An Effective Git And Org-Mode Based Workflow For Reproducible Research

Luka Stanisic, Arnaud Legrand and Vincent Danjean
firstname.lastname@imag.fr
CNRS/Inria/Univ. of Grenoble, Grenoble, France

ABSTRACT

In this paper we address the question of developing a lightweight and effective workflow for conducting experimental research on modern parallel computer systems in a reproducible way. Our approach builds on two well-known tools (Git and Org-mode) and enables to address, at least partially, issues such as running experiments, provenance tracking, experimental setup reconstruction or replicable analysis. We have been using such a methodology for two years now and it enabled us to recently publish a fully reproducible article [12]. To fully demonstrate the effectiveness of our proposal, we have opened our two year laboratory notebook with all the attached experimental data. This notebook and the underlying Git revision control system enable to illustrate and to better understand the workflow we used.

1. INTRODUCTION

In the last decades, both hardware and software of modern computers have become increasingly complex. Multi-core architectures comprising several accelerators (GPUs or the Intel Xeon Phi) and interconnected by high-speed networks have become mainstream in the field of High-Performance. Obtaining the maximum performance of such heterogeneous machines requires to rely on combinations of complex software stacks ranging from the compilers to architecture specific libraries or languages (e.g., CUDA, OpenCL, MPI) and complex dynamic runtimes (StarPU, Charm++, OmpSs, etc.) [1]. Dynamic and opportunistic optimizations are done at every level and even experts have troubles fully understanding the performance of such complex systems. Such machines can no longer be considered as deterministic, especially when it comes to measuring execution times of parallel multi-threaded programs. Controlling every relevant sophisticated component during such measurements is almost impossible even in single processor cases [8], making the full reproduction of experiments extremely difficult. Consequently, studying computers has become very similar to studying a natural phenomena and it should thus use the same principles as other scientific fields that had them defined centuries ago. Although many conclusions are commonly based on experimental results in this domain of computer science, articles generally poorly detail the experimental protocol. Left with insufficient information, readers have generally troubles reproducing the study and possibly build upon it. Yet, as reminded by Drummond [4], *reproducibility* of experimental results is the hallmark of science and there is no reason why this should not be applied to computer science as well.

Hence, a new movement promoting the development of reproducible research tools and practices has emerged, especially for computational sciences. Such tools generally focus on *replicability* of data analysis [14]. Although high performance computing or distributed computing experiments involve running complex codes, they do not focus on execution results but rather on the time taken to run a program and on how the machine resources were used. These requirements call for different workflows and tools, since such experiments are not replicable by essence. Nevertheless in such cases, researchers should still at least aim at full reproducibility of their work.

There are many existing solutions partially addressing these issues, some of which we present in Section 3. However, none of them was completely satisfying the needs and constraints of our experimental context. Therefore, we decided to develop an alternative approach based on two well-known and widely-used tools: Git and Org-mode. The contributions of our work are the following:

- We propose and describe in Section 4.1 a Git branching model for managing experimental results synchronized with the code that generated them. We used such branching model for two years and have thus identified a few typical branching and merging operations, which we describe in Section 4.3

- Such branching model eases provenance tracking, experiments reproduction and data accessibility. However, it does not address issues such as documentation of the experimental process and conclusions about the results, nor the acquisition of meta-data about the experimental environment. To this end, in Section 4.2 we complete our branching workflow with an intensive use of Org-mode [11], which enables us to manage and keep in sync experimental results and meta-data. It also provides literate programming, which is very convenient in a laboratory notebook and eases the edition of reproducible articles. We explain in Section 4.3 how the laboratory notebook and the Git branching can be nicely integrated to ease the set up of a reproducible article.

- Through the whole Section 4, we demonstrate the effectiveness of this approach by providing examples and opening our two years old Git repository at http://starpu-simgrid.gforge.inria.fr/. We illustrate several points in the discussion by pointing directly to specific commits.

- Although we open our whole Git repository for illustration purposes, this is not required by our workflow. There may be situations where researchers may want to share only parts of their work. We discuss in Section 5 various code and experimental data publishing options that can be used within such a workflow.

- Finally, we discuss in Section 6 how the proposed methodology helped us conduct two very different studies in the High Performance Computing (HPC) domain. We also report limits of our approach, together with some open questions.

2. MOTIVATION AND USE CASE DESCRIPTION

Our research is centered on the modeling of the performance of modern computer systems. To validate our approach and models, we have to perform numerous measurements on a wide variety of machines, some of which being sometimes not even dedicated. In such context, a presumably minor misunderstanding or inaccuracy about some parameters at small scale can result in a totally different behavior at the macroscopic level [8]. It is thus crucial to carefully collect all the useful meta-data and to use well-planed experiment designs along with coherent analyses, such details being essential to the reproduction of experimental results.

Our goal was however not to implement a new tool, but rather to find a good combination of already existing ones that would allow a painless and effective daily usage. We were driven by purely pragmatic motives, as we wanted to keep our workflow as simple and comprehensible as possible while offering the best possible level of confidence in the reproducibility of our results.

In the rest of this section, we present two research topics we worked on, along with their specifics and needs regarding experimentation workflow. These needs and constraints motivated several design choices of the workflow we present in Section 4.

2.1 Case Study #1: Modeling and Simulating Dynamic Task-Based Runtimes

Our first use case aims at providing accurate performance predictions of dense linear algebra kernels on hybrid architectures (multi-core architectures comprising multiple GPUs). We modeled the StarPU runtime [1] on top of the SimGrid simulator [3]. To evaluate the quality of our modeling, we had to conduct experiments on hybrid prototype hardware whose software stack (e.g., CUDA or MKL libraries) could evolve along time. We had only limited control and access to the environment setup, as the machines are strictly managed by the administrators who maintain and update its configuration so that it matches the needs of most users. Moreover, we relied on code from two external repositories together with many of our own scripts. These external code sources were quite complex and frequently changed by their respective multiple developers.

