Numer. Math. Theor. Meth. Appl.
doi: 10.4208/nmtma.2015.w10si

Vol. **8**, No. 1, pp. 22-46
February 2015

Towards Textbook Efficiency for Parallel Multigrid

Björn Gmeiner[1,*], Ulrich Rüde[2], Holger Stengel[3], Christian Waluga[1] and Barbara Wohlmuth[1]

[1] *Institute for Numerical Mathematics (M2), Technische Universität München, Boltzmannstrasse 3, D-85748 Garching b. München, Germany.*
[2] *Department of Computer Science 10, FAU Erlangen-Nürnberg, Cauerstraße 6, D–91058 Erlangen, Germany.*
[3] *Erlangen Regional Computing Center (RRZE), FAU Erlangen-Nürnberg, Martensstraße 1, D–91058 Erlangen, Germany.*

Received 5 December 2013; Accepted 29 July 2014

Abstract. In this work, we extend Achi Brandt's notion of textbook multigrid efficiency (TME) to massively parallel algorithms. Using a finite element based geometric multigrid implementation, we recall the classical view on TME with experiments for scalar linear equations with constant and varying coefficients as well as linear systems with saddle-point structure. To extend the idea of TME to the parallel setting, we give a new characterization of a work unit (WU) in an architecture-aware fashion by taking into account performance modeling techniques. We illustrate our newly introduced parallel TME measure by large-scale computations, solving problems with up to 200 billion unknowns on a TOP-10 supercomputer.

AMS subject classifications: 65N55, 68W10

Key words: Multigrid, parallel computing, textbook efficiency, finite element method.

1. Introduction

Asymptotic complexity bounds for iterative solvers that contain unspecified constants provide only heuristic criteria to design efficient solvers [10, Chapter 14]. This simple observation motivated Brandt to coin the term *textbook multigrid efficiency* (TME) to characterize truly fast multigrid algorithms. In [9] he writes

> *Textbook multigrid efficiency means solving a discrete PDE problem with a computational effort that is only a small (less than 10) multiple of the operation count associated with the discretized equations itself.*

*Corresponding author. *Email addresses:* gmeiner@ma.tum.de (B. Gmeiner), ulrich.ruede@fau.de (U. Rüde), waluga@ma.tum.de (C. Waluga), wohlmuth@ma.tum.de (B. Wohlmuth)

In the context of Brandt's TME, computational *efficiency* is evaluated in terms of the number of floating point operations (FLOPS) that are needed to obtain a solution with a certain accuracy. To achieve TME, the number of FLOPS to solve the discretized system must be a small multiple of the FLOPS needed for an application of the discretized operator. Consequently, the elementary work unit (WU) is defined as the cost of one such operator application.

Unfortunately on modern architectures, the TME notion falls short in predicting the performance of an iterative solver since it does not take into account the parallel efficiency and an architecture-aware algorithm design. Nowadays even multigrid algorithms that are TME efficient are not necessarily fast. Thus there is a need to extend the TME metric so that it becomes possible to predict the time to solution also on parallel computer architectures.

The main contribution of our work lies in adapting the TME paradigm to the issues originating from the ongoing transition to massively parallel multicore systems with complex communication networks and with a hierarchical memory architecture. On such modern machines, the simple count of floating point operations of an algorithm correlates poorly with actual computational cost. In particular, energy consumption emerges increasingly as the fundamental bottleneck and as the limiting resource of large scale computing. Since the use of energy is predominantly caused by data movement, the computational cost is strongly affected by moving data. Here data movement includes parallel communication in a large supercomputer cluster, the communication between chips and memory modules in a computer node, and also the access of each processing core to the register banks or cache memory within the chip itself.

In this paper, we focus on linear finite elements (FE) on semi-structured hierarchical simplicial meshes. These restrictions yield the advantage that the complex and computationally expensive logistics of manipulating unstructured grids can be avoided on the finer levels of the multigrid hierarchy. Our implementation is based on the Hierarchical Hybrid Grid (HHG) framework [3,6], a geometric multigrid library implemented in C++ and using the MPI message passing system and hybrid programming. Using HHG, we recently demonstrated that it is possible to solve linear elliptic problems with more than 10^{12} unknowns on current TOP 10 supercomputers, using several 100 000 processor cores [18,19]. The efficiency of the framework relies on the systematic performance- and architecture- aware co-design of the multigrid algorithm and its implementation. In particular, the matrix-free design avoids the explicit storage of the stiffness matrix and thus circumvents the severe efficiency penalty in terms of data storage and redundant data transport. In the course of our work, we also demonstrate an enhanced method to deal with variable coefficients in our stencil-based framework. For illustration, we complement our considerations with large-scale computations.

Before we proceed, let us discuss some related work. Due to the extensive amount of literature dealing with multigrid methods, we must restrict ourselves to recent contributions on parallel realizations for high-end distributed memory architectures. For a further discussion of parallelization techniques for multigrid methods, we refer to [13,24,35]. To the authors' best knowledge, the first parallel multigrid solvers capa-

ble of solving systems with about a billion degrees of freedom were reported in [1, 3]. In [30] a matrix-free geometric multigrid is presented for solving elliptic partial differential equations with variable coefficients. Here, FE discretizations defined on octree-based meshes are used, enabling extreme levels of local refinement. Numerical experiments for the Poisson and linear elasticity operators demonstrate the scalability of the method; in particular, an elasticity calculation with $8 \cdot 10^9$ unknowns using 32 000 processors is presented. A similar approach is followed in [32], where, an algebraic multigrid (AMG) method is used as the coarse grid solver. Numerical experiments for the variable-coefficient Poisson problem are given to demonstrate the scalability of the method. They demonstrate the solution of a scalar Laplacian on a non-uniform mesh of the Antarctic ice-sheet with 10^{11} unknowns using 262 144 cores. A multigrid-preconditioned conjugate gradient algorithm in matrix-free form is developed in [14]. The solver is applied to an elasticity-problem and scalability up to 8 000 cores is demonstrated. In [27] the performance and scalability of conjugate gradient solvers and geometric multigrid algorithms for a scalar model equation is demonstrated on a semi-structured refinement of a spherical shell geometry. Their largest run involves more than 10^{10} unknowns on 65 536 cores. An aggregation based AMG, reaching beyond 10^{11} degrees of freedom on 300,000 cores, is reported in [8]. A study of the scalability of FE solvers on clusters with GPU accelerators is given in [20]; see also [2, 11, 23, 25, 28] for other recent contributions in this direction.

Only few realizations of multigrid algorithms are supplemented with predictive models for the parallel performance on distributed memory architectures. In [29] a new strategy is proposed that allows reliable predictions for the execution time of a given code on a large number of processors, by only benchmarking the code on small numbers of processors. Another approach is followed in [15], where a performance model is developed to predict the time to solution by linking the convergence rate to the arithmetic intensity (FLOPS per byte ratio) via the roofline model [36]. For the optimized computational kernels that are employed in the multigrid implementation of HHG, however, the roofline model fails to predict the performance properties accurately. Hence, in [18] a refined roofline and Execution-Cache-Memory (ECM) performance model for the Sandy Bridge architecture was introduced.

The remainder of the article is structured as follows: In Section 2 we present the finite element discretization of the model problems on block-structured meshes and discuss some important details of the multigrid solver employed in this work. Then, in Section 3, we recall the classical view on TME and show some computational results which allow us to determine multigrid parameters to achieve good convergence using only a small number of work-units. To extend the notion of TME to the parallel setting, we give a new characterization of the WU in an architecture-aware fashion in Section 4. We demonstrate how we can obtain reasonable scaling factors for the WU in a parallel setting using a current state-of-the-art architecture. The following Section 5 then discusses the parallel TME paradigm and illustrates the newly introduced measures by large-scale computations of up to $2 \cdot 10^{11}$ unknowns. We conclude the paper with additional remarks in Section 6.

2. Multigrid on hierarchical hybrid grids

We consider scalar elliptic PDE and systems of PDE that are discretized using standard FE on a polyhedral domain $\Omega \subset \mathbb{R}^3$. By subdividing Ω into a conforming tetrahedral triangulation $\mathcal{T}_{-2}$, we define our base mesh which is further refined using a method due to Bey [7] as illustrated in Fig. 1. More precisely, we uniformly refine this mesh twice to obtain a semi-structured coarse mesh $\mathcal{T}_0$. Based on this initial coarse grid triangulation, we construct a hierarchy of grids $\mathcal{T} := \{\mathcal{T}_l,\ l = 0, 1, \cdots, L\}$ by successive uniform refinement. We point out that this uniform refinement strategy guarantees that all our meshes consist of elements which are scaled and rotated variants of a small group of prototype elements $\widetilde{T} \in \mathcal{B}$. For each refinement of a base element in $\mathcal{T}_{-2}$ there exists 6 such prototype elements, hence $|\mathcal{B}| \leq 6|\mathcal{T}_{-2}|$. This construction not only guarantees that the resulting meshes satisfy a uniform shape-regularity assumption, it can later also be exploited for an efficient realization of matrix-free solver strategies.

