Analysis and parallelization strategies for Ruge-Stüben AMG on many-core processors

P. Zaspel

Departement Mathematik und Informatik
Fachbereich Mathematik
Universität Basel
CH-4051 Basel

Preprint No. 2017-06
June 2017

www.math.unibas.ch

Journal of Scientific Computing manuscript No.
(will be inserted by the editor)

Analysis and parallelization strategies for Ruge-Stüben AMG on many-core processors

Peter Zaspel

Received: date / Accepted: date

Abstract The Ruge-Stüben algebraic multigrid method (AMG) is an optimal-complexity black-box approach to solve linear systems arising in discretizations of e.g. elliptic PDEs. Recently, there has been a growing interest in parallelizing this method on many-core hardware, especially graphics processing units (GPUs). This type of hardware delivers high performance for highly parallel algorithms. In this work, we analyse convergence properties of recent AMG developments for many-core processors and propose to use more classical choices of AMG components for higher robustness. Based on these choices, we introduce many-core parallelization strategies for a robust hybrid many-core AMG. The strategies can be understood and applied without deep knowledge of a given many-core architecture. We use them to propose a new hybrid GPU implementation. The implementation is tested in an in-depth performance analysis, which outlines its good convergence properties and high performance in the solve phase.

Keywords Ruge-Stüben algebraic multigrid (AMG) · Graphics processing units (GPUs) · Parallelization · Many-core · Iterative linear solvers · Graph traversal

Mathematics Subject Classification (2000) 65N55 · 65F10 · 65Y05 · 68W10 · 65F50 · 65N22

1 Introduction

The solution of sparse linear systems from discretized elliptic partial differential equations (PDEs) is the time-dominant component of many numerical simulations. Iterative solvers based on *multigrid* methods solve such systems in optimal (linear) complexity. Standard *geometric* multigrid often struggles with discretizations of PDEs in complex geometries. However, *algebraic* multigrid (AMG) is able

P. Zaspel
Departement für Mathematik und Informatik
Universität Basel
Spiegelgasse 1
4051 Basel, Switzerland
E-mail: peter.zaspel@unibas.ch

to handle this special case without further adoptions. AMG builds the multigrid *hierarchy* by a purely algebraic construction involving only the entries of the underlying system matrix.

Many-core processors achieve high performance by executing a very high number of concurrent computational threads. However, they are often limited by a relatively low per-thread performance and other simplifications. One example of many-core processors are *graphics processing units (GPUs)*. Another example are *Xeon Phi* processors. Nowadays, a series of high performance computing (HPC) cluster systems is equipped with many-core processors as *accelerators*. That is, many-core processors can be used in addition to the existing multi-core processors in a given compute node. To achieve high performance on many-core processors, it is necessary to develop numerical algorithms that use the provided rather extreme amount of parallelism. Here, it is often not enough to reformulate an existing algorithm in a smart way. Instead, the applied numerical method has to be changed to achieve high (parallel) performance. However, by changing the numerical method, the solution properties are affected. As a rule of thumb, we note that a higher degree of parallelism in a numerical method is often connected to weaker solution properties. Hence, we have to make a trade-off between (parallel) performance and strong solution properties.

This article analyses this kind of trade-off in recent work, e.g. [23], on the design and implementation of Ruge-Stüben algebraic multigrid [31, Appendix A] on many-core processors. We consider Ruge-Stüben AMG as main object of study, since it is well-known for its good convergence and robustness. Based on our analysis, we propose a different balance of algorithmic components and parallel performance than used, for many-core processors, before. For a practical realization, we propose many-core parallelization strategies for major parts of the method. This leads to a hybrid GPU-based Ruge-Stüben AMG implementation, for which we analyse the performance.

1.1 Related work

Whenever we discuss *algebraic multigrid* in this article, we actually refer to Ruge-Stüben AMG. For completeness, we note however that quite some work on non-Ruge-Stüben AMG types on many-core processors has been conducted in e.g. [4, 32, 10, 9, 6, 21, 18] for (smoothed) aggregation AMG and in [34] for auxiliary grid AMG.

Algebraic multigrid has a *setup phase* with the multigrid hierarchy construction and a *solve phase* with the iterative application of recursive multigrid cycles. The solve phase is almost identical in geometric and algebraic multigrid. Hence, the core of AMG lies in the setup phase. Traditionally, the execution of the setup phase has been considered to be purely sequential [31, Appendix A]. With the need of parallelization of AMG on multi-core clusters, a parallelization of the setup phase has been mainly achieved by applying the purely sequential parts of the setup phase, i.e. the *coarsening* or *C/F splitting*, onto partitions of the full system matrix and by adding some corrections at the interface [15, 35]. Unfortunately, this approach does not scale to the degree of parallelism provided by many-core processors. In fact, a single many-core processor roughly requires the parallelism that is normally distributed over a full cluster. This leads to a bad *domain-to-interface* ratio when applying the parallelization strategy for multi-core processors to many-

core processors. One very parallel coarsening approach, which shows acceptable performance even on many-core processors, is the PMIS (parallel maximum independent set) classification [35]. However, PMIS is not always robust. This is often overcome by some stronger interpolation techniques at higher computational costs [8]. The choice of suitable algorithmic components in the setup phase is therefore one of the crucial trade-offs that has to be made in the construction of AMG on many-core hardware. These trade-offs will be discussed in this work.

Maybe the first work on (Ruge-Stüben) AMG for many-core processors has been performed in [19,14]. In that work, the setup phase is based on matrix decomposition with boundary treatment. In [16,24], only the *solve phase* of AMG was parallelized on GPU and also used in a multi-GPU setting. This approach might not yet use the full performance of GPUs. More recently, AMG has become available in the open-source libraries *ViennaCL* [27] and *Paralution* [17]. In [33,28], performance results of *ViennaCL* have been discussed for AMG, however Ruge-Stüben AMG has still only been considered either with a multi-core CPU setup or with a single-threaded GPU setup. To the author's knowledge, details on the implementation of Paralution are not published. Two current state of the art commercial many-core (GPU) implementations are *GAMPACK* [11] and *AmgX* [23]. They provide a solve *and* setup phase that runs fully on GPUs by using PMIS. In their works [11] and [23], the authors provide an overview of their implementations and some excellent performance results when comparing CPU and GPU based AMG. To keep their competitive advantage, they can, however, not uncover the full details of their parallelization approaches. Moreover, their choice of test cases for convergence results is, understandably enough, targeted mainly at their customers' needs.

1.2 Contribution of this work

The objective of this work is two-fold. First, the aim is to complement the above works by a balanced discussion of the impact of the choice of the numerical components of AMG on the robustness and solution properties. To start this discussion, we introduce the components of AMG. We then discuss choices of AMG components that have been made for many-core AMG by authors in the past. Here, we especially focus on the commercial GPU implementations [11,23] as gold standard. Within these implementations we closely follow [23] and (numerically) study the convergence and robustness properties of methods used therein. Our numerical study uses modified versions of the *AMG2013 benchmark*, which is a stripped-down version of *BoomerAMG* [15] in *hypre* [12]. Within that study, we propose a set of parameters and components for AMG that we consider to be more robust than existing choices in many-core AMG.

Second, the aim is to propose a series of parallelization strategies that allow to run the major part of the AMG setup phase on many-core processors. This discussion shall give starters in the field of many-core programming an insight into the algorithmic challenges that have to be faced when dealing with AMG on many-core hardware. In order to understand the properties of the proposed algorithms, an implementation has been performed based on *CUDA* and a sparse linear algebra library for GPUs (*CUSP* [5]). This implementation is *hybrid* in the sense that the full solve phase and almost the full setup phase is done on GPU. The only part, which is left on CPU is a subset of the operations in the C/F

splitting. The new AMG solver runs on a single GPU together with one CPU core. As we will see, the resulting AMG implementation has strong convergence and robustness properties. A performance study will outline that the solve phase of our AMG runs clearly faster than a parallel version of *AMG2013*. However, the hybrid GPU setup phase struggles in competing with *AMG2013*. Those difficulties will be analyzed and explained individually. Finally, a real-world application with linear systems from a discretized PDE in a complex geometry will be shown.

This work is structured as follows. In Section 2, we review Ruge-Stüben AMG. Section 3 analyses the convergence and robustness properties of typical AMG constructions in GPU-based AMG. Section 4 introduces parallelization strategies for a robust hybrid many-core AMG and briefly discusses our concrete GPU implementation. Numerical results and performance measurements of our implementation are outlined in Section 5. Section 6 summarizes the outcome of this work.

2 Overview of the components of Ruge-Stüben algebraic multigrid

We will closely follow [31, Appendix A] to give a short overview of Ruge-Stüben AMG. Notation from [31] will be partially adapted. We want to solve linear systems

$$A\mathbf{x} = \mathbf{b} \tag{1}$$

with $A \in \mathbb{R}^{N \times N}$ a sparse symmetric M-matrix, and $\mathbf{x}, \mathbf{b} \in \mathbb{R}^N$. These linear systems arise e.g. in the approximate solution of elliptic PDEs by finite differences. In the following, will first discuss the two-level idea of multigrid and then the multigrid algorithm. This is followed by a summary of all algorithmically important components of AMG, namely C/F splitting, interpolation and smoothers.

2.1 Two-level idea

The two-level or two-grid algorithm contains all important properties of multigrid. To explain it, we start by identifying the original linear system as system on a *fine level* or fine grid, noting that we will use the terms *grid* and *level* equivalently. On the fine grid, we thus have

$$A_f \mathbf{x}^f = \mathbf{b}^f \quad \Leftrightarrow \quad \sum_{j \in \Omega^f} a_{ij}^f x_j^f = b_i^f \quad \forall i \in \Omega^f \,,$$

with $N_f := N$, $A_f := A$, $\mathbf{x}^f := \mathbf{x}$, $\mathbf{b}^f := \mathbf{b}$ and $\Omega^f := \{1, \ldots, N_f\}$. Next, we introduce the *smoother* $S_f \in \mathbb{R}^{N_f \times N_f}$. Starting from an approximation $P_f \approx A_f^{-1}$ to the inverse of the system matrix, it is defined as

$$S_f := (I_f - P_f A_f) \,.$$

I_f is the identity matrix. One step of the smoother generates from $\mathbf{x}^f$ a smoothed vector $\bar{\mathbf{x}}^f$ with

$$\bar{\mathbf{x}}^f := S_f \mathbf{x}^f + (I_f - S_f) A_f^{-1} \mathbf{b}^f \,. \tag{2}$$

The smoother has a structure, which avoids to invert the system matrix. It shall damp the highly oscillatory modes in the error $\mathbf{e}_f = \mathbf{x}^f - A_f^{-1}\mathbf{b}^f$. Examples for smoothers, i.e. concrete choices for P_f, will be given in Section 2.5.

Having damped highly oscillatory error modes on the fine level by the *pre*-smoother, the idea of the two-level method is to construct a *coarse level* or coarse grid on which the remaining low error modes are damped. The coarse level is constructed by introducing a splitting $\Omega^f = C \dot\cup F$, the *C/F splitting*, into coarse and fine grid variables. Fine grid variables will be kept on the next level, while the information of all variables will be transferred from coarse to fine grid by transfer operators. The linear system that is solved on the fine grid is given as

$$A_c \mathbf{x}^c = \mathbf{b}^c \quad \Leftrightarrow \quad \sum_{j \in \Omega^c} a_{ij}^c x_j^c = b_i^c \quad \forall i \in \Omega^c \,, \tag{3}$$

with the obvious choice of $\Omega^c := C$ such that that $A_c \in \mathbb{R}^{N_c \times N_c}$, $\mathbf{x}^c, \mathbf{b}^c \in \mathbb{R}^{N_c}$, $N_c := |C|$. The above equation is coupled to the fine grid problem by *restricting* the residual $\mathbf{r}^f = \mathbf{b}^f - A^f \bar{\mathbf{x}}$ on the fine level to the coarse grid and by using it as right-hand side. To transfer solutions from fine to coarse level, a *restriction operator* $I_f^c \in \mathbb{R}^{N_c \times N_f}$ is introduced. The opposite mapping is done by an *interpolation* or *prolongation operator* $I_c^f \in \mathbb{R}^{N_f \times N_c}$. Together, they define the coarse grid system matrix

$$A_c := I_f^c A_f I_c^f \,,$$

which we call the *Galerkin operator*. We usually set the interpolation matrix as transposed restriction matrix $I_f^c := (I_c^f)^\top$. In a pure two-level method, the coarse-level problem is solved directly. Its solution is transfered back to the fine grid by the coarse grid correction $\mathbf{x}^f = \bar{\mathbf{x}} + I_c^f \mathbf{x}^c$ and another *post*-smoothing step is applied. Iterating this approach leads to an iterative linear two-level solver.

In AMG, we call the construction of the C/F splitting, the construction of the prolongation and restriction, the construction of the Galerkin operator and the construction of the smoother the *setup phase*. The remaining part is the *solve phase*. It will turn out, that the setup phase is actually the algorithmically more challenging part.