After a long period of development and experimentation, we have written an article [12] that builds on a completely replicable analysis of the large set of experiments we gathered. We took care of doing these experiments in a clean, coherent, well-structured and reproducible way that allows anyone to inspect how they were performed.

2.2 Case Study #2: Studying the Performance of CPU Caches

Another use case that required a particular care is the study of CPU cache performance on various Intel and ARM micro-architectures [13]. The developed source code was quite simple, containing only a few files, but there were numerous input, compilation and operating system parameters to be taken into account. Probably the most critical part of this study was the environment setup, which proved to be unstable, and thus, responsible for many unexpected phenomena. Therefore, it was essential to capture, understand and easily compare as much meta-data as possible.

Limited resources (both in terms of memory, CPU power and disk space) and software packages restricted availability on ARM processors have motivated some design choices of our methodology. In particular, this required our solution to be lightweight and with minimal dependencies. Although this research topic has not led to a reproducible article yet, we used the proposed workflow and can still track the whole history of these experiments.

3. RELATED WORK

In the last several years, the field of reproducibility research tools has been very active, various alternatives emerging to address diverse problematic. However, only few of them are appropriate for running experiments and measuring execution time on large, distributed, hybrid systems comprising prototype hardware and software. The ultimate goal of reproducible research is to bridge the current gap between the authors and the article readers (see Figure 1) by providing as much material as possible on the scientist choices and employed artifacts. We believe that opening a laboratory notebook is one step in that direction since not only the positive results are shared in the final article but also all the negative results and failures that occurred during the process.

There are successful initiatives in different computer science fields (e.g., DETER [7] in cybersecurity experimentation or PlanetLab in the peer-to-peer community) that provide good tools for their users. However, when conducting experiments and analysis in the HPC domain, several specific aspects need to be considered. We start by describing the ones related to the experimental platform setup and configuration. Due to the specific nature of our research projects (prototype platforms with limited control and administration permissions), we have not much addressed these issues ourselves although we think that they are very important and need to be mentioned.

Platform accessibility Many researchers conduct their experiments on large computing platforms such as Grid5000 or PlanetLab and which have been specifically designed for large scale distributed/parallel system experimentation. Using such infrastructures eases reproduction but also requires to manage resource reservation and to orchestrate the experiment, hence the need for specific tools. However, the machines we considered in our study are generally recent prototypes, some of them being rather unique and meant to be accessed directly without any specific protocol.

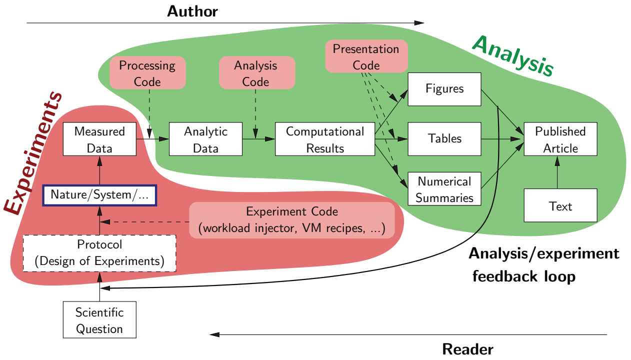


Figure 1: Ideally, the experimenter would keep track of the whole set of decisions taken to conduct its research as well as all the code used to both conduct experiments and perform the analysis.

Setting up environments It is necessary to carefully configure machines before doing experiments. Among the few tools specifically designed for this purpose and based on recipes, we can cite Kameleon [9], which allows to reconstruct an environment step by step. Another approach consists in automatically capturing the environment required to run the code (e.g., as done by CDE [14, chap.4].) or to use virtual machines so that code can be later re-executed on any other computer. In our use cases, experimental platforms are already set up by expert administrators and we had neither the permission nor particular interest to modify their configuration.

Conducting experiments Numerous tools for running experiments in a reproducible way were recently proposed. Solely for the Grid5000 platform, there are at least three different ones: Expo [10], XPflow [2] and Execo [6]. These tools are not specifically designed for HPC experiments but could easily be adapted. Another set of related tools developed for computational sciences comprises Sumatra [14, chap.3] and VisTrails [14, chap.2]. Such tools are rather oriented on performing a given set of computations and do not offer enough control on how the computations are orchestrated to measure performances. They are thus somehow inadequate in our context. Some parts or ideas underlying the previously mentioned tools could have been beneficial to our case study. In our experiments, simple scripts were sufficient although they often require interactive adaptations to the machines on which they are run, which makes experiments engine that aim at automatic execution difficult to use.

Now we detail aspects related to software, methodology and provenance tracking, which are often neglected by researchers in our community.

Accessibility It is widely accepted that tools like Git or svn are indispensable in everyday work on software development. Additionally, they help at sharing the code and letting other people contribute. Using such tool for managing experiments is however not that common. Public file hosting services, such as Dropbox or Google Drive have become a very popular way to share data among scientists that want to collaborate. The unclear durability of such service and the specific requirements scientists have in term of size and visibility has lead to the development of another group of services (e.g., figshare) that are focused on making data publicly and permanently available and easily understandable to everyone.

Provenance tracking Knowing how data was obtained is a complex problem. The first part involves collecting meta-data, such as system information, experimental conditions, etc. In our domain, such part is often neglected although experimental engines sometimes provide support for automatically capturing it. The second part, frequently forgotten in our domain, is to keep track of any transformation applied to the data. In such context, the question of storing both data and meta-data quickly arises and the classical approach to solve these issues involves using a database. However, this solution has its limits, as managing source codes or comments in a database is not convenient and is in our opinion handled in a much better way by using version control systems and literate programming.

Documenting While provenance tracking is focused on how data was obtained, it is not concerned with why the experiments were run and what the observations on the results are. These things have to be thoroughly documented, since even the experimenters tend to quickly forget all the details. One way is to encourage users to keep notes when running experiment (e.g., in Sumatra [14, chap.3]), while the other one consists in writing a laboratory notebook (e.g., with IPython).

