Reproducible Experiments in Parallel Computing: Concepts and Stencil Compiler Benchmark Study

Danilo Guerrera, Helmar Burkhart, and Antonio Maffia

University of Basel, Switzerland
{danilo.guerrera,helmar.burkhart,antonio.maffia}@unibas.ch

Abstract. For decades, the majority of the experiments on parallel computers have been reported at conferences and in journals usually without the possibility to verify the results presented. Thus, one of the major principles of science, reproducible results as a kind of correctness proof, has been neglected in the field of experimental high-performance computing. While this is still the state-of-the-art, current research targets for solutions to this problem. We discuss early results regarding reproducibility from a benchmark case study we did. In our experiments we explore the class of stencil calculations that are part of many scientific kernels and compare the performance results of four stencil compilers. In order to make these experiments reproducible from remote, a first prototype of an replication engine has been developed that can be accessed via the internet.

1 Introduction

Whenever you read a paper that reports on computational experiments, immediate questions such as the following arise:

- If we could rerun the experiment: will we get the same, similar or different results? What if we run the experiment on a different compute environment?
- Results are often for a specific problem only. What if we define a slightly different test case?
- Libraries and software components influence measurements. For instance, have compilation flags been properly set?

One would reach another level of trust in scientific results if we could somehow reproduce experiments. Such trust problems have already been reported in the pharmaceutical industry, in finance, and other fields. "It is impossible to believe most of the computational results presented at conferences and in published papers today. Even mature branches of science, despite all their efforts, suffer severely from the problem of errors in final published conclusions" [1]. It is therefore crucial to be able to test results for science to be self-correcting. "The ability to reuse and extend the results enable science to move forward" [2].

The difficulty in reproducing computational research is in large part caused by the difficulty in capturing every last detail of the software and computing

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part I, LNCS 8805, pp. 464–474, 2014.

environment, which is what is needed to achieve reliable replication [3]. As can be found, articles often do not have a sufficiently detailed description of their experiments, and do not make available the software used to obtain the results claimed. As a consequence, parallel computational results are most often impossible to reproduce, often questionable, and therefore of little or no scientific value [4].

The problem has been detected and early results have been reported. Dolfi et al. [5] propose a model for reproducible papers such that "the current manuscript already contains sufficient details, codes, and scripts to reproduce all the presented numerical results and figures". A constellation of tools and prototypes is available in order not only to document the workflow of an application but also make it reproducible. Taverna [6] and Vistrails [7] integrate data acquisition, derivation, analysis, and visualization as executable components throughout the scientific exploration process. Repeatability is facilitated by ensuring that the evolution of the software components used in the workflow is controlled by the organizations that design the workflows. This "controlled services" approach is shared by other WFMS such as Kepler [8] and Knime [9], an open-source workflow-based integration platform for data analytics.

In this paper, we explore the reproducibility of benchmark experiments and demonstrate early results of a project with the goal of building an ecosystem for reproducible computational experiments. In Section 2 we define a taxonomy which is the basis for the design of our system. Section 3 describes the functionality of our system envisaged and an early prototype of a workflow engine. In Section 4 we present a case study by introducing the stencil motif and the performance results of different compilers. Conclusions and sketches of further work are given in Section 5.

2 Taxonomy for Reproducible Benchmark Experiments

2.1 Space of Computational Experiments

Computational problem solving in general can be described as follows: A computational **problem** is solved by an algorithmic **method** on a compute **system**. We call the triple (Problem / Method / System) a *micro-experiment*, which can be considered as being one point in the space of experiments (see Figure 1).

- ***Problem***: Solve a (random) dense system of linear equations in IEEE double precision arithmetic.
- ***Method***: Two-dimensional block-cyclic data distribution. Right-looking variant of the LU factorization with row partial pivoting (see [10]).
- ***System***: Distributed-memory computer with Message Passing Interface (MPI 1.1 compliant) and Basic Linear Algebra Subprograms (BLAS) installed.