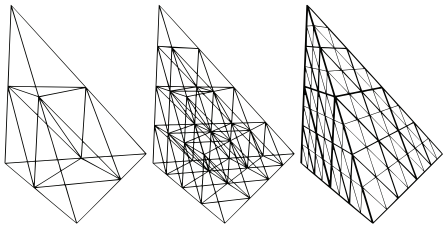

Figure 1: Structured refinement of a tetrahedron due to Bey [7].

For the discretization of the field variables on the different grid-levels, we use linear Lagrange finite elements, i.e, given a mesh $\mathcal{T}_l \in \mathcal{T}$, we define the piecewise polynomial function space

$$S_l^1 := \{v \in C^0(\overline{\Omega}) \ :\ v|_T \in P_1(T),\ \forall\, T \in \mathcal{T}_l\},$$

where $P_1(T)$ denotes the space of linear polynomials on a physical tetrahedral element T of our triangulation. Note that by construction there holds $S_m^1 \subset S_l^1$ for any $l > m \geq 0$.

For each level l, we internally organize the unknowns in groups associated with the primitives, i.e., the elements, faces, edges, and vertices of the base grid $\mathcal{T}_{-2}$. This enables the realization of efficient storage schemes and simplifies the implementation of sophisticated caching strategies to accelerate the parallel communication [5, 6, 16].

2.1. Scalar problems

As a first example, we consider a Poisson problem that serves as a model for many physical phenomena; e.g., in electrostatics, diffusion processes, heat transfer or flow problems in porous media. The governing equation in Ω is given as

$$-\text{div}\,(k\nabla u) = f, \tag{2.1}$$

where $k \in L^\infty(\Omega)$, $k \geq k_0 > 0$, denotes the conductivity parameter, u the potential field, and f the right-hand side. For the sake of simplicity, we assume homogeneous Dirichlet boundary conditions at the boundary $\partial\Omega$ of the simulation domain Ω. However, other kinds of physically relevant boundary conditions can be incorporated in the usual way. We will study the important cases of (piecewise) constant and smoothly varying coefficients.

Using the notation introduced above, we can discretize a weak form of the problem on a fixed mesh-refinement level l as: find $u_l \in V_l := S_l^1 \cap H_0^1(\Omega)$ such that

$$a(u_l, v_l) = L(v_l) \qquad \forall\, v_l \in V_l, \tag{2.2}$$

where we define the bilinear form as $a(u,v) := \int_\Omega k\nabla u \cdot \nabla v \,\mathrm{d}x$ and the linear form is given by $L(v) := \int_\Omega f\, v \,\mathrm{d}x$. As usual in finite element methods, we can expand the discrete solutions in a nodal basis ϕ_l^i, i.e., we can write $u_l = \sum_i u_l^i \cdot \phi_l^i$. In the following, we call the vector $\underline{u_l} = [u_l^i]_i$ the coefficient vector associated with a discrete finite element function u_l.

The nested hierarchy of meshes described above leads to a sequence of discrete problems of the form $A_l \underline{u_l} = \underline{f_l}$, from which a canonical multigrid solver can be constructed.

2.1.1. Basic multigrid components

The main ingredients of a multigrid method are the prolongation and restriction operators that are used to interpolate the discrete solution between two adjacent grid levels, and the smoother that is applied to relax the error on each level.

For the transfer operators, we use, as it is standard, the embedding of the nested finite element spaces $S_l^1 \subset S_{l+1}^1$ to define the prolongation I_l^{l+1}, and we define the restriction as $I_{l+1}^l := (I_l^{l+1})^T$. Consequently, the operators satisfy the Galerkin relation $A_l = I_{l+1}^l A_{l+1} I_l^{l+1}$.

As a smoother, we employ a pointwise Gauss-Seidel update that operates on a row-wise red-black-partition. On each level l, we perform an operation of the form

$$\underline{u_l}^{m+1} = \underline{u_l}^m + \mathcal{S}_l(\underline{f_l} - A_l \underline{u_l}^m),$$

where the processing structure is organized in groups of (base) primitives in the semi-structured grid: first all unknowns associated with the coarse grid vertices are updated, then all unknowns lying on coarse grid edges, followed by all faces, and finally all elements. Generally we follow a sequential update scheme, however, for the sake of parallel efficiency, we relax certain dependencies between primitives of the same dimension by treating them in a block Jacobi-like fashion. In practical applications, the effect of this simplification on the convergence rates is observed to be negligible [6], especially when taking into account the improved parallel efficiency. As previously mentioned, the cost of performing the smoother constitutes a work unit (WU). This

will become relevant in our concept of parallel TME. Hence, the smoother kernel will be analyzed in detail in Section 4.

To enable an efficient matrix-free implementation of the geometric multigrid algorithm, we base our compute-kernels on a stencil-paradigm, i.e., within uniform blocks of the mesh we apply the transfer operations as well as the discrete operator associated with our problem in the fashion of a finite difference method. For later reference, let us here define the stencil

$$s^{[l,i]} = [A_l]_i,$$

which represents the i-th row of the global stiffness matrix A_l on level l. Moreover, as most entries of A_l are zero, we define by $\mathfrak{I}_i$ the index set of non-zeros in row i, i.e., $s_j^{[l,i]} = 0$ for $j \notin \mathfrak{I}_i$. In case of the applied structured mesh refinement, the local representation of the operators of interior nodes amounts to a 15-point stencil, as depicted in Fig. 2.

Applying this stencil to a vector of unknowns constitutes the basic operation in the smoother. The innermost loop performs the update operation

$$u_l^j \leftarrow u_l^j + \frac{\omega}{s_j^{[l,j]}} \left(f_l^j - \sum_{k \in \mathfrak{I}_j} s_k^{[l,j]} u_l^k \right) \tag{2.3}$$

for each index j associated with the primitive that is currently relaxed. Here the upper index stands for the component of the associated vector. A moderate over-relaxation with $\omega > 1$ has been shown to improve the smoothing rates and thus the multigrid efficiency, especially for 3D problems [17, 37].

2.1.2. Stencil assembly

The hierarchical mesh has one important advantage when (patch-wise) constant material parameters are considered. In this case, the amount of memory needed to represent the discretized operator can be largely reduced due to the structured refinements which allow to assemble the stencils locally from pre-computed element stiffness matrices $A_{\widetilde{T}}$ for the different prototype element types $\widetilde{T} \in \mathcal{B}$ present in the mesh, see also [4] for details.

For variable coefficients, however, the stencil needs to be individually assembled for every grid point. This requires a careful implementation to retain a compute- and storage-efficient matrix-free implementation. To this end, we observe that for linear tetrahedral finite elements, the gradients of the reference shape functions $\widehat{\phi}_i$ are constant. Thus for variable coefficients $k(\mathbf{x})$, the local element contributions A_T to the stiffness matrix A_l can obviously be computed by

$$[A_T]_{i,j} := J_T^{-T} \nabla \widehat{\phi}_j \cdot J_T^{-T} \nabla \widehat{\phi}_i |\det(J_T)| \int_{\widehat{T}} \hat{k}(\widehat{\mathbf{x}}) \, \mathrm{d}\widehat{\mathbf{x}}, \tag{2.4}$$

where J_T denotes the Jacobian of the linear-affine mapping from the reference tetrahedron $\widehat{T}$ to the physical element $T \in \mathcal{T}_l$ for some $l \geq 0$, and $\hat{k}$ and $\hat{\phi}_i$ stand for

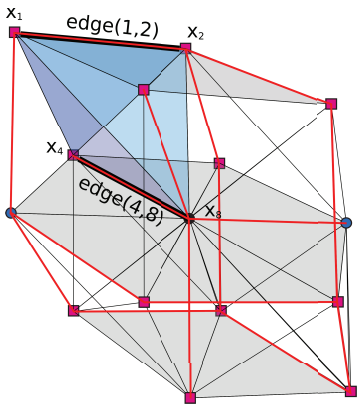

Figure 2: Coefficient averaging of a 15-point stencil. First, two nodal coefficient values on the highlighted edges are averaged, then the coefficients of each element are calculated by averaging two opposing edges.

the transformed functions from T onto $\widehat{T}$. Hence, given a suitable approximation of $\int_{\widehat{T}} k(\widehat{\mathbf{x}})\,\mathrm{d}\widehat{\mathbf{x}}$, the stencil can again be assembled from scaled and rotated variants of the precomputed prototype element stiffness matrices as for the constant coefficient case. To obtain a matrix-free implementation, we must numerically evaluate the coefficient integral in each operator evaluation. For linear tetrahedra, one evaluation at the barycentric coordinate $(1/4, 1/4, 1/4)$ with weight $1/6$, or alternatively a sum over the four nodes with the weights $1/24$ is sufficient [31], provided the coefficient is smooth enough. In our approach, we use the vertex based quadrature formula since a node-wise storage of the coefficient $k(\mathbf{x})$ for the matrix-free construction is better suited to the HHG memory layout structure and thus more efficient. The above integration rule requires three additions per element, and consequently 72 additions per stencil if we were to use a naiv implementation. To reduce the computational work, we apply a local (i.e. stencil-wise) common subexpression elimination that reduces the number of operations to only 40.

