Effective Reproducible Research with Org-Mode and Git

Luka Stanisic and Arnaud Legrand

CNRS/Inria/Univ. of Grenoble, Grenoble, France
{firstname.lastname}@imag.fr

Abstract. In this article 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 workflow simply builds on two well-known tools (Org-mode and Git) and enables to address issues such as provenance tracking, experimental setup reconstruction, replicable analysis. Although this workflow is perfectible and cannot be seen as a final solution, we have been using it for two years now and we have recently published a fully reproducible article [3], which demonstrates the effectiveness of our proposal.

1 Introduction

In the last decades, both hardware and software of modern computers became so complex that even experts have troubles fully understanding their behavior. Therefore, it could be argued that these machines are no longer deterministic, especially when measuring execution times of programs running on a large distributed computer systems or hybrid platforms. Controlling every relevant sophisticated component during such measurements is almost impossible, making the full reproduction of the 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 based on experimental results in this domain of computer science, surprisingly articles generally poorly detail the experimental protocol. Left with insufficient information, readers have generally trouble to reproduce the study and possibly build upon it. Yet, as reminded by Drummond [1], *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 [5]. 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.

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part I, LNCS 8805, pp. 475–486, 2014.

There are many existing solutions partially addressing these issues, but none of them was completely satisfying our needs and therefore we developed an alternative approach that is based on two well-known and widely-used tools: Org-mode and Git. We present our contributions in Section 3, where we first describe a specific use of Org-mode for doing the provenance tracking of the entire projects. Then, we propose a unique way to use Git for keeping synchronized experiment results and code that generated them. Finally, it will be demonstrated in Section 4 how the proposed methodology helped us conducting two very different studies in the High Performance Computing (HPC) domain. We will also state limits of our approach, together with some open questions.

2 Related Work

In the past few years, the field of reproducibility research tools has been very active, various alternatives emerging to address diverse problematic. However, in the HPC domain most of them are concentrated on platform accessibility, setting up environments and running the experiments on large clusters. Even though such tools are very useful in general, we could not benefit from them due to the specific nature of our research projects. The machines we needed to study are recent prototypes, with ever-changing libraries, set up by expert administrators and we do not have neither the permission nor the interest to do any modifications to the environment.

However, when conducting experiments and analysis, there are more aspects that need to be considered. We detail the ones 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. Making data accessible, file hosting services, such as Dropbox, Google Drive and many others, became very popular among all scientists that want to collaborate. There is another group of services that is more oriented on making data publicly available and easily understandable to everyone, e.g., figshare[1].

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., and it is often managed by experimental engines. The second, frequently forgotten in our domain, part is to track any applied transformation to the data, i.e., moving the objects from one state to another. Moreover, there is a question of storing both data and meta-data. 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 handled in much better way by using version control systems and literate programming.

[1] http://figshare.com

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 [5, chap. 3]), while the other one consists in writing a laboratory notebook (e.g., with IPython[2]).

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 system has to be flexible. In such a volatile context, integrated tools with databases, such as Sumatra, are too cumbersome for everyday extentions and modifications.

Replicable analysis. Researchers should only trust figures and tables that can be regenerated from raw data. Therefore, ensuring replicable analysis is essential to any study. Popular solution nowadays for this problem is to rely on open-source statistical software like R and knitr that simplify figure generation and embedding in final documents [5, chap. 1].

To address the previous problems, we propose to rely on a minimalist set of simple, lightweight, and well-known tools. We use **Org-mode** [2], 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 snippets). Such features follow the literate programming principles introduced by Donald Knuth three decades ago, and for which there is a renewed interest in the last years. In addition, for version control system we decided to rely on **Git**, a distributed revision control tool that has an incredibly powerful branch management system.

3 Tips and Tricks for Reproducible Research

In this section, we provide guidelines on best practices and hints on pragmatic ways to implement them. First, we illustrate how to handle provenance tracking issues with literate programming, in particular with Org-mode. The approach we propose is lightweight, to make sure the experiments are performed in a clean, coherent and hopefully reproducible manner without being slowed down by a rigid framework. However, it is tempting to sometimes break one of these rules, which hinders reproducibility in the end. This is why we harden these guidelines with a particular usage of Git that forces the user to keep data and code synchronized.

3.1 Provenance Tracking through Literate Programming

As described in Section 2, there are many tools that can help to automatically capture environment parameters, to keep track of experimentation process, to

[2] http://ipython.org

organize code and data, etc. However, it is not easy to understand how they work exactly, and additionally each of them creates new dependencies on specific libraries and technologies. 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 simple, flexible scripts, that address the following challenges.

