ExaStencils: Advanced Stencil-Code Engineering

— First Project Report —

Christian Lengauer[1], Sven Apel[1], Matthias Bolten[2],
Armin Größlinger[1], Frank Hannig[3], Harald Köstler[3], Ulrich Rüde[3],
Jürgen Teich[3], Alexander Grebhahn[1], Stefan Kronawitter[1],
Sebastian Kuckuk[3], Hannah Rittich[2], Christian Schmitt[3]

[1]Department of Computer Science and Mathematics, University of Passau
[2]Department of Mathematics and Science, University of Wuppertal
[3]Department of Computer Science, University of Erlangen-Nuremberg

UNIVERSITÄT PASSAU

Fakultät für Informatik und Mathematik

ExaStencils: Advanced Stencil-Code Engineering
— First Project Report —

Christian Lengauer[1], Sven Apel[1], Matthias Bolten[2], Armin Größlinger[1],
Frank Hannig[3], Harald Köstler[3], Ulrich Rüde[3], Jürgen Teich[3],
Alexander Grebhahn[1], Stefan Kronawitter[1], Sebastian Kuckuk[3],
Hannah Rittich[2], and Christian Schmitt[3]

[1] Department of Computer Science and Mathematics, University of Passau
[2] Department of Mathematics and Science, University of Wuppertal
[3] Department of Computer Science, University of Erlangen-Nürnberg

Abstract. Project ExaStencils pursues a radically new approach to
stencil-code engineering. Present-day stencil codes are implemented in
general-purpose programming languages, such as Fortran, C, or Java,
or derivates thereof, and harnesses for parallelism, such as OpenMP,
OpenCL or MPI. ExaStencils favors a much more domain-specific ap-
proach with languages at several layers of abstraction, the most abstract
being the mathematical formulation, the most concrete the optimized
target code. At every layer, the corresponding language expresses not
only computational directives but also domain knowledge of the problem
and platform to be leveraged for optimization. This approach will enable
a highly automated code generation at all layers and has been demon-
strated successfully before in the U.S. projects FFTW and SPIRAL for
certain linear transforms.

1 The Challenges of Exascale Computing

The performance of supercomputers is on the way from petascale to exascale.
Software technology for high-performance computing has been struggling to keep
up with the advances in computing power, from terascale in 1996 to petascale in
2009 on to exascale, now being only a factor of 30 away and predicted for the end
of the present decade. So far, traditional host languages, such as Fortran and
C, being equipped with harnesses for parallelism, such as MPI and OpenMP,
have taken most of the burden, and they are being developed further with some
new abstractions, notably the partitioned global address space (PGAS) memory
model [1] in the languages Coarray Fortran [30], Chapel [9], Fortress [38], Unified
Parallel C [8] or X10 [10]. Yet, the sequential host languages remain general-
purpose: Fortran or C or, if object orientation is desired, C++ or Java.

The step from petascale to exascale performance challenges present-day soft-
ware technology much more than the advances from gigascale to terascale and
terascale to petascale have. The reason is the explicit treatment of the massive

parallelism inside one node of a high-performance cluster cannot be avoided any longer. That is, the cluster nodes must be manycores with high numbers of cores. The reorientation of the computer market from single cores to multicores and manycores has been observed with concern [29]. In the high-performance market, the situation is somewhat alleviated by the fact that the additional cycles that large numbers of cores provide are actually being yearned for. But, the question of how to exploit them with efficient and robust software remains.

While the potential for massive parallelism on and off the chip is the single most serious challenge to exascale software technology, other challenges take on a high priority and are frequently being mentioned, such as power conservation, fault tolerance and heterogeneity of the execution platform [2]. At best, one would strive for performance portability, i.e., the ability to switch the software with ease from one platform, when it is being decommissioned, to the next, while maintaining highest performance.

2   ExaStencils Application Domain: Stencil Codes

Stencil codes have extremely high significance and value for a good-sized community of scientific-computing experts in academia and industry. They see widespread use in solving the systems arising form a discretization of partial differential equations (PDE) and systems composed of such equations. For the implementation of scalable stencil codes, the foremost requirement is to use of efficient solution algorithms, i.e., iterative solvers that rely on the application of a stencil and that provide good convergence properties. Major application areas are the natural sciences and engineering.