What we usually want is a comparison between data resulting from more than a single micro-experiment. Keeping two out of the three dimensions fixed, we get an experiment which is a function of the third one: the red line shown in Figure

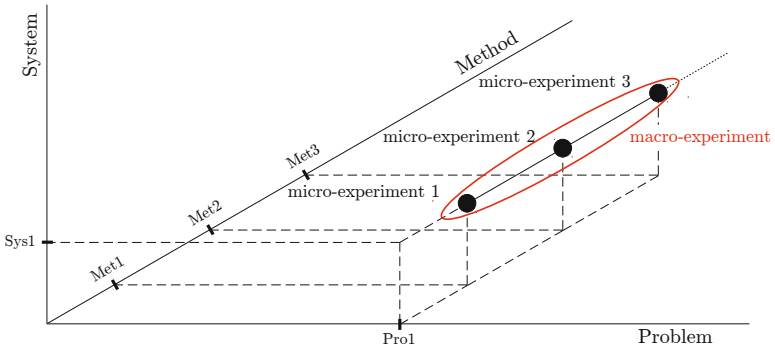


Fig. 1. Space of Computational Experiments

1 identifies such a *macro-experiment*, which is a collection of micro-experiments (e.g. the black dots in Figure 1). Macro-experiments can be categorised as being either system-oriented, method-oriented, or problem-oriented.

2.2 Replication, Recomputation, and Reproduction

Given the taxonomy of above, we obtain different levels of repeatability:

- ***Replication***: *Basic replication* means re-running the original micro- or macro-experiment with all values in the experiment space kept fixed. As a result, execution properties that have been claimed become verifiable, which guarantees a high-level of credibility. *Advanced replication* mechanisms even support the changing of certain problem parameters while keeping method and system dimensions fixed. One of the major difficulties for providing replication support are security concerns because the system needs to be accessible from outside. In addition, the workload in high-performance computing scenarios is often significant, which forces the setting of limits for the number of replication calls.
- ***Recomputation***: Portable methods are usually considered as being a good quality criterion for computational experiments. Therefore, methods used to solve a particular problem should not be bound to a specific compute environment. In the context of high-performance computing where parallel systems are used, this movement is however restricted to related machine models.
- ***Reproduction***: Computational experiments are only an aid for getting actual scientific results. If different methods end up with the same scientific results in terms of computed output, some kind of experimental proof of the scientific insight is given.

3 Towards BETSc: Basel Ecosystem for Trusted Scientific Computing

3.1 Design Issues

In addition to the taxonomy described in the previous section, we identified some features we would like to achieve. The major problem when talking about reproducibility of experiments is that people other than the original researcher have to deal with the configuration of their own environment, which likely differs from the one used in the original test: we therefore target for **dependency check** (which must represent the first step of an experiment) and **portability**. A scientific discovery process may require the application of several steps and activities and it is necessary to trace and collect sufficient provenance information over this process, thus achieving **provenance support**. Allowing to **automatically run** tests, makes it possible to generate **documentation** regarding the environment in which the experiment was executed and store it as a unique identifier. What scientists aim for are results: since they are so important, it is necessary to be sure about their correctness, therefore introducing a **correctness check** at the end of the experiment's execution, and only at this point **visualize** the outputs.

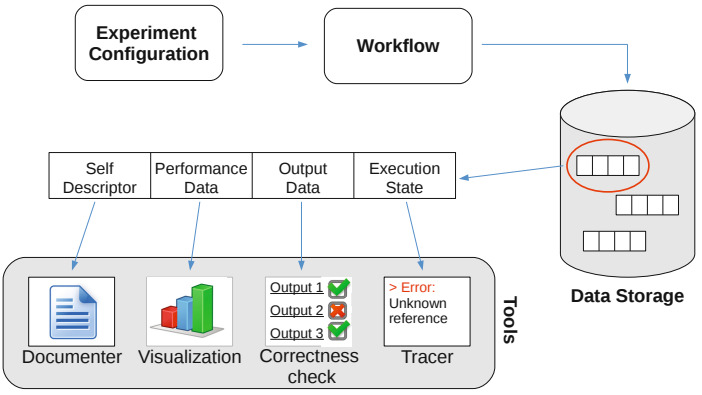


Fig. 2. Experiment flow

The key idea is that, once an experiment is characterized (i.e. the problem to be explored is stated), a user can define the macro-experiments to be executed, and pass such information to the workflow engine. The workflow sets up the environment and produces a structured output. Such an output was designed in order to allow flexibility in treating the data. As shown in Figure 2, it is composed of:

– **self-descriptor**: this field stores the information which uniquely defines an experiment. An environment stamp with libraries and dependencies is created, then all of the experiment-dependent parameters are added, together with information about hardware and threads used.

