Journal of Computational Physics 281 (2015) 237–250

Contents lists available at ScienceDirect

Journal of Computational Physics

www.elsevier.com/locate/jcp

A massively parallel solver for discrete Poisson-like problems

Yvan Notay *,[1], Artem Napov

Service de Métrologie Nucléaire, Université Libre de Bruxelles (C.P. 165/84), 50, Av. F.D. Roosevelt, B-1050 Brussels, Belgium

ARTICLE INFO

Article history:
Received 21 March 2014
Received in revised form 13 June 2014
Accepted 18 October 2014
Available online 23 October 2014

Keywords:
Parallel computation
Poisson solver
Multigrid
Algebraic multigrid
AMG
Preconditioning
Aggregation

ABSTRACT

The paper considers the parallel implementation of an algebraic multigrid method. The sequential version is well suited to solve linear systems arising from the discretization of scalar elliptic PDEs. It is scalable in the sense that the time needed to solve a system is (under known conditions) proportional to the number of unknowns. The associate software code is also robust and often significantly faster than other algebraic multigrid solvers. The present work addresses the challenge of porting it on massively parallel computers. In this view, some critical components are redesigned, in a relatively simple yet not straightforward way. Thanks to this, excellent weak scalability results are obtained on three petascale machines among the most powerful today available.

1. Introduction

Many simulation codes in physics or engineering require the repeated solution of large symmetric positive definite (SPD) linear systems

$$A\,\mathbf{u} = \mathbf{b} \tag{1.1}$$

stemming from (or closely related to) the discretization of self-adjoint elliptic partial differential equations (PDEs) of the form

$$-\overline{\nabla}(D\overline{\nabla}u) + cu = f \quad \text{in } \Omega \subset \mathbb{R}^d, \tag{1.2}$$

where $d = 2$ (2D problems) or $d = 3$ (3D problems), with D positive and c nonnegative in Ω, and with appropriate boundary conditions prescribed on $\partial\Omega$. As an example, when $D \equiv 1$ and $c \equiv 0$, one has the Poisson equation whose solution is at the heart of simulation codes in quite diverse applications: computational fluid dynamics (CFD) programs based on pressure correction techniques (e.g., [9,18,39]), some beam dynamics simulations [2,3], some plasma/flow interactions simulations [38], chemical virtual prototyping [30], biomedical modeling [31], etc.

These linear system solutions often represent both the most time consuming part of the code and the main source of performance bottleneck on parallel computers. For large 3D simulations, it is nowadays standard to use multigrid methods. These methods are indeed scalable in the sense that the overall computational work to obtain the solution up to a prescribed tolerance is proportional to the number of unknowns.

* Corresponding author.
E-mail addresses: ynotay@ulb.ac.be (Y. Notay), anapov@ulb.ac.be (A. Napov).
[1] Research Director of the Fonds de la Recherche Scientifique – FNRS.

http://dx.doi.org/10.1016/j.jcp.2014.10.043
0021-9991/

Multigrid methods use a hierarchy of progressively smaller systems to obtain fast convergence. In *geometric* multigrid methods (e.g., [36]) this hierarchy is determined by discretizing the same continuous problem on a sequence of progressively less refined grids. Such an approach is inherently problem dependent, meaning that particular problem features (boundary conditions, jumps or anisotropy in the PDE coefficients, etc.) may require a special treatment. The implementation also requires a close interaction between discretization and solution modules, which is incompatible with some software designs.

On the other hand, *algebraic* multigrid (AMG) methods generate the multigrid hierarchy automatically, starting from the original system matrix A and recursively forming a smaller (coarser) problem from a larger one. Although the resulting methods are then often slightly less efficient than their geometric counterparts, this approach avoids the just mentioned drawbacks. The algorithms tend to be more robust, and can sometimes be applied to a wide class of problems without further tuning. Moreover, the implementation of AMG methods is in no way easier than that of geometric multigrid methods, but several software packages exist that can be called in a black box fashion and require less expertise from the end user.

