Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Performance Engineering Concepts and Software Engineering Concepts for HPC PeCoH Deliverable D2.1

Nathanael Hübbe and Sandra Schröder

Work Package:	WP2
Responsible Institution:	Universität Hamburg
Date of Submission:	October 2019

Contents

Chapter 1

Introduction

Most scientists write their own scientific software in order to produce research results. However, the majority of them are not formally trained in performance and software engineering, since the main motivation of scientists is to perform science and not to care about writing efficient software in a performance and software engineering sense. This often results in decreased productivity in the whole process.

The following paragraph describes Task 2.1 as given in the project proposal:

In this task, diverse concepts from the software engineering domain in general and previous research projects (e.g., SPP 1648) are identified. These concepts will be evaluated according their suitability for the HPC community and their applicability to the scientific application software in each domain. These concepts can include but are not limited to: 1) performance patterns and antipatterns, 2) efficient algorithms and data structures, 3) optimized data exchange technologies for software component interfaces, 4) software architectures optimized for performance, 5) migration strategies towards improved software architectures, 6) performance-aware deployment strategies.

In this report, concrete performance engineering and software engineering concepts are collected that can help scientists to improve the development of scientific software. Additionally, these concepts are evaluated against selected criteria in order to show the benefits of those concepts when applied in scientific programming.

Chapter 2

Performance Engineering Concepts

In the following, selected performance engineering concepts are listed in brief that could help scientists to improve the performance of their software. Additionally, the advantages and disadvantages of the concepts are discussed. In the subsequent section, the benefits of these concepts are qualitatively assessed based on specified criteria.

2.1 Concepts

→ **In-situ methods**
The analysis of data happens at the same time as the data creation, i.e., during the run of a simulation. This means data does not need to be stored.

Advantages:

- reduction of storage requirements
- quick feedback from running process
- supports interactive data exploration and analysis

Disadvantages:

- change of analysis requires rerunning of the data generation
- significant implementation effort

→ **Big data/statistical methods**
Tuning of optimization parameters using statistical information to drive decision processes.

Advantages:

- may deliver better results than hand-tuning or without tuning
- application may be transparent
- may provide insights into system behavior

Disadvantages:

- significant implementation effort
- signal selection is crucial for success

  - possibly bad performance when encountering new software

→ **Machine Learning/AI methods**
Tuning of optimization parameters using AI algorithms like neural networks.

Advantages:

- may deliver better results than hand-tuning or no tuning
- application may be transparent
- likely more robust against new software than statistical methods

Disadvantages:

- significant implementation effort
- signal selection and algorithm design is crucial for success
- algorithm is black box that does not help understanding the system

→ **Domain Specific Languages (DSL)**
Creation of dialects of programming languages designed to support programming within a specific domain.

Advantages:

- boilerplate code that is common within the target domain can be abstracted away
- implementation of language constructs may depend on hardware, and exploit its potential without source code adaption

Disadvantages:

- performance problems become compiler problems
- abstraction means that understanding of performance problems becomes harder
- focus on single domain may unduly constrain applicability, hindering progress in science

→ **Compiler auto parallelization**
Methods of letting the compiler compile a sequential program into a parallelized executable.

Advantages:

- easy use of available hardware
- may significantly increase the amount of parallel programs run on supercomputers
- may provide backends for thread/process/hybrid parallelization

Disadvantages:

- need to perform full parallelization, otherwise Amdahl's law strikes
- parallelization may be very inferior to carefully hand parallelized code
- parallelization of I/O heavy programs may be counterproductive

→ **Compiler auto parallelization for GPUs**
Methods of letting the compiler generate code for GPU acceleration.

Advantages:

- likely easier to implement than auto parallelization
- easy use of available hardware
- may significantly increase GPU usage in supercomputers

Disadvantages:

- generated code likely inferior to hand-written GPU code
- it is virtually impossible for the compiler to know which parts will profit from GPU offloading
- may lead to programs requiring GPU that are I/O bound

→ **Performance warnings and feedback**
Methods of providing feedback on performance issues, preferably directly from the compiler. For instance, the compiler can run analyses that distinguish memory bound loops from computation bound loops. If applicable to the loop, the compiler may also output the chains of dependent operations that are likely to limit the performance.