2.2 Multigrid method

By carefully choosing all numeric ingredients it is possible to have linear complexity in the degrees of freedom for all operations on the fine grid during one iteration of the solve phase, cf. [31, Appendix A] for more details. However, directly solving the coarse problem (3) still has a computational complexity of $O({N_c}^3)$, leading to a non-optimal method. To overcome this issue, the algorithmic idea of pre-smoothing, coarse grid correction and post-smoothing can be extended such that the direct solve on the coarse grid is replaced by a recursive application of pre-smoothing, coarse grid correction and post-smoothing on the coarse level. Extending this to several levels leads to the so-called *V-cycle* in a *multigrid* method. Algorithm 1 summarizes the V-cycle. Here, we have introduced levels $l = 0, 1, \ldots, l_{max}$ with nested sets of variables $\Omega_0 \subset \Omega_1 \subset \ldots \subset \Omega_{l_{max}}$. Now, level l_{max} is the finest level Ω_f. Analogously we introduce per-level coarse/fine grid sets C^l/F^l, matrices A_l, vectors $\mathbf{x}^l$, $\mathbf{b}^l$, smoothers S_l, interpolation operators I_{l-1}^l and restriction operators I_l^{l-1}. In lines 3/4 and 9/10, the smoother is applied ν times. The construction of coarser levels is usually stopped as soon as the number of variables per level goes below a certain threshold and the coarsest grid is solved directly.

Algorithm 2 introduces AMG as iterative solver. Lines 2–7 correspond to the setup phase. Lines 8–12 describe a simple iterative solver where the function CYCLE

Algorithm 1 V-cycle in AMG

Require: complete multigrid hierarchy already set up
function CYCLE(l, $\mathbf{x}^l$, $\mathbf{b}^l$)
  if $l > 0$ **then** ▷ finer levels
    for $s = 1, \ldots, \nu$ **do**
      $\mathbf{x}^l = S_l(\mathbf{x}^l, \mathbf{b}^l)$ ▷ pre-smoothing
    $\mathbf{b}^{l-1} = I_l^{l-1}(\mathbf{b}^l - A_l\mathbf{x}^l)$ ▷ residual restriction
    $\mathbf{x}^{l-1} = 0$
    CYCLE($l-1$, $\mathbf{x}^{l-1}$, $\mathbf{b}^{l-1}$) ▷ solve on coarse grid
    $\mathbf{x}^l = \mathbf{x}^l + I_{l-1}^l\mathbf{x}^{l-1}$ ▷ update solution
    for $s = 1, \ldots, \nu$ **do**
      $\mathbf{x}^l = S_l(\mathbf{x}^l, \mathbf{b}^l)$ ▷ post-smoothing
    return $\mathbf{x}^l$
  else ▷ coarsest level
    $\mathbf{x}^0 = A_0^{-1}\mathbf{b}^0$ ▷ direct solve
    return $\mathbf{x}^0$

Algorithm 2 Iterative algebraic multigrid solver

Require: $A \in \mathbb{R}^{N\times N}$ symmetric M matrix
function AMGSOLVER(A, $\mathbf{b}$, ϵ_{res})
  $A_{l_{max}} := A, \Omega_{l_{max}} := \{1, \ldots, N\}$
  for $l = l_{max}, \ldots, 1$ **do** ▷ multigrid hierarchy setup
    $C^l \cup F^l = \Omega_l$, $\Omega_{l-1} := C^l$ ▷ C/F splitting
    construct I_{l-1}^l, $I_l^{l-1} := (I_{l-1}^l)^\top$ ▷ transfer operators
    $A_{l-1} := I_l^{l-1} A_l I_{l-1}^l$ ▷ Galerkin operator
    construct S_l ▷ smoother
  $\mathbf{x}_0 := 0, n := 0$
  repeat ▷ iterative solution
    $\mathbf{x}_{n+1}$ = AMG(l_{max}, $\mathbf{x}_n$, $\mathbf{b}$) ▷ multigrid cycle
    $\mathbf{r}_{n+1} = \mathbf{b} - A\mathbf{x}_{n+1}$
  until $\|\mathbf{r}_{n+1}\|_2 \leq \epsilon_{res}$
  return $\mathbf{x}_{n+1}$

corresponds to the launch of the *V-cycle* in Algorithm 1. Instead of applying the V-cycle in an iterative scheme, it could also be launched as preconditioner in a Krylov subspace solver, e.g. a conjugate gradient (CG) method. This second approach is often preferred due to faster convergence.

2.3 C/F splitting

The core idea of the algebraic multigrid method is to introduce the dual view of the linear system as a graph. Thereby, the C/F splitting or coarsening can be done on a graph instead of a geometry. The mapping between matrix and graph is done by considering the system matrix as adjacency matrix for a graph in which the variables are the nodes or points. Hence, weighted *connections* or *edges* between nodes exist if there is a non-zero non-diagonal entry in the system matrix A relating one variable to the other.

2.3.1 Sequential coarsening

The crucial sequential part of the Ruge-Stüben AMG method is the choice of coarse and fine grid points (C-/F-points), i.e. C/F splitting. It is based on the above discussed graph representation.

Algorithm 3 Standard coarsening algorithm

Require: level l
function AMGStandardCoarsening
  $F^l := \emptyset, C^l := \emptyset, U^l := \Omega^l$
  for $i \in U^l$ **do**
    $\lambda_i^l := \left|{S_i^l}^\top \cap U^l\right| + 2\left|{S_i^l}^\top \cap F^l\right|$
  while $\exists i$ s.th. $\lambda_i^l \neq 0$ **do**
    find $i_{\max} := \operatorname{argmax}_i \lambda_i^l$
    $C^l := C^l \cup \{i_{\max}\}$
    $U^l := U^l \setminus \{i_{\max}\}$
    for $j \in ({S_i^l}^\top \cap U^l)$ **do**
      $F^l := F^l \cup \{j\}$
      $U^l := U^l \setminus \{j\}$
    for $i \in U^l$ **do**
      $\lambda_i := \left|{S_i^l}^\top \cap U^l\right| + 2\left|{S_i^l}^\top \cap F^l\right|$
  return C^l, F^l

The neighborhood of a point/variable $i \in \Omega^l$ is given by

$$N_i^l := \left\{ j \in \Omega^l \middle| j \neq i, a_{ij}^l \neq 0 \right\} .$$

We say that a variable i is *strongly negatively coupled* to variable j if we have, for a fixed $0 < \epsilon_{str} < 1$, the relaxation

$$-a_{ij}^l \geq \epsilon_{str} \max_{a_{ik}^l < 0} |a_{ik}^l| .$$

All strong negative couplings of a variable i can be denoted by the set

$$S_i^l = \left\{ j \in N_i^l \middle| i \text{ strongly negatively coupled to } j \right\} .$$

We also need the set of all variables j which are strongly coupled to i. It is

$${S_i^l}^\top := \left\{ j \in \Omega^l \middle| i \in S_j^l \right\} .$$

The rough idea of C/F splitting is to create a rather uniform distribution of coarse and fine grid variables with fine variables being surrounded by coarse variables (in the graph). Good convergence is achieved, if fine grid points are strongly coupled to coarse grid points [31, Section A.7.1.1]. A first algorithmic idea [26,31] to achieve this, is as follows. One repeats the choice of coarse grid points until all points are classified as coarse or fine grid points. Whenever a coarse grid point has been chosen, all neighboring points, which are strongly coupled to the coarse point, are defined as fine grid points.

In real applications, this algorithmic idea is replaced by the *standard coarsening* algorithm [31, Section A.7.1.1], which is stated in Algorithm 3. This method also introduces sets of undecided variables U^l and importance measures λ_i^l. It results in a rather evenly distributed set of coarse grid points due to the indirectly imposed ordering by the weights λ_i^l.

2.3.2 Parallel coarsening

As sketched in Section 1, the standard idea to parallelize the setup phase and thus also the coarsening on multi-core processors is to apply a domain-decomposition idea. That is, the system matrix / graph is divided into subparts where the standard sequential coarsening is applied. Afterwards, the resulting splittings are connected by some boundary treatment, cf. [35,13]. However, it is well-known that the quality of the splitting gets affected if the ratio between internal points and points treated by the boundary correction becomes smaller. Applying multi-core coarsening on many-core processors is somewhat the extreme case of this situation. To be able to have a high utilization of the compute resources of a many-core processor, a large amount of threads have to be executed in parallel. Thereby, almost all points become boundary points. Consequently, it is not advisable to apply multi-core coarsening techniques on many-core hardware.

One exception to this rule is the parallel maximal independent set (PMIS) [35] coarsening. This algorithm can be parallelized on many-core processors without a degradation of the generated C/F splitting. However, PMIS, parallelized or not, often does not produce optimal splittings. To keep a relatively robust AMG implementation, PMIS usually is combined with strong, thus computationally expensive, interpolation [8] methods. Interpolation will be discussed in the next section.

2.4 Interpolation

Interpolation defines the transfer between information on different levels in AMG. As previously stated, the interpolation operator I_{l-1}^{l} transfers solutions on a coarse level to the next finer level. Since we use the transpose of the interpolation operator matrix as restriction matrix, it suffices to describe the interpolation operation and its construction.

We start by introducing some additional notation with

$$C_i^l := C^l \cap N_i^l, \quad F_i^l := F^l \cap N_i^l, \quad \bar{C}_i^l := C^l \cap S_i^l, \quad \bar{F}_i^l := F^l \cap S_i^l \,.$$

2.4.1 Direct interpolation

Direct interpolation uses the strongly coupled coarse grid points to interpolate to a given fine grid point. Thus, for each $i \in F^l$ we use the set of so-called *interpolatory variables* $P_i^l = \bar{C}_i^l$ and interpolate a given fine grid variable e_i^l by

$$e_i^l = \sum_{k \in P_i^l} w_{ik}^l e_k^l, \quad w_{ik}^l = -\alpha_i^l \frac{a_{ik}^l}{a_{ii}^l}, \quad \alpha_i^l = \frac{\sum_{j \in N_i^l} a_{ij}^l}{\sum_{k \in P_i^l} a_{ik}^l} \,.$$

We still assume here that the system matrix is an M-matrix. Therefore, we can neglect the handling of positive non-diagonal entries.

2.4.2 Standard interpolation

A much better convergence can be achieved by *standard interpolation*. It not only considers strongly connected coarse grid nodes but also includes strong connections between fine grid nodes. In order to describe this process, we have a look at a fine

grid point $i \in F^l$. The application of its corresponding matrix row to a vector $\mathbf{e}$ reads as

$$a_{ii}^l e_i^l + \sum_{j \in N_i^l} a_{ij}^l e_j^l \, .$$

To apply standard interpolation, we introduce a modified system matrix. There, we replace in those rows that are associated with a fine grid point $i \in F^l$, as above, the variables e_j with $j \in \bar{F}_i^l$, thus the strongly coupled fine grid points, as

$$e_j \longrightarrow - \sum_{k \in N_j^l} a_{jk}^l e_k^l / a_{jj}^l \, .$$

The newly generated matrix $\hat{A}_l$ has entries $\hat{a}_{ij}^l$ and possesses new neighborhood sets $\hat{N}_i^l$. By further setting for all $i \in F^l$ that $\hat{P}_i^l = \bar{C}_i^l \cup (\bigcup_{j \in \bar{F}_i^l} \bar{C}_j^l)$, we can define standard interpolation analogously to direct interpolation as

$$e_i^l = \sum_{k \in \hat{P}_i^l} \hat{a}_{ik}^l e_k^l, \quad \hat{w}_{ik}^l = -\hat{\alpha}_i^l \frac{\hat{a}_{ik}^l}{\hat{a}_{ii}^l}, \quad \hat{\alpha}_i^l = \frac{\sum_{j \in \hat{N}_i^l} a_{ij}^l}{\sum_{k \in \hat{P}_i^l} \hat{a}_{ik}^l} \, .$$

We thus extend direct interpolation to include the neighborhood of the strongly connected fine grid points.

2.4.3 Truncation

The more connections between variables are taken into account when constructing the interpolation operator, the better the interpolation. However, too many connections lead to rather densely populated interpolation operator matrices and thus to denser system matrices on coarser levels. This in turn might heavily affect the overall complexity of the multigrid method. Therefore, it is often necessary to introduce a *truncation* of the interpolation. One thus drops all entries related to connections with a strength smaller than the largest entry scaled by a threshold ϵ_{tr}. The resulting entries finally have to be rescaled accordingly.