Environment Capture. Environment capture consists in getting all the details about the code, used libraries and system configuration. It is necessary to gather as much useful meta-data as possible, to allow to compare experiment results with the previous ones and inspect if there were any changes to the experimental environment. 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.

Once meta-data is captured, it can be stored either individually or accompanying results data. Some prefer keeping these two separated, making their primary results unpolluted and easier to exploit, but they soon run into difficulties when they need to retrieve information from meta-data. Therefore, we strongly believe that the experiment results should stay together with the information about the system they were obtained on. Moreover, keeping them in the same file makes the access straightforward and simplifies the project organization, as there are less objects to handle. Consequently, 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 haven't yet met with such kind of scenario, but obviously it would require some adaptations to the approach.

Note that all the data and meta-data are gathered automatically using scripts, finally producing a read-only Org-mode document. Why the experiments were performed and what are the observations on its results is stored elsewhere, more precisely in a laboratory notebook of the project.

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 multiple purposes. Indeed, the labbook should not only serve as journal but also play the following software development roles to ensure that it can be exploited by others:

1. README: The labbook should explain ideas behind the whole project purpose and methodology, i.e., what the workflow for doing experiments is and how the code and data are organized in folders. It should state the conventions on how the labbook should be used. This part serves as a starting point for newcomers and is also a good reminder for experienced users.
2. Documentation: The labbook should detail what are the different programs and scripts, and what is their purpose. This documentation concerns source code for the experimentation as well as tools for manipulating data and analysis code for producing plots and reports. Additionally, there should be explanations on the revision control usage and conventions.
3. Examples: The labbook should contain example usages 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 everyday users, since some scripts require lots of environment variables, arguments and options.
4. Log: It is important to keep track of big changes to the source code and the project in general inside a log section. Since all modifications are already captured and commented in Git commits, the log section should offer a much more coarse grain view of the code development history. There should also be a list with descriptions of every Git tag in the repository as it helps finding the latest stable, or any other specific, version of the code.
5. Experiment results: Every set of experiment should be carefully noted here, together with the key input parameters, the motivation for running such experiment and finally observations on the results. Inside the descriptive conclusions, Org-mode allows to use both links and git-links connecting the text to the actual files in the Git repository. These hyperlinks point to the crucial data and the analysis reports that illustrate a newly discovered phenomena.

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. Org tags can be used to distinguish which collaborator conducted a given set of experiments or to list expertise requests. Although many of these information may already be present in the experiment files, having it at the journal level revealed very convenient. Experiments can also be tagged to indicate on which machine they were performed and whether the results were important or not. Again, although such tagging is not required it is very handy in practice and make the labbook much easier to understand and exploit.

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. Using a plain text file is also the most portable format across different architectures and operating systems.

Data File Organization. Having a clear, coherent and hierarchical organization of all the files is a good practice for a proper scientific work, especially when external collaborators are involved. Once again, the approach we propose is lightweight and flexible but is motivated by the three following important points:

1. There should never be any critical information in file organization. Important information should be in the files themselves. Indeed, we could as well have blobs rather than files but managing data and extracting important information would probably not be very convenient. Thus, we do not recommend to impose much on file organization so that users can organize their data in a way that is natural to them. We think this lack of rules is not an issue as long as this organization is explained in the labbook.
2. The file organization should be flexible enough to be changed and adapted as the experimental data set grows. Such reorganization could seemingly break the labbook hyperlinks. However, as we briefly mentioned, we recommend to use git-links in the labbook, which are Org-mode hyperlinks that store links to specific revisions of files (typically when they were created). So reorganizing the data files will not break the labbook information.
3. The naming convention should not impede the activity of the researcher, so here we used almost no convention at all. According to our experience, all experiment results could simply be saved in folders, each of them representing one set of measurements and having a unique characteristic name, e.g., the name of the machine on which it was performed. Inside a folder, file names could be prefixed by additional key characteristics of the experiment set, followed by an ordinal number indicating in which order experiments were run. This idea seemed the most natural one to adopt, but we are reconsidering alternatives for the future projects.

Conclusion. The approach we described implicates a partial redundancy of some data and meta-data, typically saved in both experiment result files and in the laboratory notebook. However, such information are never entered twice manually. Most data should always be automatically tracked, although when some data convey key information, they should be manually added to other places as well, since it provides a better overview of the whole project.