If $k(\mathbf{x})$ is extended to a tensor, e.g., in case of anisotropic material properties, one can store the vectors $J_T^{-T}\nabla\widehat{\phi}_j$ instead of the element matrix entries for a cheap evaluation of element matrices.

Let us briefly sketch the averaging procedure: Fig. 2 illustrates the elements that are attached to each inner node in the semi-structured tetrahedral mesh. First, we determine the average between two nodal coefficient's values on each of the 16 edges that are highlighted in the figure. Then, for each of the 24 elements, the coefficients are calculated by averaging the values of the corresponding two opposing edges. The integration weight of $1/24$ is contained in the local stiffness matrices in our implementation.

Let us remark that the node-wise organization of the coefficients does not only save roughly a factor of 6 in memory compared to an element-wise storage, but moreover it allows to implement the stencil assembly without indirect addressing, since plain array data structures can be employed. The assembly procedure for the nodes lying

on the lower-dimensional primitives of the base mesh, i.e., (base) vertices, edges, and faces, are more technical and require indirect addressing, since they are basically unstructured or at least are composed of rotated stencil parts; we refer to [16] for more technical details.

2.2. Extension to systems with saddle-point structure

As an example for systems of PDE with saddle point structure, we consider the Stokes system in a polyhedron Ω:

$$-\operatorname{div}(\nabla \mathbf{u} - p\mathbf{I}) = \mathbf{f}, \tag{2.5a}$$

$$\operatorname{div} \mathbf{u} = 0. \tag{2.5b}$$

Here $\mathbf{u} := [u_1, u_2, u_3]^\top$ denotes the fluid velocity, p the pressure, and $\mathbf{f} := [f_1, f_2, f_3]^\top$ a forcing term acting on the fluid. For simplicity, let us again consider $\mathbf{u} = \mathbf{0}$ on the boundary $\partial\Omega$. Further, to make the pressure well-defined, we set $\int_\Omega p \,\mathrm{d}x = 0$.

The discrete velocity-pressure pair is approximated in the product space

$$\mathbf{V}_l \times Q_l := [S_l^1 \cap H_0^1(\Omega)]^3 \times [S_l^1 \cap L_0^2(\Omega)].$$

By definition, the boundary conditions on the velocity $\mathbf{u}$ as well as the mean-value condition of the pressure p are directly built into the discrete function spaces.

We use the following standard weak formulation of the Stokes problem for discretization: find $(\mathbf{u}_l, p_l) \in \mathbf{V}_l \times Q_l$ such that

$$\mathbf{a}(\mathbf{u}_l, \mathbf{v}_l) + \mathbf{b}(\mathbf{v}_l, p_l) = \mathbf{L}(\mathbf{v}_l) \quad \forall\, \mathbf{v}_l \in \mathbf{V}_l, \tag{2.6a}$$

$$\mathbf{b}(\mathbf{u}_l, q_l) - \mathbf{c}(p_l, q_l) = 0 \qquad \forall\, q_l \in Q_l, \tag{2.6b}$$

where we define the bilinear forms as

$$\mathbf{a}(\mathbf{u}, \mathbf{v}) := \int_\Omega \nabla \mathbf{u} : \nabla \mathbf{v} \,\mathrm{d}x, \quad \mathbf{b}(\mathbf{u}, q) := -\int_\Omega \operatorname{div} \mathbf{u} \cdot q \,\mathrm{d}x,$$

and the linear form as $\mathbf{L}(\mathbf{v}) := \int_\Omega \mathbf{f} \cdot \mathbf{v} \,\mathrm{d}x$. Since we use equal-order approximation of the velocity and the pressure, we have to incorporate a stabilization to avoid spurious pressure modes. Here, we choose a standard approach detailed in [12], namely

$$\mathbf{c}(p, q) := \sum_{T \in \mathcal{T}_l} \alpha_T^2 \int_T \nabla p \cdot \nabla q \,\mathrm{d}x,$$

which ensures stability of the discrete problem. The stabilization parameter $\alpha_T \sim \operatorname{diam}(T)$ has to be chosen carefully to ensure uniform stability and to avoid unwanted effects due to over-stabilization.

After discretizing (2.5) with finite elements as described before, the problem can again be written as a linear system of equations:

$$\begin{bmatrix} \mathbf{A}_l & \mathbf{B}_l^\top \\ \mathbf{B}_l & -\mathbf{C}_l \end{bmatrix} \begin{bmatrix} \underline{\mathbf{u}}_l \\ \underline{p}_l \end{bmatrix} = \begin{bmatrix} \underline{\mathbf{f}}_l \\ \underline{0} \end{bmatrix}. \tag{2.7}$$

Here, the operators can be assembled similarly as in the scalar case, and $\underline{\mathbf{u}}_l$ and $\underline{p}_l$ again denote the coefficient vectors associated with the finite element functions $\mathbf{u}_l \in \mathbf{V}_l$ and $p_l \in Q_l$, respectively.

For the solution of the discrete linear problem (2.7), we use an approach outlined in [34]: By formally eliminating the velocities from the pressure equation, we can reformulate (2.7) equivalently as

$$\begin{bmatrix} \mathbf{A}_l & \mathbf{B}_l^\top \\ \mathbf{0} & \mathbf{C}_l + \mathbf{B}_l \mathbf{A}_l^{-1} \mathbf{B}_l^\top \end{bmatrix} \begin{bmatrix} \underline{\mathbf{u}}_l \\ \underline{p}_l \end{bmatrix} = \begin{bmatrix} \underline{\mathbf{f}}_l \\ \mathbf{B}_l \mathbf{A}_l^{-1} \underline{\mathbf{f}}_l \end{bmatrix}.$$

Now the discrete problem for the pressure reads

$$S_l \underline{p}_l = \underline{g}_l. \tag{2.8}$$

Here $S_l = \mathbf{C}_l + \mathbf{B}_l \mathbf{A}_l^{-1} \mathbf{B}_l^\top$ and $\underline{g}_l = \mathbf{B}_l \mathbf{A}_l^{-1} \underline{\mathbf{f}}_l$ define the pressure *Schur complement* system. In our numerical experiments, we solve (2.8) by a preconditioned conjugate gradient (PCG) method, where we choose a lumped mass matrix M_l as preconditioner that is known to be spectrally equivalent to S_l. This simple and cheap preconditioning reduces the convergence degradation effects due to varying element sizes and shapes.

As S_l is in general dense, its direct assembly cannot be performed efficiently. However, since the PCG method only requires applications of the Schur-complement operator, we can replace this operation by a series of matrix-vector multiplications, where the multiplication with the discrete inverse vector-Laplacian $\mathbf{A}_l^{-1}$ is performed inexactly by three independent multigrid iterations (one for each velocity component u_i) as described for the scalar case. This yields an iterative scheme using inner (multigrid) and outer (PCG) iterations. The dominant cost here is the inner iteration.

3. Classical textbook multigrid efficiency

We now proceed to analyze the efficiency of the solvers proposed in Section 2 in the light of the classical notion of TME. To this end, we tune the multigrid parameters to achieve fast convergence with respect to the number of WUs. Here the operator evaluations on the coarser meshes as well as the residual calculation contribute to the number of WUs, but we neglect the contribution of the inter-grid transfers since they are significantly cheaper (considering their FLOPS) than one relaxation of the system. However, for an actual implementation they require a relative high memory-bandwidth and have a poor balance between additions and multiplications. Thus they cannot exploit the architecture of current CPUs to their full capacity, and thereby increase the run-time by a small percentage.

We will present results for iterative V- and W-cycles and also for a full multigrid (FMG) implementation for the scalar case (CC) as well as for the system (SF). Here, the expression C(μ_1, μ_2) denotes the number of pre-smoothing steps μ_1, the post-smoothing steps μ_2, and the type of cycle C $\in$ { V, W } under consideration. Similarly

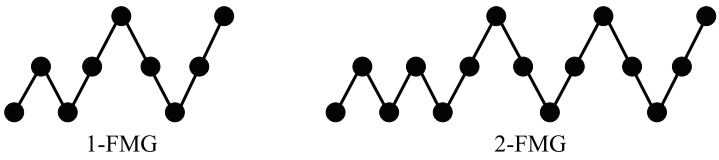

Figure 3: Illustration of FMG-1V$(\cdot,\cdot)$ and FMG-2V$(\cdot,\cdot)$ cycles.

to the V-cycle and the W-cycle, two variants of the FMG are chosen as illustrated in Fig. 3.

In the experiments, we start with a zero initial vector. To quantify discretization errors, we use a scaled Euclidean norm $\|\underline{u}-\underline{u}_l\| := h_l^{3/2}\|\underline{u}-\underline{u}_l\|_{\ell^2}$, where $\underline{u}$ denotes the coefficient vector associated with the nodal interpolation of the exact solution into the discrete function space. This enables us to also compare the iteration error (i.e. algebraic error) with the discretization error, which is essential in quantifying the efficiency of a FMG method.

