously verified. Automated and integrated test procedures remove much effort, reduce risk of error rate and enable a much more efficient development process, since the effects of every development step are continuously available.

In this thesis the testing and analyzing of a parallel file system JULEA[1] is automated. With it all these processes are integrated and linked to the version control system Git. Every checked change triggers a test run. Therefore the concept of Git hooks is used, by me of which it is possible to include testing to the common develop workflow. Especially scientific projects suffer from lack of careful and qualitative test mechanisms. In the course of this not only correctness is relevant, which forms the base for any further changes, but also the performance trend. A significant criterion of a quality of a parallel file system is its efficiency. Performance regression of a system caused by made changes can crucially affect the further development course. Hence it is important to draw conclusions about the temporal behavior instantaneously after every considerable develop step. The trends and results have to be evaluated and analyzed carefully.

The best way to recognize this kind of information is the graphical way. The goal is to generate simple but meaningful graphics out of test results, which would help along to improve the quality of the developing process and the final product. Ideally the visualization would be available on a web site for more comfortable use. Moreover to abstract from the certain project, the test system is portable and universal enough to be integrated it in any project versioned with Git.

Finally, some tests in need of improvement were located using this framework.

Contents

1. Introduction and Motivation

One of the most important and at the same time most underrated stages of software development is testing. It is a continuous process and should be performed at every stage of the project development or software development process in general. In Figure 1.1 you can see the desired workflow in software development. It consists four steps, which are largely independent from each other. After the idea was designed starts the developing step. One cycle is over after the designed and implemented software was tested and could be released. Then the cycle begins again with designing next ideas. Unfortunately it isn't the typical case.

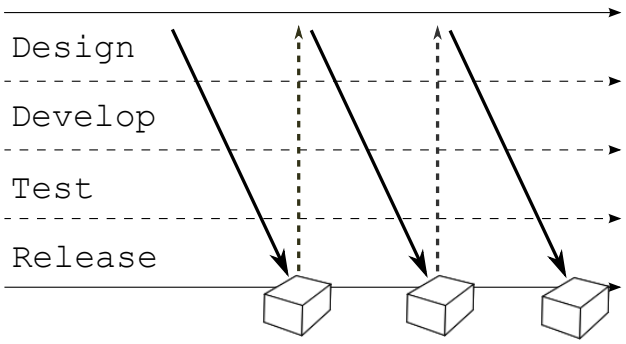


Figure 1.1.: Desired workflow in software development [2]

In fact the disadvantage of this strategy seems to be the encapsulation of the testing process in a separate stage. In the ranking of most diligent done work is testing probably on the last place. It has to be integrated in every other process, in best case in the background.

Testing is the first step toward software quality, but in the science it plays not the most important role. The reasons for this might be the non profit goal, which gives much freedom on the one hand, but also less control on the other. In commercial business there is mostly an own department for testing the software and permanent checking. Sometimes external analyst companies are hired. There are more or less strict deadlines and a direct feedback in case of any errors. The goal is to produce software for customers satisfaction by the lowest price and largest profit. In contrast the goal of science is at first to get the idea to work. The primary discipline of scientists is still science. Since they are not software engineers, the quality of code is secondary. This is less a reproach, but a fact that detects a gap in the cooperation. It gets even worse when it comes to parallel software and file systems, which are types of data storage for storing, retrieving and updating sets of data files. Parallel file systems can handle same function on data

in parallel at the same time. To make it possible without errors and with still good performance complex mechanisms are necessary. Testing parallel software is difficult, but testing a parallel file system, which is closed to hardware, require complex, resource-heavy and prolonged tests to satisfy quality claims, can be moreover annoying. Setting up a test involves considerable expense, due to parallelism and special requirements like restarting necessary daemons for each run to ensure independent results.

The problem, of testing in science has a second facet. Tests if the software is correct are neglected, but performance tests, if the optimum of possible speed is reached are mostly not present. Since software is a tool to do other science, most of scientists are already happy if the software is produces the task results correctly. Most of the tests, if they are performed at all, are about the application semantics, not the software quality. It makes it inevitable to alter the situation, to change the strategy of testing, first of all in minds of scientists. One of the ways could be to offer a more comfortable environment for development, which takes over the largest effort of testing. At the same time not all the control should be taken from the developer, since the test system should help, not disturb.

This thesis's main goal is to create a test system for the developing of the parallel file system JULEA and to motivate pursuit of intensive and careful testing. The strategy is to create an integrated test mechanism to enable easy testing during the development process. Investing time and productive testing once properly could reduce the cost and effort of development tremendously. The requirements on the system are user friendliness, universality and simplicity. The system should at first help the developer, not require more effort. And it should give trust in its reliability. Universality means it has to work with different environments without much effort to adapt the configuration. It has to be not only simple to configure, but simple to use. The developer should also make any changes and updates easily and the results should be easily understandable without spending much time on the other.

For the concept some foundations about version control systems and continuous software developing with these are introduced in chapter 2. At first it deals with Git and its internal work strategy to understand all the opportunities of working with it. The second part of chapter 2 gives an outline about the testing in software developing in general, shows problems in this area and possible solutions for these. Chapter 3 deals with the design by taking a closer look at the environment and requirements of the test system. The design has to be structured very clearly and every step has to be logical to finally show the advantage and use of the test system. Chapter 4 focuses on implementation of selected and most important aspects. It contains details about configuration and settings, which are useful to understand in case the user of the test system would like to integrate his own changes. Chapter 5 contains the presentation and evaluation of the previous work. In the focus are graphics with the performance of several commits. The chapter discusses satisfying requirements and the quality of presenting the information. Next chapter shows some already existing solutions, its advantages and disadvantages and explains why it is still worthwhile to think about an own test system. Finally the summary and conclusion are made in the last chapter. Moreover it contains remarks for the future work and additional ideas.

2. Background

At first it is necessary to know some theory and background of the task to enable sufficient work. This chapter gives an introduction to version control systems focused on distributes ones. Further it deals with Git - its usage, structure and internal details. Finally it is about testing in the software, and how to combine developing and testing in a workflow to increase the quality. Especially it is not only about basics of testing, but about the question of integrated tests with a version control system.

2.1. Git

Most software developers use a VCS[1] in their projects. These systems help to manage documents, source-code or any units of information, they enable clean working in bigger groups and on different documents at the same time. The main task of such systems is to keep all versions from the past, to enable accessing, reverting or restoring of any version at any time. It is like an advanced copy and paste or undo redo, but parallel and asynchronously. Usually all changes get an identifying number or letter code, like hash code, timestamp and at least the name of the user who made changes.

There are in general three types of VCSs - local, central and distributed. The local one is limited to a single document, which is versioned and hold a kind of index on all changes. This type is not interesting in software development, at least because there are too many documents to manage them separately. The centralized VCS is based on the idea of a server-client system, where network is necessary to access the repository. There is a single copy on the central server, where to the programmers can *commit* their changes. Committing means uploading and recording local changes in the repository, in this case on a server. The history of changes is also available only in repository and can only be reached via network. Any changes or versions of other developers can be pulled down, also only having network. The most popular VCS using this strategy is Subversion.

The next level and most important kind of VCS for this thesis is the distributed one. Here, no network is needed to do local versioned work, due to peer-to-peer structure in contrast to the centralized one. Every authorized developer has the local repository, while every repository has equal rights. Local changes in developers own repository can be committed without network and are kept on the local machine. To compare or merge any changes from other repositories or to upload and make accessible own changes there is a remote repository, which can be reached only via network. This concept has lots of

[1] Version Control System

advantages. The developer *clones* or copies the whole repository with all the history and meta data to his own local directory.

Nowadays it is no problem to hold redundant copies of text or even some pictures due to cheap hard space and efficient compression. In further chapters we will see, that it is not only about mindless copying of anything, but a smart strategy of holding only new information without copying older unchanged parts. So it is no problem to keep a copy, but has a significant benefit. Each local repository is kind of a remote backup for own changes. In case of damaging or destroying the remote, at least own versions including the history would not get lost. Particularly because te local repository is kind of its own server, it are fast, since it does not need to communicate with the central server doing local changes or asking for local history, but just access the hard drive.

One of the most used distributed VCS is Git, which is a free and open source distributed version control system and is used in this thesis. Once developed for the Linux kernel it became more and more popular in software development.

2.1.1. Workflow

There are two workflows in Git, a local and a remote one. The local work with Git is like saving new changes on the project on your local system with the possibility to go back every time to older changes or come back again to the last change. Doing this it is not necessary to copy any files. To put changes in Git they just have to be *committed*. In this moment a new version of the document will be . It gets a date stamp, an individual hash value, author of the change, person who did the commit, a commit message and maybe some more information configured by user. It looks like in the Figure 2.1. The legend for Figures can be looked up in the appendix A.

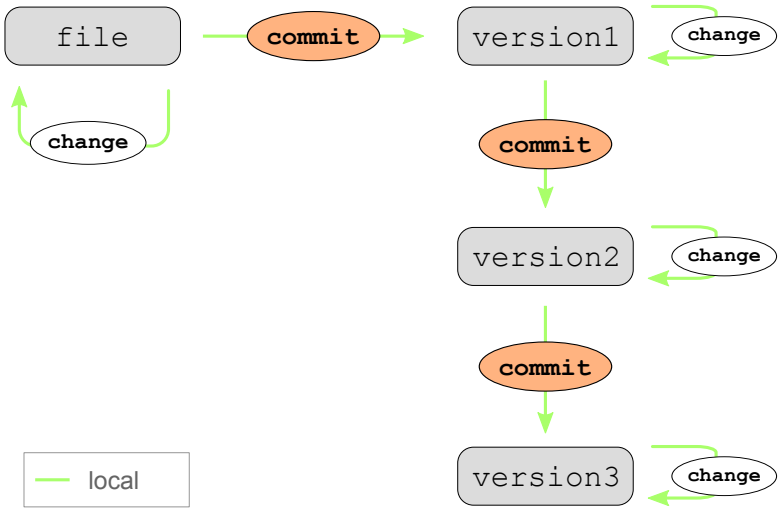


Figure 2.1.: Committing changes in Git

The remote workflow in Git requires a remote server, where all the changes can be stored and accessed by other developers. To upload their own changes developers have to

push them. Other authorized developers can get the current state by *pulling* it. Pushing the changes means to upload all not pushed commits in one operation, while every commit keeps its assigned properties like the hash values and does not get any further information like a push message. So remote work presupposes the local work with Git.

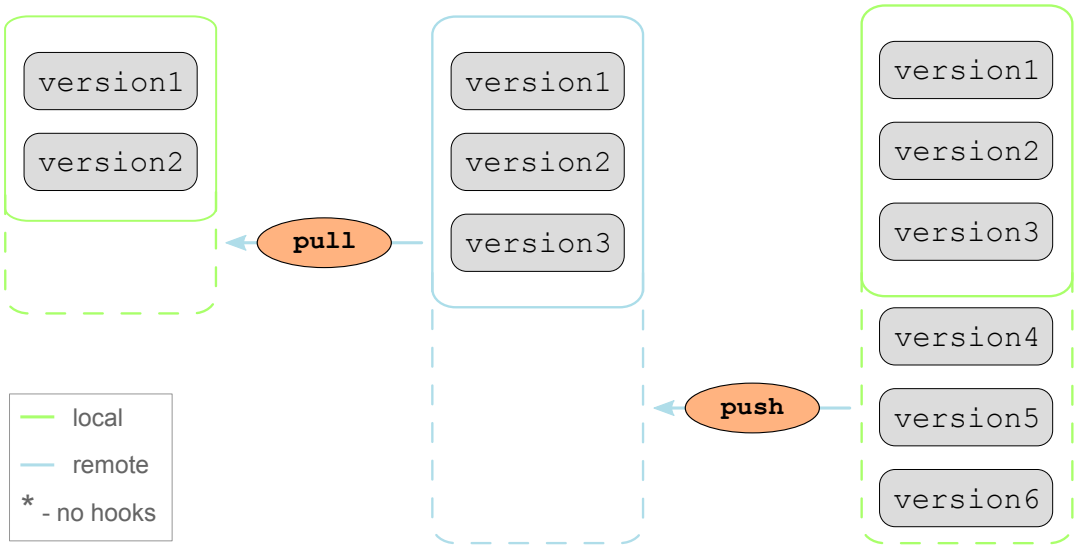


Figure 2.2.: Git pull and push changes from remote

Figure 2.2 shows the principle of pushing and pulling. The missing version three can be pulled from the remote server, and new versions four, five and six can be pushed from a local machine to the remote. There is no need to copy already available information.

The next section presents more detailed view how Git handles information.

2.1.2. Internals

To understand most of Git's features, it is very important to know how it works with information internally. First of all Git is not saving the files like a queue of changes, but in form of snapshots. A snapshot is the current state of the system. Before doing a commit, changed files have to be added to the *staging area* or the *Index* with `git add <file>`. The staging area is a kind of departure station, where is to choose which changes will board and which not. This stage between developer's work and Git's memory enables more flexibility, since there is a choice which changes are mature for commit and do not need to commit everything. Other VCSs usually allow to commit separate changed files, but not separate changes in one file. In Git larger changes can be split into several commits. Every entry in the staging area creates some objects in `.git/objects`. Every file in this directory is an object of one of four types - a blob, a tree, a tag or a commit object.

