Abstract. Scientific discoveries across all fields, from physics to biology, are increasingly driven by computer simulations. At the same time, the computational demand of many problems necessitates large-scale calculations on high-performance supercomputers. Developing and maintaining the underlying codes, however, has become a challenging task due to a combination of factors. Leadership computer systems require massive parallelism, while their architectures are diversifying. New sophisticated algorithms are continuously developed and have to be implemented efficiently for such complex systems. Finally, the multidisciplinary nature of modern science involves large, changing teams to work on a given codebase. Using the example of the DCA++ project, a highly scalable and efficient research code to solve quantum many-body problems, we explore how computational science can overcome these challenges by adopting modern software engineering approaches. We present our principles for scientific software development and describe concrete practices to meet them, adapted from agile software development frameworks.

1. Introduction

High-performance computing (HPC) is approaching the exascale era, which will open up great opportunities in various research areas of computational science. Exascale supercomputers could provide the performance boost required for more precise weather forecasts [1], accelerate the computational design and discovery of functional materials [2], and make simulations of the full human brain possible [3]. The transition to exascale computing will involve a diversification of computer architectures and a vastly increasing demand on parallelism [4]. Thus, to live up to the exascale promises, computational science is confronted with the tasks of porting existing application codes, developing new scalable algorithms and addressing issues such as fault tolerance [5].

The evolution of leadership computing systems is only one facet of the complex, dynamic environment computational science is embedded in. The exploratory and iterative nature of science requires the continued development of domain specific algorithms, which tend to grow in complexity. For scientific success and impact, these algorithms have to be turned rapidly into production code. Academic integrity requires verification and validation of the codes, and reproducibility of simulation results they produce. The high personal turnover in academia and

DCA++ project: Sustainable and scalable development of a high-performance research code

U R Hähner, G Balduzzi, P W Doak, T A Maier, R Solcà and T C Schulthess

1 Institute for Theoretical Physics, ETH Zurich, 8093 Zurich, Switzerland
2 Computational Science and Engineering Division, Center for Nanophase Materials Sciences, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, USA
3 Swiss National Supercomputing Center, ETH Zurich, 6900 Lugano, Switzerland
E-mail: haehneru@itp.phys.ethz.ch

1290 (2019) 012017 doi:10.1088/1742-6596/1290/1/012017
the volatility of funding sources, in turn, cause research teams to constantly change in size and composition. This implies that new team members need to be trained and the contributions of leaving team members preserved.

To maintain scientific productivity, software development in computational science needs to sustain the changes in computer architectures, algorithms and teams. To further foster scientific productivity, software development in research groups should likewise feature scalability in the number of developers such that a growing team indeed accelerates software development and thereby the scientific progress. Sustainable and scalable software development accompanied by sufficient quality assurance can be supported by appropriate practices. Traditional software development processes that require up-front specification of requirements, such as the waterfall model [6], are in contrast to the uncertainty of the scientific discovery process [7]. Scientific software development, however, can benefit from modern software engineering methods, such as the agile methodology, which promotes an iterative, adaptive and collaborative approach [8].

The term agile software development describes practices and frameworks that embrace the values and follow the principles formulated by the Manifesto for Agile Software Development [9]. Besides the anticipation of change, the agile methodology values the people involved in the development process and emphasizes continuous improvement. While there are various successful agile frameworks such as Scrum [10] and Lean software development [11], we found the concept of Extreme Programming (XP) [12] most useful to our project. XP defines a set of practices that improve software development with respect to the agile values. These practices can be applied consecutively to the development process according to the project’s goals and needs.

In the development of the DCA++ project [13], a high-performance research software framework in the field of condensed matter physics, we are faced with the challenges of scientific software development as described above. The codebase is developed and maintained by a multi-site team located at ETH Zurich, the Oak Ridge National Laboratory (ORNL) and the affiliated HPC centers, the Swiss National Supercomputing Center and the Oak Ridge Leadership Computing Facility. Accordingly, the team’s competences range from condensed matter and computational physics to applied mathematics and computer science. The realm of the DCA++ project is the development of modern quantum cluster methods, such as the dynamical cluster approximation (DCA) [14, 15, 16] and its DCA+ extension [17, 18], and their efficient implementation for leadership computing systems. These methods can be applied to models of high-temperature superconductivity to gain understanding of superconducting materials such as the copper-oxide based compounds, usually referred to as cuprates [19, 20].