Stencil codes are algorithms with a pleasantly high regularity: the data structures are higher-dimensional grids, and the computations follow a static, locally contained dependence pattern and are typically arranged in nested loops with linearly affine bounds. This invites massive parallelism and raises the hope for easily achieved high performance. However, serious challenges remain:

  – Because of the large numbers and varieties of stencil code implementations, deriving each of them individually—even if by code modification from one another—is not practical. Not even the use of program libraries is practical; instead, a domain-specific metaprogramming approach is needed.
  – Efficiency, i.e., a high ratio of speedup to the degree of parallelism, is impaired by the low *computational intensity*, i.e., the low ratio of computation steps to data transfers of stencil codes.
  – An unsuitable use of the execution platform may act as a performance brake.

3   ExaStencils Approach: Domain-Specific Optimization

With project ExaStencils, we propose a radical departure from the traditional way of developing stencil codes. To this end, we make two major decisions.

3.1 Domain-Specific Source Languages

The first decision is to liberate ourselves from the traditional, general-purpose source languages that have historically been dominating high-performance software development, and to move to much easier languages that cater to a specific application domain. This has a serious consequence. The language technology that ensues has great power but for a, in current thinking, shockingly small domain of programs. The most striking example is FFTW (the Fastest Fourier Transform in the West) [17], which is a highly powerful optimizing compiler for essentially one problem: the fast Fourier transform. An optimizing compiler with a somewhat larger domain has been SPIRAL [34], which addresses also a number of (but not all) other linear transforms.

Domain-specific programming has become quite popular recently, and many languages (DSLs), and their compilers, have been proposed and used for specific domains [40, 27]. Alone for the domain of stencil computations, there are, e.g., Liszt [13] (or the newer DeLite), Pochoir [39], and PATUS [11]. Each one of these is pursuing specific goals: Liszt adds abstractions to Java to make stencils programming easier, also for unstructured problems; Pochoir employs a divide-and-conquer skeleton on top of the parallel C extension Cilk to make stencil computations cache-oblivious; PATUS achieves performance by auto-tuning. ExaStencils seeks highest performance via a second radical decision, which we describe next.

3.2 Domain-Specific Optimization at Every Refinement Step

None of the approaches just mentioned has the explicit goal of reaching exascale performance. This is our goal for the domain of stencil codes (thus, the name of our project: ExaStencils). In order to reach it, we insist not only on the freedom to choose or craft the DSL. Rather, we demand also the freedom to choose one dedicated language at every one of a small number of refinement steps, from the first, abstract, executable formulation of the stencil computation down to the target code actually running on the platform of our choice. With every refinement step also comes its own, dedicated, highly automated optimization technology, which exploits the domain-specific knowledge available and useful at this step.

Roughly, the ExaStencils project follows Wirth's notion of stepwise refinement [42] and Parnas' approach of program design, which has later been condensed in the paradigm of model-driven software development [35], and Parnas' notion of program families [33]. The idea is to traverse a path of refinement steps from the mathematical statement of the stencil computation to the target code to be executed on the platform at hand. In every step, choices are made that specialize the solution. These choices are governed by the implementation goals to be reachedD- –different implementation goals, different choices. The overall goal will be the same: exascale performance! But, for different stencil computations and different execution platforms, it may be reached by different choices. By developing a choice tree, we hope to achieve performance portability.

The novel contribution of ExaStencils, beyond the notions of stepwise refinement, model-driven software development, and program families, is the representation, aggregation, and employment of a knowledge base of conditions and

rules concerning stencil codes and the platforms on which they run. ExaStencils makes choices at different layers of abstraction, which form work areas in the project. Let us discuss them in turn.