There are several types of AMG methods. Options include *classical* AMG developed along the lines of the seminal works by Brandt, McCormick, Ruge and Stüben [7,32], *smoothed aggregation* AMG initiated by Vaněk, Mandel and Brezina [40], and (plain or unsmoothed) *aggregation-based* AMG as recently developed by the present authors [23,27,28].

Classical AMG methods are available in the *hypre* software package [19] (more specifically, their implementation forms the BoomerAMG [17] module of *hypre*); *hypre* is intended for both sequential and distributed computing, but experiments on massively parallel computers reveal some issues regarding the scalability of classical AMG methods; see [14] for a thorough analysis and [13,41] for the development of variants designed to face the sources of performance bottleneck.

Smoothed aggregation AMG is available in the ML software package [15]. Its parallelization is discussed in [37], and nice results on up to 2048 cores are reported in [2,3]; we are not aware of publications discussing its behavior on larger parallel machines.

Aggregation-based AMG has been made popular thanks to the AGMG software package [25], which also comes with both a sequential and a parallel version. In [27], promising numerical results are reported on a moderate size Intel cluster (with up to 48 nodes). However, the conclusions in [27] are to be toned down: on the one hand, the comparison made in [10] shows that aggregation-based AMG methods are faster than other AMG methods sequentially or on few processors, but may become slower as the number of processors increases; on the other hand, the results reported in [6] for a related method show that, on massively parallel systems, the scalability may be not fully satisfactory.

In the present paper, we report our efforts to improve the scalability of these aggregation-based AMG methods, focusing in particular on the AGMG implementation. Our motivation is manifold. Firstly, AGMG is used in a number of applications; see [30,31,35,38] for a sample of studies where the use of the package is acknowledged. Next, the sequential version of AGMG has been found robust for a wide class of problems, including problems with jumps in the PDE coefficients, strong anisotropy, unstructured meshes with strong local refinement and convection–diffusion problems with dominating convection driven by non-constant flows [23,24,27,28]. It has also been reported as significantly faster than its competitors; see [8,10,24]. We believe that this has some importance: to obtain the fastest parallel method, it is often a good idea to start from the fastest method in sequential (even if this is more challenging because less a method requires computations, less opportunity there is, in parallel, to overlap the communications with these computations).

Finally, issues to be faced are very different in nature from those raised by classical AMG methods, and therefore require different approaches to tackle them. In fact, most if not all difficulties come from the use of the K-cycle [29]. According to the results in, e.g., [12,21,27], it is indeed important to combine aggregation-based AMG methods with this cycle, despite the larger number of coarse grid solves per iteration it involves, compared with the more standard V-cycle (see the next section for details). Note that the way we address the associated difficulties in the context of AGMG is also insightful for any method that uses the K-cycle, or even the W-cycle which presents similar characteristics.

In the following, we first confirm that the naive use of AGMG on many core systems leads to severe scalability issues. Then we discuss the redesign of some critical algorithmic components, based on relatively simple yet not straightforward adaptations. Finally, we report the weak scalability results obtained on different massively parallel architectures, with up to 373,000 cores. In particular, we show that a linear system with 10^{12} unknowns is solved in less than 2 minutes; that is, in about 0.1 nanoseconds per unknown.

The paper is organized as follows. In Section 2, we review the main algorithmic components of AGMG. In Section 3, we give once for all the technical specifications of the numerical experiments reported in the different parts of the paper. The results obtained in parallel with the method as in [27] are presented in Section 4. The redesign for massively parallel computers is discussed in Section 5, and numerical results obtained on petascale machines are reported in Section 6. Concluding remarks are given in Section 7.

2. Algorithm overview

Before entering the core of this section, we make some general comments on the parallelization strategy. The unknowns of the linear system (1.1) are distributed among the *processes*, or, equivalently, MPI ranks. All vectors are distributed accordingly, as well as the rows of the matrix A. Hence, each process holds a "local" portion of the vectors and a "local" portion of the matrix rows. We further call "local diagonal block" the part of these rows restricted to columns that correspond to local unknowns. Altogether, the local diagonal blocks form the block diagonal part of A with respect to the partitioning of the unknowns induced by their distribution among the processes.