- **performance data**: each micro-experiment generates data, which are used to set up a comparison based on different performance metrics (e.g. GFlop-s/s, speedup, etc.).
- **output data**: these are simply the numerical results produced by each micro-experiment. Later on, they can be used to verify the correctness of the execution.
- **execution state**: at the end of a micro-experiment, potential fails can be detected thanks to log data generated at run time and stored in this field.

In order to analyze or visualize the information stored in the data structure described above, BETSc will provide a tool suite:

- **Documenter:** provides information about the experiments, in particular the settings as defined in the configuration phase;
- **Visualizer:** provides graphs regarding the performance output of a macro-experiment (stored in the field Performance Data of the structured output);
- **Correctness Checker:** verifies the correctness of an execution. Numerical simulation outputs are not suitable for checking bit-wise correctness due to their nature, and this is even worse when talking about numerical results coming out from a parallel computation, where not only the sequence of operations on the data can change but also the core where the operations are executed;
- **Tracer:** provides information on the state of an execution, showing what possibly went wrong and why.

3.2 Replication Prototype

We developed a prototype of a workflow engine [11]. Its first task is an automatic installation of all of the components required for running the experiments (see Figure 3): this is done with the command *install*. In order to run a test, a test suite needs to be created: this means both adding a method and a problem. A working example (fully implemented methods) is provided along with the workflow: at the current stage new methods and problems must be added manually by the user. Under development are the commands to do it automatically: adding a new method will be done by the command *add_method*, which generates a directory structure for the method passed as argument to it. A skeleton of *Makefile* will be provided in the newly created directory structure: it has to be modified by the user according to the peculiarity of the method. The same has to be done for new problems: after a problem has been added, the user has to provide the implementations for it, according to the methods he is intending to test.

After this configuration, it is possible to automatically execute all of the specified micro-experiments using the command *run* followed by a specification of the methods you want to use. Problem specific parameters as well as particular system settings (e.g. number of threads to be used) are passed through the *SetEnv* file. The workflow manages the execution of such experiments, setting the parameters, actually running the test, gathering the results, and sending

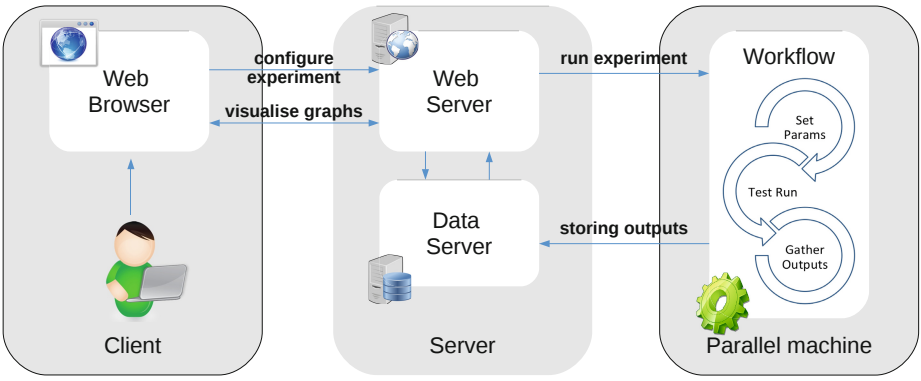


Fig. 3. Workflow architecture

them to the data server for storing. It is then possible to visualize different kinds of graphs regarding the executed experiments, starting from the fields of the structured data stored (as defined above, see Figure 2).

4 Case Study: Stencil Compiler Comparisons

4.1 Stencil Motif Background

In his 2004 lecture titled *Defining Software Requirements for Scientific Computing*, Phillip Colella identified *seven so-called dwarfs*, which are defined as algorithmic methods that capture a reusable pattern of computation and communication. The dwarf idea was later taken up by scientists at Berkeley and the list of dwarfs (renamed to motifs) was extended [12]. In this list, stencil computations are present as motif *Structured Grid*. It defines operations on a multi-dimensional grid, which are repeatedly applied such that the value of a grid point depends on the value of the point itself and the values of neighbours in a previous time step. The stencil motif has manyfold applications in science and engineering such as weather forecast, geophysics, computational fluid dynamics, and image processing.