Advantages:

- the produced output may be relatively easy to turn into significant performance gains by the programmer
- the produced output may help sharpen the understanding of the CPU by the programmer
- can be implemented incrementally

Disadvantages:

- requires the user to care about performance
- depends on an accurate CPU model
- the amount of achievable speedup is generally limited
- may easily drown the user in information

2.2 Dimensions for Assessment

The following areas which are influenced by the use of the above concepts are used for the assessment:

→ **Investment Costs**
Any development that needs to be done to create the tools that are independent of the concrete application of the method. For a DSL, this would include the definition and standardization of the DSL, along with the implementation of a compiler and necessary runtime libraries.

→ **Application Costs**
Any development that needs to be done on an application by application basis. For a DSL, this would be the effort of actually using DSL constructs in an application.

→ **CPU Usage**
The total amount of CPU time that is used for an application. This includes compilation, computation, idling while waiting for resources (I/O, network), and any other overhead that blocks a CPU.

→ **Energy Consumption**
The total amount of energy that is required to produce a result. This is clearly dominated by CPU usage, but other parts of the system must also be considered when they become relevant.

→ **Storage Space**
The integral of data size on permanent storage over time.

→ **Time To Results**
The wall-clock time difference between the formulation of the problem, and the actual display of actionable results. This time span includes any human development effort needed to tackle the problem at hand, as well as all times spent aggregating data, computing data, storing data, post-processing data, and visualizing data.

→ **Programmability**
The ease of writing software that makes use of the concept.

→ **Maintainability**
The ease of maintaining software that makes use of the concept. Mainly this depends on code readability, along with the predictability of the effects of small changes that are made without understanding the whole system.

→ **Flexibility**
How broad is the field of natural applications of the concept? "Natural" means that the concept must fit the application well, and not require significant workarounds to make the application match the concept.

2.3 Qualitative Assessment

In this section, we try to qualitatively assess the different concepts by comparing them according to the criteria. The assessment uses the dimensions defined above on the scale --, -, 0, +, ++. For consistency, + and ++ always mean better, while - and -- always mean worse.

For example, writing a fully featured compiler is a major investment cost, which calls for a -- because it is a cost. Likewise, a reduction of CPU time to 42% calls for a + assessment, because faster is better.

This assessment is very rough and qualitative. A zero means little impact, the factor of effect should be less than two if applicable. For investments, this should be less than a person year. A + or - means significant impact, the factor of effect should be between two and ten if applicable. For investments, this should be between a person year and five person years. A ++ or -- means major impact, the factor of effect should be larger than ten. For investments, this should be more than five person years.

Table 2.1 gives a quick overview of our assessments, the following list will detail our arguments for each assessment.

	Investment Costs	Application Costs	CPU Usage	Energy Consumption	Storage Space	Time To Results	Programmability	Maintainability	Flexibility
in-situ methods	0	--	-- … +	-- … +	++	-- … +	0	0	-
statistical methods	-	0	+	+	0	0	0	0	0
AI methods	-	0	0	0	0	0	0	0	0
DSL	-- … -	+	- … +	- … +	0	+	+	+	-
auto parallelization	--	+	-- … 0	-- … 0	0	0 … ++	0	0	+
auto GPU acceleration	--	+	+	+	0	0 … ++	0	0	+
performance warnings	-	-	+	+	0	0	0	-	0

Table 2.1: Overview of our qualitative assessments of the different concepts for improving performance of scientific applications.

→ **In-situ methods**

0 Investment Costs
In-situ methods are a concept that requires rewriting each application in a way that allows full processing of the data where it happens to be stored or generated. This per-application redesign cannot be outsourced to general tools. Thus, there is little general investment that can be done to reduce the workload of redesigning the applications.