Extendability It is hard to define good formats for all project components in the starting phase of the research. Some of the initial decisions are likely to change during the study, so the system has to be easy to extend and modify. In such a moving context, integrated tools with fixed database schemes, as done for example in Sumatra, seemed too rigid to us although they definitely inspired several parts of our workflow.

Replicable analysis Researchers should only trust figures and tables that can be regenerated from raw data that comprise sufficient details on how the experiments were conducted. Therefore, ensuring replicable analysis (the Analysis part of Figure 1) is essential to any study. A popular solution is to rely on open-source statistical software like R and knitr that simplify figure generation and embedding in final documents [14, chap.1].

To address the previous problems, we decided to work with a minimalist set of simple, lightweight, and well-known tools. We use **Org-mode**, initially an Emacs mode for editing and organizing notes, that is based on highly hierarchical plain text files which are easy to explore and exploit. Org-mode has also been extended to allow combining plain text with small chunks of executable code (Org-babel [11] snippets). Such feature builds on the literate programming principles introduced by Donald Knuth three decades ago, and for which there has been a renewed interest in the last years. Although such tool was designed for conducting experiments and for writing scientific articles, its use is not so common yet.

In addition, for version control system we decided to rely on **Git**, a distributed revision control tool that offers an incredibly powerful and flexible branching mechanism.

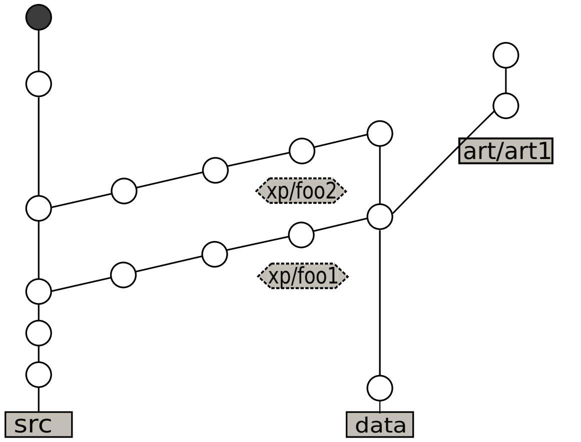


Figure 2: Proposed Git branching scheme with 4 different types of branches

4. A GIT AND ORG-MODE BASED WORKFLOW

In this section, we present our workflow for conducting experiments and writing articles about the results, based on a unique Git branching model coupled with a particular usage of Org-mode. Although these are well-known and widely used tools, to the best of our knowledge no one so far has proposed using them in a similar manner for doing reproducible research. The approach we present is lightweight, to make sure the experiments are performed in a clean, coherent and hopefully reproducible way without being slowed down by a rigid framework. It fully fulfilled our needs for the use cases presented in Section 2 and would equally help anyone doing research in a such context. However, the main ideas behind our solution are general and can be applied in other fields of computer science as well, possibly with some domain specific adjustments as the environment in which experiments are performed can be very different.

We remind the reader that every document and series of commits described herein can be found at http://starpu-simgrid.gforge.inria.fr/. Links to Git commits with examples are provided in the rest of this paper and we encourage readers to inspect them. All our documents are plain text documents and can thus be opened with any text editor or browser. However, since most of these files are Org-mode documents, we suggest to open them with a recent Emacs and Org-mode installation rather than within a web browser. Options for pretty printing in web browsers exist, but are not fully mature yet. We are currently working on making it easier for non Emacs users to easily exploit such data.

4.1 Git Branching Structure

Relying on a revision control system for under development code is nowadays a common practice. It is thus a good habit to ensure that all source code changes are committed before running the experiments. These restrictions are already implemented in many tools, such as Sumatra. Knowing which revision of the source code was used to produce a given data makes it theoretically possible to reproduce experimental results. However, in practice it can be quite burdensome.

A first problem is that code and experimental results are often treated as two separate kinds of objects. Being only loosely coupled, the link between some data and the code that produced it is often hard to find and exploit, and sometimes it can even be completely lost. Therefore, we suggest to store both of them in the same Git repository so as to ensure that they are always perfectly synchronized. This greatly increases the confidence level in the results and makes it easier to obtain the code that produced a particular data set.

Storing everything in the same repository can quickly lead to an anarchic and unexploitable system and hence it requires some organization and convention. To this end, we propose an approach that consists in 4 different types of branches, illustrated in Figure 2. The first branch, named *src*, includes only the source code, i.e., the code and scripts required for running the experiments and simple analysis. The second branch, *data*, comprises all the source code as well as all the data and every single analysis report (statistical analysis results, figures, etc.). These two branches live in parallel and are interconnected through a third type of branches going from *src* to *data*, the *xp#* branches ("#" sign means that there are multiple branches, but all with the same purpose). These are the branches where all the experiments are performed, each *xp#* branch corresponding to one set of experiment results. The repository has typically one *src* and one *data* branch, both started at the beginning of the project, while there is a huge number of *xp#* branches starting from *src* and eventually merged into *data* that accumulates all the experimental results. All these together form a "ladder like" form of Git history. Finally, the fourth type of branches is the *art#* branches which extend the *data* branch. They comprise an article source and all companion style files, together with a subset of data imported from the *data* branch. This subset contains only the most important results of the experiments, that appear in the tables and figures of the article.

By using Git as proposed, it is extremely easy to set up an experimental environment and to conduct the experiments on a remote machine by pulling solely the head of the *src* or of an *xp#* branch. This solves the problem of long and disk consuming retrieving of the whole Git repository, as the *src* and *xp#* branches are typically very small.

On the other hand, one might want to investigate all the experimental data at once, which can be easily done by pulling only the head of *data* branch. This is meant for the researchers that are not interested in the experimentation process, but only in the analysis and cross-comparison of multiple sets of results. For such users, the *src* and *xp#* branches are completely transparent, as they will retrieve only the latest version of the source code (including analysis scripts) and the whole set of data.