2.5 Standard smoothers

We already introduced smoothers $S_l \in \mathbb{R}^{N_l \times N_l}$ to be an important numerical ingredient on a given level of the algebraic multigrid method. For $P := D_l^{-1}$, $D_l = \mathrm{diag}(A_l)$ we get the *Jacobi smoother* $S_l^J := I_l - D_l^{-1} A_l$. For a given point i its application reads as

$$\bar{x}_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j \right) \, .$$

Due to its trivial nature it can be very easily applied in parallel, even on many-core hardware. Furthermore, we can introduce a relaxed Jacobi iteration with the parameter $\omega \in \mathbb{R}$ and $P := \omega D_l^{-1}$, thus $S_l^{\omega J} := I_l - \omega D_l^{-1} A_l$, resulting in a similar iteration rule. By a proper choice of the relaxation parameter, the smoothing can be improved a lot.

A stronger smoother than Jacobi is the *Gauss-Seidel smoother* with $P_l := L_l^{-1}$ and L_l^{-1} the lower triangular part of A_l including all diagonal entries. It results in the smoother $S_l^{GS} = I_l - L_l^{-1} A_l$ which has the iteration rule

$$\bar{x}_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} a_{ij} \bar{x}_j - \sum_{j=i+1}^{N} a_{ij} x_j \right) . \tag{4}$$

By iteration rule (4) we can already see that this smoother is purely sequential. A way to overcome this is to introduce a coloring [20, Section 3.6.3] for the variables of the linear system, decoupling subsets of the variables which can be then treated in a parallel way. For more details on coloring for iterative linear solvers see e.g. [29, Section 12.4]. Note however, that colored Gauss-Seidel versions are not considered here.

3 Convergence and robustness in existing many-core AMG

In this section, we review and analyse the convergence and robustness impact of typical combinations of methods and parameters in many-core AMG. In addition, we propose a set of parameter and method choices, which still cover some of the many-core performance considerations while achieving better convergence and robustness results. The analysis will be based on three model problems, i.e. discretizations of a Poisson problem in two and three dimensions as well as discretizations of an elliptic problem in two dimensions with an anisotropic Laplace operator.

As discussed in the previous section, the choice of coarsening, interpolation and smoother can strongly impact the quality of the numerical results. Choosing the best combination for a given application is a challenge by its own. Clearly, reviewing the current state of the art in this field for arbitrary applications and all available techniques is out of scope. However, we here aim at discussing choices that have been made recently in context of GPU-based AMG. In this context, we focus on the two commercial GPU AMG implementations *AmgX* and *GAMPACK*. Since both implementations use similar techniques, we have chosen to discuss results with respect to *AmgX*, more specifically results based on [23].

In [23], the authors introduce *AmgX* and compare it to the *AMG2013* benchmark [1]. The latter is a stripped-down version of *BoomerAMG* in *hypre 2.9.0b*. *AMG2013* was introduced to perform cluster scalability studies with complex work loads. The benchmark uses a pre-designed application setup and a fixed set of coarsening, interpolation and smoother choices. While these choices seemingly are favorable for a benchmark study, they do not necessarily represent the best choices for convergence and robustness. In fact, *BoomerAMG* might converge much better for other parameter choices. This should be kept in mind when assessing the impact of our and other results. Moreover, note that, at time of writing this article, the current version of *hypre* is *2.11.2* and no longer *2.9.0b*. A comparison of a recent *hypre* version against *AmgX* has been made in [25], where performance improvements for *hypre* were reported.

3.1 Benchmark applications

We report on three benchmarks, which can be performed with the AMG2013 benchmark application. The benchmarks solve linear systems from discretized elliptic partial differential equations. The first benchmark is a Poisson problem in three dimensions with homogeneous Dirichlet boundary conditions,

$$\begin{aligned} -\Delta u &= f \qquad \text{in } \mathcal{D}\,, \\ u &= 0 \quad \text{on } \partial\mathcal{D}\,, \end{aligned} \tag{5}$$

which corresponds for $\mathcal{D} = (0,1)^3$ exactly to one important benchmark in [23]. The linear systems that we consider are obtained by discretizing the above problem by a standard second-order seven-point finite difference stencil on a uniform grid with mesh width $h := \frac{1}{N_\mathcal{D}+1}$, where $N_\mathcal{D}$ is the number of (inner) grid points in each space dimension. The right-hand side is set to $f \equiv \left(\frac{1}{h}\right)^2$. We abbreviate this benchmark by the name *Poisson3D*.

It is well known that the condition number of the resulting linear system matrix in Poisson3D scales like $O(h^{-2})$. Hence, it only depends on the mesh width in one dimension. This also means that, for a given amount of unknowns, the condition number of the system matrix for a discretization of a two-dimensional Poisson problem becomes much larger than the condition number in context of the three-dimensional problem for a fixed number of unknowns. Consequently, for a given number of unknowns, considering a two-dimensional problem should be more challenging than a three-dimensional problem. Therefore, we also include the two-dimensional test case *Poisson2D* based on (5) with $\mathcal{D} = (0,1)^2$ and a standard five-point stencil discretization into our considerations.

Moreover, we are interested in the robustness of the linear solver with respect to anisotropies. Therefore we consider a third benchmark, *Anisotropic2D* in which we discretize the elliptic PDE

$$\begin{aligned} -c_1 \frac{\partial^2 u}{\partial x_1^2} - \frac{\partial^2 u}{\partial x_2^2} &= f \qquad && \text{in } \mathcal{D}\,, \\ u &= 0 && \text{on } \partial\mathcal{D}\,, \end{aligned}$$

with anisotropy coefficient $c_1 \in \mathbb{R}$ on a two-dimensional domain $\mathcal{D} = (0,1)^2$. We again use a corresponding second-order five-point finite difference stencil and the same right-hand side.

3.2 Solver test cases

In our convergence studies, we mimic the AMG2013 benchmark and the examples picked in [23]. Therefore we solve the given linear systems $A\mathbf{x} = \mathbf{b}$ for an initial solution guess $\mathbf{x}_0 = \mathbf{0}$ and let the solver run until the relative residual $\|\mathbf{r}_i\|/\|\mathbf{b}\|$ of a given iterate $\mathbf{x}_i$ drops below 10^{-6}. AMG is used as preconditioner in a conjugate gradient solver. Throughout our convergence studies, we stop coarsening, when the current level has less than 10 unknowns. Moreover, we apply a truncation with $\epsilon_{tr} = 0.2$. Whenever applicable, we further set the strength parameter to $\epsilon_{str} = 0.25$.

We use three different sets of parameters for algebraic multigrid. These parameters are summarized in Table 1. The first set of parameters is identical to

	AMG2013 param.	*AmgX* param.	parameter proposal
coarsening	PMIS	PMIS	standard coarsening
aggressive coarsening	first level	no	no
interpolation	extended	standard	standard
smoother	L1-Gauss-Seidel	Jacobi ($\omega = 0.8$)	Jacobi ($\omega = 0.8$)

Table 1 We compare convergence and robustness properties of three sets of parameters for AMG (*AMG2013* benchmark, *AmgX* parameters in [23], our own parameter set proposal).

the settings considered in the AMG2013 benchmark: The coarsening is done using PMIS. Aggressive coarsening, cf. [36], with appropriate interpolation is use on the first level. On all other levels, extended interpolation, cf. [8], is applied. The pre- and post-smoother is always an L1-Gauss-Seidel smoother.

Our second set of parameters shall reflect the settings chosen in [23] for *AmgX*. Unfortunately, [23] does not give all the details about their applied parameters. Therefore, we tried our best to reverse-engineer these parameters using the same software, hardware (Titan cluster), a similar build environment, etc. The parameters reported in Table 1 seem to fit best the results in [23]. That is, we use PMIS as coarsening without aggressive coarsening on the first level. Moreover, we apply standard interpolation and use a (twice applied) relaxed Jacobi smoother with relaxation parameter $\omega = 0.8$.

The final set of parameters is our own proposal of parameters, which is more close to the classical way to build AMG. That is, we choose standard coarsening, no aggressive coarsening on the first level, standard interpolation and a relaxed Jacobi smoother with relaxation parameter $\omega = 0.8$. As we will see in our results, the convergence properties for this approach clearly outperform the *AmgX* parameters and are at least as good as the *AMG2013* parameters. We will argue that these parameters show good convergence properties and robustness with respect to anisotropies. Meanwhile, the Jacobi smoother still allows for a very high parallelism in the solve phase. The crucial difference to *AMG2013* will be the choice of a mainly sequential AMG setup. However, as we will see, there is still a lot of room for a many-core parallelization for major pieces of the setup phase while keeping the only truly sequential part of the standards coarsening on CPU.

Note that we perform all convergence studies in this chapter with hand-modified versions of the *AMG2013* benchmark. Hence, we use a reliable, well tested CPU-based code to do the comparison of the different choices of parameter sets. We even use the same hardware as in [23], i.e. we run our studies on the compute nodes of the cluster *Titan* at Oak Ridge National Lab. The Titan cluster is a Cray XK7 system. Each node is equipped with a 16-core 2.2GHz AMD Opteron 6274 processor and 32 GB of RAM. We used the *GCC 4.9.3* compiler and *MPICH 7.5.2* and compiled *AMG2013* and the hand-modified versions with compiler flag `-O3`. Within our study, similar to [23], we also compare the convergence properties between benchmarks running sequentially on one node and benchmarks being distributed to 8 cores by the MPI-parallelization within *AMG2013* or *hypre*, respectively. As we will see, the use of a parallelized setup phase will have an impact on the setup and solve phase.

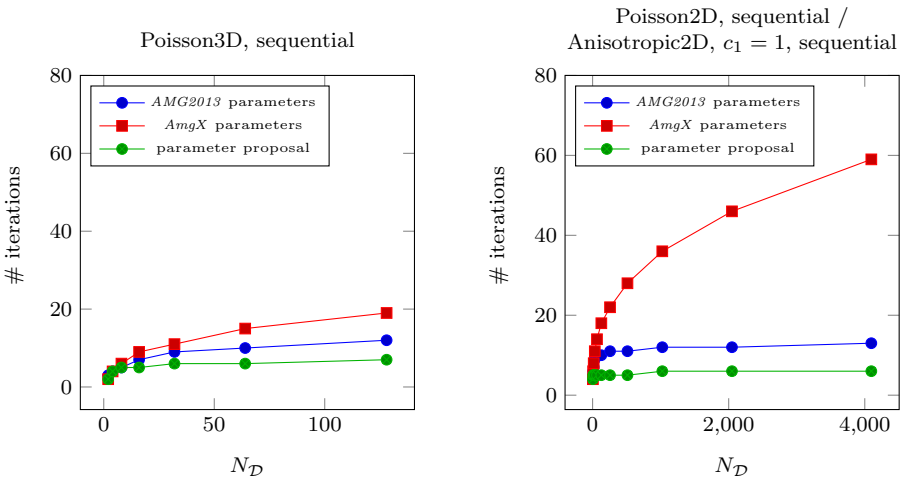

Fig. 1 In some cases, we observe significant differences in the convergence behavior for the three solver parameter settings in the Poisson3D application problem (*left*) and the Poisson2D application problem (*right*).

3.3 Numerical results

In the following, we discuss the convergence results that we obtain with the above given parameters. We always report the number of iterations of the solver with respect to the parameter $N_{\mathcal{D}}$, i.e. the discretization parameter used above for defining the finite difference discretization. This parameter is not identical to the number of unknowns which is $N_{\mathcal{D}}^3$ and $N_{\mathcal{D}}^2$ in Poisson3D and Poisson2D/Anisotropic2D, respectively.

On the left-hand side of Fig. 1, we show the results for the Poisson3D benchmark running sequentially for the different parameters sets defined before. This benchmark is shown in [23]. We observe a growing number of iterations for growing problem size. Hence, all benchmarks seem not to be in an asymptotic regime, where we would expect to see constant or at least very slowly growing number of iterations. The general tendency is that our parameters outperform the parameters of *AMG2013*. The worst results are seen for the *AmgX* parameters.

In contrast, the results in Fig. 1, on the right-hand side, are in the asymptotic regime. For the *AMG2013* parameters and our parameter proposal, we observe almost constant iteration counts when going to finer discretizations with an asymptotically high condition number. In contrast, the *AmgX* parameters do not result in the AMG-type optimal convergence. The more expensive extended interpolation in connection with the Gauss-Seidel-type smoother seem to recover the optimal convergence of AMG in presence of the less strong PMIS coarsening for *AMG2013*.