-- Application Costs
Each application that is to use in-situ methods must have the relevant parts redesigned and rewritten to implement them. This is a major change for many applications, as it touches an applications' entire I/O behavior.

-- … + CPU Usage
In-situ methods avoid storing results, and thus idling of the CPU while it's waiting for I/O. However, this comes at the price of not having the intermediate data stored for other types of analysis. Thus, the data may need to be recomputed several times to achieve the same analysis results, especially when some methods of data analysis are prompted by the results of more general types of analysis. Careful planning of the analysis may reduce this trade-off, but the fact remains that recomputation may become necessary where non-in-situ workflows would just reuse intermediate data. Thus CPU usage may see a major increase (two recomputations or more) or decrease significantly (single pass with avoided I/O time), or anything in between.

-- … + Energy Consumption
This largely is the same as the effect on CPU usage, however the savings may be larger as the storage space is used less. This may allow using less spinning disks, and thus further reduce the energy consumption. Nevertheless, if results have to be recalculated repeatedly, all the savings are gone, and a large negative impact remains.

++ Storage Space
Avoiding the need to store intermediate data may provide a major decrease in disk storage usage.

-- ... + Time To Results
The ability to analyse and/or visualize data right when it is produced may significantly decrease the time until the scientist sees first results of a simulation run. On the other hand, running a second analysis based on intermediate data requires restarting the entire simulation to reproduce the intermediate data. This may add a major delay until the results are visible.

0 Programmability
On-line analysis is not inherently easier or harder to program than analysis on stored data. It's "simply" replacing I/O with abstractions to facilitate immediate analysis of results.

0 Maintainability
As the code is not inherently more or less readable, so is maintenance not inherently easier or harder.

- Flexibility
In-situ methods may loose the flexibility of running any analysis post-mortem on the produced data.

→ **Statistical methods**

- Investment Costs
Statistical methods require a significant investment into creating the software that analyzes the performance data, and which adjusts tuning parameters accordingly. This should be a one-time investment which produces payoffs for any application.

0 Application Costs
As statistical tuning methods are expected to work system-wide, little or no adjustment of application code should be necessary.

+ CPU Usage
The goal of statistical methods is to increase the efficiency of resource usage, i.e., either CPU time and/or idling while processes are waiting for other operations (I/O or communication) to complete.

+ Energy Consumption
This follows the CPU usage.

0 Storage Space
Unless they find use within data compressors, statistical methods have no impact on the usage of storage space.

0 Time To Results
Time to results should improve along with CPU usage reduction. However, time to results includes more than plain CPU usage (like development and queue times), so the impact is not as significant as for the CPU usage.

0 Programmability
As application codes don't change, no impact on their programmability is expected.

0 Maintainability
As application codes don’t change, no impact on their maintainability is expected.

0 Flexibility
As application codes don’t change, no impact on their flexibility is expected.

→ **AI methods**

- Investment Costs
AI methods require a significant investment into creating the software that analyzes the performance data, and which adjusts tuning parameters accordingly. This should be a one-time investment which produces payoffs for any application.

0 Application Costs
As AI tuning methods are expected to work system-wide, little or no adjustment of application code should be necessary.

0 CPU Usage
The goal of AI methods is to increase the efficiency of resource usage, i.e., either CPU time and/or idling while processes are waiting for other operations (I/O or communication) to complete. However, at the same time AI methods themselves are computationally expensive, both for training and application of the neural networks, reducing the achievable impact. This is why we do not see a significant improvement for CPU usage.

0 Energy Consumption
This follows the CPU usage.

0 Storage Space
Unless they find use within data compressors, AI methods have no impact on the usage of storage space.

0 Time To Results
Time to results should improve along with CPU usage reduction. However, time to results includes more than plain CPU usage (like development and queue times), so the impact is not as significant as for the CPU usage.