Another typical use case is when one wants to write an article or a report based on the experiment results. A completely new branch can then be created from *data*, selecting from the repository only the data and analysis code needed for the publication and deleting the rest. This way, the complete history of the study behind the article can be preserved (e.g., for the reviewers) and the article authors can download only the data set they really need.

When used correctly, such Git repository organization can provide numerous benefits to the researchers. However, it is not sufficient in our setting, since commit messages in Git history give only coarse grain indications about source code modifications. There is still a lot of information missing about the environment setup of the machines, why and how certain actions were performed and what the conclusions about the results are. We address all these questions with Org-mode files, as described in the following subsection.

4.2 Using Org-mode for Improving Reproducible Research

As mentioned in Section 3, several tools can help to automatically capture environment parameters, to keep track of the experimentation process, to organize code and data, etc. However, none of them addresses these issues in a way satisfying our experimental constraints, as these tools generally create new dependencies on specific libraries and technologies that sometimes cannot be installed on experimentation machines. Instead, we propose a solution based on plain text files, written in the spirit of literate programming, that are self-explanatory, comprehensive and portable. We do not rely on a huge cumbersome framework, but rather on a set of basic, flexible `shell` scripts, that address the following challenges.

4.2.1 Environment Capture

Environment capture aims at getting every detail about the code, the libraries in use and the system configuration. Unlike the parallel computing field where applications are generally expected to run in more or less good isolation of other users/applications, there are several areas of computer science (e.g., networking, security, distributed systems, etc.) where fully capturing such platform state is impossible. However, the principle remains the same, as it is necessary to gather as much useful meta-data as possible, to allow comparison of experimental results with each others and to determine if any change to the experimental environment can explain potential discrepancies. This process should not be burdensome, but automatic and transparent to the researcher. Additionally, it should be easy to extend or modify, since it is generally difficult to anticipate relevant parameters before performing numerous initial experiments.

Thus, we decided to rely on simple `shell` scripts, that just call many Unix commands in sequence to gather system information and collect the different outputs. The meta-data that we collect typically concern users logged on the machine during the experiments, the architecture of the machine (processor type, frequency and governor, cache size and hierarchy, GPU layout, etc.), the operating system (version and used libraries), environment variables and finally source code revisions, compilation outputs and running options. This list is not exhaustive and would probably need to be adjusted for experiments in other domains.

Once such meta-data is captured, it can be stored either individually or accompanying results data. One may prefer to keep these two separated, making the primary results unpolluted and easier to exploit. Although some specific file systems like HDF5 (as used in activepapers [5]) provide a clean management of meta-data, our experimental context, where computing/storage resources and our ability to install non-standard software are limited, hinders their use. Storing such information in another file of a standard file system quickly makes information retrieval from meta-data cumbersome. Therefore, we strongly believe that the experiment results should stay together with the information about the system they were obtained on. Keeping them in the same file makes the access straightforward and simplifies the project organization, as there are less objects to handle. Furthermore, even if data sustains numerous movements and reorganizations, one would never doubt which environment setup corresponds to which results.

In order to permit users to easily examine any of their information, these files have to be well structured. The Org-mode format is a perfect match for such requirements as its hierarchical organization is simple and can be easily explored. A good alternative might be to use the yaml format, which is typed and easy to parse but we decided to stay with Org-mode (which served all our needs) to keep our framework minimalist.

A potential issue of this approach is raised by large files, typically containing several hundreds of MB and more. Opening such files can temporary freeze a text editor and finding a particular information can then be tedious. We have not yet met with such kind of scenario, but it would certainly require some adaptations to the approach.

In the end, all the data and meta-data are gathered automatically using scripts (e.g., `41380b54a7{run-experiment.sh#l220}`), finally producing a read-only Org-mode document (e.g., `1655becd0a{data-results.org}`) that serves as a detailed experimental report.

The motivations for performing the experiments, and the observations about the results are stored separately, in the laboratory notebook.

4.2.2 Laboratory Notebook

A paramount asset of our methodology is the laboratory notebook (labbook), similar to the ones biologist, chemists and scientist from other fields use on a daily basis to document the progress of their work. For us, this notebook is a single file inside the project repository, shared between all collaborators. The main motivation for keeping a labbook is that anyone, from original researchers to external reviewers, can later use it to understand all the steps of the study and potentially reproduce and improve it. This self-contained unique file has two main parts. The first one aims at carefully documenting the development and use of the researchers' complex source code. The second one is concerned with keeping the experimentation journal.

Documentation This part serves as a starting point for newcomers, but also as a good reminder for everyday users. The labbook explains the general ideas behind the whole project and methodology, i.e., what the workflow for doing experiments is and how the code and data are organized in folders. It also states the conventions on how the labbook itself should be used.

Details about the different programs and scripts, along with their purpose follow. These information concern the source code used in the experiments as well as the tools for manipulating data and the analysis code used for producing plots and reports. Additionally, there are a few explanations on the revision control usage and conventions.

Moreover, the labbook contains a few examples of how to run scripts, displaying the most common arguments and format. Although such information might seem redundant with the previous documentation part, in practice such examples are indispensable even for experienced users, since some scripts have lots of environment variables, arguments and options.

It is also important to keep track of big changes to the source code and the project in general inside a ChangeLog. Since all modifications are already captured and commented in Git commits, the log section offers a much more coarse grain view of the code development history. There is also a list with brief descriptions of every Git tag in the repository as it helps finding the latest stable, or any other specific, version of the code.

Experiment results All experiments should be carefully noted here, together with the key input parameters, the motivation for running such experiment and the remarks on the results. For each experimental campaign there should be a new entry that replies to the questions why, when, where and how experiments were run and finally what the observations on the results are. Inside the descriptive conclusions, Org-mode allows to use both links and git-links connecting the text to specific revisions of files. These hyperlinks point to crucial data and analysis reports that illustrate a newly discovered phenomenon.