For simplicity, the computational domain in all our computations is chosen as the unit cube $\Omega = (0,1)^3$ if not mentioned otherwise. For scaling and performance studies on more complex domains, we refer to our work [18], where spherical shells and complex networks of channels were considered using the same version of the HHG software framework. For such geometries the performance was observed to deteriorate by some ten percent compared to the cube domain.

Our first experiment investigates the performance of different multigrid settings, varying the number of smoothing steps, type of cycle, and using over-relaxation in the smoother. We first consider the scalar case (2.1) with a constant coefficient, i.e., $k = 1$, and an exact solution given as

$$u(\mathbf{x}) = \sin(2x_1)\sin(4x_2)\sin(16x_3). \tag{CC}$$

The boundary data is chosen according to the exact solution and the source term f on the right hand side is constructed by evaluating the strong form of the operator on the left hand side of (2.1). In Fig. 4, we display errors and residuals as functions of computational cost measured in work units. We note that in the pre-asymptotic stage, the W(3,3) cycle with an over-relaxation factor of $\omega = 1.3$ produces the most efficient error reduction, but interestingly this is not the most efficient method when considering the convergence of the residual. In this respect, the V(3,3) cycles with a mesh independent over-relaxation factor of again $\omega = 1.3$ in (2.3) are found to perform best. All methods require about $30-50$ operator evaluations (WU) to converge to the truncation error of approximately 10^{-6}. Clearly, we have not yet reached the 10 WU required for TME, but since we solve systems as large as $8.6\cdot 10^9$ unknowns, we believe that this is already in an acceptable range.

To achieve full TME, we next consider a FMG algorithm. To evaluate different variants of FMG, Table 1 lists the ratio

$$\gamma_l = \|\underline{u}-\tilde{\underline{u}}_l\|/\|\underline{u}-\underline{u}_l\|, \tag{3.1}$$

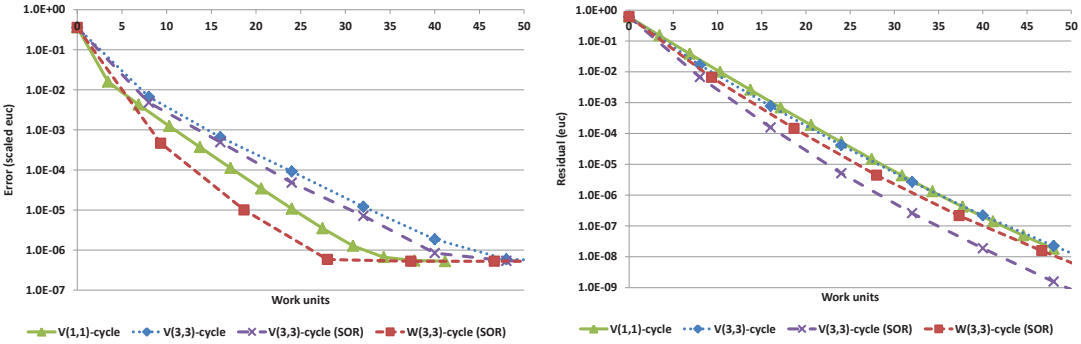

Figure 4: Error (left) and residual (right) for (CC) on a resolution of $8.6 \cdot 10^9$ grid points.

between the total error obtained by the FMG $\|\underline{u} - \underline{\tilde{u}}_l\|$ and the discretization error $\|\underline{u} - \underline{u}_l\|$, where $\underline{\tilde{u}}_l$ denotes the result obtained with FMG. For the first row of Table 1 there is only one coarser level, and we thus display the results for a two-level algorithm. In each following row of the table, the mesh size of the finest mesh is halved and thus number of multigrid levels increases by one. Here a FMG-V(2,2) cycle is a full multigrid method employing a single V(2,2) cycle on each new level. As shown in Table 1 the combined error is roughly 1.7 times larger than the discretization error, and thus discretization error and algebraic error are roughly of the same size. The FMG-2V(1,1) cycle employs 2 V-cycles on each new level and results in a relative algebraic error of $\approx 10\%$ compared to the discretization error.

Table 1: Ratio γ_l between approximation error after one FMG-cycle and the discretization error for (CC).

Resolution	Discr. Error	FMG-V(1,1)	FMG-2V(1,1)	FMG-V(2,2)	FMG-V(3,3)
129^3	$1.4 \cdot 10^{-4}$	4.3	1.2	2.3	2.0
257^3	$3.4 \cdot 10^{-5}$	4.0	1.1	1.7	1.3
513^3	$8.4 \cdot 10^{-6}$	3.9	1.1	1.7	1.3
$1\,025^3$	$2.1 \cdot 10^{-6}$	4.1	1.1	1.7	1.3
$2\,049^3$	$5.3 \cdot 10^{-7}$	4.4	1.1	1.7	1.3

A V(2,2) cycle employs on the finest grid a total of 4 WU for the smoothing steps, plus one for the residual computation. We neglect the work for restriction and prolongation, but account for the cost of V-cycle recursion with standard coarsening in 3D by the factor of $1 + \frac{1}{8} + \frac{1}{64} + \cdots = \frac{8}{7}$. Note that the second recursion in the FMG algorithm contributes another factor of $\frac{8}{7}$ to the cost estimate, such that the cost of the FMG-V(2,2) cycle can be estimated as $(\frac{8}{7})^2 \cdot 5 \approx 6.5$ WU, while the FMG-2V(1,1) cycle requires $(\frac{8}{7})^2 \cdot 2 \cdot 3 \approx 7.8$ WU. Thus, the 1.3 extra work units lead to an error reduction from 1.7 to 1.1 times the discretization error.

A heuristic explanation why the FMG-2V(1,1) cycle yields better results for this example than the FMG-V(2,2) cycle can be given as follows. On finer grids not only the solution of the model problem (CC) is smooth, but also the discretization error

is essentially a smooth function. On each new level, the V-cycles must reduce an error component that is the difference of the discretization errors of the two levels, i.e. a smooth function. However, for a smooth error, relaxation on fine grids is inefficient and thus this error must be reduced on coarser grids. Note that this remaining discretization error can only be *evaluated* on a fine grid, but it must be *removed algebraically* by recursing onto coarser grids. Thus, the extra coarse grid corrections and residual computations of the FMG-2V(1,1) cycle (as compared to the FMG-V(2,2) cycle) pay off, even though both methods eventually use an equal number of smoothing steps on each level. Note further that this is an effect in the pre-asymptotic stage of the V-cycle when considering it as an algebraic solver. However, this situation is typical when V-cycles are used within an FMG method that is itself in the asymptotic phase as a solver for the differential problem.

To assess next which of the two V-cycle methods is more efficient in error reduction with respect to the PDE problem, we can argue as follows. Comparing the FMG-2V(1,1) cycle with the FMG-V(2,2) cycle, a 35% improvement of the error is achieved by 20% higher cost. We must now compare this error reduction relative to the cost with the accuracy improvement that could be achieved by progressing to another level of mesh refinement. Here an 8-fold increase of work would asymptotically lead to 4-fold reduced error (for both the FMG-2V(1,1) and FMG-V(2,2) method). This is asymptotically less efficient than the ratio $20/35$, and thus the error reduction by the stronger FMG-2V(1,1) method must be considered as efficient in the given constellation. Progressing to a finer grid with the same solution strategy would not reduce the error with respect to the differential equation not as effectively as using the more expensive iteration on the coarser grid.

In our second example, we consider the solution of the Stokes system (2.5):

$$\mathbf{u}(\mathbf{x}) = \begin{bmatrix} -4\,\cos(4\,x_3) \\ 8\,\cos(8\,x_1) \\ -2\,\cos(2\,x_2) \end{bmatrix}, \quad \text{and} \quad p(\mathbf{x}) = \sin(8\,x_2)\,\sin(2\,x_3)\,\sin(4\,x_1). \qquad \text{(SF)}$$

Let us remark that $\mathbf{u}$ is solenoidal by construction. The boundary data and right hand side are again chosen such that the strong problem (2.5) is satisfied.

Analog to the FMG cycle, the algorithm proceeds from the coarsest to the finest level. However instead of one or two multigrid V-cycles, we apply four PCG iterations as pressure correction to the system (2.8), including one restart every two iterations on each level. As described in Section 2.2, each outer CG iteration uses one inner multigrid cycle (that is, however, applied independently to each scalar velocity unknown). Table 2 lists the ratio γ_l for two different V-cycle types. We observe an almost mesh independent convergence using around $(\frac{8}{7})^2 \cdot 6 \cdot 4 \approx 31$ WU for the case FMG-4CG-1V(2,1) that employs on each level 4 CG iterations each of which employs one V(2,1) cycle. Indeed, this is about three times more than ideal TME. Nevertheless, the operator evaluations are cheap when considering that a non-staggered discretization is often handled by significantly more expensive smoothers. But we note that while the solution process through the pressure Schur complement is algorithmically simple and

Table 2: Ratio γ_l between approximation error after one FMG-cycle and the discretization error for (SF).