The results on the right-hand side of Fig. 1 together with the two results in Fig. 2 form the robustness study Anisotropic2D for growing anisotropies $c_1 = 1, 10, 100$. The tendency of the results fits together with the results obtained in the previous benchmarks. The parameters from *AMG2013* and our parameter proposal show similar convergence results. Both parameters are almost perfectly

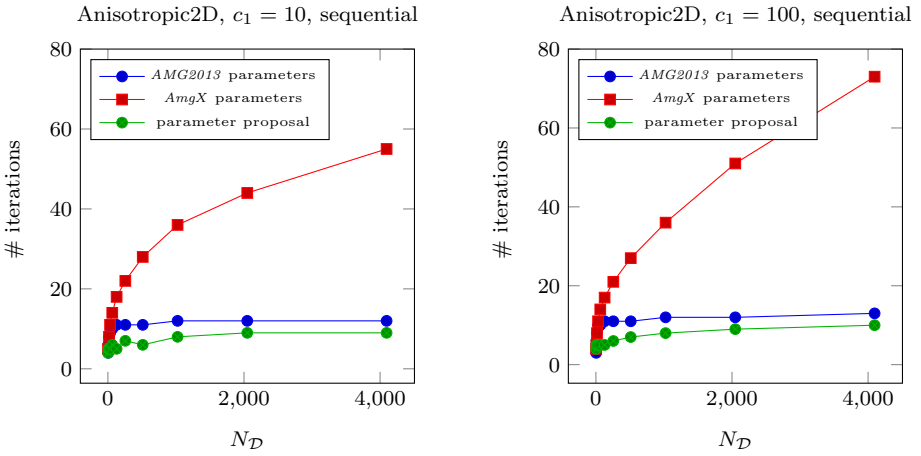

Fig. 2 The (potential) parameters of *AmgX* in [23] lead to a strong dependence on the anisotropy parameter comparing $c_1 = 1$ (Fig. 1, *right*), $c_1 = 10$ (*left*) and $c_1 = 100$ (*right*). That is, AMG is not robust with respect to anisotropies in this case.

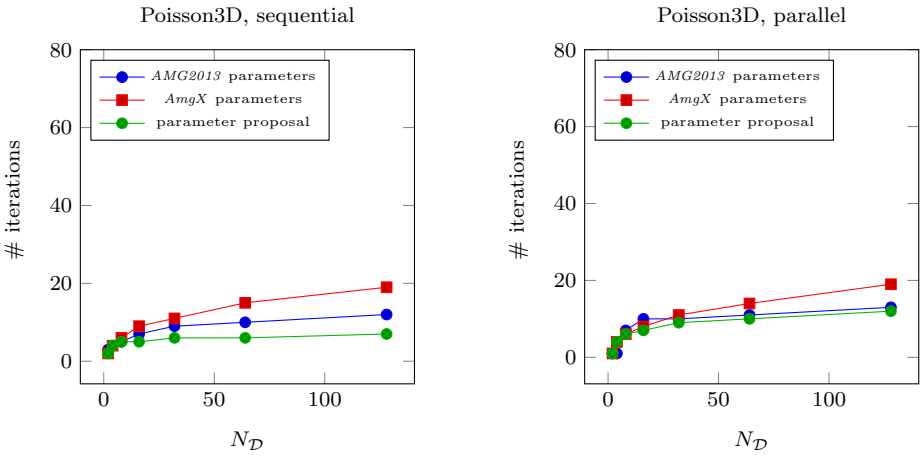

Fig. 3 When running *AMG2013 / hypre* in parallel, the coarsening in our parameters get degraded leading to a worse solver performance.

robust with respect to the anisotropy. On the other hand, the *AmgX* parameters are not robust with respect to anisotropy.

Finally, we also compare, similar to [23], convergence results for the sequential use of *AMG2013 / hypre* (as before) with results obtained when distributing the work over eight MPI processes. Fig. 3 highlights these results comparing the sequential results for Poisson3D on the left with the parallel results for Poisson3D on the right. It becomes obvious that the PMIS-based parameter sets converge almost identical in the sequential and in the parallel case. In contrast, the parallelization of the standard coarsening requires corrections at the interface of the decomposed domains. Thereby performance results degrade in the parallel case, still achieving similar convergence as with the *AMG2013* parameters.

3.4 Discussion

As expected, our results show that the best possible choice of parameters is not perfectly clear, even in our minimalistic benchmark. The standard parameters in *AMG2013* always produce almost problem-size independent convergence, are robust with respect to anisotropies and seem to be robust with respect to an increase in the number of processors for the parallel setup phase. The good parallel properties might be the reason why the authors used these parameters for the parallel benchmark *AMG2013*, even though the classical parameters in *BoomerAMG/hypre* are much more similar to our proposed parameter set. However, the *AMG2013* parameters use the sequential Gauss-Seidel-type smoother and rather expensive extended interpolation.

The *AmgX* parameters, keeping in mind that we don't know the exact parameters and just reverse engineered them, seem to be not robust to anisotropies and seem to show a growing number of iterations for larger problem sizes. However, since a Jacobi smoother is used, the solve phase will run very efficiently on many-core hardware. The setup phase will be efficient due to PMIS. This is also reported in [23], where the authors show impressive performance results comparing *AmgX* to *AMG2013*. That is, with the parameters of *AmgX* used in [23], the convergence and robustness properties of the method are lowered at the gain of a pre-asymptotically fast solution behavior. This is a typical balancing done for numerical methods for many-core processors.

Our proposed parameters shall introduce a third flavor. That is, we aim for best (asymptotic) convergence rates, still seeking for a method that will have a very parallel and efficient solve phase. However, achieving a good many-core performance in the setup phase, when using standard coarsening and standard interpolation, is challenging. In fact, parallelizing some part of the coarsening is impossible. However, as we will show, it is still possible to parallelize a major part on the setup phase even if the coarsening is done on CPU.

4 Hybrid many-core parallelization strategies

In this section, we discuss algorithms and parallelization strategies for AMG on many-core hardware. These algorithms are generic in terms of the target many-core hardware. That is, their efficient implementation shall not be restricted to GPUs but should also be possible for, e.g., Xeon Phi processors. To this end, the applied amount of hardware-specific functionalities is reduced to a minimum. We do this with an educative objective: We want to lower the bar for beginners in many-core computing to be able to write scientific codes and to focus more on the core idea in many-core parallelization, i.e. exposing a high degree of parallelism.

We first give an overview about details of our general strategy to do a many-core parallelization. Thereafter, we explain the general methodologies of sparse matrix construction and local graph traversal. This general consideration allows to simplify the discussion of parallelization details for the setup phase with the (hybrid) C/F splitting and the interpolation operator construction, afterwards. Moreover, we address the parallelization of the V-cycle and the smoothers. Finally, details of the concrete implementation on GPU are discussed.

4.1 General many-core parallelization strategy

Our general base strategy for many-core parallelization is to move a big portion of the parallelization complexity to libraries. In the context of the given application (AMG) this means that we use libraries implementing a standard set of routines for (sparse) linear algebra (sparse matrix formats, matrix–matrix products, matrix–vector products, scalar products, ...). We claim that such libraries exist with the (incomplete) list of examples containing *CUSP*, *CUSPARSE*, *LAMA*, *ViennaCL*, *Paralution*, *GHOST*, *MAGMA*, *MKL*. Similarly, we use libraries, which implement STL vector algorithm type methods in parallel, such as SUM, EXCLUSIVESCAN or MAXIMUM. Again, several libraries allow this, including but not being limited to *Thrust*, *ArrayFire* and *Boost.Compute*.

Only if we are lacking appropriate library support, we actually propose to implement new many-core parallel functions. We call such functions *kernels*. In addition to the function parameters, we pass the amount of parallel threads on which the kernel executes its code. Within the kernel code, the thread index is fetched first. Afterwards, computations are done relative to that thread index, cf. Algorithm 5 for an example. Writes into the many-core processor's memory are not considered to be consistent / thread-safe during the kernel execution. That is, while executing a kernel, two different threads shall not write into the same memory location. The only exception are *atomic* operations. They serialize conflicting parallel writes, if necessary. However, this might impact performance. A global synchronization over all threads is done at the end of the kernel execution. Note again that these assumptions are strong simplifications of the usually much more elaborated memory and thread execution models used in recent many-core processors. However, they make it much more easy to introduce many-core algorithms.

4.2 CSR matrix construction

We store matrices in the *compressed sparse row* (CSR) matrix format. It encodes a sparse matrix $A \in \mathbb{R}^{N \times N}$ with $N_{nonzero}$ non-zero entries by arrays

- *row_offsets* of indices (of length $N + 1$) describing the offset to take in *column_indices* and *values* to find entries of a given matrix row,
- *column_indices* of indices (of length $N_{nonzero}$) describing the column index of a given row entry and
- *values* of (double precision) floating point values (of length $N_{nonzero}$) describing the respective matrix entry .

Throughout the construction of the interpolation operators, we have to assemble sparse CSR matrices. This can be rather easily done in parallel, when using one parallel thread per row of the matrix that shall be filled. Algorithm 4 shows the necessary steps. First, a generic function COUNTENTRIES mimics the work that would be done for generating the matrix entries. This is done in a completely parallel way. While doing this, instead of writing the data into the new matrix, the number of generated non-zero entries per row are counted and stored in array *num_entries_per_row*. Then, a parallel sum and a parallel exclusive scan operation allow to compute the total number of non-zeros and the offsets for the CSR matrix data structure. Finally another generic function FILLCSRMATRIX does the actual work and fills the system matrix as simulated in COUNTENTRIES, before.

Algorithm 4 Generic method to construct CSR matrix in parallel

```
1: function buildCsrMatrix(A, N, ...)
2:     allocate num_entries_per_row[N]
3:     setValue⟨N⟩(num_entries_per_row,0)
4:     countEntries⟨N⟩(num_entries_per_row, ...)        ▷ count number of entries per row
5:     total_entries := sum⟨N⟩(num_entries_per_row)     ▷ compute number of non-zeros
6:     allocate num_rows[N + 1], column_indices[total_entries], values[total_entries]
7:     exclusiveScan⟨N⟩(num_entries_per_row, row_offsets)        ▷ compute offsets
8:     fillCsrMatrix⟨N⟩(num_rows, column_indices, values, ...)   ▷ fill matrix
```

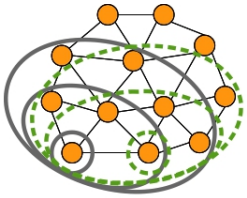

Fig. 4 In the local graph traversal, the next neighbors of each node are visited up to a fixed depth (indicated by the different ellipse sizes). The many-core parallelization makes use of the independence of each traversal (indicated by the different line types).

Even though the above approach actually requires to compute the sparse matrix twice, it is still a very effective way to construct a CSR matrix in parallel.

4.3 Local graph traversal up to a fixed depth

Several algorithms within the AMG setup phase require to iterate over neighbours and/or neighbours of neighbours of a node in the graph representation of the linear system. This can be translated to a local graph traversal up to a depth of one/two starting from each node in the graph, cf. Fig. 4. In the following, we formulate a generic many-core parallel local traversal up to depth two for a graph that is given by an adjacency matrix stored in the CSR format.

The easiest way to parallelize this embarrassingly parallel local graph traversal is by parallelizing it over the starting nodes for each of the traversal steps. Algorithm 5 formalizes this idea. It is launched in parallel with N threads and a sparse matrix $A \in \mathbb{R}^{N \times N}$ given as parameter. In each thread, first the current thread index idx is retrieved. Then, each neighbour of node idx is visited by iterating over all non-zero off-diagonal elements in the idxth matrix row. The same idea is repeated to iterate over all neighbours of a visited neighbour node. Clearly, Algorithm 5 is not the necessarily most efficient parallelization approach for a specific many-core architecture. To give an example, an implementation of the algorithm in the *CUDA* programming model and running on GPUs of *Nvidia*, might not achieve the fastest possible performance due to thread divergence. Here, e.g. [22] gives an excellent overview over optimization techniques. However, Algorithm 5 is very generic and delivers a high degree of parallelism for a lot of many-core architecures. Still, optimized graph traversal algorithms are considered future work.

Algorithm 5 Generic kernel for local graph traversal up to depth two

```
1: function TRAVERSE⟨N⟩(A, N, ...)
2:     idx := GETTHREADINDEX()
3:     for jj ∈ [A.row_offsets[idx], A.row_offsets[idx + 1] − 1] do
4:         j := A.column_indices[jj]                  ▷ get column index of current matrix entry
5:         if (idx ≠ j) AND ... then
6:             ...                                    ▷ do something with weight on edge (idx, j)
7:             for kk ∈ [A.row_offsets[j], A.row_offsets[j + 1] − 1] do
8:                 k := A.column_indices[kk]
9:                 if (j ≠ k) AND ... then
10:                    ...                            ▷ do something with weight of edge (j, k)
```

4.4 Hybrid C/F splitting

We use the C/F splitting following Algorithm 3. The coarsening algorithm is the only part of the code which is only partially parallelized on many-core hardware.

Steps three and four of Algorithm 3 compute the strong influence weights λ_i^l for the current level. This operation can be parallelized in a many-core fashion. Here, the first necessary step is to pre-compute the maximum (negative) non-diagonal entry for each matrix row, thus, $\max_{a_{ik}^l<0} |a_{ik}^l|$. The parallel graph traversal idea of Section 4.3 is applied for this. Using these maxima, it is possible to identify strongly coupled nodes in an algorithm for evaluating the λ_i^l. The weight evaluation algorithm again traverses the graph up to a depth of one. Whenever a strongly influencing node is found, the appropriate weight λ_i^l is increased by an atomic operation.