Managing efficiently all these different information in a single file requires a solid hierarchical structure, which once again motivated our use of Org-mode. We also took advantage of the Org-mode tagging mechanism, which allows to easily extract information, improving labbook's structure even further. For example, tags can be used to distinguish which collaborator conducted a given set of experiments and on which machine. Although such information may already be present in the experiment files, having it at the journal level proved very convenient, making the labbook much easier to understand and exploit. Experiments can also be tagged to indicate that certain results are important and should be used in future articles.

Several alternatives exist for taking care of experiment results and progress on a daily basis. We think that a major advantage of Org-mode compared to many other tools is that it is just a plain text file that can thus be read and modified on any remote machine without requiring to install any particular library, not even Emacs. Using a plain text file is also the most portable format across different architectures and operating systems.

We provide two examples of labbook files. The first one has only with the documentation parts related to the code development and usage (`30758b6b6a{labbook}`) and is the one obtained from the *src* branch or from the beginning of an *xp#*, while the second one (`01928ce013{labbook#l272}`) has a huge data section comprising the notes about all the experiments performed since the beginning of the project.

4.2.3 Using Literate Programming for Conducting Experiments

In our field, researchers typically conduct experiments by executing commands and scripts in a terminal, often on a remote machine. Later, they use other tools to do initial analysis, plot and save figures from the collected data and at the end write some remarks. This classical approach has a few drawbacks, which we try to solve using Org-babel, Org-mode's extension for literate programming.

The main idea is to write and execute commands in Org-babel snippets, directly within the experimentation journal, in our case the labbook. This allows to go through the whole experimentation process, step-by-step, alternating the execution of code blocks and writing text explanations. These explanations can include reasons for running a certain snippet, comments on its outputs, plan for next actions or any other useful remarks. At the end, this process can be followed by a more general conclusion on the motives and results of the whole experimentation campaign. Conducting experiments in such manner provides numerous benefits comparing to the usual way scientists in our field work.

The first problem with the classical approach is that researchers save only the experiment results (possibly with some meta-data), while all other seemingly irrelevant outputs of commands are discarded. However, in case of failures, these outputs can occasionally be very helpful when searching for the source of an error. Although, such outputs, along with the commands that produced them, can sometimes be found in a limited terminal history, their exploration is a very tedious and error-prone process. On the other hand, when using Org-babel, all snippet results are kept next to it, which simplifies the tracing of problems.

Secondly, since the preparation and management of experiments is a highly repetitive process, grouping and naming sequences of commands in a single snippet can be very beneficial. This allows to elegantly reuse such blocks in future experiments without writing numerous scripts or bulky ”one-liners”.

Additionally, Org-babel permits to use and combine several languages, each with its own unique purpose, inside the same file. This again decreases the number of files and tools required to go through the whole experimentation process, making it simpler and more coherent.

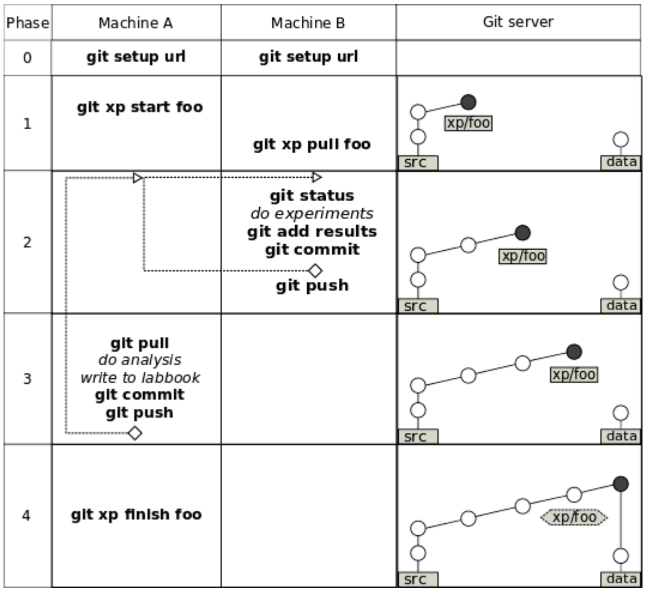
Last, and probably the most important point, using this approach avoids documenting an experimental process afterwards, which is generally tedious and often insufficient. Researchers are frequently in a hurry to obtain new data, especially under a pressure of rigorous deadlines. They do not dedicate enough time to describe why, where and how experiments were performed or even sometimes what are the conclusions about the results. At that moment, answers to these questions may seem obvious to the experimenters, therefore they neglect noting it. However, in few days or months, remembering all the details is not so trivial anymore. Following literate programming principles and taking short notes to explain the rationale and usage of code snippets, while executing them, is quite natural and solves the previous issues. From our own experience, it does not significantly slow down the experimental process, while it provides huge benefits later on.

Finally, the outcome of this approach is a comprehensible, well-commented executable code, that can be rerun step-by-step even by external researchers. Additionally, it can also be exported (tangled), producing a script that consists of all snippets of the same language. Such scripts can be used to completely reproduce the whole experimentation process.

An example of this approach is provided in `0b20e8abd5{labbook#l950}`. It is based on Shell snippets, and although it can be rerun only with the access to the experimental machines, it provides both a good illustration of Org-babel usage for conducting experiments and a faithful logging of the commands run to obtain these experimental data, which is paramount for a researcher willing to build upon it.

4.3 Git Workflow in Action

We now explain the typical workflow usage of our Git branching scheme, that is also tightly linked to the experimentation journal in Org-mode.



Phase	Machine A	Machine B	Git server
0	**git setup url**	**git setup url**	
1	**git xp start foo**	**git xp pull foo**	xp/foo src data
2		**git status** *do experiments* **git add results** **git commit** **git push**	xp/foo src data
3	**git pull** *do analysis* *write to labbook* **git commit** **git push**		xp/foo src data
4	**git xp finish foo**		xp/foo src data

Figure 3: Typical Git experimentation workflow with different phases, using two machines

However, branching and merging is a technical operation that can become cumbersome and generally scares new Git users. That is why such Git interactions should be made as easy as possible with new specific Git commands, which we are currently packaging. We introduce such commands along with their intended use without going into the details of their implementation.