4 ExaStencils Workflow

The workflow of a stencil-code generation à la ExaStencils is illustrated in Fig. 1. In a first step, a stencil algorithm is engineered by a mathematician. The solution is put into a first executable form via a cooperation of the mathematician with a software engineer. In the ExaStencils approach, the software description names a set of algorithmic and platform choices, each made from a number of options and alternatives. Then, an implementation is "woven" automatically. The weaving algorithm is capable of applying optimizations customized for the specific choices made. One powerful model exploited in ExaStencils is the polyhedron model for automatic loop parallelization. In a final step, some low-level fine-tuning for the platform at hand takes place. The target code can be in any language—or, indeed, mix of languages—that is suitable. In a preliminary code generator, this is C++ (see Subsect. 4.5). In the following subsections, we expand further on these development steps.
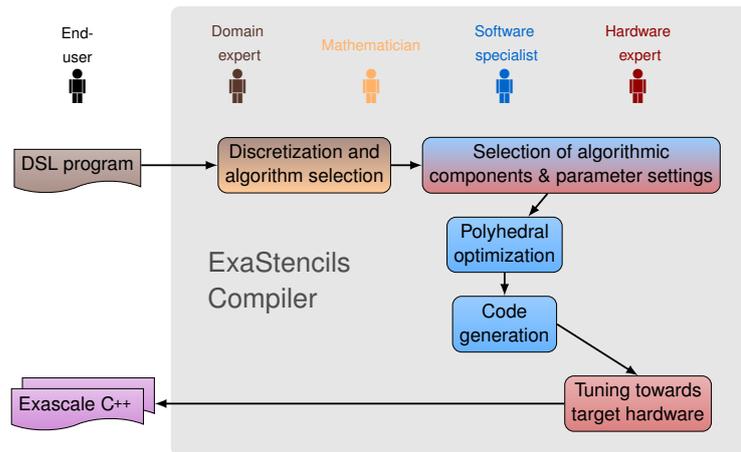

Fig. 1. The workflow of the ExaStencils programming paradigm: the ExaStencils compiler builds on the combined knowledge of domain experts, mathematicians, and software and hardware specialists to generate high-performance target code.

4.1 Algorithmic Engineering

The domain of ExaStencils is multigrid stencil codes on (semi-)structured grids, see Fig. 5. In many applications, a large, structured, linear system consisting

of hundreds of millions of unknowns or more must be solved, whose system matrix can be described compactly, memory-efficiently by one or more stencils. Multigrid methods are *asymptotically optimal* solvers for elliptic PDE, i.e. they belong to the few algorithms that qualify as starting point to implement scalable parallel solvers. Thus, multigrid methods are widely used on massively parallel computers, and different parallel implementations are available that scale on current supercomputer architectures [3, 21, 4, 5, 14, 15]. Multigrid methods involve stencil computations on a hierarchy of very fine to successively coarser grids. On the coarser grids, less processing power is required and communication dominates. A multigrid method is characterized by two strategies: (1) a smoothing strategy, which is used to smooth the sampling error of the grid at hand, and (2) a coarsening strategy, which gets one from a grid to the next coarser grid. Once one arrives at the coarsest level, one refines the grid again via some form of interpolation. This cycle of coarsening and refining is called a V-cycle (Fig. 2). A multigrid algorithm consists of a sequence of progressively deeper V-cycles. Techniques for the efficient implementation and a systematic performance engineering of parallel multigrid methods is a major current research topic [18].

Most of the computational effort in multigrid methods is spent in the smoother, which in simple cases can be a point relaxation, such as Gauss-Seidel or Jacobi. This results in a low ratio of computation to memory load and store operations, limiting the performance that can be achieved on modern architectures, as is typical for applications limited by memory bandwidth [24]. Furthermore, scaling to
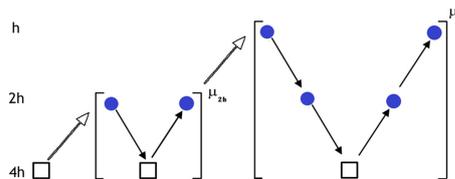
Fig. 2. Depiction of a multigrid algorithm as a succession of V-cycles

very high numbers of processors can suffer from a higher number of levels. For the latter, aggressive coarsening can be a viable option, while the number of computation steps that are necessary can be raised by pipelining of multiple steps of the iterative smoothing procedure, by using polynomial smoothers or by the use of block smoothers. These techniques typically result in a better smoothing factor yielding an overall improved convergence rate.