The remaining part of Algorithm 3 is implemented sequentially including the necessary copy operations between sequential processor (CPU) and many-core processor. To achieve an optimal computational complexity, the lookup of the largest weight λ_i^l in step 6 of the algorithm is performed with a priority queue data structure [7, Section 6.5]. Since e.g. the STL-based implementation of a priority queue does not allow to have a constant-complexity maximum-find and -removal operation (in fact that implementation has a logarithmic complexity in that case), the data structure is hand-implemented. The proposed implementation uses bucket sort [7, Section 8.4] as base algorithm and allows to achieve the designated constant complexity removal operation with a linear-complexity data structure setup time. Overall this keeps a linear complexity for the standard coarsening algorithm.

Finally, while skipped in Algorithm 3, it is necessary to identify the mapping between the newly found coarse grid points and their position in the next coarser grid level. This can be implemented by means of a many-core parallel STL vector algorithm-type operation. Fig. 5 shows an example for the approach. A given array *coarse* contains — per node — the flag whether the node will become a coarse grid node. By applying an EXCLUSIVESCAN operation to that array, an enumeration of the coarse grid nodes is generated. As long as one accesses only those entries in *fine_to_coarse* that correspond to a new coarse grid node, the correct mapping will be returned.

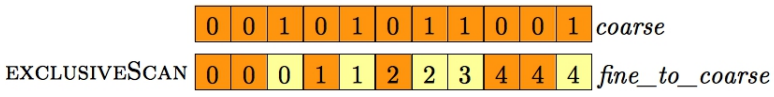

Fig. 5 An exclusive scan operation allows to compute the mapping between a coarse node on the current and next multigrid level. Only entries with light background will be accessed.

$$A_l = \begin{pmatrix} 1.00 & -0.25 & & \\ -0.25 & 1.00 & -0.25 & \\ & -0.25 & 1.00 & -0.25 \\ & & -0.25 & 1.00 \end{pmatrix} \rightarrow \begin{pmatrix} f & t & & \\ t & f & f & \\ & f & f & t \\ & & t & f \end{pmatrix}$$

Fig. 6 In the standard interpolation operator construction, we reuse the existing sparsity pattern of A_l to store the set of interpolatory points $P_i^l = \bar{C}_i^l$ via a boolean indicator (t/f).

4.5 Interpolation operator construction

4.5.1 Direct interpolation

The many-core parallel implementation of direct interpolation starts by pre-computing the intermediate coefficients

$$\alpha_i^l = \frac{\sum_{j \in N_i^l} a_{ij}^l}{\sum_{k \in P_i^l} a_{ik}^l} \, .$$

To build the two sums, parallel local graph traversal up to a depth of one becomes necessary. The core idea for this has been discussed in Section 4.3. The weights w_{ik}^l are computed while building the interpolation matrix following the ideas in Section 4.2. The positions of the interpolation coefficients in the finally constructed matrix I_{l-1}^l are determined by the lookup table created at the end of the C/F splitting as outlined in Section 4.4. Overall the direct interpolation matrix is constructed fully in parallel.

4.5.2 Standard interpolation

Implementing standard interpolation for many-core processors is more involved. Here, the algorithm starts by finding the set of interpolatory variables $P_i^l = \bar{C}_i^l$ for direct interpolation. Since this is necessary for each row, a sparse matrix is constructed to store this information. We reuse the existing data structure of the sparse system matrix and replace the array of non-zeros by boolean indicator values as shown in Fig. 6. The boolean values (t/f) store, whether a given neighbor node is part of the interpolatory set or not. Filling this indicator matrix can be done by parallelization over the number of rows of the system matrix similar to the local graph traversal in Section 4.3

Next, the original system matrix is modified such that strongly coupled fine grid points are expanded as

$$e_j \longrightarrow - \sum_{k \in N_j^l} a_{jk}^l e_k^l / a_{jj}^l \, .$$

Note, that this construction is rather compute intensive and might be replaced by a more sophisticate method in the future. It requires e.g. to implement a sparse

matrix row addition. The new set $\hat{P}_i^l = \bar{C}_i^l \cup (\bigcup_{j \in \bar{F}_i^l} \bar{C}_j^l)$ of interpolatory variables is computed, as well. Finally the algorithm reproduces the direct interpolation implementation now using the expanded system matrix and $\hat{P}_i^l$.

Truncation in the interpolation matrix can also be done in a many-core parallel way. This is done within the construction algorithm for interpolation matrices, to avoid building new sparse matrices. However, this algorithm is also available as stand-alone method.

4.6 V-cycle and smoothers

Due to the (assumed to be) available linear algebra primitives, the implementation of a V-cycle on many-core hardware is straight-forward, since the V-cycle only contains standard matrix-matrix, matrix-vector and vector-vector operations, cf. Algorithm 1. Within the V-cycle of Algorithm 1, we replace the direct solver on the coarsest level by a Jacobi-preconditioned CG solver.

For now, we only use a relaxed Jacobi iteration as smoother. The construction and application of the Jacobi smoother can be easily realized by the available linear algebra primitives

4.7 Implementation on GPU

Our concrete implementation of algebraic multigrid for a single GPU is based on the sparse linear algebra library *CUSP* [5] in version 0.4.0 and the STL Vector algorithm library *Thrust*, which comes with the *CUDA Toolkit 7.5*. The first library provides a set of linear algebra primitives (matrix-matrix products, matrix-vector products, ...), iterative solvers (CG, GMRES, ...) and preconditioners (Jacobi, ...) for sparse matrices of different sparse matrix formats, including CSR. *Thrust* provides operations such as SUM or EXCLUSIVESCAN. The implemented code integrates withing the *CUSP* framework, thus the new Ruge-Stüben classical AMG can be used as preconditioner for the standard *CUSP* solvers. It can be furthermore applied as stand-alone solver. Compute kernels are implemented as *CUDA* kernels based on the *CUDA Toolkit 7.5*.

Note that we do not use the latest *CUSP* version due to issues with the matrix-matrix product implementation. Moreover, we did not consider to use the more recent *CUSPARSE* library instead of *CUSP*, since, at the time of starting this software project, *CUSPARSE* was still at a very early development stage. Finally, the at time of writing this article latest *CUDA Toolkit* version 8.0 is currently not supported on our target benchmark system.

5 Performance of the hybrid GPU AMG

In the following, we discuss convergence and performance results of our hybrid GPU AMG. First, we double-check the convergence and robustness of our code by repeating the convergence studies of Section 3 with our specific implementation. The remaining part of the section is a performance benchmark. To this end, we first introduce our benchmark setup in terms of hardware and software. Afterwards a general performance comparison between our implementation and *AMG2013* (following [23]) is given. As we will see, the solve phase is fast, however our expectations in terms of speedup for the setup phase on GPU are not met.

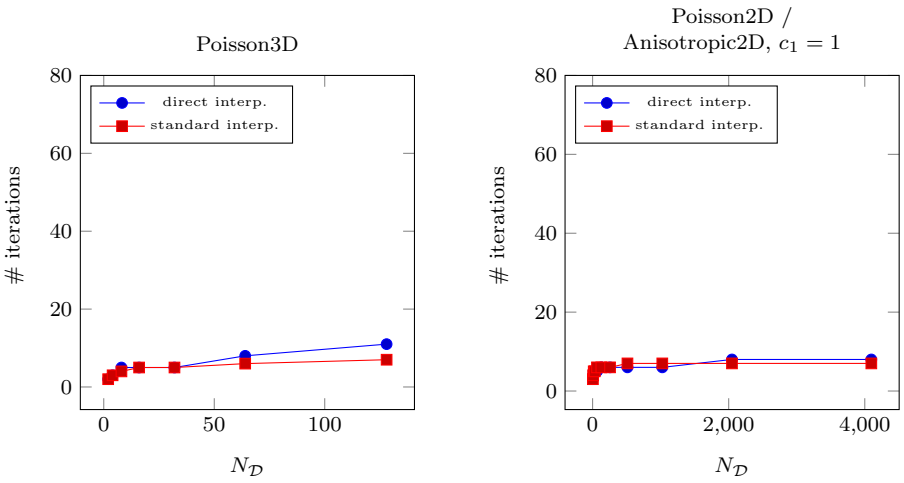

Fig. 7 Our hybrid GPU implementation of AMG attains the predicted convergence behavior for our proposed parameter set when using standard interpolation. This is shown here for the Poisson3D (*left*) and the Poisson2D (*right*) test case. Direct interpolation shows slightly weaker convergence results.

Therefore, we further give a detailed performance analysis and discussion for the setup phase. Finally, we show results for a real-world test problem on a complex geometry.

5.1 Convergence and robustness check

We repeat the convergence studies performed in Section 3, i.e. Poisson3D, Poisson2D and Anisotropic2D, for the proposed hybrid GPU AMG implementation. Hence, we show results for the parameter set that was proposed in Section 3.2, but now for our own implementation. The objective of this study is to double-check that our implementation is correct and that it is not affected by any parallelization issue.

As discussed in Section 3.2, we use standard coarsening, standard interpolation and a twice applied relaxed Jacobi pre- and post-smoother with relaxation parameter $\omega = 0.8$. Truncation is used with $\epsilon_{tr} = 0.2$ and the strength parameter is $\epsilon_{str} = 0.25$. The recursive construction of coarser levels is stopped as soon as a given level has less than 100 unknowns. The coarse grid solver is a Jacobi-preconditioned CG method converged to an absolute residual of 10^{-20}. The V-cycle is used as preconditioner within a CG method. In addition to these parameters, we also discuss convergence results for direct interpolation, since – as we will see – the GPU-parallel version of direct interpolation is much faster than the one for standard interpolation. Convergence results are again reported for the benchmark problems defined in Section 3.1 with the stopping criterion of the outer CG solver set to a relative residual norm of 10^{-6}.

In Fig. 7 we report the convergence results for the benchmark problems Poisson3D and Poisson2D on the left-hand side and right-hand side, respectively. In both cases, the convergence by using standard interpolation matches the conver-

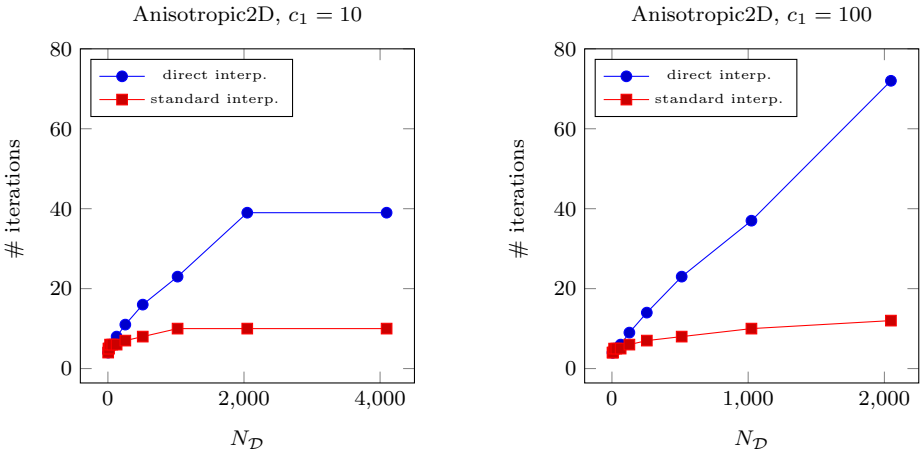

Fig. 8 Direct interpolation is not robust to the anisotropy in the Anisotropic2D benchmark. Meanwhile, our proposed parameter set applied in our hybrid GPU implementation achieves the predicted almost perfect robustness.

gence results predicted in Section 3: The three-dimensional problem still has a very slight increase in iteration count for a grid size of up to $N_{\mathcal{D}} = 128$, i.e. $N = 128^3$ unknowns. In contrast, the two-dimensional problem is already in its asymptotic convergence behavior with a constant number of iterations for growing problem size. In the two-dimensional problem, direct interpolation only slightly increases the number of iterations until convergence. This behavior is worse for the Poisson3D problem, where there is a stronger, but still acceptable, increase in the number of iterations when using direct interpolation.

The impact of the weaker direct interpolation becomes clearly visible when considering the Anisotropic2D test case in Fig. 8. Direct interpolation, together with our other parameters, is not robust to anisotropies. With growing anisotropy, the number of iterations until convergence becomes much larger. In contrast, standard interpolation shows only a very slight increase in the number of iterations, i.e. it is almost perfectly robust. Again, this matches the convergence behavior predicted in Section 3.

5.2 Performance benchmark setup

Our performance benchmark shall be comparable with the result presented in [23]. To this end, we use the same hardware and in general the same software environment as in this study. Our benchmark platform is a *Cray XK7* system, namely the GPU cluster Titan at Oak Ridge National Lab. Our performance comparisons are limited to a single node of this cluster. Each node contains an 16-core 2.2GHz *AMD Opteron 6274* processor and 32 GB of RAM. The GPU is an *Nvidia Tesla K20X*. We use the standard GNU compiler framework on that cluster, i.e. *GCC 4.9.3* with *MPICH 7.5.2*. The *Nvidia CUDA Programming Toolkit* is used in version 7.5. As stated before, we further use the library *Thrust*, as it comes with the *CUDA Toolkit* as well as *CUSP* in version 0.4.0.