4.3.1 Managing Experiments with Git

On Figure 3 we explain the typical workflow usage of our branching scheme. Although it is self-contained and independent from any other tool, we found it very practical to couple it with our laboratory notebook.

Before even starting, the whole project needs to be correctly instantiated on every machine, as shown in Phase 0. The `git setup url` command will clone the project from server, but without checking out any of the branches.

When everything is set, the researcher can start working on a code development inside the *src* branch, committing changes, as shown in Phase 1. These modifications can impact source code, analysis or even the scripts for running the experiments. Later, such modifications should be tested and the correctness of the whole workflow should be validated. Only then can one start conducting real experiments by calling `git xp start foo`. This command will create and checkout a new branch *xp/foo* from the *src*. Then, this command will create, commit and push a new folder for storing the results. We used the convention that these two (branch and folder) should always have the same name, which eases the usage of both Git and labbook. Next, the newly created branch is pulled on a remote machine B, using `git xp pull foo`. It will fetch only the last commit of the *xp/foo* branch. As a result, machines for experimentation, such as machine B, get only the code required to run the experiments, without neither Git history nor any experimental data. Meanwhile, machine A and all other users that want to develop code, do the analysis and write articles will continue using the standard `git pull` command to get full Git repository, although it can sometimes be quite memory and time consuming.

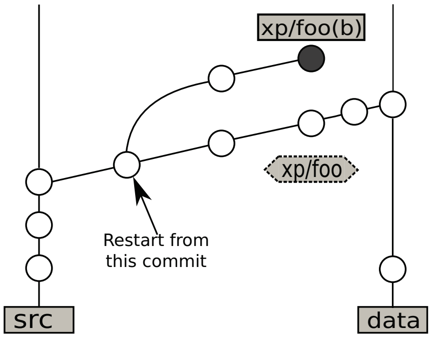


Figure 4: Restart or reproduce experiments starting from a certain commit

In Phase 2, we first verify that there has not been any code modification before running the experiment and we also automatically ensure that the latest version of the code has been compiled. Then, experiments are run, generating new data. The resulting Org-mode data files, containing experiment outputs together with the captured environment meta-data, are then committed to the Git. Such process may be repeated, possibly with different input parameters. Finally, the commited data is pushed to the server.

After that, the experiment results can be pulled on the machine that is used to do the analysis (Phase 3). Important conclusions about the acquired data should be saved either in separate reports, or even better as a new *foo* entry inside the experiment results section in the labbook. Results of the analysis could later trigger another round of experimentation and so on.

Finally, when all desired measurements are finished, *xp/foo* will be merged with the *data* branch using `git xp finish foo`, as depicted in Phase 4. This command will also delete the *foo* branch, to indicate that the experimentation process is finished and to avoid polluting the repository with too many open branches. Still a simple Git tag will be created on its place, so if needed, the closed branch *foo* can be easily found and investigated in future. Note that while in *src* branch labbook only has the documentation part, the *xp#* branches are populated with observations about the experiments. Therefore, the merged labbook in the *data* branch holds all the collected experimental entries with comments, which makes their comparison straightforward.

One interesting option is to go through the entire workflow depicted in Figure 3 directly within the labbook, using the literate programming approach with Org-babel we described in Section 4.2.3.

4.3.2 Reproducing Experiments

The main goal of such workflow is to facilitate as much as possible the reproduction of experiment by researchers. This can be done by calling the `git xp start --from foo` command, from the machine we want to repeat the experiments. As displayed in Figure 4, this command will checkout the desired revision of the code and create a new branch and folder based on the initial *xp#* branch. From there, conducting the new experiments, noting the observations and later merging with *data* branch is performed as usual.

It may happen that software components of the machines used for experiments are replaced between two series of ex-

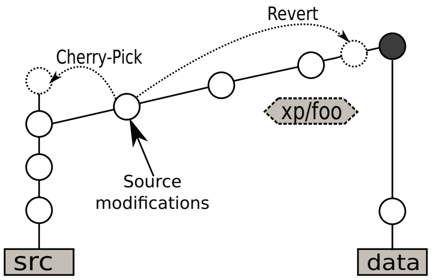


Figure 5: Handling source modifications that occurred during the experimentation

periments. In many cases, this is done by the machine administrators and the researchers conducting the experiments may have no permission to revert it. There could thus be some important changes in the environment and repeated experiments might produce different results from the initial ones. Unfortunately, when dealing with experiments that cannot be run on virtual machines, nothing can be done to avoid this problem. The best we can do is to carefully track all the software used in the experiments, so if eventually any deviation of the experimental results occurs, we can compare the meta-data and find the source of discrepancy.

It is also worth mentioning that if researchers want to reconduct a previous experiments, but on a completely new machine, they will use this exact same approach.

4.3.3 Fixing Code

Sometimes, while conducting the experiments on a remote machine, the researcher may need to make few small source code modifications. These modifications have to be committed (since measurements are never done using an uncommitted code), even though in most cases they represent an *ad hoc* change, specific to an individual machine and its current installation. These minor, local hacks would pollute the *data* branch and thus it has to be ensured that they are not propagated when branches are merged. This protection is thus implemented inside the `git xp finish foo` command.

At the end of the *foo* branch, all source code changes are reverted. This means that we create an "anti-commit" of all the previously committed source modifications inside that experimental branch, as shown in Figure 5. This way modifications remain local for *foo* and the experimental setup can still be reproduced by pulling the revision before the revert operation.

If the researcher eventually recognizes that some of the source code modifications done inside *foo* branch could be useful for the whole project, there are two ways to insert them in the *src* branch. The first one involves rewriting Git history and it is not advised as it can introduce incoherence between Git repositories. The second option is to simply cherry-pick the desired commits. Although this approach produces some redundancy, it is easier, safer and enables to keep the Git history comprehensible and consistent.