The performance of multigrid methods depends on the choice of algorithmic components for discretization, grid transfer, cycling strategy, and smoothing. They do influence the total run time, on the one hand, by their influence on the convergence rate, that is the reduction of the error per iteration, and, on the other hand, by the execution time of the individual components on a given architecture. While the former is independent of the target architecture, the latter is influenced strongly by specific hardware properties such as the cache size, the size of the vector units, if present, etc. The convergence rate can be predicted by Local Fourier Analysis (LFA), a mathematical tool that analyzes a given iterative method by freezing coefficients and neglecting boundary conditions. The LFA is used widely in the multigrid community [28, 41]. We have begun to extend the

technique to deal with block-smoothers and aggressive coarsening in addition to the standard LFA techniques [6]. The LFA tool developed will then be used to determine the convergence rate of a multigrid method in terms of the expected convergence rate a priori, i.e., without building and running the actual multigrid method. The combination with a performance model for stencil computations, in general, and the specific requirements of multigrid methods, in particular, enable a prediction of the overall run time of the method without actually running it on the target architecture. This will massively speed up the optimization process used later in the code-generation workflow.

4.2 Domain-Specific Representation and Modelling

Multigrid solvers come in thousands of variants, which differ in the shape of the stencil and the grid, the coarsening and smoothing strategy, the boundary conditions, the communication patterns, and many other conceptual and implementation-level aspects. For example, there are the special strategies necessary to exploit the resources of the execution platform at hand, e.g., caching and load balancing.

One of the radical departures from tradition in our approach happens at the layer of the most abstract executable representation of our problem solution, i.e., our stencil code: We will not consider the code as an individual program but as a member of a family of codes. That is, our domain-specific language will pinpoint the commonalities that the code shares with the other codes of the family and the variabilities in which it departs from the other codes. Each point of variability comes with a number of options or alternatives.
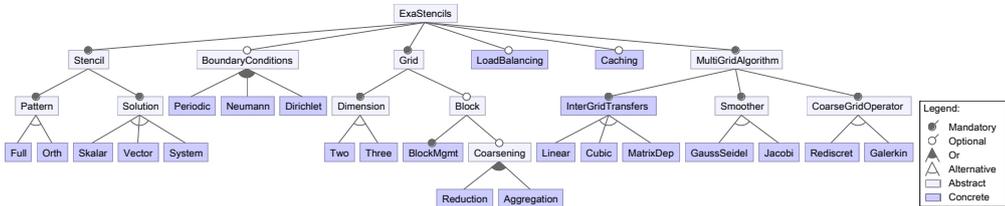
Fig. 3. Example variability model for multigrid stencil codes. We distinguish between *Abstract* and *Concrete* configuration options. Concrete introduces variability, Abstract improves understandability. A configuration option can be either *Mandatory*, i.e., required in all variants, or *Optional*. Configuration options can be grouped in *Alternative* or *Or* groups. Exactly one participant of an Alternative and at least one option of an Or must be selected in one variant.

Commonalities and variabilities are usually specified in terms of a *variability model*. Fig. 3 shows a possible variability model for stencil codes in the form of a tree, in which each node denotes a *configuration option*—in our case, the choice of algorithmic components, alternatives of data structures to be used, and possible parameter values. A selection of configuration options gives rise to an executable variant of the stencil code.

Domain-specific language elements will be our devices for specifying the choices of individual configuration options and their combinations. A review of different technologies for the implementation of DSLs [36] led us to choose Scala [31] as the host language. Actually, we will use four DSLs at decreasingly abstract layers of abstraction (Fig. 4), all hosted by a common parsing and transformation framework. Layers 1–2 address the concerns of application scientists, Layers 2–3 those of mathematicians, and Layers 3–4 those of computer scientists. At present, we are finalizing a prototype generator that will handle input code written in our DSLs.
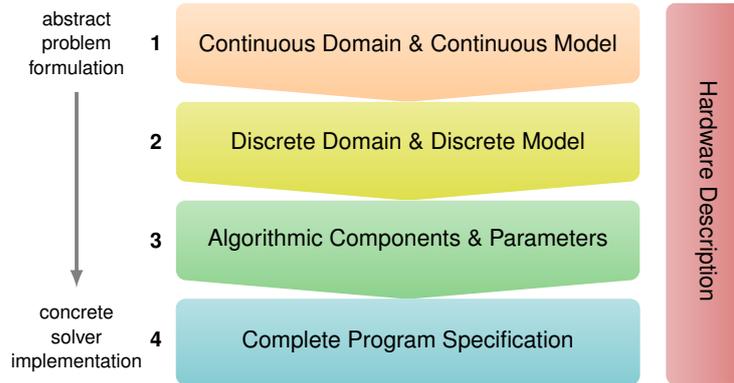