We think that following the proposed guidelines is sufficient to conduct a clean, comprehensible and reproducible research while having a very fluid workflow. However, not all scientists are rigorous enough to always follow such conventions and even those who are, occasionally have the need to bend the rules in order

to quickly get some results in a dirty way. This sometimes pollutes the whole project organization, often breaking the chains of the workflow processes and making some parts incoherent.

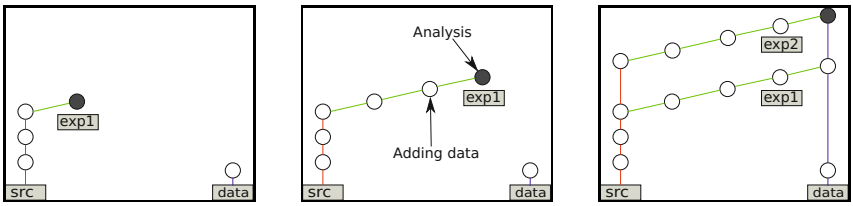
In order to force researchers to be more disciplined and help them doing their work in a reproducible manner, we propose to combine the previous approach with a particular usage of Git.

3.2 Using Git for Improving Reproducible Research

Even when the project is well organized, meta-data tracked, all the collaborators follow the conventions and take notes in the laboratory notebook, several practical issues may still arise:

1. Although a svn revision or a Git sha1 of the source code is captured, this does not guarantee that the experiment was run correctly and that the results can easily be reproduced. There could exist some uncommitted differences between the tracked and the current version of the code or the compilation could be out-of-date, i.e., code was compiled with an old revision.
   A solution proposed for example by Davison [5, chap. 3] and which we applied as well, is to force recompilation and systematically verify that everything is fully committed before running any experiment. The only exception to this rule are the tests performed to validate the workflow.
2. Even with the complete and correct meta-data and code revisions, it is not always easy to reconstruct the experimental setup, especially if it consists of the code from numerous external repositories. Multiplying repositories hinders provenance tracking, coherency and experimental reproduction. The solution we propose is to increase the reproducibility confidence level by using only revision control and a collection of simple scripts that automatically track information. Additionally, we suggest to store both code and experimental data in the same Git repository, so that they are always perfectly synchronized, which eases the obtainment of the code that produced a particular data set. Nevertheless, this introduced the following new challenges.
3. Unlike source code, data files can be large, thus keeping them together in the same repository can rapidly increase its size. Moreover, doing code modifications, analysis and experiments in the same Git branch complicates its history and makes experimental setup reproduction slightly ambiguous.
4. Another difficulty comes from managing several external beta source codes that should now coexist in the same repository. Since these codes are also under development, they are regularly upgraded by their developers and these changes need to be propagated to local project as well. Additionally, these codes typically have their own revision control, which rises many issues with potential local code modifications and commits that concern now both local and external projects.

Proposal. To solve aforementioned problems we propose an approach that uses Git with two parallel interconnected branches, displayed in Figure 1. The first



(a) Development (b) Experimentation (c) Merging results

Fig. 1. Different phases in git workflow

branch, named *src*, includes only the source code, i.e., code and scripts for running the experiments and the analysis. The second, *data*, branch consists of all the source code as well as all the data and analysis reports (statistical analysis results, figures, etc.). These two branches live in parallel and are interconnected through a third type of branches, the *exp#* branches, where the experimentation is performed. All these together form a "ladder like" Git repository, depicted in Figure 1(c).

We now explain the typical workflow usage of such branching scheme:

1. **Development phase**: Researchers work on a code development inside the *src* branch, committing the changes, as shown on Figure 1(a). These modifications can impact local or external 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 doing real useful experiments.
2. **Experimentation phase**: Researcher creates a new branch from the *src* branch containing and a new folder to store the results. We used the convention that these two (branch and directory) should always have equal names, which eases the usage of both Git and labbook. In the example of Figure 1(a), this new branch for doing the measurements is called *exp1*. Next, new experiments are executed running the scripts and generate results. The resulting data, together with the captured environment meta-data, are then committed to the Git. After that, one might want to do some basic analysis of the data, investigating the results, which may later trigger another round of experimentation and so on, as it is showed on the Figure 1(b). Finally, only when all desired measurements are finished, *exp1* will be merged with the *data* branch.
3. **Merging and reports phase**: All experimental *exp#* branches are in the end merged with *data*, as it can be seen on Figure 1(c). In addition, result observations for each *exp#* branch are written to the labbook. Afterwards, comparison of different experiments can be performed by generating figures, tables and clear explanations, to describe the newly discovered phenomena. Since the changes to the source code from *src* branch are also propagated through *exp#*, the head of the *data* branch will always contain the latest code together with all the data. Nevertheless, the older version of code responsible for producing a particular data or analysis can always be found in the Git history.