At the heart of the DCA and the DCA+ algorithms is a continuous-time quantum Monte Carlo (QMC) solver [21], which consumes most of the compute time of a typical DCA++ production run. For this reason, continuous effort is put into the optimization of these QMC algorithms to increase performance and to reduce time-to-solution. Feature extensions to the DCA++ code are usually driven by new research questions and novel theoretical models, and agility is required to develop code that satisfies the needs of our domain scientists and collaborators. To achieve sustainability and scalability, and to incorporate systematic testing in the DCA++ project, we have explored, integrated and adapted agile software development practices.

In the remainder of this paper, we first introduce four principles - inspired by the Manifesto for Agile Software Development - that we believe are necessary for successful and sustainable development of a high-performance research code. Some of the principles, such as the design principles for sustainable software, translate into immediate actions, while others, such as transparency and reproducibility in the context of open science, require suitable practices. In section 3, we describe concrete practices that we apply in the DCA++ project to comply with our principles. We conclude by describing challenges when adopting new practices in general and agile practices in a research group in particular.
2. Principles

2.1. Communication and feedback

To benefit from the multidisciplinarity of the DCA++ team, active and frequent communication between team members is at the heart of the development process. The spatial distance between the development sites makes messaging and collaboration tools like Slack [22] as well as video conferences important everyday channels of communication. To overcome language barriers that naturally exist between different disciplines, face-to-face meetings are still preferred and in many cases absolutely necessary to make efficient progress.

Developmental work on the DCA++ code can generally be divided into two categories: (1) implementation of new science capabilities, and (2) optimization of existing code for either new computing architectures or for allowing access to larger problem sizes. An initial implementation of a new science capability is usually performed by the team members closest to the science. In short feedback loops, students and postdocs develop the new feature in collaboration with their domain science advisors. To optimize existing code for execution on leadership computing hardware, expertise from all disciplines is required: domain scientists provide insight into domain specific algorithms, applied mathematicians translate the performance critical parts into the language of linear algebra, and computer scientists estimate computational work and memory requirements, and evaluate parallel programming models and design choices. The collective effort results in initial ideas for algorithmic improvements and code optimizations. As for the implementation of new capabilities, these initial ideas are followed up by an iterative and collaborative development process.

To gain confidence in suitability and correctness of developed code, it is essential to supplement feedback through discussions by verification and validation (V&V). The characteristics of scientific software are captured by the computational science specific definition of V&V given by Babuska and Oden [23]:

- **Verification**: “The process of determining if a computational model obtained by discretizing a mathematical model of a physical event and the code implementing the computational model can be used to represent the mathematical model of the event with sufficient accuracy.”
- **Validation**: “The process of determining if a mathematical model of a physical event represents the actual physical event with sufficient accuracy.”

2.2. Technical excellence

The fast evolution of leadership computer systems is both a challenge and a chance for computational science. Iterative development of scalable and efficient algorithms is required to use the growing capacity of these machines and to progress in science. DCA++ was able to exploit the first petascale supercomputer available to open science, ORNL’s Jaguar, by featuring multi-level parallelism, cache-friendly delayed Monte Carlo updates [24] and mixed precision support. This made DCA++ the first scientific application to break the sustained petaflop barrier on a general-purpose supercomputer [25]. Later, DCA++ successfully responded to the emergence of hybrid CPU-GPU systems and sustained 15 petaflop/s on ORNL’s Titan supercomputer, the successor to Jaguar [26]. The increasing diversity of architectures, from CPU-only to GPU-dominated node configurations, demands that we continue to develop the DCA++ code with regard to evolving supercomputing capabilities.