4.3.4 Making Transversal Analysis

Such Git organization, environment parameter capture and careful note taking in labbook also simplifies the comparison of data sets. Since the *data* branch aggregates all the *xp#* branches, it is the best location to analyze and compare these different results with each others. The numerous plain-text meta-data are easily exploited and incorporated in the analysis. Since each data file comprises the revision of the source code used to generate it, it is easy to backtrack to specific commits and to exploit the labbook to explain unexpected behaviors.

4.3.5 Writing Reproducible Articles

Using the described workflow on a daily basis makes the writing of reproducible articles straightforward. Figure 2 shows how researchers can create a new branch from the *data* branch that contains only useful experimental results by using the `git article start art1` command. This command deletes all the unnecessary data from the new branch, keeping only the experiments that are previously tagged in labbook with *art1* keyword. This approach is convenient for the authors that are collaborating on the article writing and are not concerned with the analysis since it is possible for them to pull only the Git history of the article, not the whole project.

Occasionally, some important experiment results may have been overlooked or conducting additional measurements becomes necessary. In such case, these new results can be added later through merging with the updated *data* branch.

The same principles used for conducting experiments with Org-babel can be applied when writing an article since it is possible to combine the text of the paper with data transformations, statistical analysis and figure generation, all using generally different programming languages. A major advantage of this methodology is that a lot of code and text can be recycled from previous analysis scripts and from the labbook.

Keeping everything in the same file rather than to have it scattered in many different ones, makes everything simpler and greatly helps the writers. We did not encounter any particular issue when multiple authors did collaborate on the same paper. This also simplifies modifications and corrections often suggested by reviewers since every figure is easily regenerated by calling the code snippet hidden next to it in the article.

The final result of the whole workflow, is an article containing all the raw data that it depends on together with the code that transformed it into tables and figures, possibly along with the whole history with the detailed explanations of how this data was obtained. This is very convenient not only for the authors but also for the readers, especially reviewers, since all experimental and analysis results can be inspected, referenced, or even reused in any other research project.

5. PUBLISHING RESULTS

Making data and code publicly available is a good practice as it allows external researchers to improve or build upon our work. However, at least in our domain it is not that commonly done, in particular because it is not that trivial to do when the study was not conducted with a clean methodology in mind from the beginning. If such intentions are not envisioned from the beginning of the project, it is generally very tedious to document and package afterwards. Gathering all the data required for an article can be cumbersome, as it is typically spread in different folders on different machines. Explaining experiment design and results is even harder, since notes that were taken months ago are often not precise enough. In the end, few researchers somehow manage to collect all the necessary elements and put them in a tarball, to accompany the article. Nevertheless, such data without appropriate comments is hardly understandable and exploitable by others. This lowers the researchers' motivation to share their data as it will not be widely used.

The question of what parts of this whole history should go public can remain a sensitive topic. We think that, at the very least, the data used to produce the article figures and conclusions should be made available. Of course, providing only already post-processed .csv tables with only carefully chosen measurements can make the article replicable, but will not guarantee anything about reproducibility of the scientific content of the paper. Therefore, meta-data of all experiments should be made available as well. Likewise, providing more material than what is presented, may be desirable as it allows to illustrate issues that cannot be included in the document due to lack of space. The most extreme approach would be to publish everything, i.e., the whole laboratory notebook, acquired data and source code archived in a revision control system. Yet, some researchers may have perfectly valid reasons for not publishing so much information (copyright, company policy, implementation parts that the authors do not wish to disclose now, etc.).

The methodology we propose allows to easily choose which level of details is actually published. From the wide spectrum of possible solutions, we present two we used so far.

5.1 The Partially Opened Approach with Figshare Hosting

When we first started writing the article on the modeling and simulation of dynamic task-based runtimes [12], our Git repository was private and we had not considered to open it. To publish our experimental data, we decided to use Figshare, which is a service that provides hosting for research outputs, that can be shared and cited by others through the DOI mechanism.

Although our article was managed within an internal Git, publishing to figshare required to select, archive and upload all the data files and to finally annotate them in the web browser. This could probably have been automated, but the REST API was not completely stable at that time, so we had to do everything manually. Likewise, the github-figshare project could help, but it was at the early development stage and requires the whole Git repository to be hosted on GitHub, which may raise other technical issues (in particular the management of large files, whose size cannot exceed 100MB).

Hosting all our raw data on figshare also required adjusting our reproducible article. Data are first downloaded from figshare, then untared and post-processed. To this end, we again used the literate programming feature of Org-babel and the way we proceeded is illustrated in `e926606bef{article#l85}`.

Finally, this resulted in a self-contained article and data archive [15]. This approach was not so difficult to use, although the interaction with figshare was mostly manual, hence not as effective as it could have been.

5.2 The Completely Open Approach with Public Git Hosting

Using the previous approach, we somehow lost part of the history of our experimental process. Some data sets were not presented, some experiments where we had not properly configured machines or source codes were also missing. Nevertheless, it is clear that with highly technical tools and hardware such as the ones we experimented with, good results are not only the consequence of an excellent code, but also of expertise of the experimenters. Making failures available can be extremely instructive for those willing to build upon our work and thus publishing the whole labbook and Git history seemed important to us. In our case, this did not require additional work except to move our Git repository to a public project. With all the information we provide and an access to similar machines and configurations, others should be able to to repeat our experiments and to reproduce our results without much burden.

In the end, it is important to understand that even though we decided to completely open our labbook to others, this step is not a prerequisite for writing reproducible articles. The level of details that is made public can be easily adapted to everyone's preferences.