0 Programmability
As application codes don’t change, no impact on their programmability is expected.

0 Maintainability
As application codes don’t change, no impact on their maintainability is expected.

0 Flexibility
As application codes don’t change, no impact on their flexibility is expected.

→ **DSL**

-- ... - Investment Costs
Use of a Domain Specific Language requires the development of a suitable compiler for it first. Obviously, if the DSL is very similar to an established language for which free (FOSS) compilers are available, the required effort may be relatively small. However, the more domain specific features are included,

and the more complex these features get, the greater will be the required effort for the compiler.

- + Application Costs
  The aim of a DSL is to make the development of the application code easier.
- -…+ CPU Usage
  Depending on the goal of the DSL, CPU usage may either increase or decrease due to the use of a DSL. It may increase if the DSL is only geared towards usability from a programmers perspective, introducing costly code into the executable. If the DSL provides garbage collection, for instance, the resulting memory allocating code will require much more CPU time compared to the same program written in C. Providing garbage collection is a sensible feature to want from a programmers perspective, as it does preclude important classes of bugs, but it has a significant run time cost. On the other hand, a DSL may choose to provide abstractions that allow it to generate more efficient code. Such a DSL would help reducing the CPU usage of the applications.
- -…+ Energy Consumption
  This follows the CPU usage.
- 0 Storage Space
  DSLs are generally not expected to change the storage of results.
- + Time To Results
  The declared goal of a DSL is to provide abstractions to the programmer that are closer to the problem domain that they work for. This added ease should provide for faster development.
- + Programmability
  Abstractions that are a better/closer fit to the problem domain should provide for easier, quicker programming with more concise resulting code.
- + Maintainability
  Abstractions that are a better/closer fit to the problem domain should provide for more concise and readable resulting code.
- - Flexibility
  The abstractions of a DSL will have the effect of a vendor lock-in. Once an application uses the abstractions provided by a specific DSL, it will become hard to port the application to another language. The stronger the abstractions of the DSL, the more thorough the lock-in becomes. Thus, an application that makes heavy use of a high-level DSL will be chained to the fate of the DSL forever. If the DSL does not support a platform, so won't the application. If the DSL is not maintained to provide top performance on current machines, the application won't show top performance.

$\rightarrow$ **Compiler Auto Parallelization**

- -- Investment Costs
  Auto parallelization by the compiler requires writing a compiler that can do it. Unfortunately, this compiler needs to be able to perform auto parallelization on virtually every part of the application. If the resulting executable would only be partially parallelized, Amdahl's law would strike and immediately limit the maximum parallelization. (A 10x speedup through parallelization requires parallelization of more than 90% of the application's computations, and to put

a single 36 core server to good use, no more than 2% of the code may be left sequential.) As such, there is simply no way of implementing this incrementally: Either the compiler can do parallelization of virtually everything, or it will be useless on modern supercomputers. The consequence is, that the full compiler development has to be paid for up-front.

+ Application Costs
  The upshot of automatic parallelization is, that application codes may be left unchanged and still profit from parallelism.

-- ... 0 CPU Usage
  Any parallelization incurs overheads. Automatic parallelization is no exception, so the total CPU time consumption will go up.

-- ... 0 Energy Consumption
  This follows the CPU usage.

0 Storage Space
  Auto parallelization is not expected to change data storage formats on disk.

0 ... ++ Time To Results
  This is, what any parallelization aims to improve. Nevertheless, success depends heavily on what the compiler can do for the given code, and is not guaranteed. A central data structure that happens to be partitioned in a sub-optimal way can kill all the performance gains.

0 Programmability
  As application codes are not changed from their sequential versions, auto parallelization has no impact on programmability.

0 Maintainability
  As application codes are not changed from their sequential versions, auto parallelization has no impact on maintainability.

+ Flexibility
  As auto parallelization only adds a compilation mode, it adds the flexibility to run sequentially written applications in a parallel fashion.