At the same time, we seek for algorithmic improvements that overcome current limitations and make parameter ranges accessible that have been out of reach before. The DCA+ algorithm, for example, introduces a continuous lattice self-energy in the DCA formalism, which further cures the fermionic sign problem of the embedded quantum Monte Carlo method [17]. This advancement enables large cluster calculations of the 2D Hubbard model with values
of the Coulomb interaction that are relevant for the study of models of high-temperature superconductors [26].

Technical excellence is not limited to algorithm development, but also applies to software design. We strive for designs that permit clear expression of algorithms, and feature flexibility to support various types of theoretical models and alternative algorithms to solve them. Designs also need to ensure extensibility to new models and algorithms, and allow easy adaptation to changing requirements (see section 2.3). The solution is an object-oriented and generic programming model. The C++ programming language provides us *inheritance* and *templates* to implement it.

### 2.3. Sustainability

We pursue sustainability as the ability to continuously develop software that meets our needs of today and that is prepared to adapt to our needs of tomorrow. The anticipated change of needs is driven by both the progress in science (e.g. development of new algorithms) and the evolution of leadership computing platforms (e.g. emergence of new architectures). Consequently, we characterize sustainable software as being *maintainable* and *extensible*, in line with current suggestions of the software engineering community [27].

Development of sustainable software can be enabled through appropriate (design) principles and practices. In object-oriented programming, the *SOLID* design principles provide a guideline for writing maintainable and extensible code. The acronym SOLID encapsulates five individual principles [28]:

- **Single responsibility principle:**
  “A class should have only one reason to change.”

- **Open/closed principle:**
  “Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.”

- **Liskov substitution principle:**
  “Subtypes must be substitutable for their base types.”

- **Interface segregation principle:**
  “Clients should not be forced to depend on methods they do not use.”

- **Dependency inversion principle:**
  A. “High-level modules should not depend on low-level modules. Both should depend on abstractions.”
  B. “Abstractions should not depend upon details. Details should depend upon abstractions.”

We refer to the literature for the details of each principle, but we want to emphasize that *SOLID code* features high *cohesion* and low *coupling*, attributes of a maintainable and extensible design.

In addition to these concrete principles for object-oriented design, sustainability can be fostered by the following general design principles:

- **KISS ("Keep it simple, stupid"):** Simple designs are easier to understand and maintain.

- **YAGNI ("You aren’t gonna need it"):** Incremental design [12] combined with refactoring (see section 3.4), instead of up-front design, is in accord with the exploratory nature of research.

- **DRY ("Don’t repeat yourself") [29]:** Duplication multiplies the cost of change.

A further key principle to sustainable software development is valuing *defect prevention* over *defect detection* [30]. The cost of a defect ranges from fixing it, to repeating expensive production runs, to possibly retracting publications, and increases significantly with the time between
deployment and discovery. Thus, catching defects early and preventing them from entering production code minimizes maintenance cost and ensures sustainability. In practice, defect prevention can be accomplished by creating comprehensive automated tests for any functional code (see 3.3. Test-driven development) and having code pass two or more pairs of eyes (see 3.4. Code reviews).

2.4. Open science
Transparency and reproducibility of scientific research is of paramount importance for scientific integrity. Routines that aid in transparency and reproducibility are adopted by an increasing number of journals. Many publications now contain raw data associated with published results in additional online material, sometimes even integrating code, as well as details of the analysis that was used in additional supplemental material.

Scientific work produced with the DCA++ application follows most of the “ten simple rules for reproducible computational research” set up by Sandve et al. [31]. A typical workflow consists of running the DCA++ code on an HPC system, generating data, which then in most cases is transferred to a local workstation for analysis. Since this workflow does not employ an integrated workflow environment, most of the the rules are followed explicitly and sometimes manually. As a minimal requirement, we track sufficient information during the workflow so that we can reproduce any results presented in published work at a later date. The DCA++ output files include full build details: source code version information (see section 3.6), loaded modules and configuration options (see section 3.2). These output files, the raw data as well as the input files used for generating them are stored and archived together with the manuscript files, as are the scripts that create published figures.