Fig. 4. The DSL hierarchy of ExaStencils

4.3 Domain-Specific Optimization and Generation

Which configuration options (i.e., which choices of algorithmic components, alternatives of data structures, and parameter values) contribute to maximal performance is obvious in some cases and very surprising in others. To make matters worse, certain combinations of options can interfere with each other with respect to performance in subtle ways (which is an instance of the feature-interaction problem [7, 37]). To make this problem tractable, ExaStencils will provide a capability of recommending suitable combinations of configuration option, based on a machine-learning approach. The objective is to make sufficiently accurate performance predictions on the basis of performance measurements of only a small number of concrete stencil-code variants. The latest innovation here emerged from recent work on automated software configuration [37]: The key idea is to detect and handle explicitly interactions among configurations options—even among numeric parameters, rather than simply using black-box auto-tuning [12] or machine-learning approach [22].

We started experiments with the Highly Scalable Multigrid Solver [26]. This solver tolerates a limited lack of structure in the grid by considering so-called hierarchical hybrid grids, as depicted in Fig. 5. At the coarsest level, on the left,

the grid is unstructured, but refinements of each segment (middle and right) must be homogeneous, though each segment may exhibit a different structure.
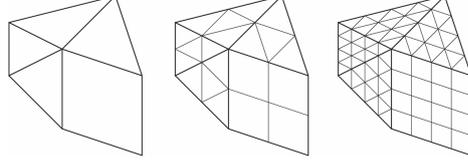
Fig. 5. Successive refinement of a hierarchical hybrid grid

The variability model for the Highly Scalable Multigrid Solver is illustrated in Fig. 6. First experiments have demonstrated already that a machine-learning approach based on the explicit detection and treatment of configuration-option interactions can predict the performance of individual stencil-code variants with a high accuracy [19]. We are only just beginning to exploit domain knowledge but obtained promising results in a first pass even without it. With domain knowledge, notably about already-know configuration-option interactions, we will be able to reduce the number of measurements needed for the prediction further.
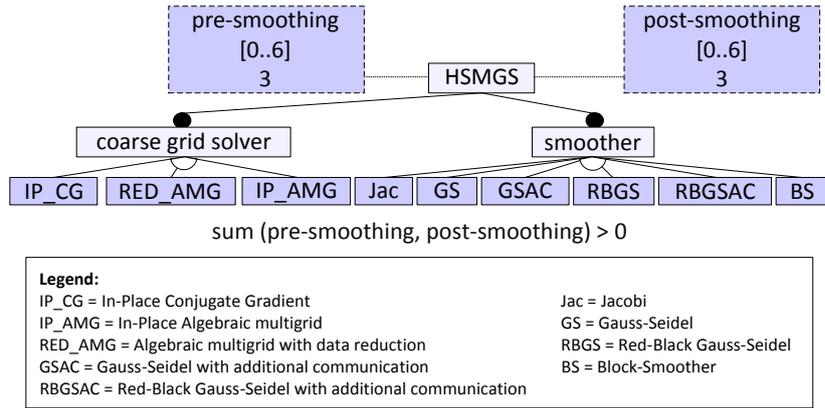
Fig. 6. Concrete variability model for the Highly Scalable Multigrid Solver (HSMGS). In this experiment, we consider the performance contributions and interactions between three different *coarse grid solver*s and five different *smoother*s. Furthermore, we considered the impact on performance of the number of *pre-smoothing* and *post-smoothing* steps, which we vary from 0 up to 6 with 3 as default value.