We solve the linear system (1.1) using AGMG as a preconditioner for the flexible conjugate gradient (FCG) method [26]. In parallel, following ideas that trace back to [33], we consider an implementation of FCG that allows to compute the needed inner products with a single global communication; see steps 5, 6 and 9 in Algorithm 1 below.[2] The equivalence of this latter with the original FCG algorithm from [26] is shown in Appendix A.

Algorithm 1 (Parallel FCG to solve $A\,\mathbf{u} = \mathbf{b}$)

1. Initialization: select initial approximation $\mathbf{u}_0$ and set $\mathbf{r}_0 = \mathbf{b} - A\,\mathbf{u}_0$
2. **for** $k = 0, 1, \ldots$ until convergence
3. $\quad \mathbf{v}_k = \mathcal{B}(\mathbf{r}_k)$
4. $\quad \mathbf{w}_k = A\,\mathbf{v}_k$
5. $\quad \alpha_k = \mathbf{v}_k^T \mathbf{r}_k$
6. $\quad \beta_k = \mathbf{v}_k^T \mathbf{w}_k$
7. $\quad$ **if** $k > 0$
8. $\quad\quad$ **then**
9. $\quad\quad\quad \gamma_k = \mathbf{v}_k^T \mathbf{q}_{k-1}$
10. $\quad\quad\quad \mathbf{d}_k = \mathbf{v}_k - (\gamma_k/\rho_{k-1})\,\mathbf{d}_{k-1}$
11. $\quad\quad\quad \mathbf{q}_k = \mathbf{w}_k - (\gamma_k/\rho_{k-1})\,\mathbf{q}_{k-1}$
12. $\quad\quad\quad \rho_k = \beta_k - \gamma_k^2/\rho_{k-1}$
13. $\quad\quad$ **else**
14. $\quad\quad\quad \mathbf{d}_k = \mathbf{v}_k$
15. $\quad\quad\quad \mathbf{q}_k = \mathbf{w}_k$
16. $\quad\quad\quad \rho_k = \beta_k$
17. $\quad \mathbf{u}_{k+1} = \mathbf{u}_k + (\alpha_k/\rho_k)\,\mathbf{d}_k$
18. $\quad \mathbf{r}_{k+1} = \mathbf{r}_k - (\alpha_k/\rho_k)\,\mathbf{q}_k$

Besides the inner products, the multiplication by A (matrix vector product, or *Matvec* at step 4) also requires communications. However, because the matrix is sparse, each process has only to communicate with a few neighbors, and this communication can further be overlapped with the part of the Matvec related to the local diagonal block. The critical stage is therefore the application of the preconditioner to the residual vector $\mathbf{r}$ (step 3).

If $\mathcal{B}(\cdot)$ corresponds to the multiplication by a given SPD matrix $\mathcal{B}$, FCG is equivalent to the standard preconditioned conjugate gradient (PCG) method, and its convergence is then known to be related to the spectral properties of $\mathcal{B}A$, namely to the ratio of its largest and its smallest eigenvalues. However, FCG allows to maintain approximately the same convergence rate when $\mathcal{B}$ only approximates a SPD matrix, for instance because computing $\mathcal{B}(\mathbf{r})$ involves solving linear subsystem(s), which is done in practice only up to a given accuracy.

The speed of convergence obtained with algebraic multigrid preconditioning – and AGMG is not an exception – depends on the interplay between the *smoothing iterations* and a *coarse grid correction*. Smoothing iterations are simple stationary iterative methods; those used in AGMG are described below. Coarse grid correction amounts to a solution of a smaller system associated with a coarser grid. In the case of AGMG the coarse grid is obtained by a partitioning of the n unknowns in $n_c < n$ *aggregates* G_k, $k = 1, \ldots, n_c$. The matrix of the smaller system, referred to as *coarse grid matrix*, is then given by

$$(A_c)_{s\ell} = \sum_{i \in G_s} \sum_{j \in G_\ell} (A)_{ij}.$$

One application of the AGMG preconditioner is sketched in Algorithm 2 below.