As part of open science, any new developments in the DCA++ code are publicly available as open source, or are made available with a certain delay. By open sourcing the code, we want to further increase transparency and reproducibility, drive transfer of knowledge and promote software reuse in our domain. Open source scientific software likewise facilitates the important process of verifying and validating codes. An independent implementation of the same algorithm provides a useful baseline for verification, while validation can be performed against predictions of alternative methods.

3. Practices
3.1. Version control system
The benefits of using a version control system (VCS) are substantial to the success of a long-lived software project. The full long-term history of every file in terms of documented commits allows tracking changes and rolling back to a specific version if necessary. This helps to understand how the design of the software has evolved in general and to analyze and resolve bugs in particular. Storing versions properly likewise contributes to reproducibility of simulation results.

A VCS also provides the means for scalable collaborative software development. When team members work concurrently on the same part of the code, the VCS can often automatically merge these changes into a common version. If conflicting changes arise, the VCS assists to detect and resolve them. Most VCS allow tracking an arbitrary number of branches of a codebase, where a branch is a particular history of changes made to the code. The process of bringing two branches into the same state, where they contain identical code, is referred to as a merge. By branching and merging, independent features can be implemented in isolation in dedicated branches and integrated back into the main trunk, usually called master branch, when they are ready. That way, the master branch, holding the production code, is operational at all times.

For the DCA++ project, we chose the widely-used setup of a Git repository [32] hosted on the GitHub platform [33]. Git is a distributed VCS, where every developer’s local working copy of the codebase is a full-fledged clone of the repository. Git excels at performance and makes
branching and merging fast and easy. Besides hosting Git repositories, GitHub offers a full stack of features to maintain, manage and share a software project through an accessible web interface.

The full benefits of a version control system can only be experienced when the entire codebase is maintained in a single repository. Within this repository, branching capabilities should not be abused to create de facto divergent production codebases on long-lived branches. This one codebase per researcher issue is a familiar and avoidable waste in computational research that prevents progressive improvements of scientific codes as well as scalability and sustainability of scientific software projects. Instead, as outlined in section 3.6, we recommend a single production branch, the master branch, and few short-lived feature extension branches that will be rapidly integrated into master.

3.2. Automated build and test

Scientific software is usually built from source by the user. Diverse computing environments, long lists of dependencies and various possible configurations, however, can cause the build process to be time-consuming and error-prone, likely overburdening the user.

CMake [34] is a tool to automate configuring and building software, while being cross-platform and flexible to handle the variety of systems. CMake provides mechanisms to automatically detect compilers and external libraries, but also offers the possibility to manually specify these and other configuration options.

On HPC systems, we complete the build automation with the EasyBuild framework [35]. EasyBuild allows us to pack all dependencies of DCA++, i.e. CMake, compilers and all external libraries, into a single module. This module can be loaded like any other system module to obtain the full software stack required to build DCA++, as shown in the example steps of Listing 1.

```bash
$ module load DCA++/1.0-CrayGNU-17.08
$ git clone https://github.com/CompFUSE/DCA.git dca_source
$ mkdir build && cd build
$ cmake ../dca_source/ \
  -DCMAKE_BUILD_TYPE=Release \ 
  -DDCA_HAVE_LAPACK=TRUE
$ make
```

Listing 1. CMake based building procedure using Make as the native build system. We start by loading the DCA++ module, which provides all dependencies and has been installed with EasyBuild. Next, we clone the DCA++ repository to obtain the source code. In the third line, we create an empty build directory and change to it. Then, we execute the `cmake` command with the path to the source and appropriate options to configure the build, which generates a hierarchy of Unix Makefiles. Finally, we call `make` to build the applications.

Automated tests that run fast are a great way for developers to receive quick feedback about the changes they have made. Test-driven development (see section 3.3) applies this to the extreme. In the DCA++ project, all tests are written with Google Test [36], a C++ xUnit testing framework. Building and running these tests is easily automated by CMake and its test driver program CTest.