In the treatment of values of numerical parameters, we employ a function-learning approach: We deduce one polynomial function for each pair or binary option and numerical parameter. Again, so far, we did not exploit domain knowledge, such as the degree of the function that describes the contribution of the parameter values best. Measurements of 10.2% of all stencil-code variants resulted in performance predictions of an accuracy of 89%, on average.

4.4 Loop Parallelization

One important issue at Layer 4 is loop parallelization, since loop nests exhibit the highest potential for a speed gain. The polyhedron model for automatic loop parallelization [16] is a powerful platform for static, i.e., compile-time program optimization. However, it comes with some restrictions that are easily violated by stencil codes. Most importantly, it requires the linear affinity of the loop bounds and the array index expressions. For example, consider the following code for an update of a linearized triangular grid, as depicted in Figure 7:
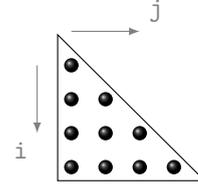

Fig. 7. Triangular grid

```
for (int i = 0; i < n; ++i)
  for (int j = 1; j < i; ++j)
    A[(i*i+i)/2+j] = 0.5 * (B[(i*i-i)/2+j] + B[(i*i+i)/2+j-1]);
```

The linearization avoids a waste of memory that would occur with the use of a two-dimensional, rectangular array. However, this is relevant only in the final target code. During the optimization, one can work with the domain-specific knowledge of the triangularity of the two-dimensional grid and let the polyhedron model loose on the corresponding code, whose two-dimensional accesses are affine:

```
for (int i = 0; i < n; ++i)
  for (int j = 1; j < i; ++j)
    A[i][j] = 0.5 * (B[i-1][j] + B[i][j-1]);
```

Another concern is to optimize reductions in stencil codes effectively. An iterative reduction via a scalar accumulator leads to flow dependences which prevent a direct parallelization. But, with the domain-specific knowledge that the reduction operator is associative and commutative, a corresponding extension to the polyhedron model makes a multitude of optimizations available, such as loop splitting, fusing, or blocking.

The restriction to the domain of stencil codes allows us to perform suitable optimizations, such as temporal or spatial blocking or a combination of both, according to the target architecture [25]. Here, the use of the polyhedron model also ensures a correct boundary handling, regardless of its complexity caused by the combination of different transformations.

4.5 Preliminary Code Generator

The ExaStencils vision that a wide range of stencil codes can be engineered with the same automatic tool –even only that target code for them can be generated with the same code generator– has been met with disbelief. For this reason, we decided to give an immediate proof of concept by developing a preliminary prototypical code generator in Scala at the start of the project [23]. It is lacking

many features that one would expect of a mature code generator, and it is completely unoptimized. However, it is already able to generate code for a non-trivial configuration space, as summarized in Table 1. Note that DSL Layer 4 not in the table; it is too concrete to have meaningful variabilities.

Variability	Layer	Options
Computational domain	DSL 1	UnitSquare, UnitCube
Operator	DSL 1	Laplacian, ComplexDiffusion
Boundary conditions	DSL 1	Dirichlet, Neumann
Location of grid points	DSL 2	node-based, cell-centered
Discretization	DSL 2	finite differences, finite volumes
Data type	DSL 2	single/double accuracy, complex numbers
Multigrid smoother	DSL 3	ω-Jacobi, ω-Gauss-Seidel, red-black variants
Multigrid inter-grid transfer	DSL 3	constant and linear interpolation and restriction
Multigrid coarsening	DSL 3	direct (re-discretization)
Multigrid parameters	DSL 3	various
Platform	Hardware	CPU, GPU
Parallelization	Hardware	serial, OpenMP

Table 1. A variability model for the preliminary Scala prototype. Variabilities in italics must be specified by the application expert, all others can be derived from them.

5 Conclusions

The DFG programme SPPEXA contains two tiers of research. The *conservative tier* is for incremental research on contemporary technologies. One important rôle of this tier is to bring legacy software up to speed for exascale. The *radical tier* is for completely new ways of treating high-performance software. Exa-Stencils belongs to the radical tier. The ultimate goal of the project is to provide proof of the application relevance of the ExaStencils paradigm and to encourage experts of other suitable domains to take a similar approach.