A blob (binary large file) is the basic data storage unit. The `add` command computes at first the hash value of the added file content. It does not matter what the name or author or the date of file is, relevant is only its content. Two different named files with

same content would cause the same hash value. The generated hash value, called SHA-1, is a 40 digits hexadecimal expressed 160 bit hash value. First two digits of the value are used to create a directory, in which a file with the name of other 38 digits is stored - the blob object with the content of added file. This file is added to the index list. Git is intelligent in packing files to save storage space. Similar files with similar contents can packed by Git using the delta compression[3].

At this stage exists only the content addressable storage, since the name or path of the added file was nowhere used. The problem of storing the filename is solved by tree objects. The Git tree is similar to UNIX file system directory. A tree object contains at least one tree entry with each a SHA-1 pointer to the blob or to another tree entry. Additionally and in contrast to the blob object a tree has type, filename and mode[4].

After a commit is done, Git creates a commit object. The object contains the tree hash, author and the commit message. This object itself also has a hash value to be referred to the database. The object also contains information about the commit parents. The first commit has of course no parents, a commit after merging has at least two parents.

In the Figure 2.3 is shown a scheme how objects in Git are structured.

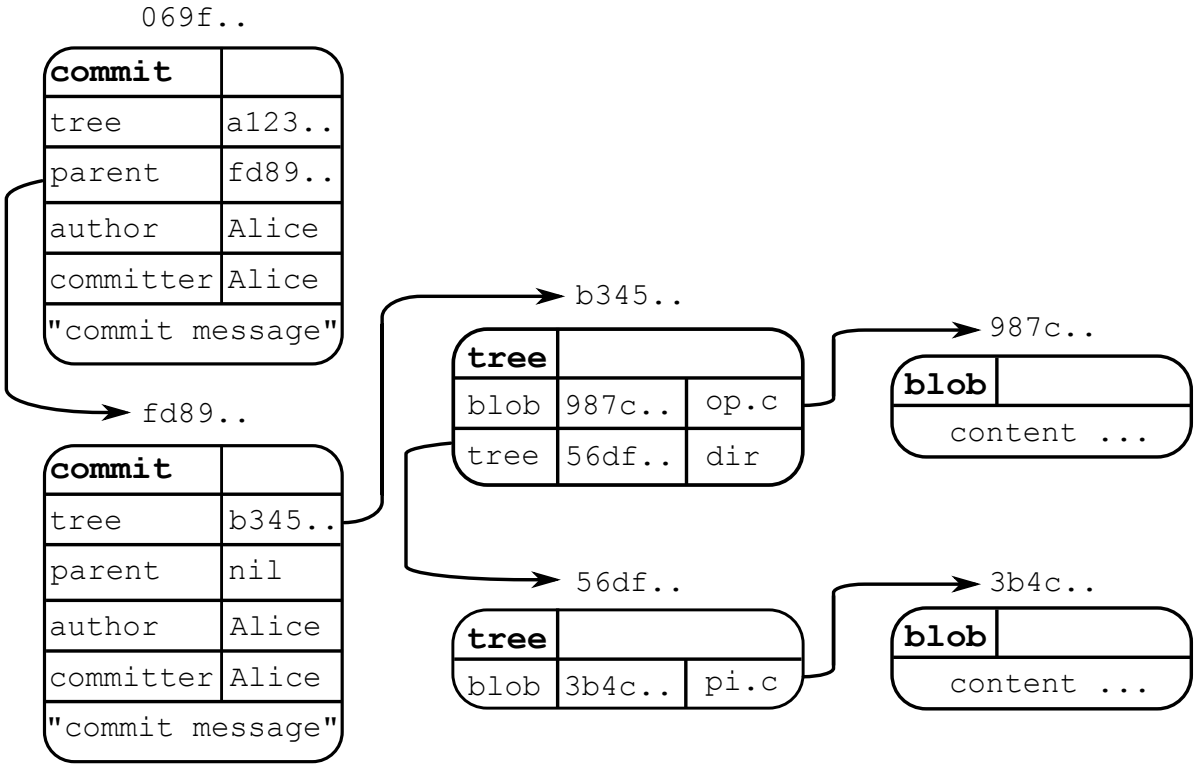


Figure 2.3.: Git object structure

The first commit with the first four characters of SHA-1 value `069f` has a pointer to a tree object, a pointer to the commit parent and a commit message. Alice is author and

committer. If we follow the pointer to the parent commit, we see the second commit object with the first four characters of SHA-1 value `fd89`. This has a pointer to a tree, no parent commit pointer, which means it is the first commit, the same author as above and the commit message. The tree object itself has also a SHA-1 value, and consists two pointers, to a blob object of the file named `op.c` and to another tree in directory `dir`. The other tree has only a pointer to a blob object, which keeps information only about the content of committed files. Once the committed version is ready for release, the developer may want to give a certain commit an additionally name. For this they can tag the hash value of the commit as a new name. This tag object has an author, the new name, tag message and of course the hash of the commit to be tagged. All these objects after one commit together build the snapshot. Since creating a new version or a snapshot is so cheap, Git becomes very powerful. In the next section is one of Git's best features presented - the branching.

2.1.3. Branching

Git got so popular not least because of its continuous development, practical use and outstanding concept of branching, which is mostly very expensive in other version control systems. Branching means to create a new parallel independent working line without disturbing the others.

As soon as the first commit is done, the first branch is automatically created, which is called master. A new branch means nothing else than a new pointer to one of the commits. Doing every further commit, the branch pointer just moves with it. After a new branch is created, it has at first a pointer to the same commit, as the old one or to the specified commit. Creating a new branch does not imply changing on it. There exists a special pointer called `HEAD`, which points to the current local branch. This pointer has to be moved to the new branch to *checkout* the branch, means to move on it. Doing a new commit on the new branch does not mean to copy the whole history to the new branch. The older branch still points to the last commit made on it, while the new branch creates a parallel working line like shown in Figure 2.4. There are three branches with some commits. After the second commit is the new `debug` branch created and two commits are made on it. Once all bugs are fixed, the two branches can be merged or combined to one working line. The work on branches can happen parallel like on the feature branch called `idea`. Here the new idea is implemented, while the work on branch master goes on. If the same code was changed, merging these branches could cause conflicts, which have to be solved manually. Git does not duplicate older changes and commits, but shares them as far as it is possible. In consequence, it is no problem to change the branch at any time, or to create a new one. It happens very simple and fast, because only a new pointer has to be created.

This concept opens up great possibilities not only for developing itself, but also for testing and trying out new possible risky ideas without being afraid of irreversible changes. At the same time the developer takes the responsibility for regular updates (`pull` and `push`) of the working directory, especially because Git is distributed, which means that

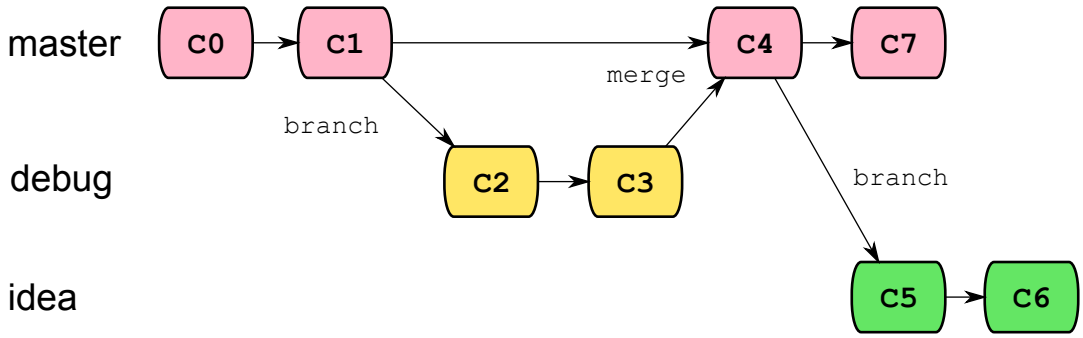


Figure 2.4.: Branches in Git

lots of persons can make changes at the same time. Sometimes it gets difficult to manage all changes and still make sure to have the latest version, which additionally is correct.

In the best case everyone should perennially test every change, so that every pushed version is correct. Beyond that, it is often necessary to check the correctness in the context of the whole project considering changes of other developers. Doing that in parallel programming causes additional hurdles. These tests are more complex and resource-intensive. More than that, in software development you should not only try to ensure correctness of your changes, but also their efficiency. Because of the relatively high effort for all these tests and doubtfulness whether all the developers reliably test their changes, it is expedient to integrate automated tests for correctness and efficiency, which still can be controlled or disabled by developers if needed. Git has some mechanisms for triggering actions after Git-specific commands like `commit` or `push`, called *hooks*.

2.1.4. Hooks

Hooks are custom scripts, which can be written in any language. They are running (if enabled) after or before important actions. Hooks enable in general programs to run automated closely linked to Git actions. It supports a better organized workflow in bigger projects.

Hooks are divided into two groups – client-side hooks and server-side hooks. Client-side hooks, as the name suggests, are scripts running after local Git commands on the client side, like `commit` or `merge`. Server-side hooks concern server operations like pushing or receiving changes. There are 16 hooks which sometimes have a certain order of running [5]. The first client-side hooks are

1. `commit-msg`
2. `prepare-commit-msg`
3. `pre-commit`

in the stated order. All these hooks can abort the commit if exiting non-zero. It is very important to know that the `pre-commit` hook has no input arguments, while the others take at least the name of the file containing the commit message.

The hook triggered after the commit is the `post-commit` hook, which takes no arguments and cannot affect the commit.

As the names suggest, there is a kind of advice for purpose of some hooks, like the `commit-msg` hook can be used to normalize the message into some project standard format or to refuse the commit after inspecting the message file" But in fact there exits no technical rule for the aim of the hook. If you need to further process the commit message, you should use one of the first two hooks, since there is no way to get the commit message in the `pre-commit` hook itself.

There is an option `-n` or `--no-verify` to bypass the `commit-msg` and `pre-commit` hooks, but is ignored by the `prepare-commit-msg` hook.

There are some more hooks for the e-mail workflow. This will not be used in this thesis. The necessary hooks here would be

- `applypatch-msg`
- `pre-applypatch`
- `post-applypatch`

More than that there are some other command specific hooks, like

- `pre-rebase`
- `post-checkout`
- `post-merge`

These hooks work same as the other pre and post operation hooks.

All previously discussed hooks were client-side hooks. Server-side hooks run before or after pushes, which enable better control of your project changes. `pre-receive` and `post-receive` hooks work like the similar client-side hooks and also can abort pushing or print an error message. They both take no arguments, which is very important for the future work. Since there are no push message or any other arguments, doing a push it is not possible to use them for passing any keywords to imitate any control options.

To name the complete list of hooks - there are `update`, `post-update`, `pre-auto-gc` and `post-rewrite` hooks, which can be looked up in the man page

To use any hook you have to rename the file in `.git/hooks` from `hook-name.sample` to `hook-name` and possibly make it executable (usually changing the mode). If there exists no custom file just create a new one with the correct name. A hook can be written in any programming language, shell script being the most common. In the Figure 2.5 you see a common workflow with some hooks, which are especially important for this thesis.

Tasks, like tests, which have to be done in fixed and organized order, can be integrated in the hooks to automate the process. The 16 Git hooks consider any need of the developer and offer all combination at every step of the work - before or after Git actions, with or without input parameters. There is hardly an unfulfilled desire for additional

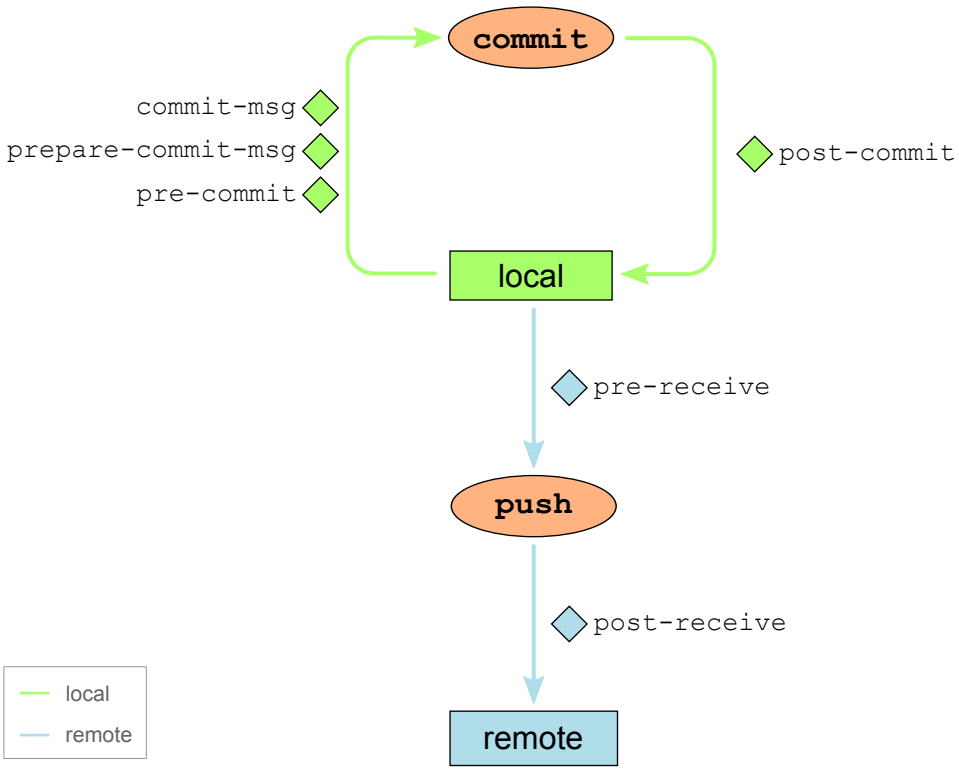


Figure 2.5.: Overview of local and remote Git hooks

hooks. It is in general not provided to define own type of hooks. The already available hooks are a great opportunity to combine developing and testing of software. The issue about testing presents the next section.