Automated continuous testing, i.e. executing the full test suite automatically within the development process, is a cornerstone in our workflow (see section 3.6) and performed by a Jenkins automation server [37]. Jenkins is connected to the GitHub repository that hosts the source code and triggered by pull requests and whenever commits are pushed to the master branch. Jenkins then builds all applications and tests on a production system, runs the tests and reports the results. Once per week, Jenkins does a full build, which includes all dependencies.
3.3. Test-driven development

Test-driven development (TDD) is a software development practice that produces self-testing code and fosters simple designs by following two basic rules [38]:

- Write new code only in response to a failing automated test.
- Eliminate duplication.

TDD implements these two rules with the Red-Green-Refactor development cycle:

1. Red: Create a small, automated test that fails.
2. Green: Write just enough code to make the new test and all other tests pass.
3. Refactor: Clean up and remove duplication.

The three steps are repeated continuously. The amount of functionality covered by a cycle and the number of intermediate steps in the refactoring stage depend on the complexity of the problem and the experience of the developer. Smaller steps, which can be taken with higher confidence, are generally favorable, though [38].

Software written with TDD features automated unit tests with 100% code coverage. Besides delivering assurance on correctness of new code and preventing regression of existing code, these tests are prerequisite for safe and fast refactoring (see section 3.4). Unit tests also provide an extremely valuable form of documentation. They describe specifications and usage by example and, as long as they pass, can never get out of date.

TDD often has a positive effect on the design of the code, as well. The design principles KISS and YAGNI (see section 2.3) are built into TDD: Code driven by tests does not do more than the tests require. “Don’t repeat yourself” (DRY) is the main goal of the Refactor step. TDD does not enforce the SOLID design principles (see again section 2.3), yet testable code is often SOLID. The SOLID principles, in turn, promote testability. The Liskov substitution principle (the L of SOLID), for example, enable us to substitute a dependency with a mock or stub without altering the behavior of the object to test.

The TDD cycle not only produces simple and testable code, its three steps also focus software development on one task at a time. In the Red step, we are concerned with the public interface of the new feature, which we describe with an automated test. The failing of the new test confirms the test’s validity for the feature. During the Green step, what only matters is getting the new test (and all other tests) to pass. The Refactor step is the moment to get the design right. The tests ensure that we do not break functionality in the process.

3.4. Refactoring

It is common that a software project accumulates technical debt over time, knowingly and unknowingly. We introduce technical debt when we explicitly or implicitly duplicate code, use an incomprehensible naming convention or add an unnecessary dependency, that is whenever we make a poor design choice. Paying off this technical debt and improving readability, maintainability and extensibility of the code is the goal of refactoring. By refactoring, we improve the internal design of existing code without changing the external behavior.

Well-practiced refactoring is a process of many tiny, behavior-preserving code changes. In isolation, these changes are almost unnoticeable, but their cumulative effect leads to a significant design improvement. Small, focused changes, compared to large, sweeping transformations, are simpler to implement, thus minimizing the risk of introducing bugs. Small changes are also easier to revert in case they do not provide the intended enhancement or do alter the external behavior.

There is a large catalog of standard refactoring techniques [39], often simply referred to as refactorings. Each of these techniques, such as Move Method or Remove Parameter, performs
a small, concrete code transformation. Many integrated development environments (IDEs) automate several of the refactorings, making them faster and safer to apply.

To ensure that external behavior is truly preserved, fast, automated tests are required, that sufficiently cover the part of the code to be refactored. In a short feedback loop, we first make a small change, then run the tests and only repeat if they pass. If a test fails, we need to revisit the change we have just made, and fix or undo it.

Refactoring code without adequate tests, *legacy code* according to the definition of Feathers [40], often puts us in a chicken-and-egg dilemma: We need to refactor to get tests in place to refactor. These situations require extra careful techniques and practices to safely break out of this cycle.

In the DCA++ project, we frequently refactor code before we add new functionality. It is a way to get a better understanding of the existing code that interfaces the new feature and to restructure it such that adding the new code is easy. Refactoring can also be an instrument for tracking down a bug in the code. Refactoring techniques like *Decompose Conditional reduce complexity*, while others like *Extract Method* isolate the defect [39].