6 Acknowledgements

References

1. Almasi, G.: (PGAS) Partitioned global address space languages. In: Padua et al. [32], pp. 1539–1545
2. Ashby, S., Beckman, P., Chen, J., Colella, P., Collins, B., Crawford, D., Dongarra, J., Kothe, D., Lusk, R., Messina, P., Mezzacappa, T., Moin, P., Norman, M., Rosner, R., Sarkar, V., Siegel, A., Streitz, F., White, A., Wright, M.: The opportunities and challenges of exascale computing – Summary report of the advanced scientific computing advisory committee (ASCAC) subcommittee. Tech. rep., Office of Science, U.S. Department of Energy (Fall 2010)

3. Bergen, B., Gradl, T., Hülsemann, F., Rüde, U.: A massively parallel multigrid method for finite elements. Computing in Science and Engineering 8(6), 56–62 (2006)
4. Bolten, M.: Multigrid Methods for Structured Grids and their Application in Particle Simulation. Ph.D. thesis, Bergische Universität Wuppertal (2008)
5. Bolten, M.: Evaluation of a multigrid solver for 3-level Toeplitz and circulant matrices on Blue Gene/Q. In: Binder, K., Münster, G., Kremer, M. (eds.) Proc. NIC Symp. 2014. NIC Series, vol. 47, pp. 345–352. John von Neumann Institute for Computing (2014)
6. Bolten, M., Kahl, K.: Using block smoothers in multigrid methods. Proc. Appl. Math. Mech. 12(1), 645–646 (2012)
7. Calder, M., Kolberg, M., Magill, E., Reiff-Marganiec, S.: Feature Interaction: A Critical Review and Considered Forecast. Computer Networks 41(1), 115–141 (2003)
8. Carlson, W., Merkey, P.: UPC. In: Padua et al. [32], pp. 2118–2124
9. Chamberlain, B.L.: Chapel. In: Padua et al. [32], pp. 249–256
10. Charles, P., Grothoff, C., Saraswat, V.A., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). pp. 519–538 (2005)
11. Christen, M., Schenk, O., Burkhart, H.: PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: Proc. IEEE Int. Parallel & Distributed Processing Symp. (IPDPS). pp. 676–687. IEEE (2011)
12. Datta, K.: Auto-tuning Stencil Codes for Cache-Based Multicore Platforms. Ph.D. thesis, EECS Department, University of California, Berkeley (2009)
13. DeVito, Z., Joubert, N., Palaciosy, F., Oakleyz, S., Medinaz, M., Barrientos, M., Elsenz, E., Hamz, F., Aiken, A., Duraisamy, K., Darvez, E., Alonso, J., Hanrahan, P.: Liszt: A domain specific language for building portable mesh-based PDE solvers. In: Proc. Conf. High Performance Computing Networking, Storage and Analysis (SC 2011). ACM (2011), paper 9, 12 pp.
14. Falgout, R.D., Jones, J.E., Yang, U.M.: The design and implementation of hypre, a library of parallel high performance preconditioners. In: Bruaset, A.M., Tveito, A. (eds.) Numerical Solution of Partial Differential Equations on Parallel Computers, chap. 8, pp. 267–294. LNCSE 51, Springer (2006)
15. Falgout, R.D., Yang, U.M.: hypre: A library of high performance preconditioners. In: Sloot, P.M.A., Tan, C.J.K., Dongarra, J.J. (eds.) Computational Science (ICCS 2002), Part III. pp. 632–641. LNCS 2331, Springer (2002)
16. Feautrier, P., Lengauer, C.: Polyhedron model. In: Padua et al. [32], pp. 1581–1592
17. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE 93(2), 216–231 (Feb 2005)
18. Gmeiner, B., Köstler, H., Stürmer, M., Rüde, U.: Parallel multigrid on hierarchical hybrid grids: a performance study on current high performance computing clusters. Concurrency and Computation: Practice and Experience 26(1), 217–240 (2014)
19. Grebhahn, A., Siegmund, N., Apel, S., Kuckuk, S., Schmitt, C., Köstler, H.: Optimizing performance of stencil code with SPL conqueror. In: Größlinger and Köstler [20], pp. 7–14
20. Größlinger, A., Köstler, H. (eds.): Proc. Int. Workshop on High-Performance Stencil Computations (HiStencils). www.epubli.de (Jan 2014)