A peculiar situation occurs when there are source modifications inside the experimental branches. They have to be committed (as measurements are never done with an uncommitted code), even though in most cases they represent an ad hoc change, specific to an individual machine and its installation. These minor, local hacks would pollute the *data* branch and thus it has to be ensured that they are not propagated. It is done by using a special script for merging the branches instead of classical Git merge command. At the end of the *exp#* branch, all source code changes (not the data) have to be invalidated by using `git revert`, i.e., the "anti-commit" of all the previously committed modifications inside that branch. This way modifications remain local for that *exp#* branch and the experimental setup can still be reproduced by pulling the revision before the revert operation, i.e., the one used to generate the data.

If the researcher eventually realizes that some of the source code modifications done inside *exp#* branch could be useful for the whole project, there are two ways to inserted them in the *src* branch. The first one involves rewriting Git history and it is not advised as it leads to incoherences between Git repositories. The second option is to cherry-pick the desired commits. Although this approach produces some redundancy, it is very easy and safe and keeps the Git history clear and coherent.

External software. One more challenge arises when there are external software repositories imported inside a local project. For example, one could have external source code B that is a part of a bigger local project A. Since B is also under development, one occasionally needs to pull the updates from its server, which can cause conflicts with local modifications to the code. Resolving these conflicts manually can sometimes be very tedious.

Even bigger problem occurs if one wants to push such local changes, as they can be committed to either our project A, or external project B, or even to both of them together. We decided to propagate, by default, these modifications only to the project A, keeping the Git sha1 of A always valid and up-to-date. Later if necessary, they can be pushed to B as well, but this has to be done by explicitly calling the necessary commands.

Dealing with described challenges is error-prone, thus we started using `git-subrepo` tool for cleaner and semi-automated management of external Git projects inside our local one. However, we still have to do everything manually when working with codes that are using other version control systems, notably svn.

Analysis. We now analyze the proposed solution and investigate how it addresses the stated problematic.

First, a complete synchronisation of code, data and analysis is ensured. The convention to use the same name for Git experiment branches and folders containing experiment results, additionally carefully noting it in the labbook, makes exploration of project history very smooth. This way one can easily find when a particular data or a figure was created, pull the revision used to generate it, inspect the code, reconstruct the environment and finally reproduce the object.

Using experimental branches also allows some local source modifications, that are specific for the remote machine or some other part of the experimentation setup. These changes stay local for that *exp#* branch avoiding to pollute main *src* and *data* branches, elegantly making the project easier to read while still keeping it coherent. Additionally, Git permits to put tags on certain commits, which can be used to annotate an important revision, such as the ones with the stable source code or the ones with some specific adjustments. With the evolution of the study, Git history becomes large and harder to explore, thus these tags can help to quickly find a desired state of the project.

By using Git as proposed, it is extremely easy to set up an experimental platform on a remote machine by pulling only the head of the *src* branch. This solves the problem of long and memory consuming retrieving of the entire data and Git history, as the *src* branch is typically very small.

On the other hand, one might want to gather all the experimental data at once, which can be easily done by pulling only the *data* branch. This is the case for the researchers that are not interested in the experimentation process, but only at the analysis of the whole set of results. For them, *src* and *exp#* branches are completely transparent, as they will retrieve only the latest version of the source code (including needed analysis scripts) and the entire data.

Another use case is when someone wants to write an article or a report based on the experiment results. A completely new branch can be created from *data*, selecting from the repository only the data and analysis code needed for the publication and deleting the rest. This way complete history of the study behind the article is preserved for the reviewers.

Holding external projects inside a local one allows to do `git pull` or `svn checkout` in sub-directories, keeping them up-to-date. Small problems arise when some modifications to the external code are done. These changes *de facto* influence both external and local repository but our solution ensures that they are committed only to the local revision control. Therefore in the meta-data part of the experiment files, tracked revision of the external code corresponds to the version pulled before any of the local changes, which is not strictly legitimate. Nevertheless, this small issue is not critical, since the revision in local project is stored in meta-data as well, and this value is always perfectly correct. Pulling this version will bring the right code, keeping the research reproducible.