3.5. Mutual code reviews

While code reviews have become standard practice in the software industry, it is our experience that scientific software development often lacks this activity. However, particularly when non-software engineers write code, reviews as part of the development cycle provide a way to guarantee code quality.

Code quality evidently includes correctness and sufficient test coverage to avoid the worst case scenario of having to retract one or more publications due to erroneous numerical results. Code reviews should likewise cover design, readability and conformity with an agreed coding style to ensure long-term maintainability of the software project, especially when the original author some day leaves the team.

Benefits of code reviews exceed quality assurance, though. The code review process offers a bidirectional learning opportunity. By receiving feedback on the submitted code, the author can improve their software development skills. The reviewer, on the other end, might see an as yet unknown programming language feature or is able to transfer an elegant solution to their own problem.

Code reviews also propagate knowledge of and behind the codebase across the team. That way, duplication of code and work is avoided and knowledge preserved in changing teams. Having multiple informed developers gives more flexibility in assigning tasks and allows to discuss important design decisions.

In the DCA++ project, we follow a strict policy and require everyone’s code to go through review. Code reviews are integrated in our workflow as part of the pull request (see section 3.6). This final examination before a feature gets merged into the master branch, however, does not replace early and frequent discussions as well as feedback.

Every developer on the team is a potential reviewer and can self-assign themselves to do a review. To get feedback on performance critical parts of the code, we also involve software engineers of the HPC centers we work with.

3.6. Issue branch workflow

A workflow to develop a high-performance research code has to be able to rapidly turn new algorithms and features into quality code that produces reliable results. At the same time, we want to continuously deliver improvements, as result of refactoring (see section 3.4) or in terms of bug fixes, to internal and external users. Like all other practices, the workflow has to be scalable in terms of developers and sustainable in a multidisciplinary, changing team.
We deal with these needs by breaking down all development work into tasks that are tracked as issues in the GitHub repository. Issues can be feature requests by group leaders or collaborators, bug reports by users, or a developer’s remark about the need to refactor a certain function or class. To ensure quick feedback and to prevent the so-called merge hell, issues are sized such that they can be implemented and integrated into the main line within a few weeks. Large or complex features, which we refer to as epics, such as the implementation of a new solver, are decomposed into small, independent tasks.

An open issue is assigned to a team member, who works on it in a dedicated branch (see section 3.1). When the work is completed, the code is submitted via a pull request into the master branch. A code review (see section 3.5) and automated testing (see section 3.2) control code quality before the changes become production code. Fig. 1 shows an exemplary network of branches corresponding to the described workflow.

![Figure 1](image_url) (Color online) Issue branch workflow. To guarantee stability of the master branch and to facilitate concurrent development, each issue is implemented in its own branch.

A large software project involves many independent as well as dependent tasks, either left to do or worked on in parallel. To visually track all tasks and to coordinate the development work in the team, we use a virtual Kanban board. Each task, i.e. open issue in the repository, is represented by a card that is moved across this board through four different stages: Backlog, In progress, Review, and Done (see Fig. 2). From our experience, the board is a great help in limiting work in progress, detecting impediments and prioritizing work.

### 4. Concluding remarks

We have presented four principles and six practices to address the challenges of developing a high-performance research code. The principles, communication and feedback, technical excellence, sustainability, and open science, have been treated separately for purposes of focus and clarity. However, they are not independent, but influence one another. Communication between team members is essential to achieve technical excellence, defect prevention as part of sustainable software development requires feedback through tests and from people, technical excellence includes sustainable software solutions, and open science allows for feedback from the entire community. Likewise, some practices are prerequisites for, or elements of, other practices. Test-driven development requires a framework for automated tests. Refactoring is a step of the test-driven development cycle and an essential practice on its own. Automated testing and code reviews are part of the issue branch workflow, which requires the branching capabilities of a version control system.