All software is compiled with compiler flag `-O3` for optimization. The GPU code is compiled for the specific *compute capabilities* of the *Tesla K20X*. That

is, we use the additional compiler flag `-arch sm_35`. To comply with [23], the *AMG2013* benchmark is run either sequentially or in parallel using MPI on eight cores of a single node of Titan. The parallel MPI processes are distributed evenly over the physical cores of the Opteron processor. Reported *AMG2013* timings correspond to the timings reported as *wall clock time* by the *AMG2013* benchmark. Timings reported for our hybrid AMG GPU implementation were measured with the `gettimeofday` command and thereby also represent wall clock times. Our GPU code fully runs at double precision. The GPU AMG implementation is not intended to be an accelerated part of an existing CPU code but a solver within a full GPU code. This is why, in the hybrid GPU case, it is expected that the system matrix, right-hand side and initial guess are stored in GPU memory before the benchmark starts. Nevertheless, all remaining transfer times between CPU and GPU memory, as well as the full CPU and GPU compute time, are included for the hybrid GPU setup/solver test runs.

5.3 Performance comparison against *AMG2013*

In the following, we report the performance of our hybrid GPU AMG implementation, comparing it to the performance of the *AMG2013* benchmark for the benchmark applications defined in Section 3.1. It shall be noted here again that the specific set of AMG parameters used in *AMG2013* might not reflect the best possible choice of parameters for a CPU-based AMG method. However, for the sake of comparable results with [23], we keep these parameters even though BoomerAMG might be much faster for other parameters (and more recent versions of *hypre*).

5.3.1 Setup phase

In Figures 9 and 10, we show the different runtimes for the setup phase of the Poisson3D, Poisson2D and Anisotropic2D benchmarks using *AMG2013* sequentially or in parallel on CPU and using our hybrid GPU AMG implementation on GPU. It turns our that the setup phase of our GPU AMG implementation is surprisingly slow, especially in comparison to the *parallel AMG2013* test case. For direct interpolation, we often outperform the *sequential AMG2013* benchmark. However, the proposed use of standard interpolation on GPU is always the slowest method. To study the reasons for this rather pessimistic performance result, we do a detailed performance analysis of the setup phase for the Poisson3D test case for $N_{\mathcal{D}} = 128$. This study is presented and discussed in Section 5.4.

5.3.2 Solve phase

The performance results for the GPU solve phase are much better than the performance results for the setup phase. In Fig. 11, we give on the left-hand side the performance results of the solve phase for the Poisson3D test case and on the right-hand side the results for the Poisson2D test case. In the three-dimensional test case, the GPU-parallel solve phase (for standard interpolation) is by a factor of roughly 3.3 faster than the *AMG2013* benchmark parallelized on eight cores. Moreover, the GPU AMG is faster by roughly a factor of 7.1 in the two-dimensional test case and the same interpolation. Note again that we use here only eight of the available 16 cores of a Titan node, as done in [23]. Also, there exists a newer version of *hypre*, which is a superset of the *AMG2013* benchmark with more recent performance optimizations [25].

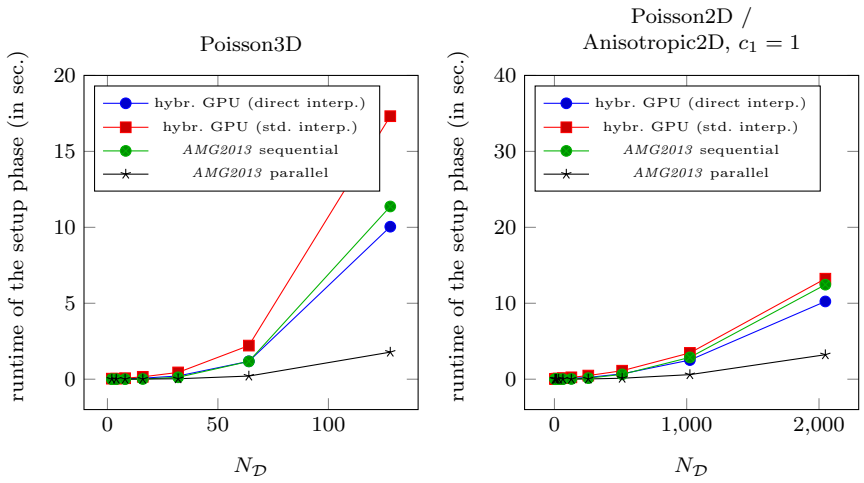

Fig. 9 We compare the runtime of the setup phase for our hybrid GPU implementation (using direct and standard interpolation) with the runtime of the setup phase of the *AMG2013* benchmark (*left:* Poisson3D benchmark, *right:* Poisson2D benchmark).

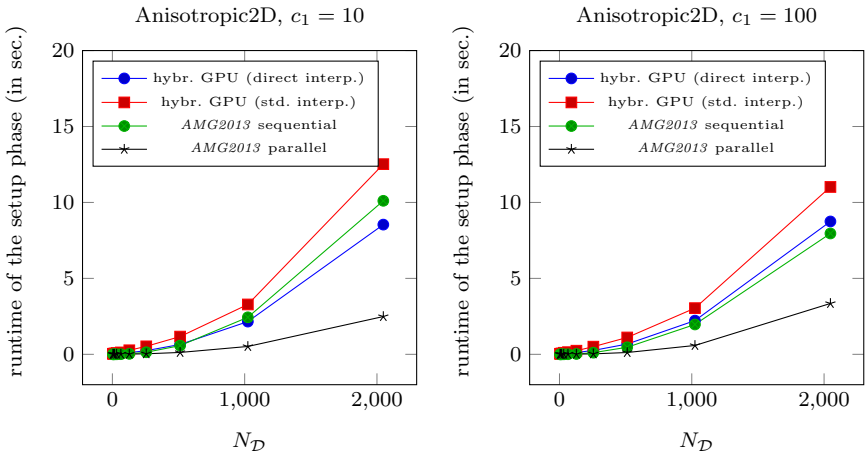

Fig. 10 The Anisotropic2D benchmark for $c_1 = 10$ (*left*) and $c_1 = 100$ (*right*) outlines a similar low performance for the setup phase of our hybrid GPU AMG compared to the *AMG2013* benchmark.

It has to be made clear at this point that our positive performance results for the solve phase could only be achieved since the chosen parameters in the setup phase lead to an iterative method with strong convergence and robustness, as discussed before. To underline this, we show the performance of the solve phase for the Anisotropic2D test case in Fig. 12. Using the weak direct interpolation, we only get a low ($c_1 = 10$) or no speedup ($c_1 = 100$) compared to the parallel *AMG2013* benchmark. However, the robust standard interpolation leads to speedups similar to the Poisson2D test case.

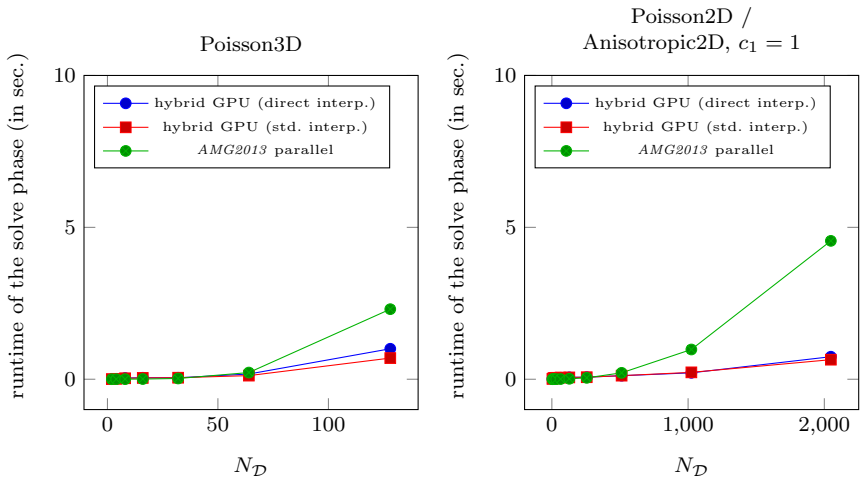

Fig. 11 In the solve phase, the hybrid GPU AMG implementation clearly outperforms the parallel *AMG2013* benchmark, both for the Poisson3D (*left*) and the Poisson2D (*right*) test case.

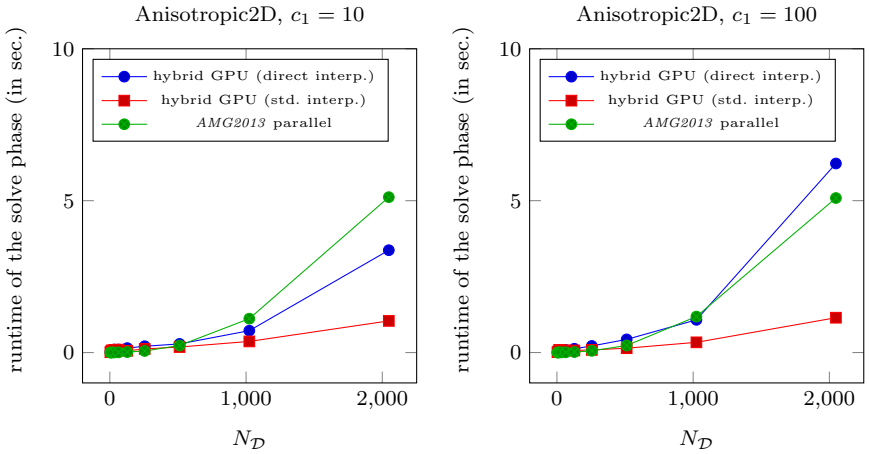

Fig. 12 The hybrid GPU AMG solve phase also clearly outperforms the parallel *AMG2013* benchmark when using standard interpolation in the Anisotropic2D test case. However, the performance advantage of the GPU gets lost, if the chosen set of parameters for AMG does not lead to a robust method as shown here for direct interpolation.

In Fig. 13, we summarize the performance results for the setup and the solve phase for the largest test cases of the Poisson3D and the Poisson2D problem. The parallel version of the *AMG2013* benchmark is faster than the hybrid GPU AMG in the setup phase, while the GPU code outperforms *AMG2013* in the solve phase. Very often, it is possible to reuse an existing multigrid hierarchy from one setup phase for several solve phases, e.g. if a single problem is solved for many right-hand sides. In this case, relatively speaking, the GPU-based AMG could become much faster. Meanwhile, we only show in Fig. 13 the total compute time for a single setup and a single solve phase. Here, *AMG2013* outperforms our implementation.

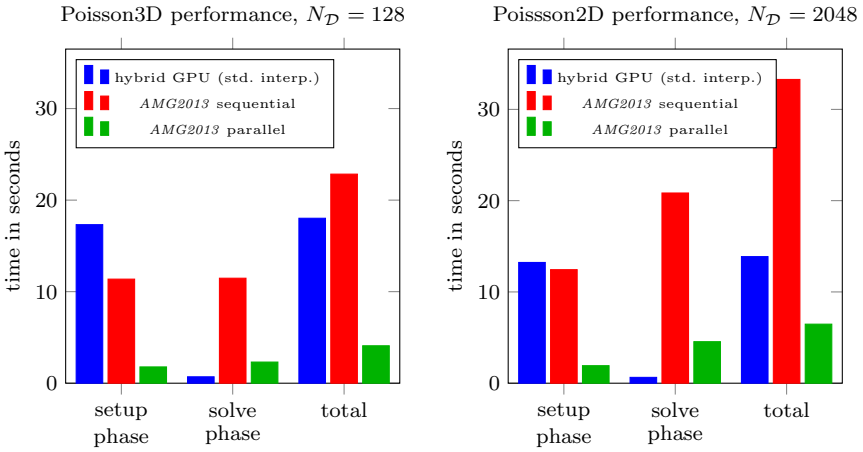

Fig. 13 In summary, the parallel version of the *AMG2013* benchmark is faster for the setup phase, while the hybrid GPU AMG is faster for the solve phase.

5.4 Detailed performance analysis of the setup phase

As seen in Section 5.3.1, the performance of the hybrid GPU AMG does not meet our performance expectations, when comparing it to the *AMG2013* benchmark on CPU. Therefore, we make an in-depth analysis of the performance of all components of the setup phase. To abbreviate the discussion, we focus on a single test case, the Poisson3D problem with $N_{\mathcal{D}} = 128$. Moreover, we only discuss the performance for the setup phase on the finest grid level.

In our study, we would like to analyse the impact of our parallelization of the setup phase. To do this, we compare our many-core parallel implementation with a sequential base CPU implementation that was initially done by the author. Note again that comparing a single-core CPU implementation to a many-core implementation is of course unfair. However, if we do not see significant speedups within this comparison, we have no chance to get a fast many-core implementation, anyway.