Stencil computations expose a high degree of parallelism, however performance is not for free. They are memory-bound as typically only a limited amount of computation is performed per grid point (i.e. low arithmetic intensity). Because of this memory bandwidth limitation, different optimization strategies are possible. For example, if the application requires that the stencil has to be applied multiple times, there is potential to exploit temporal data locality, i.e. reuse cache data across iterations. A fair amount of research has addressed the question how temporal and spatial optimization can be done and what algorithmic changes and code transformations are needed. It is the purpose of our case study to experimentally explore different compilation methods.

4.2 Macro-experiment Definition

We set up and execute a macro-experiment using five different methods and compare the performance achieved.

Problem: We solve the classical wave equation $u_{tt} - c^2 \Delta u = 0$ with a fourth order-in-space and second order-in-time finite difference method. After discretizing with step sizes h in space and t in time, we obtain the following stencil computation formula:

$$u_{ijk}^{t+1} = 2u_{ijk}^{t} + u_{ijk}^{t-1} + \frac{\Delta t^2 c^2}{h^2}(-\frac{15}{2}u_{ijk}^{t} + \frac{4}{3}(u_{i\pm1,j,k}^{t} + u_{i,j\pm1,k}^{t} + u_{i,j,k\pm1}^{t}) - \frac{1}{12}(u_{i\pm2,j,k}^{t} + u_{i,j\pm2,k}^{t} + u_{i,j,k\pm2}^{t})).$$

We use a 3-dimensional grid of 200^3 points and calculate 100 timesteps.

Method: The following five methods are used in micro-experiments:

- **Directive-based approach** exploits a set of compiler directives that influence run-time behaviour. OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer.
- **Cache-oblivious algorithm** for stencil computations of Frigo and Strumpen [13]. An algorithm is cache-oblivious when it does not contain parameters (set at either compile-time or runtime) that can be tuned to optimize the cache complexity for the particular cache size and line length. POCHOIR[14], extends the cache-oblivious algorithm by using hyperspace cuts which improve parallelism. The compiler translates the embedded stencil code into Cilk code.
- **Auto-tuning** is the use of search to select the best performing code variant and parameter configuration from a set of possible versions. A compiler following this approach is HALIDE[15], which is specialized for image processing pipelines. It separates the algorithm from the scheduling task which uses an auto-tuning approach. LLVM is used for just-in-time compilation and parallelization is realized with pthreads.
- **Polyhedral model** [16] is a framework for automatic optimization and parallelization. It is applicable to loops with affine index functions and affine loop bounds, interprets the iteration space as a polyhedron, and loop transformations correspond to operations on or affine transformations of that polyhedron.PLUTO [17], is a source to source compiler that uses the polyhedral model approach for compiler optimization. The compiler uses C code as input and provides as output an OpenMP parallelized C code.
- **DSL + Auto-tuning** approach. Domain Specific Languages allow the programmer to express a stencil computation in a concise way independently of hardware architecture-specific details. PATUS [18], developed at University

of Basel, separates the specification of the stencil operation from strategy specifications (i.e. optimization and parallelization methods such as cache-blocking). Auto-tuning is used to find the best strategy and code generation-specific parameters.

Source code for all variants is available at [11].

System: We ran the benchmarks on an AMD Opteron 6274 CPU with a total number of 16 cores which has a 16 KB L1 cache, a 2 MB shared exclusive L2 cache and a 6MB shared L3 cache, with 2.2 GHz clock rate, running Ubuntu 12.04.4 LTS (kernel 3.8.0-38-generic).

The following C/C++ compiler were used: GNU gcc 4.6.3 and Intel icc 13.1.2.

4.3 Results

Stated system configuration and parameter setting, we want to obtain a comparison between the GFlops (calculated as product of *number of timestep* * *grid size* * *stencil floating point operations*, out of five executions) produced by the methods presented above. The result of such a comparison is shown in Figure 4. It is a visualization mash-up of 6 macro-experiments using different number of cores (1, 2, 4, 8, 16 and 32).