2.2. Testing

The main goal of this thesis is to integrate testing process in developing the software. To understand how to help the developer in best way, first is necessary to differentiate goals and requirements of certain test processes. The timing when to run tests to get the most profit of test results is enormous important. Also the hierarchy of testing can save effortless. The strategy should be to first catch gross errors and then gradually refine the quality.

This section helps to bring order to the test structure.

2.2.1. Classification

There exist many classifications of tests in software like blackobox or whitebox tests, system or integration tests or many more combinations. In this thesis the decisive criterion are the consequences depending on test results. Critical tests cause active influence in development, like interrupting or aborting any process if the result is not

satisfactory. Uncritical tests do not have any effect on processes, but just deliver results, which can be evaluated and considered by the developer.

In general we differentiate two groups of tests - correctness and performance tests. While correctness is a relatively strict criterion, performance isn't, which means, it is in general more difficult to decide if the software is efficient, than if it is correct. There are often some rules for sufficiency of some characteristics for the respective develop step. In every case correctness should be the condition of any further testing and should urgently trigger consequences if not fulfilled. It is hard to say if the efficiency of actual state is good enough. But it is much easier to compare the actual state with previous one and to decide if it has become better or worse.

Regression tests are in general used for common testing to prove if a bug, which already was fixed comes up again. In this context we expand this notion and look up if the performance we already reached, is exceeded or reduced by the next change.

To summarize, correctness tests are better decidable or more clear, since there are most two answers: "yes" or "no, the result is correct or it is not. Performance measurements are in best case subjectively or relatively decidable, means it is not black or white, but better or worse.

This point is very important since correctness (compiling included) tests are kind of critical tests in this thesis, which will abort uploading changes in case of errors. If it was not clear if the results are correct or not, it would cause problems in argumentation why this commit was aborted. More than that, a correct software is a basis for any further tests. It makes no sense to check efficiency of faulty software.

While correctness tests have to be done anyway, there is a question about the extent of performance tests - in which stage is it meaningful to run all tests to get extensive results, the price of which is certainly time and resources. These questions are discussed in chapter 3.1.

2.2.2. Features

There are existing solutions for local testing, but when it comes to remote parallel tests it becomes difficult. Doing local changes, the tests could be also run manually without any greater effort. The effort doing parallel testing is much bigger, so that it is hard or impossible to do it manually. This task seems to be too special to integrate it in existing solutions, some of which are presented in chapter 6.

The main part of test results is the comparability, which is the base for any further decisions. Removing all of the influences and organizing automated parallel tests in a comfortable structure is not a trivial job. In particular this task has a strong psychological component. The aim is to be prepared for any situation to enable a correct parallel test run, since the developer would probably not want to invest time in correcting the test system to make a run. So if anything goes wrong with the test system, it would be difficult to motivate the developer to care about it. This fact causes almost greater responsibility and harder requirements not only at automated awareness but also at universality. It is a new feature, which is not so strongly asked in other cases.

3. Test Suite Design

To build a universal test system means at first to test the system itself. Therefore the requirements have to be placed and checked. The system has to work with any initial conditions and react correctly on any unexpected effects. Even if the user of the test system does anything wrong, the system should motivate him to correct mistakes, not to skip the test at all.

This chapter describes requirements on the test system and the strategy of how to solve problems.

3.1. Tests

The structure of tests in this thesis was already sketched in chapter 2.2.

The critical test group consists of two tests, compile and correctness tests, which both would abort the commit done by the developer in case of failure. But even automated tests should not get too much control over the workflow. It is important to control and improve bad behavior, but it is much more important not to lose any information. In case of error the commit should not be applied in its original form, but preserved in a special area to allow the developer to repair bugs. The branching concept seems to be a good way to solve the problem, since it is cheap to create a new branch. Once a test fails, the faulty version would be committed to an automatic generated branch, which contains not only the changes, but also all the commit information, like commit message. It enables to split the workflow not interrupting it. The aim is not to lose any information, no matter what happens.

Another case to handle is consciously committing faulty changes, if there is no time or opportunity to fix them. In this case the user would like to inform the test system to disable the testing framework temporarily. For this use case, any test has to be switched off separately. The granularity is also an indication of quality of the test system. It should consider any information the developer wants to incorporate. The critical tests depend only in one direction, means, if compiling failed, no need to prove correctness. But if the developer knows, correctness would fail, he or she could still want to check if the version compiles. Independent steps should be runnable separately. Otherwise the developer has to run the compile test manually, which causes no confidence to the test system. Figure 3.1 shows the hierarchic a structure of tests.

Despite the dependence direction, it is possible to run for example only the correct test without compile test. It is not recommended and causes some effort to set the necessary keywords, but it is still possible. It is not the task of the test system to take

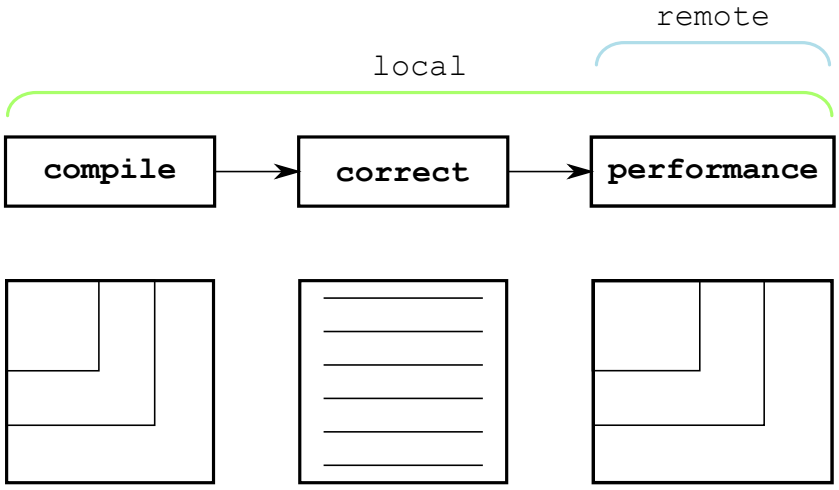


Figure 3.1.: Hierarchical test structure

the responsibility of correct order of tests, since they cause no fatal influence on the code base. A wrong ordered run would just not have the expected effect.

More detailed information how the certain tests work follows in next sections.

Compile

The compile test is the first and basic test for any further actions, so it runs locally. It seems to be trivial, but protects the developer against preventable failures.

There are two categories of the test. The simple compile test just builds the available version after cleaning the old binary files up. It is the lowest effort level and should not give much trust. Especially after the changes from other developers pulled, it could be dangerous to rely on the results of this test case. The second category sets all the configuration new. This test case is safest but the most expensive at the same time. The full compiling of the JULEA project takes about several minutes in contrast to few seconds in the first case.

The developer should be able to chose the level of compiling tests, while there should be a default test, which causes no noticeable effort if not disabled. The keyword in the commit message would overwrite the set defaults.

Correctness

The correctness test should run only in case of successful exiting of previous compile test and locally. The test contains about 40 tests, some of which test negative behavior, means to test if the program exits nonzero in faulty case. The script exits non-zero if at least one test behave unexpected (fail or pass if shouldn't). In case of failing a test, it would be anyway nice to have the results of other tests and the error code to improve the debugging.

This test is enabled per default, but can be disabled in the commit message.

Performance

This group of tests run after the changes are already done, means after the Git commands `commit` and `push` and can not take any influence on them.

There are two types of performance tests which differ in extent and duration. The largest test group contains all the other performance test cases. Since one of the groups is performed in local work it should not take much time and delay the developer in his intent. For this reason the smallest and shortest tests runs locally per default.

The results of all performance test are saved in a separate directory, which is a separate repository. Local results are visualized and both, raw data and finished diagram files, are committed in this repository. The scheme is shown in Figure 3.2 focused only on the performance tests. All the results are committed to an own branch, which name contains the hostname of hardware the tests ran on. It creates an ordered structure due to lots of different machines the tests run on. This way is necessary to avoid overwriting old results or combining and comparing results from different hardware.

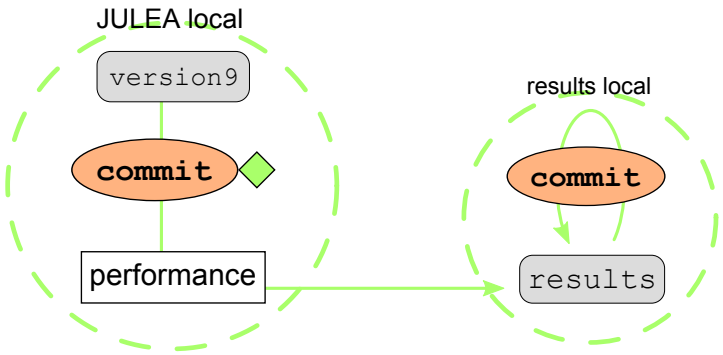


Figure 3.2.: Local performance tests

The remote performance tests are per default parallel and require some more effort. To trigger the tests after some changes are pushed, two different push actions are necessary. The common push to upload the data happens on a web server, where is no opportunity to run any tests due to missing hardware. To trigger the test a second push to another remote repository is needed. Once pushed there, the test case set in the configuration will run on the suitable hardware (look up the defaults in Listing 4.17 or in the documentation in C). This is at the same time the control mechanism of the tests. There is no opportunity to use a commit message or any other option to enable or disable the tests. Once pushed to the repository with the `post-receive` hook, you can not take any more influence on the run. The Figure 3.3 shows the global scheme of the performance tests. There are two remotes - the web and the cluster. The web remote server is used to store the project files, run the web server and additionally supports tools like redmine (project-management and bug-tracking tool [6]). For this reason there are no hooks triggered here. The second remote keeps also the project files and is able to compile them and run applications like performance tests. The results of the remote

tests are committed in the local results repository on the remote machine. All the results, whether from local work or from remote tests are pushed to the web remote server.

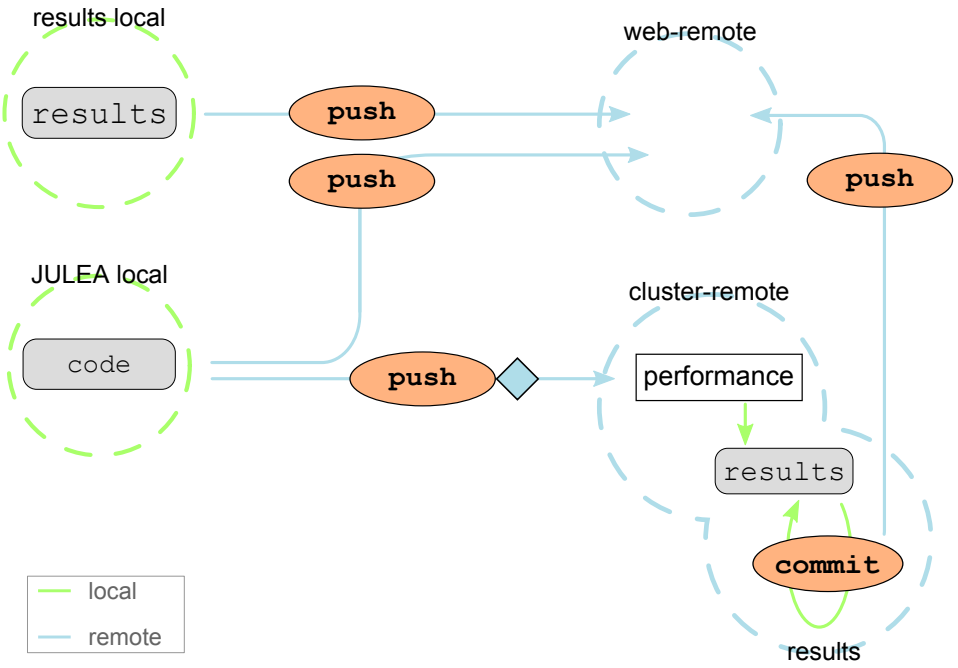


Figure 3.3.: Remote performance tests

The hook will currently not be disabled if the developer tries to push an auto generated branch, which is believed to contain errors. For this case is supposed to think the developer forgot to clean up the branches and the test runs are permitted. It is a question of experience with this concept to integrate stricter conditions. Since remote test runs produce lots of output files, take much time and have high resource requirements, it could become a problem to often run these tests on faulty software. If the developer proves to be forgetful, the solution is a `pre-receive` hooks. This hook would check the name of the pushed branch and is able to abort the push and respectively the remote test run to prevent sequence errors from a faulty branch.

In case you do not want to trigger any remote performance tests, just skip the second push action.

The performance tests are used to check the performance of the file system but can uncover some errors, which the correctness tests did not register. For this reason the output of the performance tests should still be readable and the error report well ordered.

3.2. Result Visualization

The goal of all the project is to allow better access to the evaluated results of the performance tests, which is the basis for conclusions for further development. The best way to get fast access to any information seems to be visualization.

The final step in the test system is generating of graphs and diagrams which show the performance of every development step. For the local work it is every commit, for the remote - every push. Due to the big number of pushes and even bigger of commits made in a project, it is not very useful to picture all the results. An appropriate solution is to look at results on a monthly basis. Since there usually exist more then one commit per day it is not enough to provide the x axis with the day, but is necessary to identify each commit. For this plan the commit ID or its hash value is very useful, which is with 38 characters length too long though and should be shortened to a few characters. These will probably not repeat in this relatively short period.