In Table 2, we report the results of our performance comparison against the reference CPU implementation, both for direct and standard interpolation. We include direct interpolation, since we want to find the reason for the strong performance differences between direct and standard interpolation.

5.4.1 C/F splitting

The first two timings in Table 2 correspond to the initial part of the C/F splitting, which are parallelized on many-core hardware. We compare the CPU and GPU timings for computing the maximum non-diagonal entry for each matrix row and the strong influence weights λ_i^l, as discussed in Sections 2.3 and 4.4. Based on the speedups, we conclude that the parallelization of this part of the C/F splitting is very successful. Actually, speedups in a range of 100 would correspond to a speedup of 6-10 when comparing equally priced hardware.

The next block of timings corresponds to the part of the setup phase, which is done on CPU. In the many-core implementation, we first have to copy the data

	direct interp.			std. interp.		
	CPU [s]	GPU [s]	factor	CPU [s]	GPU [s]	factor
C/F splitting						
max. neg. nondiag.	0.0508	0.0003	169.3	0.0507	0.0003	169.0
strong influence	0.0779	0.0002	389.5	0.0780	0.0003	260.0
GPU to CPU transfer	n/a	0.4537	n/a	n/a	0.4565	n/a
CPU data structure	0.3319	0.3296	1.0	0.3320	0.3299	1.0
CPU coarsening	2.4872	2.4908	1.0	2.488	2.4929	1.0
CPU to GPU transfer	n/a	0.0107	n/a	n/a	0.0107	n/a
coarse node count	0.0033	0.0853	0.0	0.0033	0.0854	0.0
fine to coarse map	0.0121	0.0013	9.3	0.0121	0.0013	9.3
interpolation						
coeffs. for interp.	0.0890	0.0004	222.5	n/a	n/a	n/a
direct interp. matrix	0.6149	0.0243	25.3	n/a	n/a	n/a
interp. point set	n/a	n/a	n/a	0.0631	0.0002	315.5
modified matrix	n/a	n/a	n/a	8.0044	0.3880	20.6
max. neg. nondiag.	n/a	n/a	n/a	0.1146	0.0006	191
coeffs. for interp.	n/a	n/a	n/a	0.0809	0.0124	6.5
direct interp. matrix	n/a	n/a	n/a	0.5821	0.0200	29.1
restriction operator	0.2253	0.0185	12.2	0.1213	0.0184	6.6
Galerkin operator	2.7489	0.6378	4.3	2.7043	0.6384	4.2
Jacobi smoother	0.2852	0.0061	46.8	0.2891	0.0061	47.4

Table 2 The detailed benchmark analysis for the setup phase (Poisson3D, $N_{\mathcal{D}} = 128$, finest grid) compares runtimes of all components of the proposed hybrid GPU implementation with those of a base CPU implementation running on *one* CPU core.

back to CPU memory. Afterwards, the CPU part of the coarsening is applied and the results are copied back to GPU. We can make several observations, here. First, the CPU-based coarsening in the GPU implementation is the most expensive part of the setup phase. Here, it has to be noted that the standard single-core performance of the AMD Opteron 6274 processor is relatively low and Turbo CORE, i.e. the AMD technology to overclock a single core, seems not to be switched on on Titan. Second, our CPU coarsening implementation seems to be clearly slower than coarsening implementations in e.g. *AMG2013*. This is not surprising, knowing that the base library *hypre* has been developed since many years. Third, it would be worth, overlapping the CPU data structure construction with major parts of the GPU to CPU data transfer. However, this is not possible with *CUSP* in version 0.4.0, at least to the author's knowledge, since an access to *CUDA Streams* is not implemented in this version, yet.

In the final many-core parallel part of the C/F splitting, we compute the number of coarse nodes and get the fine to coarse node mapping as discussed in Section 4.4. The reduction operation, to compute the number of coarse nodes, is (surprisingly) slower on GPU than on CPU, but in general neglectable, due to low overall performance impact. The exclusive scan operation for the node mapping is faster on GPU by a factor of nine. This is the (acceptable) speedup delivered by the *thrust* library and corresponds to an almost equal performance in a hypothetical equally-priced-hardware comparison.

5.4.2 Interpolation

Next, we consider the different interpolation strategies. In direct interpolation, we first pre-compute the coefficients α_i^l by the (parallel) graph traversal strategy.

This is fast on GPU with a speedup of beyond 200, i.e. even a decent speedup in a multi-core to many-core comparison. The actual construction is by a factor of 25 faster on GPU than on CPU. This is in the range of a speedup of two, when comparing equally priced hardware. This is a good result.

The overall time spend in the standard interpolation is much higher. Our parallelization strategy, to compute the set of interpolatory points, is successful, with a speedup in the range of 300. However, seemingly, our CPU base implementation to construct the modified system matrix, cf. Section 4.5, is too slow. Its parallelized version achieves a speedup of roughly 20, which is acceptable. However, the general approach to compute the modified system matrix has to be re-thinked. Afterwards, the maxima for the negative non-diagonal weights have to be updated. As in the C/F splitting, this is very fast on GPU. The computation of the coefficients α_i^l for the modified system matrix is much slower than in the direct interpolation case. First, we use a slightly different version of the compute kernel here, since we also include truncation. Second, the modified system matrix has a (significantly) larger number of non-zeros per row. Thereby, each compute kernel has much more sequential work to do, presumably leading to a higher thread divergence. Consequently, only a moderate speedup of above 6 is achieved. Finally, the kernel to compute direct interpolation is applied to the modified system matrix. This kernel is again fast.

5.4.3 Restriction operator, Galerkin product, smoother

The final three steps of the setup phase are the construction of the restriction operator by transposing the interpolation matrix, the triple matrix-product to construct the Galerkin operator and the construction of the Jacobi smoother. Transposing on GPU is more than a factor of 6 faster than on CPU. This speedup is delivered by the *CUSP* library and is acceptable. The triple product is again done using *CUSP* and is by a factor of 4 faster. We would have hoped to see higher performance, here. The *CUSP*-based setup of the Jacobi smoother is fast on GPU, anyway.

5.4.4 Summary and discussion

As it turns out, our general many-core parallelization strategies (parallel local graph traversal, matrix construction, ...) are fast. Whenever we use hand-implemented kernels, we get a decent performance improvement. The speedup obtained using the *CUSP* library is sometimes low. Note however that we do not use the latest version of *CUSP* as we discussed before. *Nvidia* has more recently introduced *CUSPARSE*, a *commercialized* version of *CUSP*, which might deliver higher performance. Using this library is future work. Our way to construct the standard interpolation by the modified system matrix seems to be a major performance bottleneck. This has to be re-thinked. Finally, the major performance limitation of the hybrid GPU AMG setup phase is the sequential CPU part. We still argue that it is more important to get a robust numerical method. Therefore using the sequential C/F splitting is favorable. Nevertheless, research on truely parallel and robust versions of C/F splitting should be undertaken. In addition, we argue that a strong GPU or many-core processor should be coupled with a few very fast CPU cores, in the future. This would reduce the negative impact of tightly coupled hybridization approaches as the one discussed here.

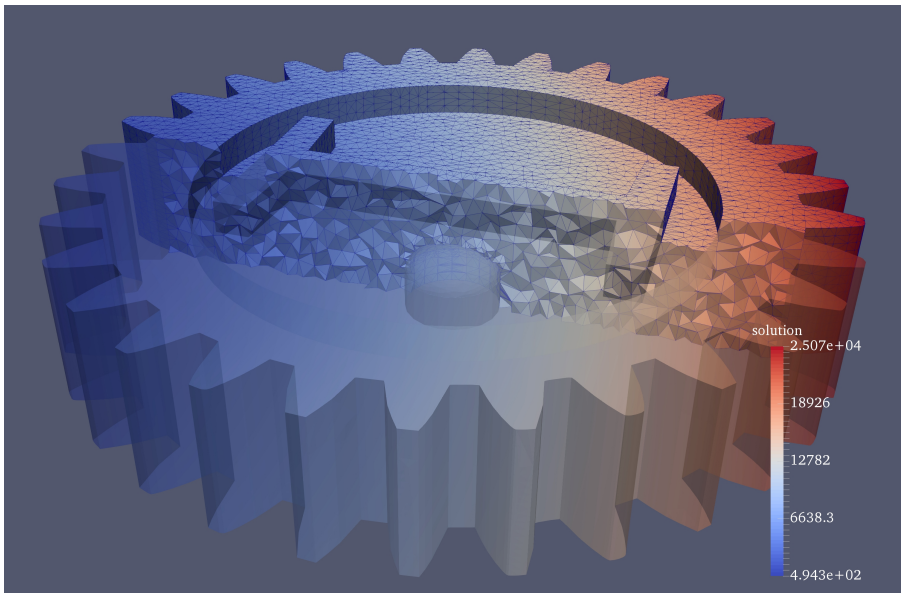

Fig. 14 The finite element discretization of the Poisson problem (6) solved on a gear geometry [2] is used as real-world application test case.

5.5 Real world application example

We finish this section with a more realistic application example. It is a Poisson problem solved on the complex gear geometry shown in Fig. 14. The exact mathematical problem is

$$\begin{aligned} -\Delta u &= -6 && \text{in } \mathcal{D}\,, \\ u &= 1 + x_1^2 + 2x_2^2 && \text{on } \partial\mathcal{D}\,, \end{aligned} \tag{6}$$

with $\mathcal{D} \subset \mathbb{R}^3$ the inner part of the gear geometry. The exact solution of this problem is $u = 1 + x_1^2 + 2x_2^2$ on the full domain $\overline{\mathcal{D}}$ and thereby constant with respect to the third coordinate direction. Note that it would be rather involved to solve such a complex geometry problem with a geometric multigrid method. However, AMG can solve such problems out-of-the-box.

The mesh that defines the geometry, i.e. $\partial\mathcal{D}$, is available at [2]. An initial meshing of $\mathcal{D}$ is done using *tetgen 1.5.1-beta1* [30]. It generates meshes of tetrahedrons. To generate meshes at different resolutions, we use the `-a` switch of *tetgen* and impose maximum cell volumes of 2^{2-l}, with $l \in \{0, \dots, 5\}$. Each mesh is imported into FEniCS 2016.2.0 [3], a finite element framework, where it is used in a standard linear finite element discretization of the PDE (6). FEniCS allows to assemble a system matrix and right-hand side for the discretization using the `assemble_system` command. An export of matrix and right-hand side is done using the linear algebra back end *Eigen*. The resulting system matrices and right-hand sides are finally used in our hybrid GPU AMG.

Fig. 15 summarizes the convergence results for the gear application problem on the left-hand side. We use the same parameters as in the performance and convergence studies before. These lead to an almost perfect convergence for growing

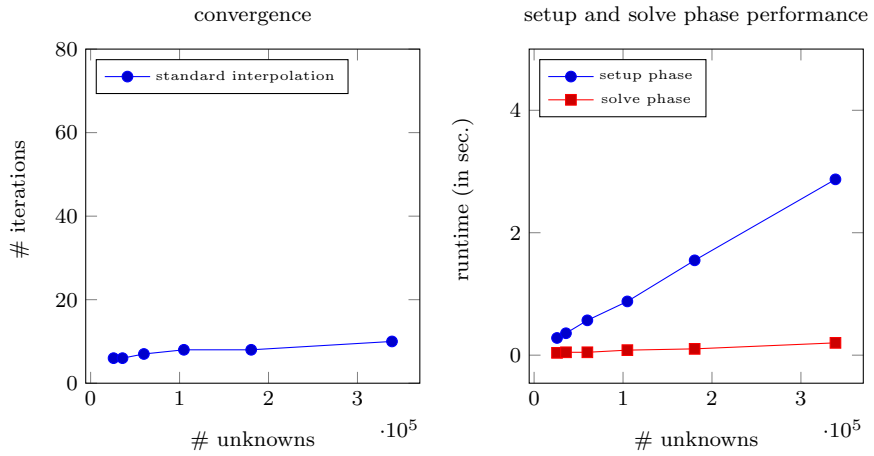

Fig. 15 *Left:* The hybrid GPU AMG shows excellent convergence results for growing problem size of the discretized Poisson problem in a gear geometry. *Right:* The setup phase needs less than three seconds for the largest problem size. The solve phase is then done in 0.2 seconds.

problem size with only a very slight increase of the number of iterations. Moreover, we report the runtime for the setup and the solve phase on the right-hand side of Fig. 15. For the largest problem size of more than 300,000 unknowns, the setup phase lasts less than three seconds when using standard interpolation. As before, the solve phase is very fast. Here, the hybrid GPU AMG only needs about 0.2 seconds to converge to the prescribed relative residual norm tolerance of 10^{-6}.