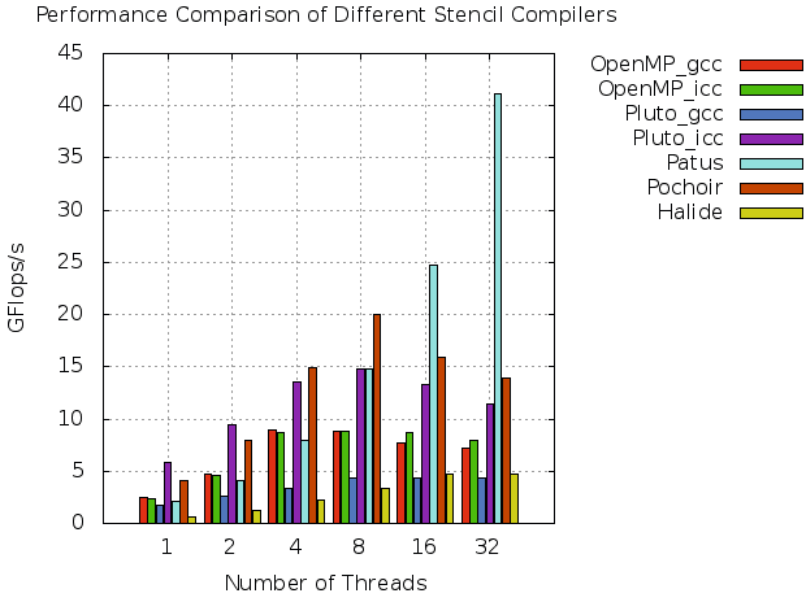
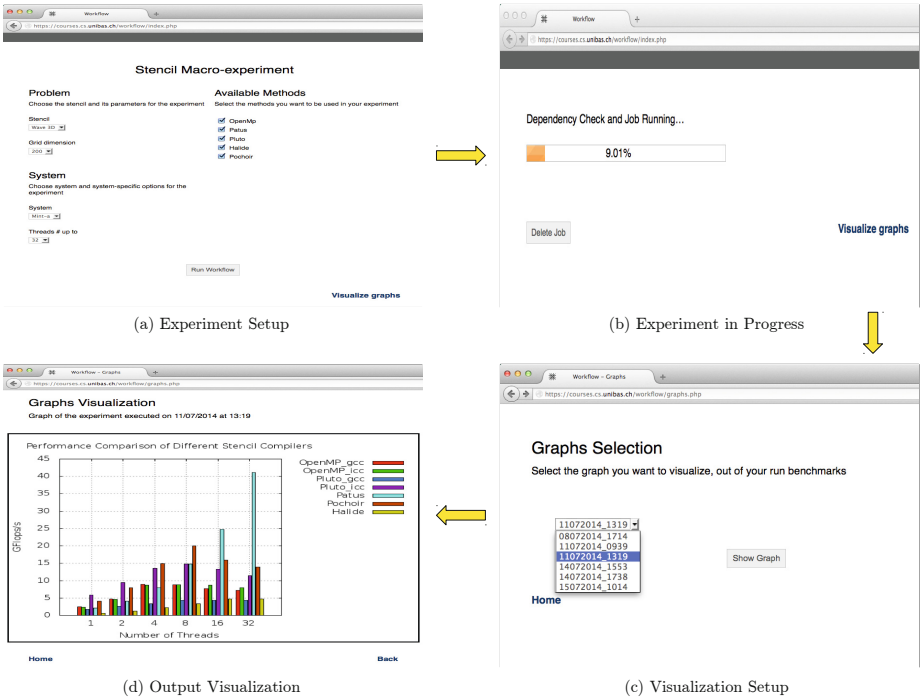


Fig. 4. Benchmark results with 100 timesteps

4.4 Replication Proof via Internet

The question which now arises is: can external people verify these measurements? In order to allow it, we set up a web interface (see Figure 5): you can configure the micro- or macro-experiments and submit them to our parallel machine. The experiment can be called, in principle, from outside, but at this stage it is only possible from our internal network, due to security and load problems.