The next question is how much and which information is relevant enough to be plotted. There should not be too many different graphs in one diagram. To still hold it clear about three diagrams and two parameters, which are plotted, should be a good guideline. In case of our performance tests there are in general two significant parameters - time and throughput. The first one is available as total run time and time without setup. Both is given in seconds and could be united in an elegant way, since we can always say, the total time is never less than the time without setup. It allows to anticipate the relative trend of the graphs and put in diagram without overloading it. The throughput is given in operations per second or in byte/sec, depending on the test case. Reading or writing operation usually measure byte/sec.

Over the time the diagram could become very wide, so that it is difficult to compare results from the first and last days of the month. With every new commit the developer wants to improve or at least to hold the best present result. With a straight line over the best performance it will be easier to compare current results. It is a kind of philosophic question to set the line on the best result and compare if the change got better, or to the worst result and compare if the change got not worse. Technically both ways are possible. In this thesis the optimistic way is chosen.

The proportion of runtime and global time is not the only trend we can expect. Moreover, the behavior of time and throughput are not independent. In case if the time graph goes higher, meaning the run took more time, we expect the throughput to go down (provided no changes on test cases). So we expect the maximum of the throughput graph at the top and the minimum of time at the bottom of the diagram. This fact excludes collision of both straight lines, which would be very inconvenient.

A possible sketch, how the finished diagram could look like is shown in Figure 3.4.

Here we can see some more details to note. We could exclude overlapping of the straight lines of maximum and minimum of the graphs, but not the overlapping with the x-axis. Since the minimum is on the bottom, it needs some distance to the x-axes to avoid overlapping. The distance should not be substantial, but large enough to differentiate the both lines.

Another question is the range of the y-axis. Like in the sketch, it does not start at zero. Starting at zero, the changes of different runs could get lost in the graph due to relatively little small percentage of the total value. It depends on numbers - for example the difference of ten seconds in two runs would still look like a straight line, if the total value amounts to thousand. On the other hand, especially for time measurements is

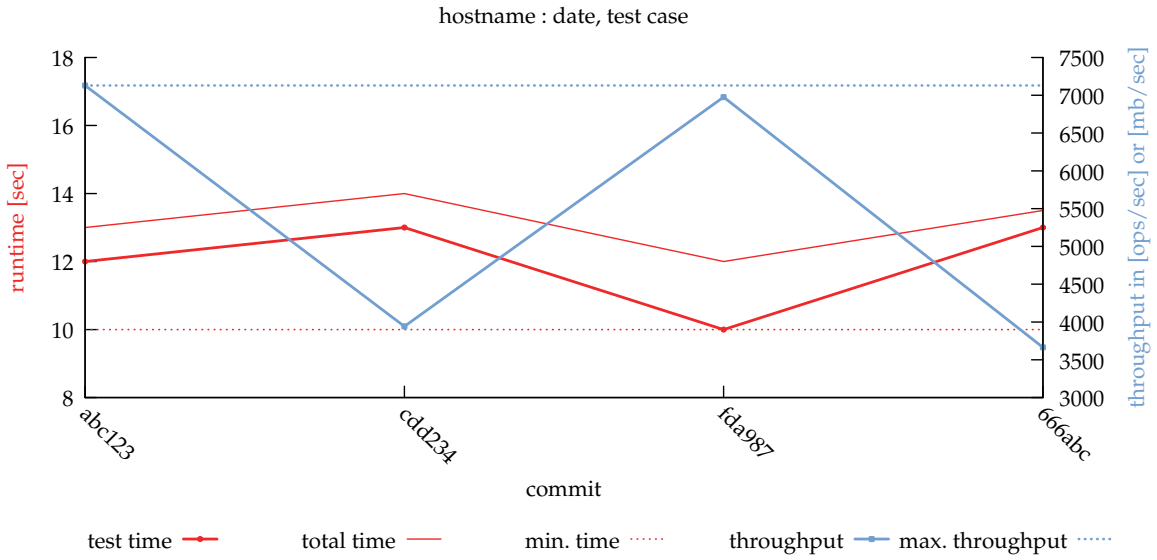


Figure 3.4.: Visualization sketch

important not to exaggerate with accuracy of changes. In case of looking only at changes, a difference of some hundredths of seconds by a total run of ten seconds could look in a graph like serious increase. These could cause wrong conclusions if not looking carefully. Here it is very important to find a balance. For the time it could be reasonable to display a range of about 20% of the total value, but at least three seconds, to account for both problems.

The next important technical requirement on the visualization script is the independence of visualization step from the result generating step. The script should visualize available results separately in case of some parameters in the visualization (like color of lines) were changed. So it has to be independent from time, hardware, directory, repository or any other resources it is run on. That means, the results must have all the information needed for plotting. But at the same time they have to be in raw format, in case the visualization tool, which requires a certain output format is changed. This proceeding makes the test system more flexible, although at the same time integrated.

An alternative would be to trigger the visualization as soon as the results are ready and committed, for example in a post-commit hook of the result repository. Once the results are committed, this commit would trigger the visualization, which could be also committed without any hooks later.

In this thesis gnuplot will be used to visualize the results. See more in chapter 4.2.4

4. Implementation

This chapter shows the implementation of ideas presented in previous chapters. It begins with an introduction to shell scripting language which is used for implementation of the scripts. In the following the most important aspects of scripts and tricky problems, which require not banal solutions are discussed.

The tests are distributed to three hook scripts depending on their meaning - the `commit-msg`, the `post-commit` and the `post-receive` hooks. The last hook is the only hook used for remote work. These hooks consist of several scripts, some of which are built in the hooks, some are called, like the gnuplot script for the result visualization.

The last section of the chapter presents some hints how to debug shell scripts.

4.1. Shell

The shell is a command-line interpreter, that provides an interface for users and a scripting language at the same time. Nowadays there exist many popular shells - bourne shell (`sh`), bourne-again shell (`bash`), c shell, debian almquist shell(`dash`), used in Debian/Ubuntu and many others. Here the common bourne shell is used, which goes as a symlink to the `dash` since the last one is aiming at POSIX compliance [7]. It is smaller and faster than the bourne again shell, contains less dependencies, but accordingly is less features-rich. For the asked tasks the `dash` is nevertheless sufficient.

Working with a shell is like doing an input to the terminal and awaiting an output. There are three channels, through which the information goes in or out between the program and the environment (e.g. terminal). Channel 0 is standard input or `stdin`, normally it is done via keyboard. Channel 1 - standard output or `stdout` and channel 2 - standard error output or `stderr` are usually printed to the monitor. The numbers of channels are integer values of the file descriptor, an abstract indicator for accessing a file, used by the kernel.

Everything typed in the terminal goes to the shell, which execute the commands. The terminal is the `stdin` of the shell. Possible input are for example built-in programs, like `cd` to change the directory or user specified programs. But shell provides not only program calls. Every shell can keep information in variables. Each variable has a name and a value, which can be empty. To define a variable in command line a common = is used without whitespace before or after. Referencing a variable is possible with the dollar character in front of it. To look up the value it is not enough to type the referenced variable in the terminal, because shell would interpret it as an input and try to execute the command. To provide the variable value onto the screen use for example the `echo` command. It is also possible to nest variables, like to write the output of a program into

a variable and then print it to the screen.

The shell keeps all values as long as the session is not exited. Long sequences of commands can be combines in scripts. Analogously to source code, scripts contain commands to be executed by shell. They can be used to automate regular workflows. Just call the script by its name in the command line. The first line defines the executing application, shown in Listing 4.1.

Listing 4.1: Define script language

```
1 #!/bin/sh
```

Note, there must not be any new line or whitespace or tabulator. `/bin/sh` is the path to the standard shell on probably all systems, which is a symbolic link to the `dash` here.

Shell will return the control after all commands are executed and finished or if the script was interrupted by an explicit exit or any other trouble.

It is not only possible to sum up the input in a script for safe and comfortable work, but also to redirect the output in an specified file. Redirection means capturing the output (from a program, command, script etc.) and providing it as an input, so eliminate intermediate result. Especially for debugging it is often annoying to search for an error in thousands of lines output on the screen. To redirect `stdout` to a file use

Listing 4.2: Redirect stdout

```
1 command > file
```

Note, if `file` does not exist, it will be created, if it does, it will be overwritten without asking or warning the user before. This operator can be also used for redirecting an empty output to truncate a file to length zero if present, or to create a new empty file if not. Since not every shell provides a whitespace as an empty output, use the ":" command.

Listing 4.3: Truncate or create an empty file

```
1 : > file
```

This way is more elegant than removing a file, caring for error if not existent and creating a new one with `touch` command. The double >> character behaves like the >, except appending output on the older and not overwriting it. The two examples refer to `stdout`. Using file descriptors it is possible to control any channel separately. The redirections can also be concatenated with a whitespace. Moreover use the ampersand syntax to redirect one channel to the destination of an other already set channel, but care of sequence. An example is shown in Listing 4.4

Listing 4.4: Redirection of channels

```
1 # redirect stdout to file.out and stderr to err.out
2 program 1>file.out 2>err.out
3 #redirect stdout to file and stderr to file
4 program 1>file 2>&1
```

```
5 #redirect stderr to what 1 was before and stdout to file
6 program 2>&1 1>file
```

Unless input is expected from the keyboard, redirection of `stdin` can be helpful to pass input parameters (no options like `-v`)to a program from a file, if there are many:

Listing 4.5: Redirection of input

```
1 # redirect stdout to file.out and stderr to err.out
2 program <input.txt
```

A special way of providing input to commands especially in scripts is a *heredoc* or *here document*. The UNIX shell syntax consists of << followed by an identifier and closed by the same identifier. In between all the information including newlines, tabs etc. is provided as an input to the command:

Listing 4.6: heredoc

```
1 cat << EOF
2 hallo
3 EOF
```

`cat` concatenates input and prints on standard output. To make up the difference, it is not the same like `cat hallo`, which would cause an error. `EOF`, stand here for "End of File", is a keyword to choose. This keyword without quote interpolates all variables and commands in backticks or with `$(command)` syntax. Putting it in single quotes disables interpolation. The herestring with syntax <<< is not available in shell.

One more kind of redirection is the pipe. A pipeline consists of chain of processing elements, where output of one command is input of the next one. The data stream works like FIFO (first in first out). The elements are separated by the pipe character "|". Every following command applies on the result of the left neighbor.

Listing 4.7: Pipe usage

```
1 echo $(ls -t1 | head -n 2 | sed -e 's/\(.*\)\..*/\1/')
```

The example 4.7 returns the name of the second file in the current directory without the file extension.

4.2. Scripts

All the scripts are written in bourne shell, except the gnuplot script, which contains a big part of gnuplot syntax.

4.2.1. commit-msg

This hook is triggered after the user called the commit command, and before the actual commit is done. The hook contains two tests, compile and correctness tests. While

compiling commands are build in the hook, the correctness test is one call of a separate test script already present in the repository.

The simple compile test just calls `waf clean` and `waf` to compile the version with all set dependencies and environment variables. `waf` is an open source tool for build automation. It is more comfortable to use and produces nicer output than the common `make` command in GCC[1]. The most extensive level removes all the dependencies and resets the whole configuration with

Listing 4.8: Setting configuration

```
1 waf configure --prefix=\$HOME
```

The level can be chosen by the developer setting the keyword `"Compile-Test: yes|clean|no"` at any position of the commit message, while the simple compile test enabled by `yes` is default. All the output goes to the screen.

The script for the correctness tests contains about 40 test cases. To run them `gtester` can be used, which is a test running utility using the GLib test framework. It is a kind of wrapper, which has some useful options like `--keep-going` to continue if one test failed and produces ordered information output. The output of these tests goes like previously to the screen, since there is no need to archive these results.

The hook before the commit is finished is the critical one because it is the only one, which can abort the commit if any - compile or correctness tests fail. In this case, shown in Figure 4.1 with the exit code `fail`, it nevertheless is guaranteed, that the changes do not get lost.

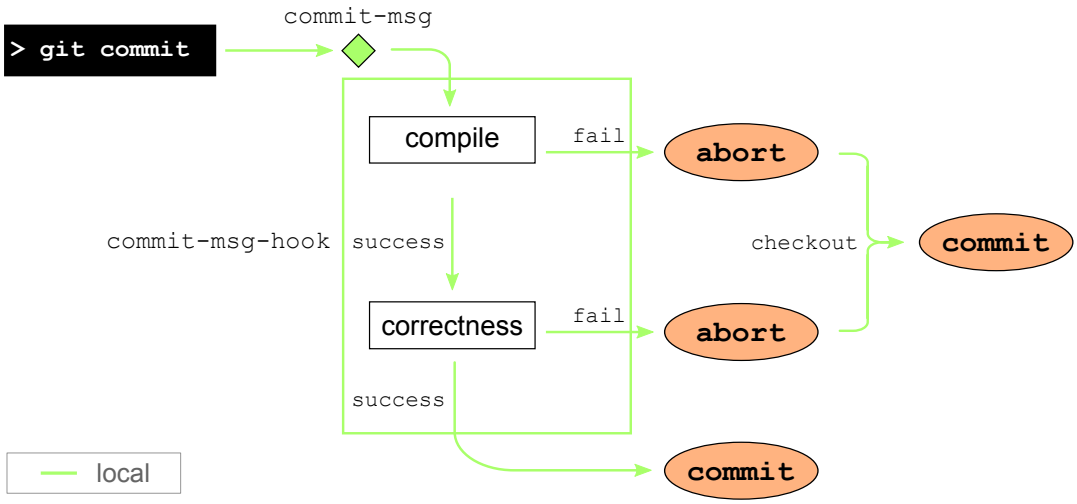


Figure 4.1.: commit-msg hook

Provided working on a correct branch, in case of failing a test, the changes would be committed to the debug branch, created automatically if not existent. Once all errors are eliminated, it is the developer's duty to merge or checkout the correct branch. In

[1]GNU Compiler Collection

general it causes problems if the working tree on the auto-generated debug branch was not clean. In this case losses are not acceptable. Every time a commit to the debugging branch fails due to any conflicts, another branch with identifying name will be created. The name is generated using the number of seconds from 1970. The name is not very nice, but practical and probably reminds the user to clean up the branches if there are lots if those. It seems to cause much overhead, but due to cheap branching concept in Git it does not really matter. The only problem is to manage several branches after all bugs are fixed. They will be not cleaned automatically, since the system is not intelligent enough to note when debugging is over. Any removal or other operations which cause information losses can only be done by user of the test system manually.