Resolution	Discr. Error	FMG-2CG-1V(2,1)	FMG-4CG-1V(2,1)	FMG-4CG-1V(2,2)
129^3	$8.5 \cdot 10^{-4}$	2.1	1.2	1.2
257^3	$2.1 \cdot 10^{-4}$	3.0	1.1	1.1
513^3	$5.2 \cdot 10^{-5}$	5.4	1.2	1.1
$1\,025^3$	$1.3 \cdot 10^{-5}$	9.7	1.3	1.2
$2\,049^3$	$3.2 \cdot 10^{-6}$	19.1	1.4	1.3

easy to implement, it seems to be less efficient (in the sense of TME) than the multigrid strategies based on distributive smoothers as proposed for staggered and non-staggered discretizations in [10].

4. Architecture-aware characterization of a work unit (WU)

The TME metric used in the previous results illustrates the convergence of the multigrid algorithm, but it does neither reflect how the algorithm performs on single-nodes nor on massively parallel computer architecture. Thus there is a need to quantify the performance of an algorithm with respect to run time. The goal of this section is a systematic assessment of this issue for the presented multigrid solver. This requires that we develop estimates or upper bounds for the cost of a WU. Consequently, we will now proceed to develop a systematic performance analysis based on the notion of TME for the HHG multigrid implementation.

The architectural complexity of modern computers makes the prediction of compute times and other performance metrics increasingly difficult. For example, reliable estimates of the run-time require elaborate performance models. We advocate with this article, that the development of appropriate quantitative and predictive performance models should be seen as an essential step in the systematic design of scientific computing software for high performance computers.

For the development of a quantitative model we first have to identify the *critical resource* in the computationally intensive parts of the algorithm. In the light of TME, our starting point is thus a careful inspection of the multigrid smoother on the finest mesh level that defines the work unit (WU). The basic operation (2.3) in the smoother corresponds to a 15-point stencil applied in a point-wise Gauss–Seidel update scheme, see (2.3). In the simplest case of a scalar PDE with constant coefficients (CC), we count 15 multiplications, 14 subtractions/additions, 15 loads (14 for u, 1 for f), and one store operation per stencil update. Executing this loop for all nodes of the finest FE mesh constitutes one WU. Note that the relaxation loop uses a stride of 2 due to the row-wise red-black partitioning. However, we significantly gain performance by the design of HHG using patch-wise structured meshes. Within each tetrahedral HHG patch, the connectivity structure at each node is identical such that a matrix row can be described as a stencil and 3D-arrays can be used to store the solution and load vectors. This permits significantly more efficient machine code, since the stencil update can be

implemented with only direct addressing and efficient index incrementing schemes for all data access operations. This also helps the compiler to analyze data dependencies and better exploit instruction level parallelism, leads to a better use of caches and prefetching hardware, etc.

Let us next consider the case of variable coefficients (VC). Following Section 2.1.2, the stencil at each node $s^{[l,i]}$ (i.e. one row of the stiffness matrix) is assembled from the local FE stiffness matrices in the surrounding tetrahedra as illustrated in Fig. 2. Note that each of the local stiffness matrices is just a scaled variant of the local stiffness matrix of the same tetrahedron for constant coefficients. Thus there exists only a small number of different local stiffness matrices on each structured block of the HHG grid that can be pre-computed and then re-used when assembling the stencil at a node; cf. [18] for details. Altogether, 136 subtractions/additions, 121 multiplications, 1 division, 30 loads (15 k, 14 u, 1 f) and 1 store must be performed in one stencil update.

We proceed by evaluating how much time is required to execute a single elementary relaxation. Therefore, we first focus on the elementary compute unit, i.e., a single processor core of the supercomputer system under consideration. The performance is here expressed in form of the reciprocal value, as *lattice updates per second* (Lups/s), see [21], where 1 GLups/s $= 1000$ MLups/s $= 10^9$ Lups/s. This metric connects the relevant useful work performed to the run-time.

For the convenience of the reader, let us first summarize the results of the performance predictions obtained by the different models and our measurements, and then discuss in detail how these are obtained. Table 3 lists the theoretically maximal performance for both smoother kernels (i.e. one WU in each case) on a single core of an Intel Sandy Bridge (SNB) system under different assumptions on what the limiting resource is. Here the values that assume memory bandwidth as the bottleneck are based on the measured STREAM [26] bandwidth of $16\,\mathrm{GB/s}$ for a single core of the architecture, while the FLOPS based limiting values are derived from the theoretical peak performance. Note here that the memory limit for case (VC) is only moderately lower than for the (CC) case. This is a result of the assembly of the matrix on the fly with only a minimal storage requirement for the coefficient k, as described in Section 2.1.2. If the stiffness matrix A_l were stored in a conventional sparse matrix format, requiring read access for 15 entries per row, plus the indices/pointers needed for indirect addressing, this would lead to almost an order of magnitude higher memory traffic and a correspondingly sharper bandwidth performance limit.

Table 3: Predicted versus measured computational performance for one WU (one Intel SNB core).

	assumed limiting resource			
	FLOPS throughput	memory bandwidth	ECM model	measured
const. coeff. (CC)	720 MLups/s	667 MLups/s	159-189 MLups/s	176 MLups/s
var. coeff. (VC)	79.4 MLups/s	500 MLups/s	40.0-42.5 MLups/s	39.5 MLups/s

The performance limits of Table 3 show that for (CC), both memory bandwidth and FLOPS throughput would lead to similar limitations, but that the measured perfor-

mance is significantly lower. This can be explained by the more involved ECM model that we will introduce below. Similarly, the (VC) case is severely limited by FLOPS throughput, and will not be bandwidth limited on a single core. However, considering measurements we note that an accurate prediction again requires the use of a more advanced model.

In the following sections, we will analyze the limiting resource for the smoothers in detail. This must take reference to a concrete compute system, consisting of hard- and software. For the experiments below, the code is compiled with the Intel compiler version 13.1.3 build 20130607 (using flags `-O3 -xAVX -fno-alias`) and linked to Intel MPI version 4.1.1.036. For optimal vectorization, the `#pragma simd` compiler directive is employed where appropriate. For benchmarking purposes, we use a dual-socket system with eight-core SNB (Xeon E5-2680) processors and DDR3-1600 memory. The clock frequency is fixed to 2.7 GHz.

4.1. Roofline performance model

For first insights into the expected performance, we map the smoother kernel code to a basic machine model and determine the quantitatively dominant operation. In the constant coefficient case (CC), loads and multiplications are on par with 15 appearances each. Due to its higher computational intensity, 136 additions are required in the variable coefficient code (VC). On the SNB micro-architecture, one `mul` or `add` instruction can be executed per cycle. It is assumed that the most effective instructions employing single instruction multiple data (SIMD) can be used, which results in 4 arithmetic operations per instruction using the Advanced Vector Extensions (AVX) to the instruction set with 256 bit register width. Under the further assumptions that all pipelines can operate in parallel without dependencies and that all data resides in L1 cache, one iteration of the AVX vectorized inner loop takes 15 cycles (CC) and 136 cycles (VC), to perform 4 Lup. This results in a single-core performance of

$$P^{\max}_{\text{core}}(\text{CC}) = \frac{2.7\cdot 10^9 \text{ cycles/s}}{15/4 \text{ cycles/Lup}} = 720\,\text{MLups/s}\ , \tag{4.1}$$

$$\text{and} \qquad P^{\max}_{\text{core}}(\text{VC}) = \frac{2.7\cdot 10^9 \text{ cycles/s}}{136/4 \text{ cycles/Lup}} = 79.4\,\text{MLups/s}, \tag{4.2}$$

for the constant and variable coefficient case, respectively. These values take the role of a *speed of light* for these particular algorithms, since they mark the maximal throughput under best case assumptions. If further assuming perfect parallel scalability, the single-core performance can be extrapolated to $P^{\max}_{\text{socket}}(\text{CC}) = 5.76\,\text{GLups/s}$ and $P^{\max}_{\text{socket}}(\text{VC}) = 635\,\text{MLups/s}$ for the full socket (8 cores).

Next, we turn to the limitations that are caused by memory throughput and determine the actual memory data traffic for one stencil update. Due to the presence of caches not all load instructions executed by the processor cause data transfer from memory (see following sections for details), resulting in $24\,\text{bytes}$ (CC) and $32\,\text{bytes}$ (VC) of mandatory main memory traffic for a single update ($8\,\text{bytes}$ per double precision element). For single-core execution, a memory bandwidth of $720\,\text{MLups/s}\cdot$

24 bytes/Lup $= 17.3\,\mathrm{GB/s}$ would be required for (CC), and 79.4 MLups/s$\cdot 32$ bytes/Lup $= 2.5\,\mathrm{GB/s}$ for (VC).