Algorithm 2 (AGMG preconditioner: $\mathbf{v} = \mathcal{B}(\mathbf{r})$)

1. $\mathbf{v}_1 = L^{-1}\mathbf{r}$ *(pre-smoothing)*
2. $\tilde{\mathbf{r}} = \mathbf{r} - A\,\mathbf{v}_1$ *(residual update)*
3. $\tilde{\mathbf{r}}_c$ from $(\tilde{\mathbf{r}}_c)_s = \sum_{i \in G_s} (\tilde{\mathbf{r}})_i$, $s = 1, \ldots, n_c$ *(restriction)*
4. Solve (approximately) $A_c \mathbf{v}_c = \tilde{\mathbf{r}}_c$ *(coarse grid solution)*
5. $\mathbf{v}_2$ from $(\mathbf{v}_2)_i = (\mathbf{v}_c)_s$ for s such that $i \in G_s$, $i = 1, \ldots, n$ *(prolongation)*
6. $\bar{\mathbf{r}} = \tilde{\mathbf{r}} - A\,\mathbf{v}_2$ *(residual update)*
7. $\mathbf{v}_3 = U^{-1}\bar{\mathbf{r}}$ *(post-smoothing)*
8. $\mathbf{v} = \mathbf{v}_1 + \mathbf{v}_2 + \mathbf{v}_3$ *(final result of the preconditioner application)*

[2] The main idea is the following. With a classical implementation, $\rho_k = \mathbf{d}_k^T A\,\mathbf{d}_k$ is computed only when $\mathbf{d}_k$ and $\mathbf{q}_k = A\,\mathbf{d}_k$ are available; that is, only when the inner product γ_k (needed to compute $\mathbf{d}_k$) has been finalized with a first global communication. Hence the finalization of ρ_k requires a second one. The trick consists in: (1) instead of computing directly $\mathbf{q}_k = A\,\mathbf{d}_k$, use a recursion similar to that used for $\mathbf{d}_k$, based on the pre-computation of $A\,\mathbf{v}_k$; (2) obtain ρ_k via an indirect formula involving $\beta_k = \mathbf{v}_k^T A\,\mathbf{v}_k$ and γ_k. Thus, both can be finalized with a single global communication, which can further be used to finalize α_k as well.

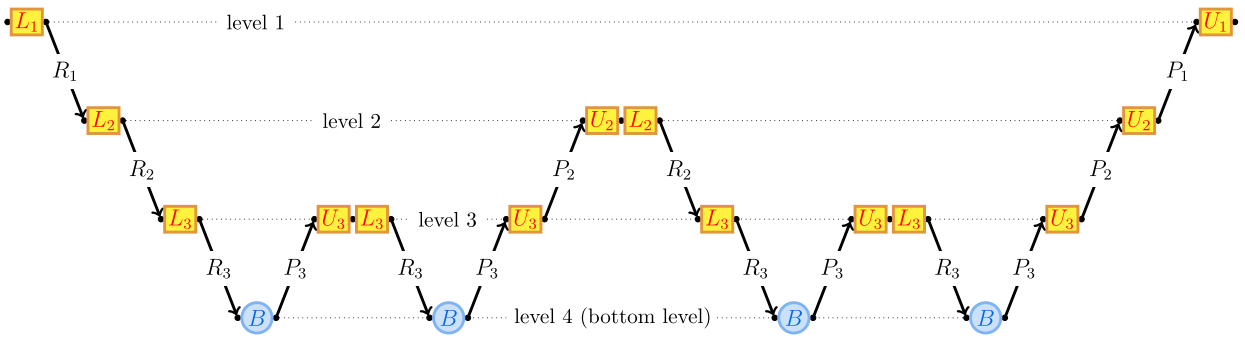

Fig. 1. The steps corresponding to one iteration of AGMG preconditioner with 4 levels; for $i = 1, \ldots, 4$, L_i is for forward Gauss–Seidel with subsequent residual computation (steps 1 and 2 of Algorithm 2); R_i is the restriction to the coarse grid at level i (step 3 of Algorithm 2); B is the bottom level solver at level 4 (step 4 of Algorithm 2); P_i is the prolongation at level i (step 5 of Algorithm 2); U_i is for residual computation, backward Gauss–Seidel and the sum of corrections at level i (steps 6–8 of Algorithm 2).