There is a possibility to bypass this hook by the developer – like said before in 2.1.4 `-n` or `--no verify`. The actual commit can be done after the hook exits successfully.

If enabled, the already done commit triggers the `post-commit` hook.

4.2.2. post-commit

This hook is only triggered if previous tests did not fail, meaning, if the committed branch is not automatically generated. In this case the default (see Listing 4.17) sequential performance test will run. The default can be overwritten in the commit message setting the keyword `"Performance-Test: yes|parallel|no"`. Before running the necessary configuration and paths are set.

The produced results arrive in an the result repository. These are visualized with gnuplot script, which produces several PDF[2] files. All the PDFs and raw results will be commited and pushed. Both actions serve the archiving of results and should not trigger any further tests, so perform without any hook. The workflow is sketched in Figure 4.2.

4.2.3. post-receive

This script is executed on a machine with suitably hardware for performance tests. The hook is triggered after the changes are pushed. The default performance tests in this hook are parallel (see Listing 4.17). The default is a parallel performance test, which can not be changed by the commit message in contrast to the others. Nevertheless the default can be changed (look up in the documentation in appendix C)

First of all the current code version has to be compiled. The next commands set the environment for the remote tests and would normally constitute a separate batch script. Due to common use of several values in the whole script the batch script in integrated in the hook using heredocs.

Batch Script

The larger performance tests simulate real conditions and run at bigger systems to provide more information about the behavior of the parallel file system. As the name reveals, to test a parallel system, we need parallel tests running on a distributed system.

[2]Portable Document Format

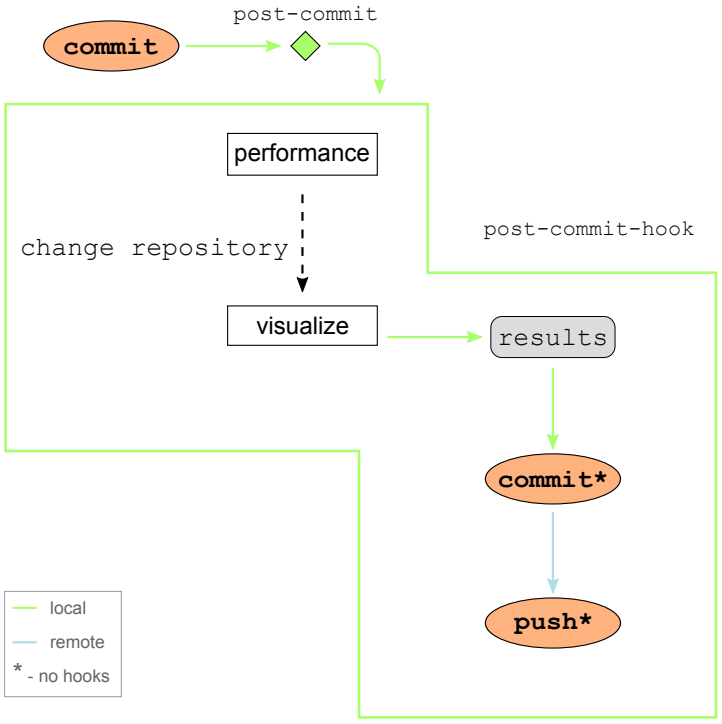


Figure 4.2.: post-commit hook

A common cluster contains at least several worker nodes and one or more master nodes, which usually are connected to the storage disks. In best case all the programs run on worker nodes, while master node is for login, compiling programs or any other smaller tasks and contains the `/home`-directory. To run any program on a node, you can connect the node via SSH and run it manually or submit a job from the master, which contains all the information to execute the application on nodes automatically. This requires a batch system, which monitors and controls the jobs. Additionally there is a job-scheduler (not always part of the batch system), which manages which job gets the required resources at which time.

All the nodes are connected via network like Ethernet. The worker nodes should have access to files in users home directory to run any programs or compute data. Via protocols like NFS[3] it is possible to access remote files, like they were on the local disk. Accessing `/home` is not to be mixed up with having a copy of the master. The system configuration and packages installed on master are not automatically the same like on other nodes. Some required system software is not working on nodes, if not installed there, no matter of NFS. Moreover to note is that usually worker nodes have no Internet access due to security. After a program run finished on a node, it might be helpful to commit or push the results to a repository. Since pushing any changes requires Internet access, it makes any remote work with Git from a worker node impossible.

The structure of a cluster looks like in Figure 4.3.

To run any software on a distributed system batch scripts are needed to execute

[3]Network File System

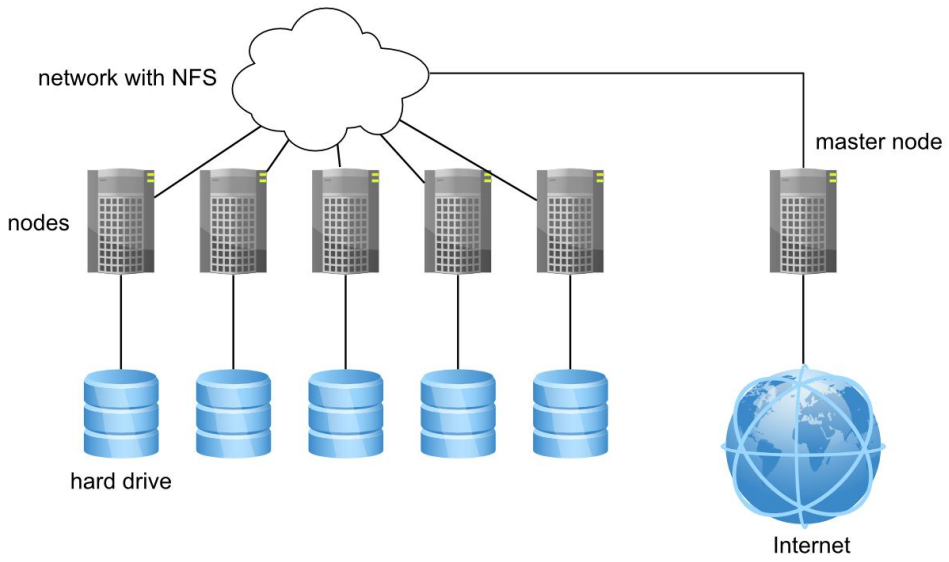


Figure 4.3.: Common cluster structure

serial or parallel programs on several nodes. The batch system on the used hardware is SLURM[4], which is an open source system with a job scheduler. The scripts are at first common shell scripts with common syntax but with some additional options to be set. These options provide resources needed for the job and are set mostly in the first lines of the script in comment syntax without whitespace shown in Listing 4.9

Listing 4.9: Example of options for distributed job

```
1 #!/bin/sh
2
3 # Time limit is one minute.
4 #SBATCH --time=1
5 # Run 10 tasks on 2 nodes.
6 #SBATCH -N 2 --ntasks-per-node 10
```

There are much more options which can be set, like output destination, email of submitter to inform in case of any trouble, job name, partition and many others (look up in bib). The system interprets these options only if the job executed with `sbatch` - a SLURM command to submit a job. the way the actual batch script is integrated in the hook is shown in the Listing 4.10.

Listing 4.10: Integrate batch script in post-receive hook

```
1 #!/bin/sh
2 ...
3 sbatch << EOF
4 #!/bin/sh
```

[4]Simple Linux Utility for Resource Management

```
5
6 #SBATCH --time=5
7 #SBATCH -N 3 --ntasks-per-node 1
8 ...
9 EOF
```

After the script is submitted, SLURM allocates required resources if they are free. All the further commands are executed in the context of specified resources. Any program ran in this script will be executed on one of the allocated resources, but usually not on master where the job was submitted. To run the program in parallel on several specified nodes the SLURM specific command like `srun` has to be set before the actual call. Figure 4.4 shows how a script works in general.

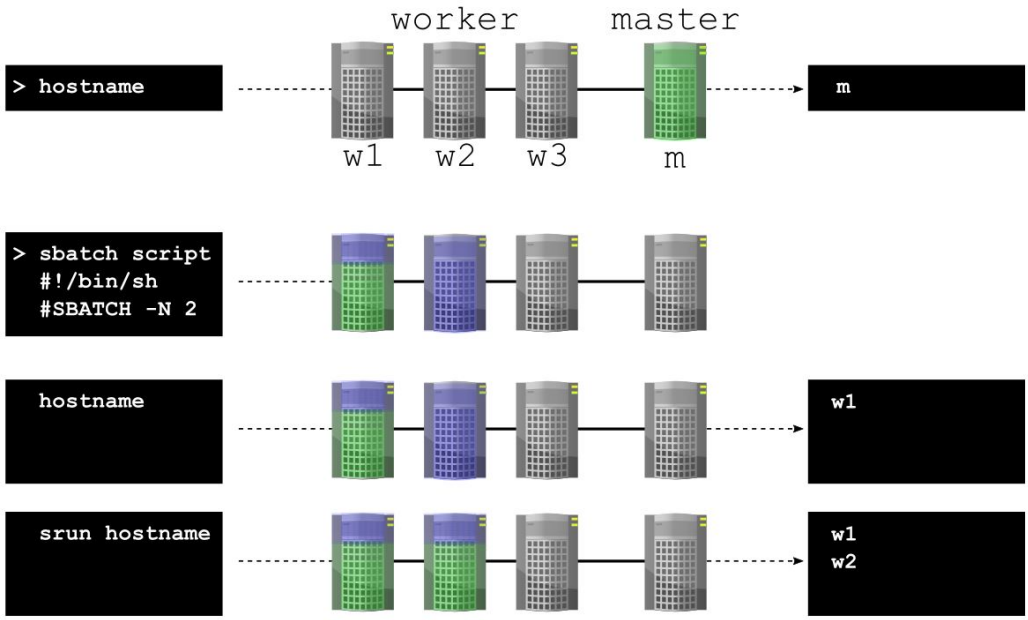


Figure 4.4.: Executing a batch script

The first step shows a command typed in the common terminal. `hostname` is a command to show the hostname of the current host. After the login the user is on the `master` node (shown in green color), whose hostname `m` is the output. The second step shows submitting the batch script. The options set after `#` ensure the reservation of two nodes (blue color) and change to one of the nodes, here to the node `w1`. Executing the command `hostname` in the batch script causes now an output from the node `w1`. Putting the command `srun` before `hostname` ensures that all the nodes involved execute the command `hostname`, so the output comes from the both nodes.

It is possible to find out which nodes were reserved for the job to target the specific node separately. This information is very useful for the test setup, since all the daemons of JULEA have to be restarted after each test case. In a loop over all involved nodes the setup is called via SSH.

Submitting a batch script in contrast to a common program call will return the control immediately after the execution, no matter if the content of the script has finished or not. All the actions to be done after the results of distributed benchmarks in the batch script are finished, have to happen in the batch script, since an action outside it would not note if the results already exist or not. The results have to be visualized with gnuplot, which is additionally not necessarily available on all worker nodes. So we need to connect the master node from the batch script. This fact leaves hardly any other opportunity than to use heredoc again. Looking irritating, nested heredocs cause no technical problems. Only the keywords have to be different:

Listing 4.11: post-receive hook structure using heredocs

```
1  #!/bin/sh
2  ...
3  name=$(hostname)
4  sbatch << EOF
5    #!/bin/sh
6    srun <test.sh> ...
7    ...
8    ssh $name << __EOF
9      ...
10     <gnuplot.sh> ...
11
12     git commit ...
13     git push ...
14   __EOF
15 EOF
```

The name of the master node is defined before the control goes to one of the nodes defined in the batch script. Within the batch script part all the commands execute sequentially, so that it is guaranteed, that the test run is finishes before connection to the master. From the master runs the visualization of results. The necessary git action can also be done on master.

4.2.4. Gnuplot

Gnuplot is a command-line program for generating graphs, visualization of functions and data. Gnuplot is portable and is available on several popular operating systems. Since gnuplot commands require its own context, there are some options how to use it in a script. One of these is to create a pure gnuplot script which will not contain any shell commands, but will be called from another script. The second possibility, chosen here, is to integrate gnuplot commands in a common shell-script. To integrate gnuplot command into shell script a trick is necessary. The naive solution seems to write down gnuplot commands like it would be in the terminal. This process is shown in Figure 4.5. At the line where gnuplot is called, opens the context of gnuplot awaiting commands. But all

the following commands are interpreted by the shell script in contrast to how it would be in the terminal. Empty gnuplot context returns to the main line of shell script and proceeds with errors.