6. CONCLUSION

In this paper, we did not intend to propose new tools for reproducible research, but rather investigate whether a minimal combination of existing ones can prove useful. The approach we describe is a good example of using well-known, lightweight, open-source technologies to properly perform a very complex process like conducting computer science experimentation on prototype hardware and software. It provides reasonable reproducibility warranties without taking away too much flexibility from the users, offering good code modification isolation, which is important for *ad hoc* changes that are ineluctable in such environments. Although the two use cases we presented are quite different, most of the captured environment meta-data is the same for both projects. Since all the source code and data are in Git repository, reconstructing experimentation setup is greatly simplified. One could argue that not all elements are completely captured, since operating system and external libraries can only be reviewed but not reconstructed. To handle this, researchers could build custom virtual appliances and deploy them before running their experiments but this was not an option on the machines we used. Using virtual machines to run the experiments is not an option either, since in our research field we need to do precise time measurements on real machines and adding another software layer would greatly perturb performance observations. Finally, after applying such a methodology throughout the whole research process, it was extremely easy to write an article (`c836dde8f5{article}`) in Org-mode that was completely replicable. Along with the text, this Org-mode document contains all the analysis scripts and the raw data that can be inspected by reviewers.

The biggest disadvantage of our approach is that it has many not so common conventions along with a steep learning curve, hence it is difficult for new users. Moreover, it requires an expertise in Org-mode, preferably using Emacs text editor, together with a good understanding of Git. We acknowledge that some researchers are more used to other editors such as Vi/Vim and will not switch them easily. Although it is still possible to use them in our context, as Org-mode is plain text file that can be edited anywhere, it would be much harder to benefit from many of its special features. We believe that the tools we used provide benefits that are worth investing time but we also understand the need to simplify its use. There are thus currently many initiatives to port Org-mode to make it work completely in Vi or in web browsers. Some of them already work, but are not fully mature or complete yet. We are thus quite confident that Org-mode will be completely Emacs independent in the near future.

There is also a problem regarding the management and storing of large data files in repositories, and which is well-known to the community. This has been already solved for the Mercurial revision control tool, but even after an exhaustive research we could not find a satisfactory solution for Git. Many tools have been proposed, leading with `git-annex`, but they all have their shortcomings. Such tools are generally meant to be alternatives to synchronization services like Dropbox and Google Drive rather than to help dealing with large data traces originating from remote machine experiments. Having large Git repositories of several GB does not hinder daily committing, but can significantly slow down pull and checkout operations of branches comprising a huge number of data sets (typically *data* and *art#* branches).

It is still unclear how this approach would scale for multiple users working simultaneously, doing code modifications and experiments in parallel. In theory, it should work if everyone has sufficient experience of the tools and workflow, but we have never tried it with more than two persons. Another interesting feature that we have not yet experienced is collaboration with external users. These researchers could clone our project, work on it on their own, try to reproduce the results and build upon our work, potentially improving the code and contribute data sets back. Even though such utilization should work smoothly, there could be some pitfalls that we have not anticipated yet.

One could also ask the question of whether providing so much information is of any interest as too much information may make the most important things harder to distinguish. Regardless of the answer to this question, we believe anyway that beyond the actual experimental content of our open laboratory notebook, its structure and the techniques we used to keep track of information or to make analysis could be useful to others.

In the near future, we plan to write several simple scripts that will completely automate our workflow. These scripts will be packaged and available on the debian Linux system, in the same way as the *git-flow* approach for software development, only this time for managing experimental research.

Although, our methodology is undoubtedly improvable and there are several alternatives, we nonetheless found it very fast and efficient for a daily usage and extremely beneficial to our work. We can only encourage people to build on such simple workflows to conduct their own studies, as it is a very effective way to conduct a reproducible research.

7. REFERENCES

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23:187–198, Feb. 2011.

[2] T. Buchert, L. Nussbaum, and J. Gustedt. A workflow-inspired, modular and robust approach to experiments in distributed systems. Research Report RR-8404, INRIA, Nov. 2013.

[3] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.

[4] C. Drummond. Replicability is not reproducibility: Nor is it good science. In *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*, 2009.

[5] K. Hinsen. A data and code model for reproducible research and executable papers. *Procedia Computer Science*, 4(0):579 – 588, 2011. Proceedings of the International Conference on Computational Science.

[6] M. Imbert, L. Pouilloux, J. Rouzaud-Cornabas, A. Lèbre, and T. Hirofuchi. Using the EXECO toolbox to perform automatic and reproducible cloud experiments. In *1st International Workshop on UsiNg and building ClOud Testbeds (UNICO, collocated with IEEE CloudCom 2013)*, Sept. 2013.

[7] J. Mirkovic, T. B. S. Schwab, J. Wroclawski, T. Faber, and B. Braden. The DETER Project: Advancing the Science of Cyber Security Experimentation and Test. In *Proceedings of the IEEE Homeland Security Technologies Conference (IEEE HST)*, 2010.

[8] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 265–276. ACM, 2009.

[9] C. Ruiz, O. Richard, and J. Emeras. Reproducible software appliances for experimentation. In *Proceedings of the 9th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom)*, 2014.

[10] C. C. Ruiz Sanabria, O. Richard, B. Videau, and I. Oleg. Managing large scale experiments in distributed testbeds. In *Proceedings of the 11th IASTED International Conference*. ACTA Press, 2013.

[11] E. Schulte, D. Davison, T. Dye, and C. Dominik. A multi-language computing environment for literate programming and reproducible research. *Journal of Statistical Software*, 46(3):1–24, 1 2012.

[12] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut. Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. In *Proceedings of the 20th Euro-Par Conference*. Springer-Verlag, Aug. 2014.

[13] L. Stanisic, B. Videau, J. Cronsioe, A. Degomme, V. Marangozova-Martin, A. Legrand, and J.-F. Méhaut. Performance analysis of hpc applications on low-power embedded platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 475–480. EDA Consortium, 2013.

[14] V. Stodden, F. Leisch, and R. D. Peng, editors. *Implementing Reproducible Research*. The R Series. Chapman and Hall/CRC, Apr. 2014.

[15] Companion of the StarPU+SimGrid article. Hosted on Figshare: http://dx.doi.org/10.6084/m9.figshare.928338, 2014. Online version of [12] with access to the experimental data and scripts (in the org source).