Again, we extrapolate to the socket level. For a bandwidth limited code this is the relevant scaling unit. In case (CC) a main memory bandwidth of roughly 138 GB/s would be needed, and 20.3 GB/s for (VC). However, the SNB system can sustain approximately 42 GB/s for an update data access pattern. Based on this first estimate, we conclude that case (CC) is main memory bandwidth limited on the given architecture when all cores are employed. The prediction for socket performance must be adjusted to

$$P^{\max}_{\text{socket}}(\text{CC}) = \frac{42GB/s}{24B/\text{Lup}} = 1.75\,\text{GLups/s}. \tag{4.3}$$

We further conclude that case (VC) is limited by FLOPS throughput and will not be affected by the memory bandwidth limit of $\frac{42GB/s}{32B/\text{Lup}} = 1.31$ GLups/s.

4.2. ECM performance model for constant coefficients (CC)

Actual single-core measurements result in a performance of 176 MLups/s, equivalent to approximately $4.2\,\mathrm{GB/s}$ main memory bandwidth. This suggests that the arithmetic instruction throughput is not sufficient to explain and predict the performance of executing the (CC) smoother stencil. The prediction of the estimate in Section 4.1 suggests that we should find a severe bandwidth saturation. However, the estimate yields only an upper bound and the benchmark suggests a significantly lower degree of bandwidth saturation, making it unlikely that bandwidth is truly the critical resource. The degree of bandwidth saturation is also an important indication of potential code optimizations.

Since the performance predictions obtained as above by the roofline model are unsatisfactory, we must proceed to develop a more accurate analysis. To this end, a more careful inspection of the code reveals the following: The vectorization report by the Intel compiler confirms that the innermost loop of the stencil code is optimally executed. However, the roofline model in its simplest setting does not account for a possible non-optimal instruction code scheduling. Also the run-time contributions of transfers inside the cache hierarchy are not taken into account. To include these effects, we employ the *Execution-Cache-Memory* (ECM) model of [22, 33], and we refer to [18] for a thorough description of the modeling process for this kernel.

An overview of the input parameters to the model is displayed in Fig. 5. Different assumptions which operations can be overlapped lead to a corridor of performance predictions of $159 - 189$ MLups/s for a single core. The obtained measurements are within this range, indicating that the micro architecture is capable of overlapping at least some of the operations (cf. Fig. 6). We also note that the (CC) kernel cannot saturate the maximal memory bandwidth of 42 GB/s with 8 cores. A comparison of the different run-time contributions shows that the single-core performance is in fact dominated by code execution and that — in contrast to the initial analysis — instruction throughput and not memory bandwidth is the critical resource. The poor predictive results that

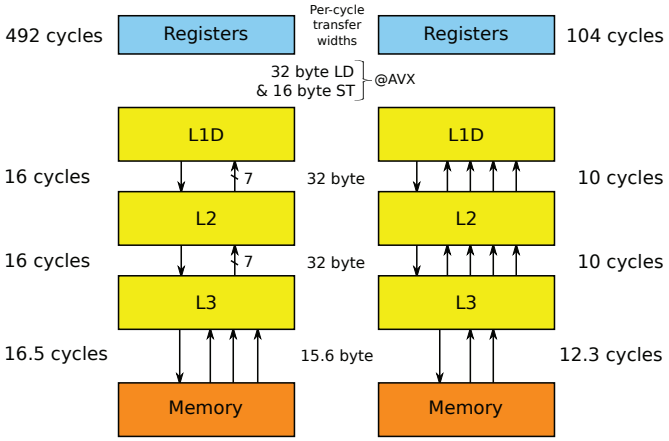

Figure 5: ECM model for the 15-point stencil with variable coefficients (left) and constant coefficients (right) on SNB core. An arrow indicates a 64 Byte cache line transfer. Run-times represent 8 elementary updates.

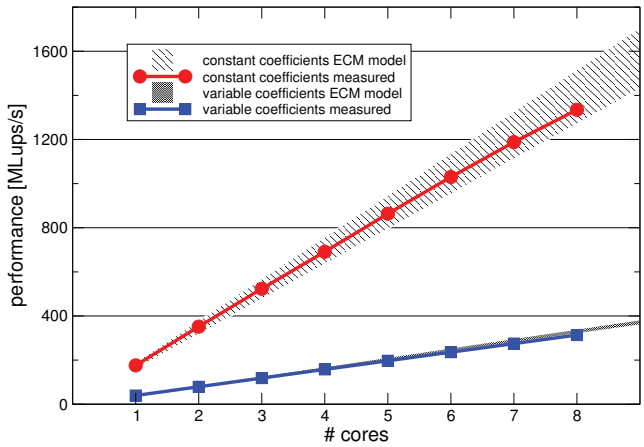

Figure 6: SNB single-chip performance scaling of the stencil smoothers on $\mathbf{256}^3$ grid points. Measured data and ECM prediction ranges are shown.

were obtained with the simple model are mainly caused by an over-optimistic assumption for the peak performance value that was based on FLOPS throughput alone. For the (CC) smoother, additional instruction overhead has to be considered. We remark that this is caused by the red-black pattern of updates that requires a high number of logistic machine instructions before vectorized operations can be used. In such cases, a thorough understanding of the performance depends on a careful code analysis. As demonstrated, this leads to a realistic prediction interval for the single-core performance.

We can additionally conclude that code optimization strategies that try to speed up the execution of a WU by reducing the memory bandwidth requirements, in particu-

lar cache blocking strategies, would fail. The above discussion demonstrates that — different form conventional wisdom — memory bandwidth is not the critical resource and thus an any code tuning aimed at this alleged bootleneck would not help to speed up the code. Code optimization would rather first have to focus on the memory layout that prohibits a more efficient vectorization of the code.

4.3. ECM performance model for variable coefficients (VC)

As for the constant coefficient kernel, there is a considerable deviation of the roofline model prediction of 79.4 MLups/s for one WU and actual performance measurement at 39.5 MLups/s. Here, the measurement immediately confirms the assumption that memory bandwidth is not the critical resource. We therefore again employ the ECM model with particular interest in refining our understanding of the instruction throughput limitation. In contrast to the roofline model, the ECM model considers the necessary data transfers through the memory hierarchy as well as run-time for code execution for estimating the single-core execution time. As a cache line ($64\,\mathrm{byte}$, 8 double precision elements) is the smallest unit of data that can be handled in the memory hierarchy, all input values to the ECM model have to be scaled to represent this size. Run-time contributions for data transfers have to be determined separately for each layer of the memory hierarchy.

For case (VC), as described in Section 2.1.2, we do not determine the execution time by manual code inspection, but we employ the Intel Architecture Code Analyzer (IACA, v. 2.01). In the (assembly) code, the dominant loop has to be marked and IACA determines (based on a simplified micro-architecture model of the processor) the number of cycles required for one iteration as well as the throughput bottleneck. 246 cycles for 4 updates (4-way SIMD) are reported, limited by the maximal throughput for load instructions. The reason for the difference to the 136 cycles that resulted from the manual code inspection is a lack of floating point registers. In summary, the processor cannot hold all required variables for one loop iteration in its registers. Therefore, register contents have to be repeatedly transferred to and from cache (register spill), resulting in excess load and store instructions. This is apparent from the assembly code. For a full update of one cache line size two iterations of the loop have to be accounted and the input value to the ECM model is therefore $2 \cdot 246 = 492$ cycles.

Next we analyze the data transfer between the levels of the memory hierarchy. Loading data from main memory to L3 cache is limited by the memory bandwidth of the system ($42\,\mathrm{GB/s}$, which is equivalent to $15.6\,\mathrm{B/cycle}$ at $2.7\,\mathrm{GHz}$). The L3 cache of SNB has 2.5 MB/core and is consequently large enough to hold at least two layers (in our setting one layer has a size of at most $256 \cdot 256 \cdot 8\,\mathrm{B} = 512\,\mathrm{kB}$) of the coefficient vector k and the solution vector u (*layer condition* - (relevant) previously loaded values can be reused from cache). Therefore, only one load stream is necessary per vector. In total, four load streams for cache lines are required: one load for k, u, f each, and one store to u. This costs additionally 16.5 cycles ($4 \cdot 64\,\mathrm{B}/15.6\,\mathrm{B/cycle}$) of data transfer time.

The *layer condition* is not fulfilled for the smaller caches L2 and L1. Therefore, two additional load streams for both, k and u, have to be accounted, resulting in 8 streams between adjacent caches. As $32\,\mathrm{bytes}$ can be transferred between caches per cycle, the transfer of one cache line takes 2 cycles. This translates to 16 cycles for 8 streams.

Fig. 5 illustrates the previously established run-times as input to the ECM model. According to our experience, they do not overlap for codes that are dominated by load instructions. Overall, run-time according to the ECM model is therefore the total aggregate (540.5 cycles), which is equivalent to a performance of 40.0 MLups/s using one core. Due to the massive dominance of core execution time, overlap of transfers would not have a significant influence, resulting in a narrow prediction range (cf. Fig. 6). As a consequence, the performance scales perfectly on the socket-level, since instruction throughput on each core is the limiting resource. The memory subsystem operates clearly below its capability and the performance values are at the lower limit of the predicted range, confirming the overlap hypothesis.