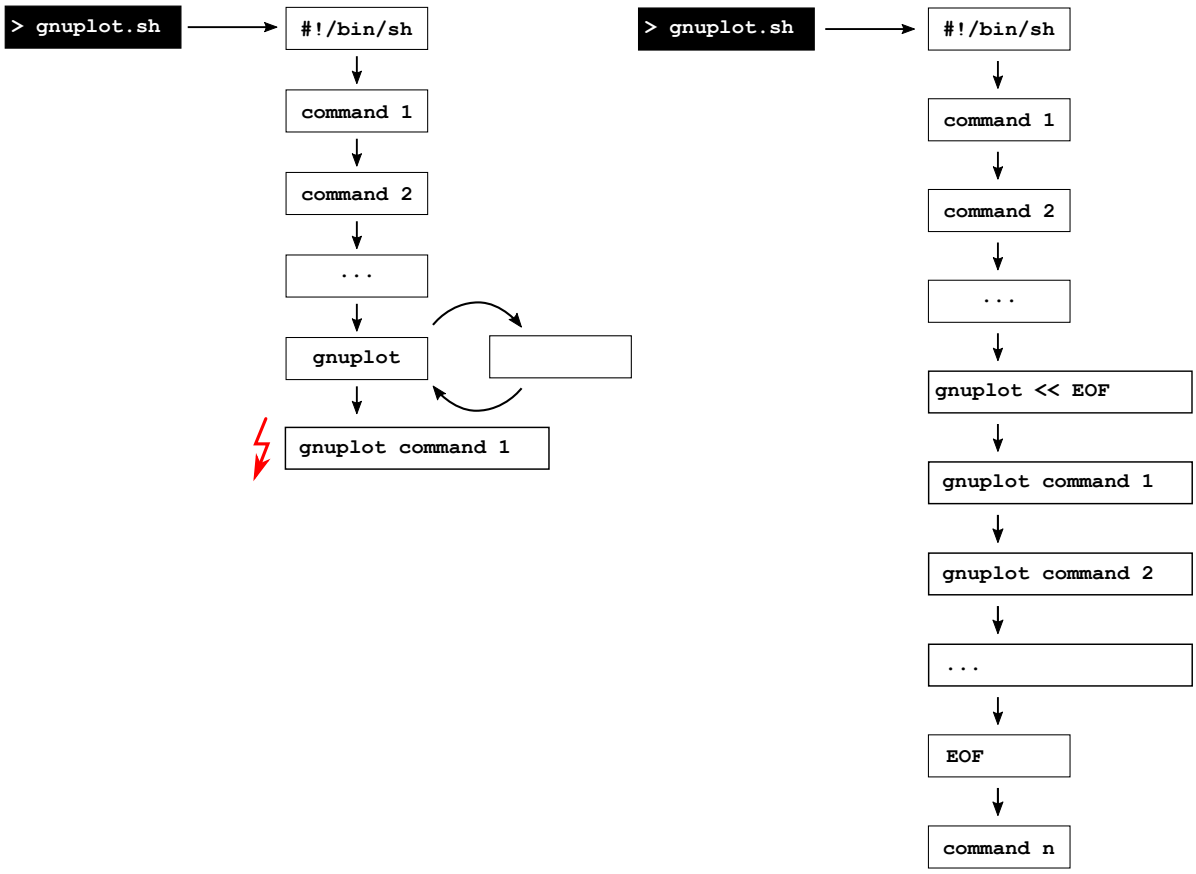


Figure 4.5.: Integration of gnuplot commands in a shell script

All the gnuplot commands have to be provided to the input of gnuplot context with the heredocs explained in 4.1. The problem to note is, that all the variables beginning with "$" will be interpreted by the script like parameter variables form the shell terminal. It causes problems by using the syntax for processing gnuplot variables, like values of certain columns in gnuplot. For example $1 in shell script is the first passed parameter, in gnuplot it would be the first column of datafile. To use it nevertheless, the dollar character of gnuplot variables has to be escaped with a backslash e.g. `\$column_number`. The integrated variant was used because of its advantages, namely the use of shell processed variables. For this reason it is not the solution to completely disable interpolation of variables. It is not gnuplot's forte to handle any strings or other variables or to apply functions on them, which is much better possible with shell. For this kind of luxury it is worth to accept the inconvenience of dollar-character syntax.

The major part of the script are settings and options. The type of diagram and its grid have to be configured. Here you can set the width of lines, the color, the position in the whole output file, title and so on. The labels of axes, its scalability and range

and many others. The axes configuration is important. Here you can chose the scaling, title, mirror axes (up and down for x, and left and right for y). Moreover two different x or y axis with different scales are possible (gnuplot can manage also more dimensions, which is not important in his thesis). Here one x-axes shows the time in seconds and the other throughput in operation per second or MB per second. The choice depends on the availability of the last named data. For this gnuplot supports the condition statement.

The `autoscale` option for the axis scales the range depending on data to be plotted. The range goes from the minimum to the maximum of the values and cross the x-axis. To avoid it set the range manually above the min and max values. The time range starts at the minimum testrun time value minus three seconds and goes to the maximum total time value. The maximum value of the test run time is not suitable, since the total time could still be more than three seconds longer, so this graph would be outside the diagram range. Negative values are rounded to zero. In case of the trhroughput there are no absolute numbers to use like three seconds. To enable a distance from the axes there is a percentage of 15% to subtract and add to the minimum and maximum values accordingly. In case the both values are similar, the graph would look like a more or less straight line.

Listing 4.12: Calculate ranges for the x-axis

```
1 y_from=((loc_time_min-3)<0 ? 0 : (loc_time_min-3))
2 y_to=(glob_time_max+3)
3 set yrange [y_from:y_to]
4
5 y2_from=($col==5 ? ((thr_min-(thr_min*0.15))/1024.0/1024.0)
    ↪ : (thr_min-(thr_min*0.15)) )
6 y2_to=($col==5 ? ((thr_max+(thr_max*0.15))/1024.0/1024.0) :
    ↪ (thr_max+(thr_max*0.15)) )
7 set y2range [y2_from : y2_to ]
```

The `col` variable is set before the gnuplot commands part began and has the value of the preferred available column (4 if no data in the fifth column is set). The condition syntax is similar to the if statement in other programming languages.

Gnuplot gives the opportunity to produce plots in a large number of different formats. The output of gnuplot can be produced in two ways - on the screen or in different format files in specified directory. At first a terminal has to be set. Different terminals behave differently with fonts, encoding like UTF, colors or line types. There are also different default values for the file size or other settings. One of the supported formats is PDF, which will be used here regarding further possible use for the website. The terminal `pdfcairo` supports also the correct UTF-8 encoding. Additionally to the terminal you can define some setting like font and file size. Additionally to the terminal you have to set the path to the output file. Some terminals like `wxt` do not need this path since the output goes goes directly to the screen on a separate window. This example shows how to configure the output and some additional options:

Listing 4.13: set terminal and output path

```
1 set terminal pdfcairo enhanced font "Palatino,10" fontscale
    ↪ 0.6 linewidth 2 rounded size 5+($number)/2cm,7.5cm
2 set output "$ROOT/pdf/${toplot%.toplot}.pdf"
```

Once the output is set, any plot command will generate the according graph. Gnuplot supports plotting not only mathematic functions, but also data. Moreover it is possible to apply any function on the data before plotting. To plot data the path to the data file has to be given in quotes as first argument of the plot command (single or double depends on interpreted or ignored variables). It follows the keyword `using` to define which data is plotted against which axes:

Listing 4.14: Plotting data

```
1 plot '$ROOT/toplot/$toplot' using 3:xticlabels(1) title
    ↪ "total time" with linespoints linewidth 2 linetype 1
    ↪ pt 7 ps 0.2 axes x1y1
```

In this example the data of the third column is plotted against the first column values while these are also labels on the x-axes. Additionally the title and line characteristics are specified. At the end you can set certain axes to be used, if there are several x or y axes with different meanings. To plot additional curves in one graph you do not need to repeat the whole command, but to follow after a comma. In Listing 4.15 is shown the example.

Listing 4.15: Plotting additional data columns

```
1 '' using ($col==4 ? \$4 : (\$5 / 1024 )) title "throughput"
    ↪ with linespoints linewidth 2 linetype 3 pt 5 ps 0.2
    ↪ axes x2y2
```

Here an if statement is also possible. Depending on the value of `col`, the fourth or fifth column will be plotted. Since these two variables come from gnuplot, they have to be escaped with a backslash. Combinations with any other mathematic operations or functions are possible, like dividing by 1024 to plot Megabytes instead of bytes.

But it is not the only use of the plot command. Two functions exist to compute minimum and maximum values of any columns:

Listing 4.16: Minimum and maximum functions in gnuplot

```
1 max=GPVAL_DATA_Z_MAX
2 min=GPVAL_DATA_Z_MIN
```

These two functions are available since version 4.2. These functions can be applied to the data defined before. To fill the data also the plot command is used, with the difference of no output. It means, the defined data exists in gnuplot, but produces no output with its visualization. It is also possible to apply the functions to already visualized data. In case the raw data visualization is not needed, it would be an additional effort to plot the unnecessary data first and remove it later. The plot command to only fill data has

to be used before any output is defined, or after the output is redirected for example to `/dev/null`. The output destination can be reset any time. The variables with the minimum and maximum values can now be used no matter of further output reset. The value can be plotted as a line parallel to x-aces or printed in a label.

Finished diagrams are presented in the chapter 5.

There are lots of settings to improve the graph, some of which are linked in the Bibliography.

4.3. Configuration

Automated tests should work on any directory structure. The one path needed have to be set in the configuration. It is the path to the result directory, where all the benchmark results are. There is no default for this path to results directory. It has to be set by the developer, since the output consists of a huge number of files, which destination should be defined by the developer, not by default. Look up in the documentation C how to set the path. The path of the build directory, from where commits are done, is automatically known by Git. To get the path of repository root use `git rev-parse --show-toplevel`.

The other kind of configuration relates on all the kinds of test defaults. The set defaults are listed in Listing 4.17.

Listing 4.17: Defaults for testing

```
1 commit-msg          enabled
2 compile test        "yes"
3 correctness test    "yes"
4 post-commit         enabled
5 -performance test   "yes"
6 post-receive        enabled
7 -performance test   "parallel"
```

The enabled hooks can be disabled by the user by renaming them to `<hook.sample>`. Other keyword defaults can be changed in the configuration file (see appendix C). Every option or keyword except the remote performance test can still be overwritten by the certain Git action in the commit message.

4.4. Debugging and Error Handling

Debugging is always a part of software development. Even if there exist lots of debuggers, finding errors is still a challenge. There is a crucial difference between debugging a program in a compiled language and in an interpreted language. In compiled languages to look up a value of a variable you have to integrate the output commands in your program. For every forgotten a value, it needs to be compiled and run again. Debuggers help to access any variable at runtime, but also have their disadvantages like influence on

runtime and changing the run. Interpreter languages execute the commands at runtime. The necessary command to output a value could just be placed between the lines. Using a script you still have to run it again if any new commands added. There are nearly no debuggers for interpreter, which would help to handle errors. So the developer has to step through the whole scripts alone to find the critical line.

There are some hints, which could be very helpful debugging shell scripts. First of all it is helpful to output everything happens in the script, line by line. There are two options [8], which should be set in the script commonly at the beginning, since they would have no influence on previous commands:

Listing 4.18: Set debugging options for output

```
1 #!/bin/sh
2 set -ex
```

These and others can be looked up in help for set command:

Listing 4.19: `help set`

```
1 -x  Print commands and their arguments as they are executed.
2 -e  Exit immediately if a command exits with a non-zero
    ↪ status.
3 -v  Display shell input lines as they are read
```

Exiting immediately helps to find out the critical fails, because the program would stop exactly at the line, that causes abort.

Error which should not cause exit can be caught with this syntax:

Listing 4.20: Catching exit status

```
1 rc=0
2 git checkout -b auto_debug || rc=$?
3
4 if [ $rc -ne 0 ]
5   then ...
```

The double pipe has a special meaning - if the first command exited with non zero status, then execute the command after the double pipe. Exit status is an 8 bit integer returned to the parent process in case if the subprocess exits [9]. The second command is an assignment, while $? holds the exit status of the last command. So the exit status is caught and does not cause an abort. Further it can be asked and used for error message. Using the if statement be aware of whitespace before and after brackets.

Not exactly debugging, but a kind of avoiding errors is the concept of traps. A trap is a command used to associate some code with a particular signal like in the Listing 4.21. The reaction if the signal is received is the executing the specified code. Signals are triggered under certain conditions. In case of termination the Signal 15 or SIGTERM is sent. The number depends on the architecture.

Listing 4.21: Shell trap

```
git stash -q --keep-index
trap "git stash pop -q; rm -f ${config_file}" HUP INT TERM 0
...
git stash pop -q
```

The trap in the Listing waits for a signal `HUP, INT, TERM` or `0`. The meaning of these signals is shown in the Listing 4.22. There exist many other signals and two special signals, which can not be trapped - `SIGKILL` and `SIGSTOP`. These signals are handled on the kernel level.

Listing 4.22: UNIX signals

```
0       Command completed successfully.

SIGHUP  Hangup detected on controlling terminal or death of
   ↪ controlling process
SIGINT  Interrupt from keyboard
SIGTERM Termination signal

SIGKILL Kill signal
SIGSTOP Stop process
```

If the signal is caught, the command in double quotes will run. Several commands are separated by a semicolon. The trap is placed on lines where it is necessary guarantee that the command will be executed.