(a) Experiment Setup

(b) Experiment in Progress

(d) Output Visualization

(c) Visualization Setup

Fig. 5. Phases during the experiment: *(a)* shows the set-up of an (macro- or micro-) experiment, while in *(b)* the experiment is running on the parallel machine; it is then possible to choose *(c)* and visualize *(d)* the output

5 Conclusions and Future Work

If computational and computer science want to be a science, reproducibility needs to be emphasized. This is a challenge on both the technical and social side. On the technical side we need tools and platforms that allow us to store experiments, remotely access them, and offer collaboration support for team efforts but also security and protection guarantees from misuse. On the social side we have to think about incentives for those who spend time in making their experiments reproducible.

We so far achieved a replication prototype for stencil experiments that can be remotely recalled via the Internet. Taking this as a starting point, our taxonomy defines the roadmap for future work. We will extend the distributed architecture, develop security layers and load attack fences, and formalize data and workflow descriptions at the micro- and macro-experiment level. Future system versions should thus support not only replication but also recomputation and reproduction, as well as more generalized experiment settings in computational science.

Acknowledgement. We thank Severin Gsponer for his early contributions to the workflow engine [19].

References

1. Victoria, S.: Trust your science? Open your data and code. Amstat News (2011)
2. Freire, J., Silva, C.T.: Making computations and publications reproducible with VisTrails. Computing in Science Engineering 14(4), 18–25 (2012)
3. Davison, A.P.: Automated capture of experiment context for easier reproducibility in computational research. Computing in Science Engineering 14(4), 48–56 (2012)
4. Hunold, S., Träff, J.L.: On the state and importance of reproducible experimental research in parallel computing. CoRR abs/1308.3648 (2013)
5. Dolfi, M., Gukelberger, J., Hehn, A., Imriska, J., Pakrouski, K., Rønnow, T.F., Troyer, M., Zintchenko, I., Chirigati, F.S., Freire, J., Shasha, D.: A model project for reproducible papers: critical temperature for the Ising model on a square lattice. CoRR abs/1401.2000 (2014)
6. Missier, P., Soiland-Reyes, S., Owen, S., Tan, W., Nenadic, A., Dunlop, I., Williams, A., Oinn, T., Goble, C.: Taverna, Reloaded. In: Gertz, M., Ludäscher, B. (eds.) SSDBM 2010. LNCS, vol. 6187, pp. 471–481. Springer, Heidelberg (2010)
7. Scheidegger, C.E., Vo, H.T., Koop, D., Freire, J., Silva, C.T.: Querying and re-using workflows with VisTrails. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, pp. 1251–1254. ACM, New York (2008)
8. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E.A., Tao, J., Zhao, Y.: Scientific workflow management and the Kepler system. Concurrency and Computation: Practice and Experience 18(10), 1039–1065 (2006)
9. `https://www.knime.org/knime`
10. HPL - A portable implementation of the high-performance Linpack Benchmark for distributed-memory computers (2008), `http://www.netlib.org/benchmark/hpl/`
11. Danilo Guerrera and Antonio Maffia. Workflow for reproducibility (2014), `https://github.com/sguera/workflow_repro`
12. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (December 2006)
13. Frigo, M., Strumpen, V.: Cache oblivious stencil computations. In: Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, pp. 361–366. ACM (2005)

14. Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.-K., Leiserson, C.E.: The Pochoir stencil compiler. In: Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2011, pp. 117–128. ACM, New York (2011)
15. Kelley, J.R., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. SIGPLAN Not. 48(6), 519–530 (2013)
16. Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A., Bastoul, C.: The Polyhedral Model Is More Widely Applicable Than You Think. In: Gupta, R. (ed.) CC 2010. LNCS, vol. 6011, pp. 283–303. Springer, Heidelberg (2010)
17. Bondhugula, U., Ramanujam, J., Sadayappan, P.: PLuTo: A practical and fully automatic polyhedral parallelizer and locality optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University (October 2007)
18. Christen, M., Schenk, O., Burkhart, H.: PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: 2011 IEEE International Parallel Distributed Processing Symposium (IPDPS), pp. 676–687 (2011)
19. Gsponer, S.: Stencil compilers in practice: Workflow engine and code generation issues. Master's thesis, University of Basel (2014)