To be effective, the practices need to be adopted by the whole team. However, applying a new practice within a team is a difficult task. Practices cannot be imposed. Adopting new practices often implies changing the work routine and therefore requires the team’s openness for change and motivation to improve. The motivation to improve can be fostered through leading by example and demonstrating the benefits of a particular practice. Moreover, to preserve
Figure 2. (Color online) Version of a Kanban board for software development as it is used in the DCA++ project. Each card represents an open issue and can be labeled according to type or priority. All cards start in the Backlog column. When a team member begins working on an issue, its card is moved to In progress. The card proceeds to the Review column as soon as the corresponding pull request is opened. Closed issues are dropped in the Done column.

openness for change and to prevent overwhelming people, new practices should be introduced one at a time.

The iterative and collaborative nature of research meets well with the ideas of agile software development. Yet the surrounding conditions in research projects can hinder the successful application of agile practices. Team members and collaborators are often split across different institutions. This prevents regular face-to-face meetings as, for example, required for Scrum’s daily stand-up. Researchers usually work on multiple projects or have teaching responsibilities, which leads to scheduling conflicts for practices like XP’s pair programming. Adaption of practices is also necessary due to the differences between commercial and scientific software development. Agile software development values customer collaboration. To adopt corresponding practices in a research group, the customer role has to be defined.

Despite the challenges of introducing new practices into a team and applying agile practices in a research environment, the usage of the practices described in this paper contributed substantially to sustainability and scalability in the development of the DCA++ project. Believing in the principles of agile software development, we still want to improve. For example, we want explore more extreme programming practices such as pair programming.

Acknowledgments
U.R.H. and G.B. acknowledge the support by the NCCR MARVEL, funded by the Swiss National Science Foundation. The work of P.W.D. and T.A.M. was supported by the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and Basic Energy Sciences, Division of Materials Sciences and Engineering. This research used resources of the Oak Ridge Leadership Computing Facility (OLCF) awarded by the INCITE program, and of the Swiss National Supercomputing Center (CSCS). OLCF is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.
References