To summarize debugging scripts is not trivial. Shell provides some helpful mechanisms, but debugging gnuplot commands redirected with heredoc is nearly impossible, since the debugging options not apply to these commands. Every change has to be tried out and considered at the final with a PDF-viewer.A more realistic way for searching errors is searching by testing.

5. Demonstration

The previous chapter presented the idea in theory and implementation. This chapter shows the final results in the form of diagrams in PDF format. The basis for the results are several commits from the JULEA repository. The default performance tests ran within the local hook `post-commit` on one of the nodes of a parallel cluster. The node has 2 processors (Intel Xeon Westmere 5650 @ 2.67GHz) with each 6 cores while only one processor was used, since the local tests are not parallel. Generating PDFs of deafult test results over 15 commits tales about 5 seconds.

The first Figure 5.1 shows the diagram for the test case `item/write`.

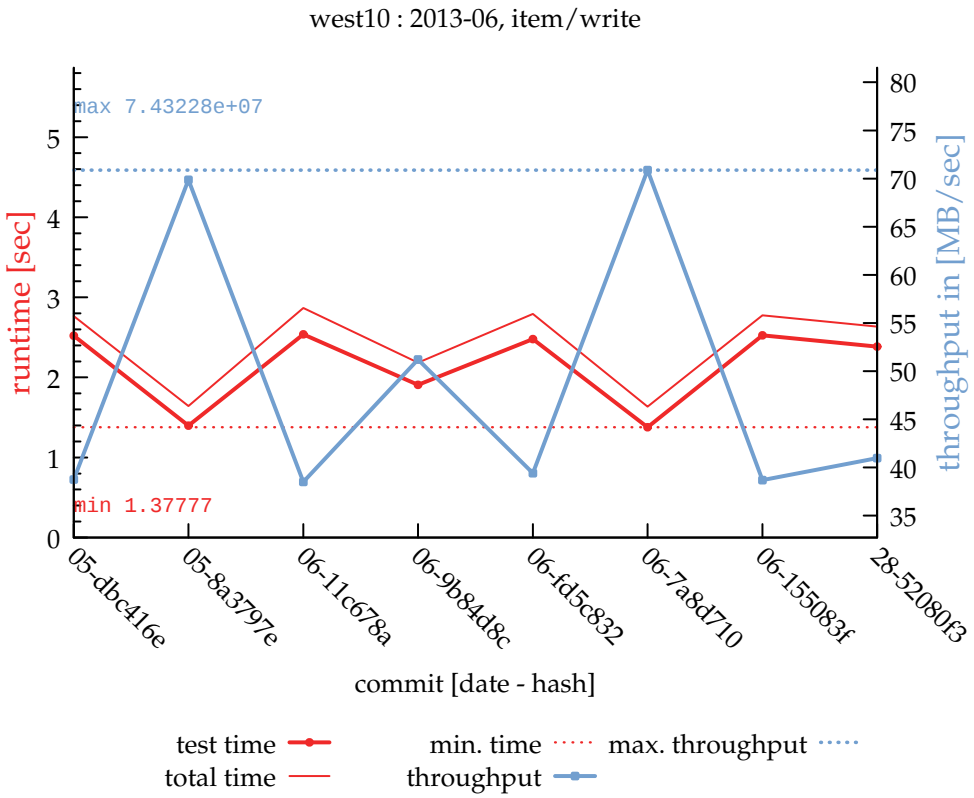


Figure 5.1.: Diagram for test case item/write

The left y-axes shows the runtime in seconds. The second shows throughput in Mbyte per second. The commits were created in the July of 2013. The accompanying days are shown on the x-axes, represented by the first two digits. The rest of the label are the first

digits of the commit hash. The tests ran on the `west10` node. The colors of the y-axis labels correspond to the colors of the graphs. The test time is plotted with thicker red line, like said in the legend. In this test case it varies in the range of less than two seconds, while the total time is constantly few tenth of a second higher. The total time measure includes the time for the setup and cleaning up after each run. The lower the runtime the higher is the throughput, while less time for the run and higher throughput mean more efficiency. Moreover, halving the time, no matter whether test time or total time, means doubling the throughput. The graphs look symmetric, but not exactly mirrored - the throughput graph is stretched vertically. The reason is the different scaling of the y-axis. Nevertheless it is at first difficult to say whether these dependencies are always valid for other test cases. The runs presuppose, that the benchmarks do not change, otherwise it would be impossible to draw conclusions using results of different test conditions.

The next example shows the test case `background/operation`. The graph of total time seems to be not available, but is in fact overlapping the test time graph. The time graph varies hardly, while the throughput is varying clearly. The scalings seem not to perfectly match and falsify the effect.

The benchmark's main task is to help analyzing the codes performance. But sometimes the graphs show limitations of the benchmarks themselves. While the correct axis scaling is the problem of the visualization algorithm, the low absolute runtime of few seconds does let the developer get just a superficial impression. To draw any firm conclusions the runtime has to be higher. However, this might still be sufficient for local sequential tests.

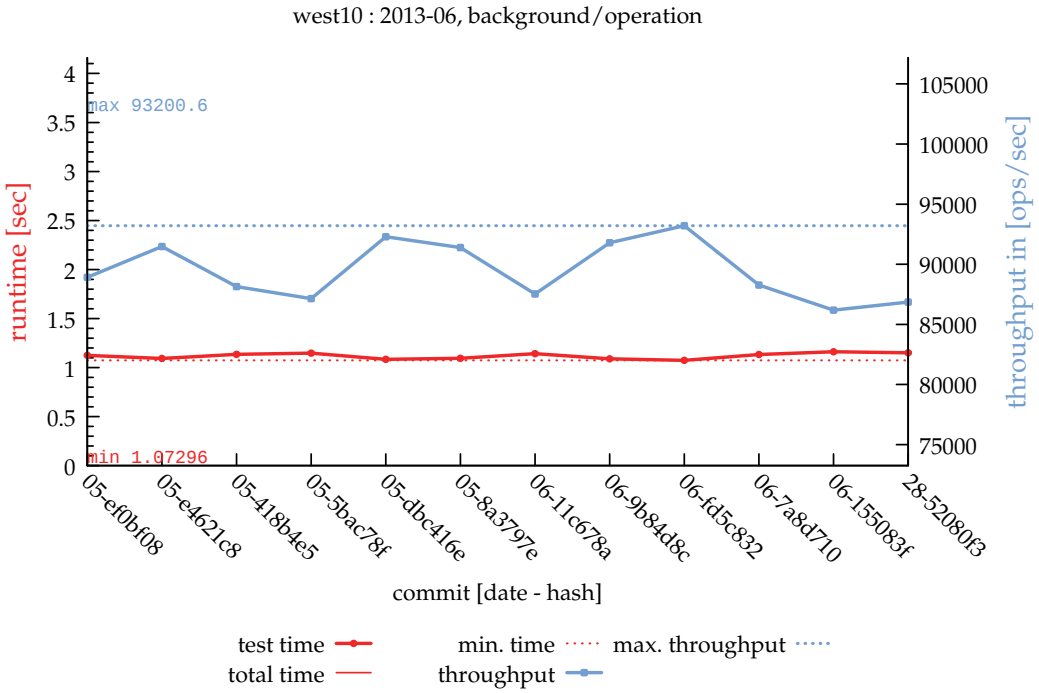


Figure 5.2.: Diagram for test case background/operation

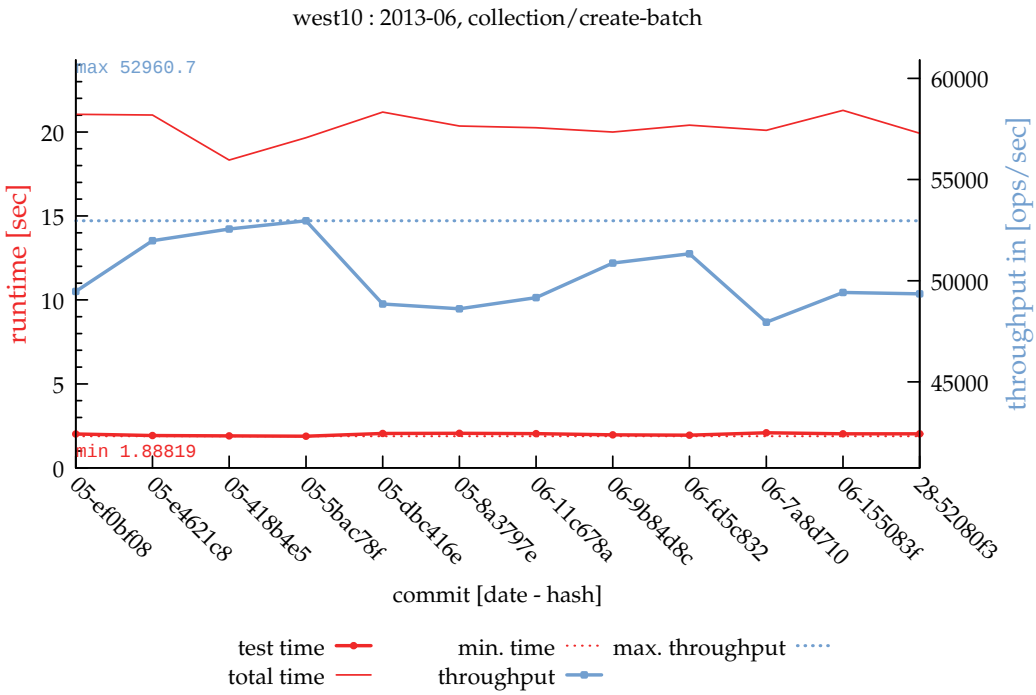


Figure 5.3.: Diagram for test case collection/create-batch

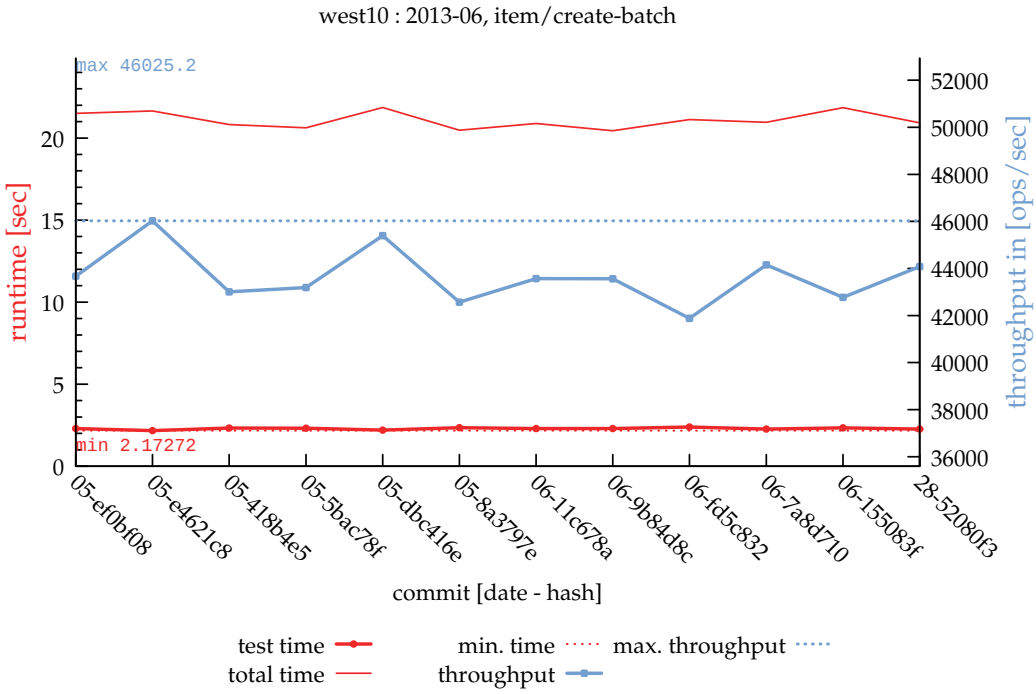


Figure 5.4.: Diagram for test case item/create-batch

The two Figures 5.4 and 5.5 to find analogies in dependencies between the test time, total time and the throughput. The test cases are similar - creating batch once for the directory (`collection`) and once for a single file (`item`). The total time of both is much higher than the test time. In these diagrams the throughput does not seem to depend on any time, since the test time is hardly varying and the total time trend does not match the throughput trend. Taking a look at the raw data we can clearly say that the throughput is inversely proportional to the time, which is not clearly apparent looking at the graph. This example shows, that is can be helpful to compare different test cases to see if there is anything wrong. It would suspicious if the total time would not be as much higher in one diagram than in the other.

The extreme during the tests detected case is shown in Figure 5.5. Here the test time is about 0 seconds constantly over all commits. The throughput value is not only changing greatly, but is also very high in absolute numbers. This benchmark can hardly give any helpful information, since there are probably not enough objects or operations involved to make a reasonable run. Here it is urgently necessary to find out the reasons and improve the source code or the benchmark test, depending on what of those both lead to these results.