→ **Compiler Auto Parallelization for GPUs**

-- Investment Costs
  Like auto parallelization, auto GPU acceleration requires a massive investment into the compiler infrastructure. Unlike parallelization, GPU acceleration can provide gains without the need to be able to handle all codes equally well. However, the auto GPU acceleration needs to be good enough to justify running the resulting executable on a machine with a GPU. Thus, the requirement to provide comprehensive transformation of the code is less strict, but still very significant.

+ Application Costs
  As the application does not need to change, its developments costs stay the same.

+ CPU Usage
  Any work that is offloaded to the GPU does not need to be done by a CPU. Nevertheless, GPU accelerated processes still need a CPU, even when that is only idling while it is waiting for the GPU to do its work.

+ Energy Consumption
  With GPU acceleration, work is moved from CPUs to GPUs. Since GPUs are typically much more energy efficient than CPUs, this move should result in a significant reduction of energy consumption. The effect is smaller than the reduction of the CPU usage, as the GPU also needs power.

0 Storage Space
  GPU acceleration has nothing to do with storage formats.

0 … ++ Time To Results
  This is one of the goals of GPU acceleration, to decrease the time to results. This works especially well for visualizations, providing massive benefits, and can work well for scientific codes as well. The question remains, how well compiler based auto acceleration can utilize the GPU. If it fails to use it well, the result may still be that the application runs faster without acceleration.

0 Programmability
  No change of source code, so no change in programmability.

0 Maintainability
  No change of source code, so no change in maintainability.

+ Flexibility
  Auto GPU accelerations adds an option to the compilation, providing the flexibility to run efficiently on hardware with accelerators.

→ **Performance Warnings**

- Investment Costs
  Like other compiler based concepts, performance warnings require an upfront investment into compiler development. However, they can be implemented incrementally, providing one feature after another, each one providing benefits for code performance.

- Application Costs
  Since performance warnings are not directly changing the code translation, but rather give feedback to the developer on how to best improve the performance of their code, the application development itself is burdened with actually implementing improvements.

+ CPU Usage
  The aim of performance warnings is to help developers use the available computing resources more efficiently. However, the gain that can be achieved is typically limited, as the suggestions will be more of the micro-optimization kind than of the large scale algorithmic-optimization kind.

+ Energy Consumption
  This follows the CPU Usage.

0 Storage Space
  Performance warnings have nothing to do with storage formats.

0 Time To Results
  While applications may run faster due to the optimizations suggested by performance warnings, these gains are set off by the need to implement those optimizations first. Thus, we expect the gains not to be significant.

0 Programmability
  Performance warnings only add a feedback to facilitate better performing programs, they do not change the way that an application is designed or implemented.

- Maintainability
  However, the micro-optimizations suggested by performance warnings may decrease the readability of the finished code, hurting its maintainability.

0 Flexibility
  Performance warnings do not change the way the compiler generates the executable from its source, so no impact on flexibility is expected.

Chapter 3

Software Engineering Concepts

In an iterative process, we collected software engineering concepts that could help scientists in writing efficient software and to improve their software engineering processes. In the following, the categories summarizing the collected software engineering concepts are listed and described. This section refers to the first step which was performed as described in Task 2.1.

3.1 Concepts

→ **Programming Concepts for HPC**
Scientists need to understand algorithms and data structures especially with respect to parallelization independent of whether implementing for shared or message passing systems in order to write efficient scientific software. This includes to understand programming languages that are widely used in HPC.

→ **Programming Best Practices for HPC**
This category summarizes software development best practices that help scientists to develop high-quality scientific software. This includes knowledge about Integrated Development Environments (IDEs), debugging, programming idioms, logging concepts, and exception handling.

→ **Software Configuration Management**
Version control and change management in scientific development processes is as important as in traditional software engineering processes. Scientific applications are characterized by frequent changes due to improvements and modifications in mathematical models. That is why we have included this category for summarizing software configuration management concepts. This category encompasses basic terminologies and concepts of version control, issue and bug tracking, release management, and deployment management.

→ **Agile Software Development**
Scientific computing can benefit from agile development practices, since scientific research and the development of scientific software have similarities with processes following the agile principles, namely responsiveness to change and collaboration. This category suggests to include test-driven development and agile testing, extreme programming, and SCRUM to the portfolio of a scientists who write scientific software.