21. Hülsemann, F., Kowarschik, M., Mohr, M., Rüde, U.: Parallel geometric multigrid. In: Numerical Solution of Partial Differential Equations on Parallel Computers, pp. 165–208. Springer (2006)
22. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: An automatic algorithm configuration framework. J. Artificial Intelligence Research 36, 267–306 (2009)
23. Köstler, H., Schmitt, C., Kuckuk, S., Hannig, F., Teich, J., Rüde, U.: A Scala prototype to generate multigrid solver implementations for different problems and target many-core platforms. Tech. rep., Department of Computer Science, University of Erlangen-Nuremberg (Jun 2014)
24. Kowarschik, M., Rüde, U., Weiss, C., Karl, W.: Cache-aware multigrid methods for solving Poisson's equation in two dimensions. Computing 64(4), 381–399 (2000)
25. Kronawitter, S., Lengauer, C.: Optimization of two Jacobi smoother kernels by domain-specific program transformation. In: Größlinger and Köstler [20], pp. 75–80
26. Kuckuk, S., Gmeiner, B., Köstler, H., Rüde, U.: A generic prototype to benchmark algorithms and data structures for hierarchical hybrid grids. In: Proc. Int. Conf. on Parallel Computing (ParCo). pp. 813–822. IOS Press (2013)
27. Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.): Domain-Specific Program Generation. LNCS 3016, Springer (2004)
28. MacLachlan, S.P., Oosterlee, C.W.: Local Fourier analysis for multigrid with overlapping smoothers applied to systems of PDEs. Num. Lin. Alg. Appl. 18, 751–774 (2011)
29. Manferdelli, J.L., Govindaraju, N.K., Crall, C.: Challenges and opportunities in many-core computing. Proc. IEEE 96(5), 808–815 (Apr 2008)
30. Numrich, R.W.: Coarray Fortran. In: Padua et al. [32], pp. 304–310
31. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Press (2010)
32. Padua, D.A., et al. (eds.): Encyclopedia of Parallel Computing. Springer (2011)
33. Parnas, D.L.: On the design and development of program families. IEEE Trans. on Software Engineering (TSE) SE-2(1), 1–9 (Mar 1976)
34. Püschel, M., Franchetti, F., Voronenko, Y.: Spiral. In: Padua et al. [32], pp. 1920–1633
35. Schmidt, D.C.: Model-driven engineering. Computer 39(2), 25–31 (Feb 2006)
36. Schmitt, C., Kuckuk, S., Köstler, H., Hannig, F., Teich, J.: An evaluation of domain-specific language technologies for code generation. In: Proc. Int. Conf. on Computational Science and its Applications (ICCSA) (Jun–Jul 2014), to appear
37. Siegmund, N., Kolesnikov, S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., Saake, G.: Predicting Performance via Automated Feature-Interaction Detection. In: Proc. Int. Conf. on Software Engineering (ICSE). pp. 167–177. IEEE (2012)
38. Steele, Jr., G.L., Allen, E.E., Chase, D., Flood, C.H., Luchangco, V., Maessen, J.W., Ryu, S.: Fortress. In: Padua et al. [32], pp. 718–735
39. Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.K., Leiserson, C.E.: The Pochoir stencil compiler. In: Proc. 23rd ACM Symp. on Parallelism in Algorithms and Architectures (SPAA). pp. 117–128. ACM Press (2011)
40. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages. In: Kent, A., Williams, J.G. (eds.) Encyclopedia of Microcomputers, vol. 28, pp. 53–68. Marcel Dekker, Inc. (2002)
41. Wienands, R., Joppich, W.: Practical Fourier Analysis for Multigrid Methods, Numerical Insights, vol. 4. Chapman & Hall/CRC (2004)
42. Wirth, N.: Program development by stepwise refinement. Comm. ACM 14(4), 221–227 (Apr 1971)