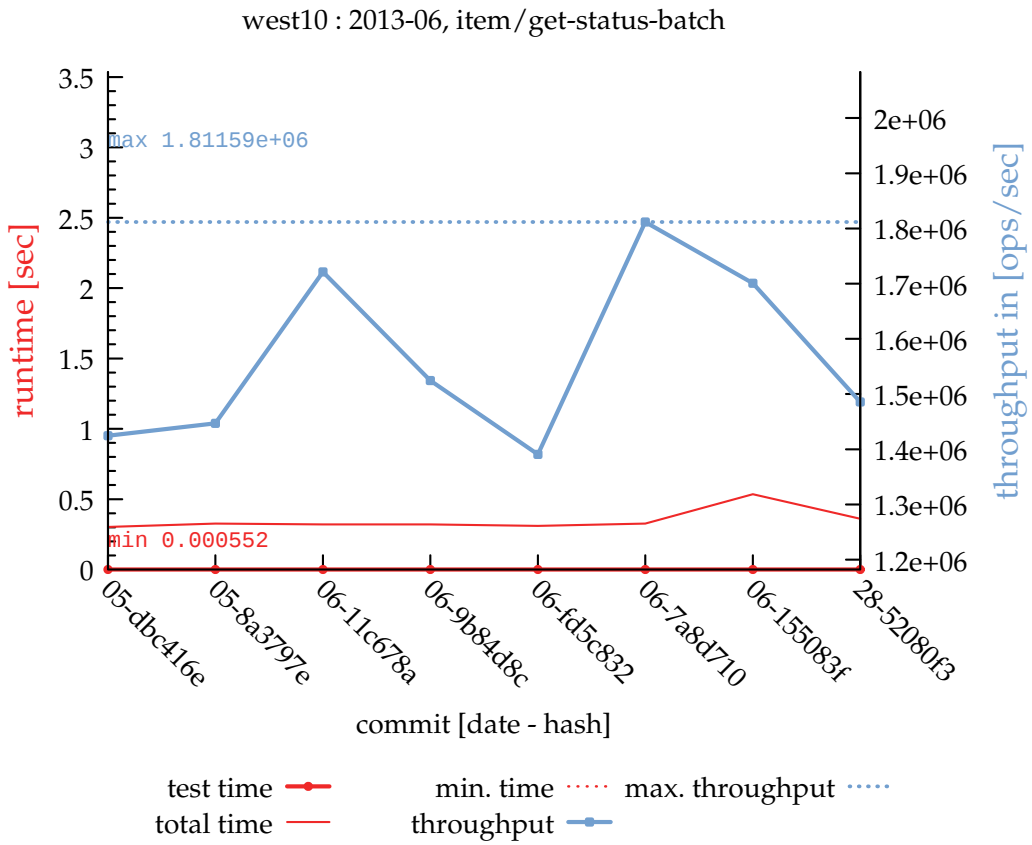


Figure 5.5.: Diagram for test case item/get-status-batch

6. Related Work

Automated testing is a part of continuous integration system, which consists of several steps. These depend on the system and definitions, but in general are [10] [11]

- monitoring the VCS
- updating the local system in case of remote changes
- compile tests
- integrating tests
- generate test report

These systems require a common code base and a VCS on it. Building the system has to work automated without any manual influence. In the best case, builds should be quick not to delay the development process. Further there have to be fully automated running tests.

There are several smaller and at least two big open source continuous integration tools. These are Jenkins [12] and Hudson [13], both written in Java and came from the same roots, while Jenkins is the fork of Hudson. Both are web based and support at least CVS[1] and Subversion, while many others including Git can be installed additionally. These systems are capable of monitoring any activity in the project, to update changes and to test them, reporting to the developer in case of any bugs. Using additional tools can be checked if any appointed rules or the syntax are broken or variable names do not match the agreements.

The two screenshots below [14] show the use of Jenkins in a project. The build history and build queue are the main parts, since Jenkins was designed for triggering system builds ans reporting in case of bugs. In a table below there are the datum and the status of the certain build presented. Colored highlighting helps to get a quick overview. The second Figure shows even more linked details, for example to the last build, the last stable or broken build. The changes in the code are also linked. All these features are very helpful not only for orientation in a big project, but for debugging in case of any errors.

[1] Concurrent Versions System

Figure 6.1.: Jenkins build diagram

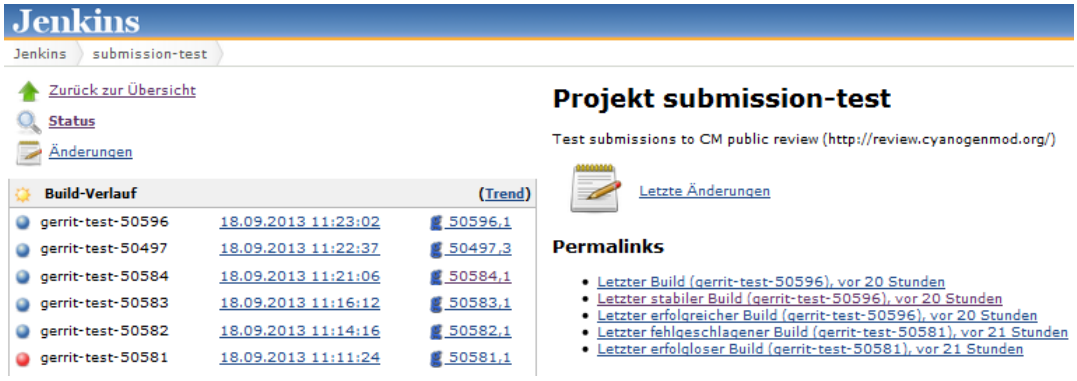


Figure 6.2.: Jenkins submission-test

This is also the main advantage of the integration tools. They work good in bigger projects with many developers whose work has to be managed. First of all the CI[2]

[2]Continuous Integration

systems support a better communication between developers and help them to report about their changes [15]. At the same time they check the versions whether there are bugs and automatically report it to all the developers. The most checks although happen after the changes were checked in, since the system requires a server and cannot be integrated in local work.

At the same time it is a big effort for developer's own work to setup such a big system. The system developed in this thesis is lightweight, better integrable and considers lots of developer's wishes for configuration. It consists of few scripts in contrast to a big server needed for most CI system. The most important difference is that the system enables to integrate testing in local work before changes become public, not only "post factum". So it supports more flexible and careful testing from the beginning. The systems helps not only to debug errors or bad trends, but to avoid them at all. It is possible to configure an individual test system for every developer of the project, while the CI system is central and equal for everyone. Using hooks enables fine-grained work during the whole development process with continuous feedback.

7. Summary, Conclusion and Future Work

In this thesis the idea of a test environment for a parallel file system was discussed in theory and implemented in practice. The theoretical background included a wide area of general problems in terms of testing in software development and in scientific software especially. The encapsulating of the test stages in the develop process seems to be one of the reasons why testing is increasingly neglected. Additionally the great effort needed to operate deters from testing. The theoretical chapters discuss the need to rethink the goals. It turns out that it is not enough to strive for software, which fulfills the purpose and produces results correctly. Considering the quality and efficiency from the beginning of the project minimize risk not only of further errors, but of the wrong development trend in general. It increases the chance to produce correct and prolific software. From there the motivation was to facilitate the processes and integrate testing in the development in a very user friendly way to eventually improve the quality of software, both development and the final product.

The basis of the project is the use of Git. The whole thesis is oriented on its use and presents in a significant proportion how to work with Git. It is important to understand how Git handles the information to capture all abilities the system offers. The knowledge of Git's internals enables the use of very powerful mechanisms like hooks and branches. Especially because of Git is a distributed system a very extended and still effective test environment was possible.

After the technical tools were familiar, the question is how and what to test. The tests were classified in two groups - critical, which check gross errors and cause active influence on the develop process and uncritical, which help to analyze the actual state and draw correct conclusions. The first group was implemented in a client side hook and includes compile and correctness tests of different adjustable extent. The second group represents the performance tests implemented in post-actions hooks on the client and server sides.

The main requirements on the system - to be universal, user friendly and simple were met. The configuration to be done by the system user is minimal. The control over all the information and processes is still in the hands of its user. In case of any unexpected errors the effort to repair the system remains comprehensible due to the extensive support of clean reports and logs. The test results visualized with gnuplot in form of diagrams are easy understandable and concrete and allow quick conclusions.

The implementation is a framework and gives guidelines how to join developing and testing using Git. Even though it remains very individual due to lots of rules and defaults, which can be configured according to the personal preferences in future.

The visualization of the test results allows many fantasies. Accordingly the desire to

enable viewing the results on a web site, there are many options how to handle the results. The one category is to use scripts like gnuplot to produce output files with diagrams. Here any other tool can be used, from commercial Matlab, Maple or Excel for Windows to self written Python, C or Java programs. The other possibility is to embed some web APIs[1] for graphics directly on the web site. An open source candidate would be the Google Chart API, which is admittedly deprecated since 2012 but still works. The tools used by Google now is SVG[2], which is an XML[3] based tool for two dimensional graphics. The main advantage of those APIs is the interactivity, which enables better analyses of the result data. The static diagram has a limit of information it can present. Not every value or trend can be displayed. Using an interactive tool many more important details like the last values, the minimum ad maximum or any other certain value can be shown to exhaust the information potential of the results.

Talking about web site support it would be ideal to link the performance graphics to the source code directly (e.g. via redmine). The commit hash shown on the x-axis would be a link to the committed version. Here the developer could very comfortably recognize the roots and reasons for any performance changes.

It is not unlikely that benchmarks can change during the time. The system could note it and set a visible sign in the diagram. Especially running the remote parallel performance tests it can be useful, if the tests run using different resources configuration, like number of nodes and processes.

There is a great potential for improvement of the visualization with the goal to promote the users efforts.

[1] Application Programming Interface
[2] Scalable Vector Graphics
[3] Extensible Markup Language

Bibliography

[1] Michael Kuhn. A Semantics-Aware I/O Interface for High Performance Computing. (7905):408–421, 06 2013.

[2] Software Development Workflow. http://www.thoughtworks.com/de/continuous-integration. [Online; accessed 01-September-2013].

[3] Git Compression of Blobs and Packfiles. https://gist.github.com/matthewmccullough/2695758. [Online; accessed 19-September-2013].

[4] Juno C Hamano. *Pro Git*. apress, 2009.

[5] githooks(5) manual page. https://www.kernel.org/pub/software/scm/git/docs/githooks.html. [Online; accessed 05-Juni-2013].

[6] Redmine. http://www.redmine.org/. [Online; accessed 01-September-2013].

[7] DashAsBinSh. https://wiki.ubuntu.com/DashAsBinSh. [Online; accessed 05-August-2013].

[8] HowTo: Debug a Shell Script Under Linux or UNIX . http://www.cyberciti.biz/tips/debugging-shell-script.html. [Online; accessed 19-September-2013].

[9] Projects, Tips, Tricks, and How-Tos Writing Better Shell Scripts. http://innovationsts.com/?p=1896. [Online; accessed 19-September-2013].

[10] Continuous Integration. http://www.martinfowler.com/articles/continuousIntegration.html. [Online; accessed 05-September-2013].

[11] CI Paper. http://download.red-gate.com/HelpPDF/ContinuousIntegrationForDatabasesUsingRedGateSQLTools.pdf. [Online; accessed 20-September-2013].

[12] Jenkins. http://jenkins-ci.org/. [Online; accessed 19-September-2013].

[13] Hudson. http://hudson-ci.org/. [Online; accessed 03-September-2013].

[14] Jenkins cyanogenmod. http://jenkins.cyanogenmod.com/job/submission-test/. [Online; accessed 19-July-2013].

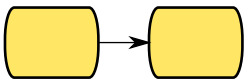
[15] CI System Paper. http://buildrelease.googlecode.com/hg-history/c9dc4d3d963169947bfd3e124f0c027a032613ee/Trunk/BRESystem/theory/martinfowler-com_articles_continuousIntegration-html_qa1qtrlm.pdf. [Online; accessed 21-September-2013].

List of Figures

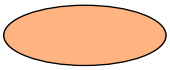
List of Listings

Appendices

A. Figure Legend



Git branch



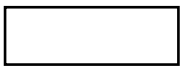
Git action

Git hook trigger

local

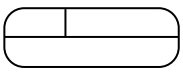
remote

user input



script, program

file



Git object

B. List of Acronyms

API	Application programming interface
CI	Continuous Integration
CVS	Concurrent Versions System
GCC	GNU Compiler Collection
NFS	Network File System
PDF	Portable Document Format
SLURM	Simple Linux Utility for Resource Management
SVG	Scalable Vector Graphics
VCS	Version Control System
XML	Extensible Markup Language

C. Documentation

1. Configure Git
   - `apt-get install git`
   - `git config --global user.name "Name"`
   - `git config --global user.email "mail@mail.xy"`
2. Clone the project
   - `git clone http://redmine.wr.informatik.uni-hamburg.de/git/julea <your repository>`
3. Copy the hooks, rename and change the mode to make them executable
   - `cp julea/tools/<hook.sample> .git/hooks`
   - `mv <hook.sample> <hook>`
   - `chmod +x <hook>`
4. Create a results repository where you want and set the path to it.
   - `git config --local test.results-path "<your path>"`
   - Using following commands you can overwrite existent defaults. Note the whitespace after the colon. Common default values are listed in Listing 4.17. All the settings except the last one can be overwritten in the commit message doing a commit.
   - `git config --local test.compile-test "Compile-Test:  yes|clean|no"`
   - `git config --local test.correctness-test "Correctness-Test:  yes|no"`
   - `git config --local test.performance-test-local "Performance-Test: yes|parallel|no"`
   - `git config --local test.performance-test-remote "Performance-Test: yes|parallel|no"`

Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen, als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen – benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Optional: Ich bin mit der Einstellung der Bachelor-Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik einverstanden.

Hamburg, den 20.09.2013 .....