5. Towards parallel textbook multigrid efficiency

Based on the elaborate characterization of a WU for the system architecture under consideration, we can now proceed to define a new metric for the efficiency of parallel finite element solvers. Assuming that the performance of a single core is μ_{sm} Lups/s, then a perfectly scalable multiprocessor with U processor cores should ideally perform at $U\mu_{\text{sm}}$Lups/s. When employed for relaxing N discrete points, i.e., when performing one WU, we should ideally observe a compute time of

$$T_{\text{WU}}(N,U) = \frac{N}{U\mu_{\text{sm}}} \quad \text{(seconds)}.$$

As part of the analysis we will also quantify, how many WU are needed to solve a problem. If full textbook efficiency were achieved, we would expect that the total cost remains at less than 10 WU. As shown above, this alone is an ambitious goal, but we must now additionally account for the overhead of parallel execution. For a problem that requires a compute time of $T(N,U)$ seconds for N unknowns, when run on a computer with U processor cores, we define the *parallel textbook efficiency factor*

$$E_{\text{ParTME}}(N,U) = \frac{T(N,U)}{T_{\text{WU}}(N,U)} = T(N,U)\frac{U\mu_{\text{sm}}}{N}. \tag{5.1}$$

This is a metric where algorithmic efficiency (in the sense of Brandt's classical TME) is combined with implementation scalability.

In the following, we will evaluate this factor for a variety of problems. The performance μ_{sm} for one work unit are the ECM model estimations (i.e. $\mu_{\text{sm}} = 189 \cdot 10^6$ and $\mu_{\text{sm}} = 42.5 \cdot 10^6$ for the constant and variable case, respectively) from Table 3. We conduct all our experiments on SuperMUC, an IBM System x iDataPlex system with a peak performance of more than 3.2 petaflop/s, using the SNB processor architecture, and that is currently ranked as number 10 on the TOP-500 list (November 2013).

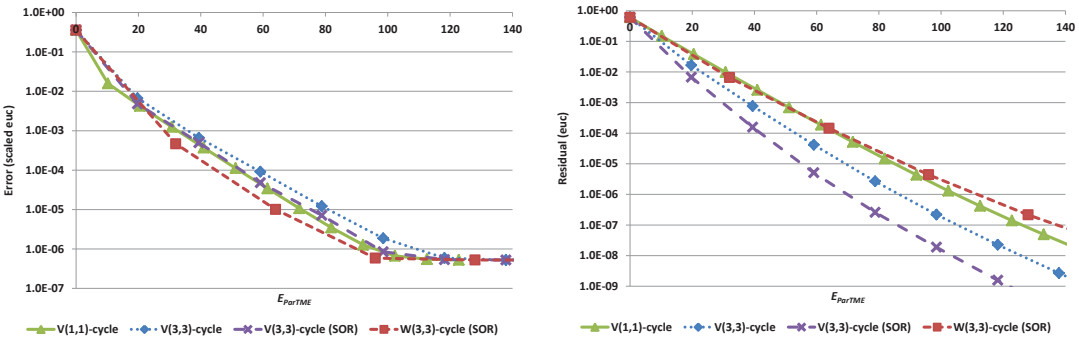

Figure 7: Error and residual for (CC) on a resolution of $8.6 \cdot 10^9$ grid points on $4\,096$ cores.

The performance data in Fig. 7, we reconsider different non-FMG cycles as in Section 3. Different, from the original E_{TME} definition, the E_{ParTME} factor is approximately worse by a factor of 2.5 compared to classical TME. We also point out that the coarsest mesh level is not yet accounted in the measurements, since it heavily relies on the coarsest level solver and number of input elements per core. In our current implementation and examples, this results in an additional overhead of around 15% per V-cycle.

Below we consider again the case of constant coefficients (CC) and the more expensive operator involving variable coefficients (VC). In the latter case the solution is kept identical to (CC), but the coefficient is chosen such that it smoothly varies, i.e.,

$$k(\mathbf{x}) = \sin(x_1 + x_2 + x_3) + 2. \qquad \text{(VC)}$$

Moreover, we consider the Stokes problem with the exact solution (SF) as an example for a system of equations. We assume that one WU of the Stokes system is equivalent to five WU for the case (CC), three for the velocity components, one for the gradient operator, and one for the divergence operator.

Table 4 lists the efficiencies for the (CC), (VC), as well as the (SF) case. In the latter case, we employ the FMG-4CG-1V(2,1) pressure correction cycle. All cycles are chosen such that they are as cheap as possible, but still keep the ratio $\gamma_l \approx 2$. We observe a larger gap between the E_{TME} and the new E_{ParTME} for (CC) and (SF) compared to the

Table 4: TME factors for different problem sizes and discretizations.

Setting/Measure	E_{TME}	E_{SerTME}	E_{NodeTME}	E_{ParTME1}	E_{ParTME2}
Grid points	-	$2 \cdot 10^6$	$3 \cdot 10^7$	$9 \cdot 10^9$	$2 \cdot 10^{11}$
Processor cores U	-	1	16	$4\,096$	$16\,384$
(CC) - FMG(2,2)	6.5	15	22	26	22
(VC) - FMG(2,2)	6.5	11	13	15	13
(SF) - FMG(2,1)	31	64	100	118	-

previous cases. We also observe the effect that traversing the coarser grids of the multigrid hierarchy is less efficient than the processing the large grids on the finest level. This is particularly true because of communication overhead and the deteriorating surface to volume ratio on coarser grids, but also because of loop startup overhead. When there are less operations, the loop startup time is less well amortized.

The case (VC) exhibits a better relative performance, since time is here spent in the more complex smoothing and residual computations. However, for the quality of an implementation it is crucial to show that the WU is implemented as time efficient as possible for a given problem. We remark that a good parallel efficiency by itself has limited value when the serial performance of the algorithm is slow. In this sense, multigrid may exhibit a worse parallel efficiency than other solvers while it is superior in terms of absolute execution time for a given problem and a given computer architecture.

E_{SerTME} and E_{NodeTME} are measured using only one core ($U = 1$) and one node ($U = 16$), respectively. We observe that the transition from one core to one node causes a similar overhead as the transition from one node to 256 nodes (E_{ParTME1}). For the (CC) case, roughly a factor of two lies between the TME and the serial implementation, and a further factor of two between the serial and the parallel execution. To interpret these results, recall that TME and ParTME would only coincide if there were no intergrid transfers, if the loops on coarser grids were executed as efficiently as on the finest mesh, if there were no communication overhead, and if no additional copy operations, or other program overhead occurred.

For the last setup E_{ParTME2}, we increase the number of grid points per core to the maximal capacity of the main memory. This also provides a better (lower) surface to volume ratio and consequently decreases the E_{ParTME} even for more compute cores.

The HHG multigrid implementation is already designed to minimize these overheads so that we can solve very large finite element systems. The largest Stokes system presented in Table 4 is solved on 4096 cores and has 3.6×10^{10} degrees of freedom. Note that though this compares favorably with some of the largest FE computations to date, it is already achieved on a quite small partition of the SuperMUC cluster that has in total 155,656 cores. For scaling results to even larger computations with more than 10^{12} degrees of freedom we refer to [18, 19].

6. Conclusion/outlook

In this paper, we extended the classical definition of TME to a more general concept taking into account the cost associated with the implementation on a parallel computer. In the paper, we have demonstrated that Brandt's seminal idea of TME can be used to effectively assess the efficiency of a multigrid algorithm. However, the classical notion of TME does not provide criteria that account for how fast a WU should be executed on a given architecture and it cannot predict the performance of an implementation on a massively parallel supercomputer. For this scenario we have developed in this article a refined TME paradigm. As central conclusion, we believe that in high performance scientific computing, the discretization, the solver algorithms, and their implementation

should be developed in a co-design process in which also the target computer architecture is taken into account. Thus, the algorithm development should be accompanied by a systematic and critical performance engineering process.

Our results demonstrate that very large FE systems can be solved in moderate compute times on current parallel architectures. Based on our experience, we believe that achieving a E_{ParTME} factor of 10 may not be realistic with current hard- and software systems. Currently we achieve a E_{ParTME} factor that is significantly below 20 for the scalar problems and slightly above 100 for the Stokes system. Taking unavoidable parallel overhead into account, we believe that these are respectable values. Our quantitative performance analysis shows that the HHG implementation is highly efficient, in the sense that it achieves a high percentage of the expected peak performance on the architectures considered. For this, we have introduced the advanced ECM-performance model and have carefully analysed what the critical resource is. In particular, HHG has reached a development stage that the potential for further code optimization is quite limited.