→ **Software Quality**
Scientific software is often written by a team of scientists. Additionally, scientific software is reused and modified by other teams (maybe from other projects). This means that the source code needs to be readable, reusable, and testable, i.e., the source code needs to be of high quality. This category summarizes concepts related to software quality including coding standards, code quality, refactoring, and code reviews.

→ **Software Design and Software Architecture**
Scientists often neglect to consider the software on a higher level of abstraction, i.e., software architecture and design. This category therefore includes concepts that should help scientists to understand the importance and impact of software architecture during software development. Additionally, this category includes basic knowledge about systematic approaches in order to appropriately collect and analyze requirements for the application and in order to develop an appropriate software architecture based on these requirements. Following a systematic approach helps the user to significantly improve quality of scientific software.

→ **Documentation**
This category summarizes documentation practices for all necessary phases in the development process, e.g., requirements, software architecture, source code, and user documentation. Documentation is an important artifact (and activity) for enabling reproducibility of scientific results.

3.2 Suitability and Benefits of Software Engineering Concepts for the HPC community

In this section, the suitability of the previous described software engineering concepts in the context of HPC is described (second step of Task 2.1). For this, we define and describe common requirements that are characteristic for software development in HPC. For these requirements, we qualitatively assess the suitability and benefits of the selected concepts. The assessment is supported by resources found in literature that reports about experiences applying specific software engineering techniques in the context of scientific computing.

The overall goal is to increase the productivity of scientists in developing scientific software. This goal is further divided into the following requirements:

→ **Credibility of Research Results**
Verification and validation plays an important role in scientific computing. Extensive checks of the scientific model and the code implementing the model are necessary to ensure the correctness of the source code and, consequently, the credibility of research results. That is why, scientific programmers should be aware of testing methods, among others, that help to detect crucial faults and errors in their code. Logging can help to record and to subsequently analyze faulty behavior in the software system. Exception handling allows to recover software from unforeseen situations.

supported by: Test-driven Development and Agile Testing, code review, documentation, Version Control, debugging, Logging, exception handling, issue and bug tracking

evidence from literature: [KB18], [CR11], [FPG$^+$11], [KTH11], [Roy09], [KS08], [HC15]

→ **Reproducibility of Research Results**
Reproducibility is an essential requirement in the scientific process [IT18]. A result is said to be reproducible if another researcher can take the original code and input data, execute it, and re-obtain the same result [PDZ06]. Sufficient details of a computation must be available, so that the results can be verified, reproduced, and extended by other researchers. Reproducibility also allows researchers to build upon the work of another research.

Reproducibility 1) requires a program works the way it should, 2) knowing the input conditions and system used to produce the results, and 3) recognizing and tracking program bugs.

The practices *Test-driven Development and Agile Testing* supports to ensure that the program works according to the specified requirements, e.g., produces the expected results.

Documentation greatly helps to ensure the reproducibility of the results. For example, researchers should document the libraries, compilers, and runtime systems that must be installed and configured in order to run the experiment. Literate programming – as implemented in the Jupyter Project[1] – combines documentation, code, data, mathematical equations, plots, and rich media in notebooks that can be shared by researchers to enable reproducibility.

Release management and version control – as additional exemplary methods – also supports reproducibility. The purpose of release management is to ensure that a consistent method of deployment is followed whereas this process is ideally documented and supported by automatic processes that can be reproduced easily. It ensures that only tested and accepted versions of hardware and software are installed.

supported by: Test-driven Development and Agile Testing, software configuration management, code review, documentation

evidence from literature: [IT18], [BDFR06], [HC15]

→ **Reusability of Source Code**
Scientific software must be adapted for different execution environments, problem sets, and available resources to ensure its efficiency and reliability. Normally, scientific code is not developed to be reused by other researchers which makes the code hard to reuse and to adapt. That is why scientific programmers need to manually implement solutions that are again hard to maintain and reuse.