4 Evaluation

We used the described methodology in two very different use cases. The first one is a part of the study of CPU cache performance on various Intel and ARM micro-architectures [4]. The developed source code was very simple, containing only a few files, but there were numerous input parameters to be taken into account. Probably the critical part of this study is about 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. Although it did not lead to a reproducible article as we were only discovering such tools, we used this workflow and can still

track the whole history of these experiments. The second use case [3] aims at providing accurate performance predictions for dense linear algebra kernels, using the StarPU runtime on top of the SimGrid simulator. By contrast, input parameters and environment here are fixed, but the source code is very complex and in constant evolution. Moreover, we had to manage code from two external repositories as well many of our own scripts.

The proposed solution proved generally successful in both use cases. We have determined several good and bad sides, while for some aspects still remain rather uncertain.

Pros. Our approach is fast and efficient for a daily usage. It provides reasonable boundaries without taking away too much flexibility from the users. It offers good code modification isolation, which is important for *ad hoc* changes. Perfect provenance tracking, which was painless to extend and explore, was crucial for the cache measurement study. Although these two use cases are very different, most of the captured environment meta-data is the same for both projects: date and time, hostname, Linux and gcc version, users logged on the machine, environment variables, used external libraries, code revisions, memory hierarchy of the machine, CPU governor and frequency, compilation outputs, etc. Since all the source code and data is in Git repository, reconstructing experimentation setup is straightforward. 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 would have to use virtual machines to run the experiments, which would introduce many new performance issues. Finally, after applying such a methodology throughout the whole research process, it was extremely easy to write an article [3] in Org-mode that was completely reproducible as well. Along with the text, this Org-mode document contains all the analysis scripts and the raw data is provided as the article companion [6] and can be inspected by reviewers.

Cons. The biggest disadvantage of our approach is that it has many not so common conventions along with a steep learning curve workflow, 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. However, we believe that these tools provide benefits that are worth investing time.

Additionally, we find current way of managing external source repositories slightly cumbersome, and we are searching for a better solution. One path could be to use recipes or experiment engines, that would do the checkout of external sources for us and would only apply the right modifications before compiling.

The problem of storing large data files in repositories is well-known to the community. It has been already solved for the Mercurial revision control tool, but even after an thorough research we could not find a satisfactory solution for Git. Git repositories can quickly reach a few Gigabytes, which does not hinder daily committing but significantly slows down rebase operations to move back to previous experimental conditions of a specific dataset.

Open Questions. 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 haven't 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 haven't anticipated.

These are only few of the unknown, and as there are certainly many more, we are hoping for the audience suggestions and remarks.

5 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 described is a good example of using well-known, lightweight, open-source technologies to properly perform a very complex process like computer science experimentation. Although our methodology is undoubtedly improvable and similar results could be obtained with other frameworks, we nonetheless find it very smooth 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 clearly a very effective way to produce a reproducible research.

References

1. Drummond, C.: Replicability is not reproducibility: Nor is it good science. In: Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML (2009)
2. Schulte, E., Davison, D., Dye, T., Dominik, C.: A multi-language computing environment for literate programming and reproducible research. Journal of Statistical Software 46(3), 1–24 (2012), http://www.jstatsoft.org/v46/i03
3. Stanisic, L., Thibault, S., Legrand, A., Videau, B., Méhaut, J.-F.: Modeling and simulation of a dynamic task-based runtime system for heterogeneous multi-core architectures. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014 Parallel Processing. LNCS, vol. 8632, pp. 50–62. Springer, Heidelberg (2014)
4. Stanisic, L., Videau, B., Cronsioe, J., Degomme, A., Marangozova-Martin, V., Legrand, A., Méhaut, J.F.: Performance analysis of hpc applications on low-power embedded platforms. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2013, EDA Consortium, pp. 475–480 (2013), http://dl.acm.org/citation.cfm?id=2485288.2485403
5. Stodden, V., Leisch, F., Peng, R.D. (eds.): Implementing Reproducible Research. The R Series. Chapman and Hall (2014), https://osf.io/s9tya/wiki/home/
6. Companion of the StarPU+SimGrid article. Hosted on Figshare online version of this article with access to the experimental data and scripts (in the org source, 2014), http://dx.doi.org/10.6084/m9.figshare.928338