Our evaluation of E_{ParTME} factors as high as 100 for the Stokes system, however, indicates that compared to the classical results in [10] it should be possible to improve the efficiency of the current Stokes solver significantly. The potential, lies in developing better algorithms. Several approaches seem promising, e.g. by considering other types of smoothers, in particular distributive smoothers. These, however, will typically lead to more complex data access patterns which will then imply more complex communication requirements in a distributed memory system, and thus also increased parallel execution overhead. Further work on improving the Stokes solver should also include the construction of the discretization itself into the co-design process and should develop a meaningful way to assess the efficiency of a discretization in the parallel context. To this end, modern higher order finite element (FE) discretizations seem to be very attractive. With higher order approximations and in the case of smooth solutions, less unknowns might be required to achieve the same accuracy. However, it is far from obvious that the cost of computing a solution of prescribed accuracy is reduced for higher order schemes when compared to low order methods, since they may lead to denser linear systems than simple low oder discretizations. Thus the cost of the discretization may increase significantly, since constructing it via quadrature formulas and applying it (i.e., a WU in the sense of the TME) may become significantly more expensive.

At this point, we also remark that although computational cost will be increasingly dominated by data movement, we are far from a situation where FLOPS can be considered free. This will remain so for the foreseeable future, if only because FLOPS operate on data that must come from somewhere. For this see e.g. the discussion of register spills in Section 4 that shows how a high density of operations can indirectly lead to an overhead for data transfers (in the cache hierarchy). Hence, if the commonly used phrase *FLOPS are free* should mean more than an excuse for using underperforming (yet scalable) algorithms, we have to investigate further in the development of modern discretization and solver techniques that are suitable for modern architectures.

References

[1] M. F. Adams, H. H. Bayraktar, T. M. Keaveny, and P. Papadopoulos. Ultrascalable implicit finite element analyses in solid mechanics with over a half a billion degrees of freedom. In *ACM/IEEE Proceedings of SC2004: High Performance Networking and Computing*. IEEE Computer Society, 2004.

[2] Nathan Bell, Steven Dalton, and Luke N Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing*, 34(4):C123–C152, 2012.

[3] B. Bergen, F. Hülsemann, and U. Rüde. Is 1.7×10^{10} Unknowns the Largest Finite Element System that Can Be Solved Today? In *ACM/IEEE Proceedings of SC2005: High Performance Networking and Computing*. IEEE Computer Society, 2005.

[4] B. K. Bergen. *Hierarchical Hybrid Grids: Data structures and core algorithms for efficient finite element simulations on supercomputers*. SCS Publishing House eV, 2006.

[5] B. K. Bergen, T. Gradl, F. Hülsemann, and U. Rüde. A massively parallel multigrid method for finite elements. *Computing in Science and Engineering*, 8(6):56–62, 2006.

[6] B. K. Bergen and F. Hülsemann. Hierarchical hybrid grids: data structures and core algorithms for multigrid. *Numerical linear algebra with applications*, 11(2-3):279–291, 2004.

[7] J. Bey. Tetrahedral grid refinement. *Computing*, 55(4):355–378, 1995.

[8] M. Blatt, O. Ippisch, and P. Bastian. A massively parallel algebraic multigrid preconditioner based on aggregation for elliptic problems with heterogeneous coefficients. *arXiv preprint arXiv:1209.0960*, 2012.

[9] A. Brandt. *Barriers to achieving textbook multigrid efficiency (TME) in CFD*. Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, 1998.

[10] A. Brandt and O. E. Livne. *Multigrid Techniques: 1984 Guide with Applications to Fluid Dynamics, Revised Edition*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 2011.

[11] James Brannick, Yao Chen, Xiaozhe Hu, and Ludmil Zikatanov. Parallel unsmoothed aggregation algebraic multigrid algorithms on gpus. In *Numerical Solution of Partial Differential Equations: Theory, Algorithms, and Their Applications*, pages 81–102. Springer New York, 2013.

[12] F. Brezzi and J. Pitkäranta. On the stabilization of finite element approximations of the Stokes equations. In W. Hackbusch, editor, *Efficient Solutions of Elliptic Systems*. Springer, 1984.

[13] E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro, and U. Meier-Yang. A survey of parallelization techniques for multigrid solvers. In M. A. Heroux, P. Raghavan, and H. D. Simon, editors, *Parallel processing for scientific computing*, number 20 in Software, Environments, and Tools, pages 179–201. Society for Industrial and Applied Mathematics, 2006.

[14] C. Flaig and P. Arbenz. A scalable memory efficient multigrid solver for micro-finite element analyses based on CT images. *Parallel Computing*, 37(12):846–854, 2011.

[15] P. Ghysels and W. Vanrose. Modeling the performance of geometric multigrid on many-core computer architectures. Technical report, ExaScience Lab, Intel Labs Europe, Kapeldreef 75, 3001 Leuven, Belgium, 2013. Submitted.

[16] B. Gmeiner. *Design and Analysis of Hierarchical Hybrid Multigrid Methods for Peta-Scale Systems and Beyond*. PhD thesis, University of Erlangen-Nuremberg, 2013.

[17] B. Gmeiner, T. Gradl, F. Gaspar, and U. Rüde. Optimization of the multigrid-convergence rate on semi-structured meshes by local Fourier analysis. *Computers & Mathematics with Applications*, 65(4):694–711, 2013.
[18] B. Gmeiner, U. Rüde, H. Stengel, C. Waluga, and B. Wohlmuth. Performance and scalability of hierarchical hybrid multigrid solvers for Stokes systems. *SIAM Journal on Scientific Computing*, 2015, accepted.
[19] Björn Gmeiner, Harald Köstler, Markus Stürmer, and Ulrich Rüde. 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.
[20] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. M. Buijssen, M. Grajewski, and S. Turek. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Computing*, 33(10–11):685 – 699, 2007.
[21] G. Hager and G. Wellein. *Introduction to high performance computing for scientists and engineers*. CRC Press, 2010.
[22] Georg Hager, Jan Treibig, Johannes Habich, and Gerhard Wellein. Exploring performance and power properties of modern multi-core chips via simple machine models. *to appear in: Concurrency and Computation: Practice and Experience*, 2014.
[23] V. Heuveline, D Lukarski, N. Trost, and J.-P. Weiss. Parallel smoothers for matrix-based geometric multigrid methods on locally refined meshes using multicore CPUs and GPUs. In *Facing the Multicore-Challenge II*, pages 158–171. Springer, 2012.
[24] F. Hülsemann, M. Kowarschik, M. Mohr, and U. Rüde. Parallel Geometric Multigrid. In A. M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, number 51 in Lecture Notes in Computational Science and Engineering, pages 165–208. Springer, 2005.
[25] Harald Koestler, Daniel Ritter, and Christian Feichtinger. A geometric multigrid solver on gpu clusters. In *GPU Solutions to Multi-scale Problems in Science and Engineering*, pages 407–422. Springer Berlin Heidelberg, 2013.
[26] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.
[27] E. H. Mueller and R. Scheichl. Massively parallel solvers for elliptic PDEs in numerical weather- and climate prediction. *ArXiv e-prints*, July 2013.
[28] A. Neic, M. Liebmann, G. Haase, and G. Plank. Algebraic multigrid solver on clusters of CPUs and GPUs. In *Applied parallel and scientific computing*, pages 389–398. Springer, 2012.
[29] G. Romanazzi and P. K. Jimack. Parallel performance prediction for multigrid codes on distributed memory architectures. In *High Performance Computing and Communications*, pages 647–658. Springer, 2007.
[30] R. S. Sampath and G. Biros. A parallel geometric multigrid method for finite elements on octree meshes. *SIAM Journal on Scientific Computing*, 32(3):1361–1392, 2010.
[31] V. V. Shaidurov. *Multigrid methods for finite elements*, volume 318. Kluwer Academic Publishers (Dordrecht and Boston), 1995.
[32] H. Sundar, G. Biros, C. Burstedde, J. Rudi, O. Ghattas, and G. Stadler. Parallel geometric-algebraic multigrid on unstructured forests of octrees. In *ACM/IEEE Proceedings of SC2012: High Performance Computing, Networking, Storage and Analysis*, page 43. IEEE Computer Society, 2012.
[33] J. Treibig and G. Hager. Introducing a performance model for bandwidth-limited loop kernels. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors,

Parallel Processing and Applied Mathematics, volume 6067 of *Lecture Notes in Computer Science*, pages 615–624. Springer Berlin / Heidelberg, 2010.

[34] R. Verfürth. A combined conjugate gradient – multi-grid algorithm for the numerical solution of the Stokes problem. *IMA Journal of Numerical Analysis*, 4(4):441–455, 1984.

[35] C. Wieners. A geometric data structure for parallel finite elements and the application to multigrid methods with block smoothing. *Computing and visualization in science*, 13(4):161–175, 2010.

[36] S. W. Williams, A. Waterman, and D. A. Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical Report UCB/EECS-2008-134, EECS Department, University of California, Berkeley, Oct 2008.

[37] I. Yavneh. On red-black SOR smoothing in multigrid. *SIAM Journal on Scientific Computing*, 17(1):180–192, 1996.