[1] Schulthess T C, Bauer P, Fuhrer O, Hoefler T, Schär C and Wedi N 2018 Comput. Sci. Eng. 10.1109/MCSE.2018.288878
[2] MaX Centre of Excellence – Materials Design at the Exascale http://www.max-centre.eu
[3] Jordan J, Ippen T, Helias M, Kitayama I, Sato M, Igarashi J, Diesmann M and Kunkel S 2018 Front. Neuroinform. 12 2
[4] Shalf J, Dosanjh S and Morrison J 2010 Proc. 9th Int. Conf. on High Performance Computing for Computational Science (Berkeley) (Berlin, Heidelberg: Springer) pp 1–25
[5] Alexandrov V 2016 J. Comput. Sci. 14 1–4
[6] Royce W W 1987 Proc. 9th Int. Conf. on Software Engineering (Monterey) (Los Alamitos: IEEE Computer Society Press) pp 328–338
[7] Segal J and Morris C 2008 IEEE Softw. 25 18–20
[8] Sletholt M T, Hannay J, Phihl D, Benestad H C and Langtangen H P 2011 Proc. 4th Int. Workshop on Software Engineering for Computational Science and Engineering (Honolulu) (New York: ACM Press) pp 1–9
[9] Beck K, Beedle M, van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R, Kern J, Marick B, Martin R C, Mellor S, Schwabe K, Sutherland J and Thomas D Manifesto for Agile Software Development http://agilemanifesto.org
[10] Schwabe K and Beedle M 2001 Agile Software Development with Scrum (Upper Saddle River: Prentice Hall PTR)
[11] Poppendieck M, Poppendieck T D and Poppendieck T 2003 Lean Software Development: An Agile Toolkit (Boston: Addison-Wesley Professional)
[12] Beck K and Andres C 2004 Extreme Programming Explained: Embrace Change (2nd Edition) (Boston: Addison-Wesley Professional)
[13] Hähner U R, Alvarez G, Maier T A, Solcà R, Staar P, Summers M S and Schulthess T C 2015 J. Comput. Sci. 14 1–4
[14] Royce W W 1987 Proc. 9th Int. Conf. on Software Engineering (Monterey) (Los Alamitos: IEEE Computer Society Press) pp 328–338
[15] Segal J and Morris C 2008 IEEE Softw. 25 18–20
[16] Sletholt M T, Hannay J, Phihl D, Benestad H C and Langtangen H P 2011 Proc. 4th Int. Workshop on Software Engineering for Computational Science and Engineering (Honolulu) (New York: ACM Press) pp 1–9
[17] Beck K, Beedle M, van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R, Kern J, Marick B, Martin R C, Mellor S, Schwabe K, Sutherland J and Thomas D Manifesto for Agile Software Development http://agilemanifesto.org
[18] Schwabe K and Beedle M 2001 Agile Software Development with Scrum (Upper Saddle River: Prentice Hall PTR)
[19] Poppendieck M, Poppendieck T D and Poppendieck T 2003 Lean Software Development: An Agile Toolkit (Boston: Addison-Wesley Professional)
[20] Beck K and Andres C 2004 Extreme Programming Explained: Embrace Change (2nd Edition) (Boston: Addison-Wesley Professional)
[21] Hähner U R, Alvarez G, Maier T A, Solcà R, Staar P, Summers M S and Schulthess T C 2015 J. Comput. Sci. 14 1–4
[22] Royce W W 1987 Proc. 9th Int. Conf. on Software Engineering (Monterey) (Los Alamitos: IEEE Computer Society Press) pp 328–338
[23] Segal J and Morris C 2008 IEEE Softw. 25 18–20
[24] Sletholt M T, Hannay J, Phihl D, Benestad H C and Langtangen H P 2011 Proc. 4th Int. Workshop on Software Engineering for Computational Science and Engineering (Honolulu) (New York: ACM Press) pp 1–9
[25] Beck K, Beedle M, van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R, Kern J, Marick B, Martin R C, Mellor S, Schwabe K, Sutherland J and Thomas D Manifesto for Agile Software Development http://agilemanifesto.org
[26] Schwabe K and Beedle M 2001 Agile Software Development with Scrum (Upper Saddle River: Prentice Hall PTR)
[27] Poppendieck M, Poppendieck T D and Poppendieck T 2003 Lean Software Development: An Agile Toolkit (Boston: Addison-Wesley Professional)
[28] Beck K and Andres C 2004 Extreme Programming Explained: Embrace Change (2nd Edition) (Boston: Addison-Wesley Professional)
[29] Hähner U R, Alvarez G, Maier T A, Solcà R, Staar P, Summers M S and Schulthess T C 2015 J. Comput. Sci. 14 1–4
[30] Royce W W 1987 Proc. 9th Int. Conf. on Software Engineering (Monterey) (Los Alamitos: IEEE Computer Society Press) pp 328–338
[31] Segal J and Morris C 2008 IEEE Softw. 25 18–20
[32] Sletholt M T, Hannay J, Phihl D, Benestad H C and Langtangen H P 2011 Proc. 4th Int. Workshop on Software Engineering for Computational Science and Engineering (Honolulu) (New York: ACM Press) pp 1–9
[33] Beck K, Beedle M, van Bennekum A, Cockburn A, Cunningham W, Fowler M, Grenning J, Highsmith J, Hunt A, Jeffries R, Kern J, Marick B, Martin R C, Mellor S, Schwabe K, Sutherland J and Thomas D Manifesto for Agile Software Development http://agilemanifesto.org
[34] Schwabe K and Beedle M 2001 Agile Software Development with Scrum (Upper Saddle River: Prentice Hall PTR)
[35] Poppendieck M, Poppendieck T D and Poppendieck T 2003 Lean Software Development: An Agile Toolkit (Boston: Addison-Wesley Professional)
[36] Google Test https://github.com/google/googletest
[37] Jenkins https://jenkins.io
[38] Beck K 2002 Test Driven Development: By Example (Boston: Addison-Wesley Professional)
[39] Fowler M 1999 Refactoring: Improving the Design of Existing Code (Boston: Addison-Wesley)
[40] Feathers M 2004 Working Effectively with Legacy Code (Upper Saddle River: Prentice Hall PTR)