6 Summary

In this work, we analysed recent approaches to run the setup and solve phase of Ruge-Stüben algebraic multigrid on many-core hardware. Our focus was on the numerical methods that were specifically used in [23], in which one of the gold-standard commercial GPU AMGs is presented. Our empirical analysis showed that the methods applied therein — together with an expert-class implementation — lead to a very good performance of the solver. However, depending on the problem, the applied methods tend to show robustness issues, e.g. for larger anisotropies. To overcome these, we proposed to use a more classical set of components for AMG, leading to much more robust results. Unfortunately, this comes at the disadvantage, to keep a small part of the setup phase on CPU.

Based on these considerations, we developed and presented strategies to still parallelize a major part of the AMG setup phase on many-core hardware. Our aim here was to show starters in the field of many-core computing, how to approach the many-core parallelization of the rather complex algorithms in AMG. Our abstract strategies were realized in a specific hybrid GPU AMG code, which we implemented on top of *CUSP* and using *CUDA*.

A detailed performance and convergence analysis finally showed that the implemented code shows excellent convergence and robustness properties and has a fast solve phase, when comparing it to the *AMG2013* benchmark. However, the

setup phase is rather slow. To understand this performance mismatch, we analyzed each component of the setup phase intensively. It turned out that our parallelization strategies always lead to a considerable performance improvement, at least by comparison to our base CPU implementation. Nevertheless, our specific approach to construct the standard interpolation (already on CPU) is not as optimal as possible. Moreover, the CPU based part of the standard coarsening is surprisingly slow due to a non-optimal GPU to single-core CPU ratio. Finally, we also studied the performance in context of a real-world application.

Overall, we conclude that robustness in the numerical results comes, as expected, at a certain cost. More research on parallel methods for the AMG setup should be done. However, from a technological point of view, it would be also favorable to always couple many-core processors with a few extremely fast standard compute cores (with a fast interconnect). In that case, our discussed approach for the setup phase should perform much better. If we focus more on the solve phase, we conclude that it pays off to have a robust method combined with many-core parallelism. This is underlined by the excellent speedups reported for the solve phase.

Acknowledgements This work is funded by the Swiss National Science Foundation (SNF) under project number 407540_167186. The author was also partially supported by the project EXAHD of the DFG priority programme 1648 "Software for Exascale Computing" (SPPEXA). Furthermore, this research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

1. AMG2013 benchmark (LLNL-CODE-659229). URL `https://codesign.llnl.gov/amg2013.php`
2. GRABCAD Community. URL `https://grabcad.com`. The used gear model was uploaded by Brian Majors on March 30, 2017.
3. Alnæs, M.S., Blechta, J., Hake, J., Johansson, A., Kehlet, B., Logg, A., Richardson, C., Ring, J., Rognes, M.E., Wells, G.N.: The FEniCS Project Version 1.5. Archive of Numerical Software **3**(100) (2015)
4. Bell, N., Dalton, S., Olson, L.: Exposing fine-grained parallelism in algebraic multigrid methods. SIAM Journal on Scientific Computing **34**(4), C123–C152 (2012)
5. Bell, N., Garland, M.: Cusp: Generic parallel algorithms for sparse matrix and graph computations (2012). URL `http://cusplibrary.github.io`. Version 0.4.0
6. Brannick, J., Chen, Y., Hu, X., Zikatanov, L.: Parallel unsmoothed aggregation algebraic multigrid algorithms on GPUs. In: O. Iliev, S. Margenov, P. Minev, P. Vassilevski, L. Zikatanov (eds.) Numerical Solution of Partial Differential Equations: Theory, Algorithms, and Their Applications, *Springer Proceedings in Mathematics & Statistics*, vol. 45, pp. 81–102. Springer New York (2013)
7. Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Introduction to Algorithms. MIT Press (2001)
8. De Sterck, H., Falgout, R.D., Nolting, J.W., Yang, U.M.: Distance-two interpolation for parallel algebraic multigrid. Numerical Linear Algebra with Applications **15**(2-3), 115–139 (2008). doi:10.1002/nla.559. URL `http://dx.doi.org/10.1002/nla.559`
9. Emans, M., Liebmann, M.: Efficient setup of aggregation AMG for CFD on GPUs. In: PARA, pp. 398–409 (2013)
10. Emans, M., Liebmann, M., Basara, B.: Steps towards GPU accelerated aggregation AMG. In: 2012 11th International Symposium on Parallel and Distributed Computing, pp. 79–86 (2012)

11. Esler, K., Natoli, V., Samardzic, A.: GAMPACK (GPU accelerated algebraic multigrid package). In: ECMOR XIII - 13th European Conference on the Mathematics of Oil Recovery (2012)
12. Falgout, R., Yang, U.: hypre: A library of high performance preconditioners. In: P. Sloot, A. Hoekstra, C. Tan, J. Dongarra (eds.) Computational Science ICCS 2002, *Lecture Notes in Computer Science*, vol. 2331, pp. 632–641. Springer Berlin Heidelberg (2002)
13. Griebel, M., Metsch, B., Oeltz, D., Schweitzer, M.: Coarse grid classification: A parallel coarsening scheme for algebraic multigrid methods. Numerical Linear Algebra with Applications **13**(2–3), 193–214 (2006)
14. Haase, G., Liebmann, M., Douglas, C., Plank, G.: A parallel algebraic multigrid solver on graphics processing units. In: W. Zhang, Z. Chen, C. Douglas, W. Tong (eds.) High Performance Computing and Applications, *Lecture Notes in Computer Science*, vol. 5938, pp. 38–47. Springer Berlin Heidelberg (2010)
15. Henson, V., Yang, U.: BoomerAMG: A parallel algebraic multigrid solver and preconditioner. Appl. Numer. Math. **41**(1), 155–177 (2002)
16. Kraus, J., Förster, M.: Facing the multicore-challenge II. chap. Efficient AMG on Heterogeneous Systems, pp. 133–146. Springer-Verlag, Berlin, Heidelberg (2012)
17. Labs, P.: Paralution v1.0.0 (2015). `http://www.paralution.com/`
18. Lewis, T.J., Sastry, S.P., Kirby, R.M., Whitaker, R.T.: A GPU-based MIS aggregation strategy: Algorithms, comparisons, and applications within AMG. In: High Performance Computing (HiPC), 2015 IEEE 22nd International Conference on, pp. 214–223. IEEE (2015)
19. Liebmann, M.: Efficient PDE solvers on modern hardware with applications in medical and technical sciences. Ph.D. thesis (2009)
20. Lukarski, D.: Parallel sparse linear algebra for multi-core and many-core platforms. Ph.D. thesis, Karlsruher Instituts fr Technologie (KIT) (2012)
21. Lukarski, D.: PRALUTION user manual (2014). Version 0.7.0
22. Merrill, D., Garland, M., Grimshaw, A.: Scalable GPU graph traversal. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '12, pp. 117–128. ACM, New York, NY, USA (2012)
23. Naumov, M., Arsaev, M., Castonguay, P., Cohen, J., Demouth, J., Eaton, J., Layton, S., Markovskiy, N., Reguly, I., Sakharnykh, N., Sellappan, V., Strzodka, R.: AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods. SIAM Journal on Scientific Computing **37**(5), S602–S626 (2015)
24. Neic, A., Liebmann, M., Haase, G., Plank, G.: Algebraic Multigrid Solver on Clusters of CPUs and GPUs, pp. 389–398. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
25. Park, J., Smelyanskiy, M., Yang, U.M., Mudigere, D., Dubey, P.: High-performance algebraic multigrid solver optimized for multi-core based distributed parallel systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, pp. 54:1–54:12. ACM, New York, NY, USA (2015)
26. Ruge, J., Stüben, K.: Algebraic multigrid (AMG). In: S. McCormick (ed.) Multigrid Methods, *Frontiers in Applied Mathematics*, vol. 3, chap. 4, pp. 73–130. SIAM (1987)
27. Rupp, K., Rudolf, F., Weinbub, J.: ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. In: Intl. Workshop on GPUs and Scientific Applications, pp. 51–56 (2010)
28. Rupp, K., Weinbub, J., Rudolf, F., Morhammer, A., Grasser, T., Jüngel, A.: A performance comparison of algebraic multigrid preconditioners on CPUs, GPUs, and Xeon Phis. Under Review (2015)
29. Saad, Y.: Iterative Methods for Sparse Linear Systems, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2003)
30. Si, H.: Tetgen, a delaunay-based quality tetrahedral mesh generator. ACM Trans. Math. Softw. **41**(2), 11:1–11:36 (2015)
31. Trottenberg, U., Schuller, A.: Multigrid. Academic Press, Inc., Orlando, FL, USA (2001)
32. Vratis Ltd.: SpeedIT 2.3 reference manual (2012)
33. Wagner, M., Rupp, K., Weinbub, J.: A comparison of algebraic multigrid preconditioners using graphics processing units and multi-core central processing units. In: Proceedings of the 2012 Symposium on High Performance Computing, HPC '12, pp. 2:1–2:8. Society for Computer Simulation International, San Diego, CA, USA (2012)
34. Wang, L., Hu, X., Cohen, J., Xu, J.: A parallel auxiliary grid algebraic multigrid method for graphic processing units. SIAM Journal on Scientific Computing **35**(3), C263–C283 (2013)

35. Yang, U.: Parallel algebraic multigrid methods – High performance preconditioners. In: A. Bruaset, A. Tveito (eds.) Numerical Solution of Partial Differential Equations on Parallel Computers, *Lecture Notes in Computational Science and Engineering*, vol. 51, pp. 209–236. Springer Berlin Heidelberg (2006)
36. Yang, U.M.: On long-range interpolation operators for aggressive coarsening. Numerical Linear Algebra with Applications **17**(2-3), 453–472 (2010)

LATEST PREPRINTS

No.	Author: *Title*
2016-16	**A. Hyder** *Conformally Euclidean metrics on R^n with arbitrary total Q-curvature*
2016-17	**G. Mancini, L. Martinazzi** *The Moser-Trudinger inequality and its extremals on a disk via energy estimates*
2016-18	**R. N. Gantner, M. D. Peters** *Higher order quasi-Monte Carlo for Bayesian shape inversion*
2016-19	**C. Urech** *Remarks on the degree growth of birational transformations*
2016-20	**S. Dahlke, H. Harbrecht, M. Utzinger, M. Weimar** *Adaptive wavelet BEM for boundary integral equations: Theory and numerical experiments*
2016-21	**A. Hyder, S. Iula, L. Martinazzi** *Large blow-up sets for the prescribed Q-curvature equation in the Euclidean space*
2016-22	**P. Habegger** *The norm of Gaussian periods*
2016-23	**P. Habegger** *Diophantine approximations on definable sets*
2016-24	**F. Amoroso, D. Masser** *Lower bounds for the height in Galois extensions*
2016-25	**W. D. Brownawell, D. W. Masser** *Zero estimates with moving targets*
2016-26	**H. Derksen, D. Masser** *Linear equations over multiplicative groups, recurrences, and mixing III*
2016-27	**D. Bertrand, D. Masser, A. Pillay, U. Zannier** *Relative Manin-Mumford for semi-abelian surfaces*
2016-28	**L. Capuano, D. Masser, J. Pila, U. Zannier** *Rational points on Grassmannians and unlikely intersections in tori*
2016-29	**C. Nobili, F. Otto** *Limitations of the background field method applied to Rayleigh-Bénard convection*

Preprints are available under *https://math.unibas.ch/research/publications*

LATEST PREPRINTS

No.	Author: *Title*
2016-30	**W. D. Brownawell, D. W. Masser** *Unlikely intersections for curves in additive groups over positive characteristic*
2016-31	**M. Dambrine, H. Harbrecht, M. D. Peters, B. Puig** *On Bernoulli's free boundary problem with a random boundary*
2016-32	**H. Harbrecht, J. Tausch** *A fast sparse grid based space-time boundary element method for the nonstationary heat equation*
2016-33	**S. Iula** *A note on the Moser-Trudinger inequality in Sobolev-Slobodeckij spaces in dimension one*
2016-34	**C. Bürli, H. Harbrecht, P. Odermatt, S. Sayasone, N. Chitnis** *Mathematical analysis of the transmission dynamics of the liver fluke, Opisthorchis viverrini*
2017-01	**J. Dölz and T. Gerig, M. Lüthi, H. Harbrecht and T. Vetter** *Efficient computation of low-rank Gaussian process models for surface and image registration*
2017-02	**M. J. Grote, M. Mehlin, S. A. Sauter** *Convergence analysis of energy conserving explicit local time-stepping methods for the wave equation*
2017-03	**Y. Bilu, F. Luca, D. Masser** *Collinear CM-points*
2017-04	**P. Zaspel** *Ensemble Kalman filters for reliability estimation in perfusion inference*
2017-05	**J. Dölz and H. Harbrecht** *Hierarchical Matrix Approximation for the Uncertainty Quantification of Potentials on Random Domains*
2017-06	**P. Zaspel** *Analysis and parallelization strategies for Ruge-Stüben AMG on many-core processor*

Preprints are available under *https://math.unibas.ch/research/publications*