Best practices and principles, e.g., SOLID, from object-oriented programming can support programmers to write reusable code. It may help programmers to avoid

[1]https://jupyter.org/

bad practices like code duplication and to use code structures that promote reusable, modular, and extendable code structures.

Documentation, e.g., in form of code comments can help to reflect the reasons of the decisions that were made during programming. Valueable information about how a given parameter was chosen, why a specific library was used (and not another one), can be captured in code comments. For this, the programmer can make use of code comment style guides for a uniform way of documenting the code.

supported by: Software quality, Documentation, Release Management, Software Design and Software Architecture

evidence from literature: [KTVR11], [FHHS16], [BDFR06], [HC15]

Chapter 4

Summary

In this report, we have outlined a collection of performance engineering and software engineering concepts. With this collection we aim for supporting scientists in improving the performance of their scientific software and increasing the productivity of the software development. We have assessed each concept according to selected criteria and have shown qualitatively that these concepts can greatly support the above mentioned aspects.

This collection is not complete. Nevertheless, it represents a good starting point for a further refinement and identification of performance engineering and software engineering concepts. In the deliverable D2.2, we report on our experiences on applying a selected set of concepts on a real-world project. The success story described in D2.2 shows that scientists can already benefit from integrating simple software engineering techniques, e.g., adhering to coding styles or refactoring.

Acknowledgement

The PeCoH project has received funding from the German Research Foundation (DFG) under grants LU 1353/12-1, OL 241/2-1, and RI 1068/7-1.

Bibliography

[BDFR06] Susan M Baxter, Steven W Day, Jacquelyn S Fetrow, and Stephanie J Reisinger. Scientific software development is not an oxymoron. *PLOS Computational Biology*, 2(9):1–4, 09 2006.

[CR11] Thomas Clune and Richard Rood. Software testing and verification in climate model development. *IEEE software*, 28(6):49–55, 2011.

[FHHS16] Jörg Fehr, Jan Heiland, Christian Himpe, and Jens Saak. Best practices for replicability, reproducibility and reusability of computer-based experiments exemplified by model reduction software. *CoRR*, abs/1607.01191, 2016.

[FPG$^+$11] P. Farrell, Matthew Piggott, G. Gorman, D. Ham, Cian Wilson, and T. Bond. Automated continuous verification for numerical simulation. *Geoscientific Model Development*, 4, 05 2011.

[HC15] Dustin Heaton and Jeffrey C. Carver. Claims about the use of software engineering practices in science: A systematic literature review. *Information and Software Technology*, 67:207 – 219, 2015.

[IT18] Peter Ivie and Douglas Thain. Reproducibility in scientific computing. *ACM Comput. Surv.*, 51(3):63:1–63:36, July 2018.

[KB18] Upulee Kanewala and James M. Bieman. Testing scientific software: A systematic literature review. *CoRR*, abs/1804.01954, 2018.

[KS08] Diane Kelly and Rebecca Sanders. The challenge of testing scientific software. In *Proceedings of the 3rd annual conference of the Association for Software Testing (CAST 2008: Beyond the Boundaries)*, pages 30–36. Citeseer, 2008.

[KTH11] D. Kelly, S. Thorsteinson, and D. Hook. Scientific software testing: Analysis with four dimensions. *IEEE Software*, 28(3):84–90, May 2011.

[KTVR11] Pilsung Kang, Eli Tilevich, Srinidhi Varadarajan, and Naren Ramakrishnan. Maintainable and reusable scientific software adaptation: Democratizing scientific software adaptation. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 165–176, New York, NY, USA, 2011. ACM.

[PDZ06] Roger D Peng, Francesca Dominici, and Scott L Zeger. Reproducible epidemiologic research. *American journal of epidemiology*, 163(9):783–789, 2006.

[Roy09] Christopher Roy. Practical software engineering strategies for scientific computing. *19th AIAA Computational Fluid Dynamics Conference*, 06 2009.