The *pre-smoothing* (step 1) amounts to perform a stationary iteration with L. Because sequential AGMG uses forward Gauss–Seidel pre-smoothing, L is the lower triangular part of A. In parallel, this is modified into a method called *Processor Block* Gauss–Seidel in [1] and *hybrid* Gauss–Seidel in [4]: one uses, for each process, a local L equal to the lower part of the local diagonal block (that is, one discards from the "sequential L" the entries connecting unknowns assigned to different processes). Hence step 1 is purely parallel. The same remark applies to *post-smoothing* (step 7), which is based on (hybrid) backward Gauss–Seidel: U is the upper part of A in the sequential case, and, in parallel, locally the upper part of the local diagonal block. Note that because A is supposed symmetric, U is the transpose of L in all cases.

On the other hand, steps 2 and 6 (residual updates) require the same type of communication as the Matvec operation already discussed above. Hence the tricky part is concentrated in steps 3–5, which altogether form the coarse grid correction. For aggregation-based methods, the restriction of the residual from the fine grid to the coarse one (step 3), and the prolongation of the coarse update on the fine grid (step 5), are both easily made purely parallel by enforcing the aggregates G_i to contain only fine grid unknowns assigned to a same process. Then the points that remain to discuss are how is solved the coarse grid system (step 4), and, finally, how are formed the aggregates.

Solving the coarse grid system. AGMG resorts to the K-cycle [29]: the system at step 4 is solved with two FCG iterations. That is, $A_c \mathbf{v}_c = \tilde{\mathbf{r}}_c$ is also solved with Algorithm 1, using the zero vector as initial approximation and only two iterations. Moreover, this application is recursive: as for the initial linear system, the preconditioner is based on a multigrid method as sketched in Algorithm 2, but now at a coarser level; which also means that a further coarser level is used for the coarse grid correction, the related systems being again solved with two FCG iterations, etc. The recursion is stopped at a properly determined *bottom level*, and the corresponding coarse grid system is solved using an alternative (non-recursive) solution method, referred to as *bottom level solver*. Often the bottom level solver is a direct solver, but changing this is crucial to obtain good scalability on massively parallel systems; see the discussion in Section 5. On Fig. 1, we depict the work flow associated with this recursive use of the algorithms in the four level case (the fine level corresponding to the systems (1.1) to be solved, two intermediate levels, and the bottom level).

Aggregation procedure. The aggregation scheme used by AGMG is described in detail in [23]. It consists in several passes of a pairwise matching algorithm. Hence if m passes are performed, aggregates will contain at most 2^m unknowns. Not all of them reach that size because a quality control is performed, aiming at guaranteeing some minimal convergence speed. However, for the problems under consideration in this work, the ratio n/n_c is effectively close to 2^m when using two or three passes ($m = 2$ or $m = 3$). In parallel, AGMG uses the same pairwise matching algorithm independently on each process, with the additional constraint that pairs should be formed with local unknowns only. By default, the sequential version uses two passes of pairwise aggregation ($m = 2$). This is increased to three ($m = 3$) for the parallel version, which also uses a slightly relaxed value of the quality control parameter. This aims at favoring a faster coarsening (a more rapid decrease of the number of unknowns from one level to the next), to reduce the number of levels and hence the overall amount of time spent on coarse levels with few unknowns. Indeed, in parallel implementations the time spent on the coarse levels typically scales less well than the time needed for the fine grid computation.

It is worth discussing briefly why the K-cycle uses exactly two iterations. Using more appears in practice useless (the number of iterations at fine grid level is essentially unaffected), and this could also significantly impact the overall computational cost of the method when $m = 2$ (see [23] for a detailed discussion of the cost). There is more margin when $m = 3$ as by default with the parallel version, but with more than two iterations the issues discussed in Section 5 would be just more critical.

On the other hand, using only one iteration is roughly equivalent to the V-cycle (which, more precisely, approximately solves the coarse grid system with one application of the coarse level preconditioner). This cycle is often the preferred option for geometric multigrid methods, and also classical and smoothed aggregation AMG methods. However, for aggregation-based methods, using the more sophisticated K-cycle is an essential ingredient to ensure that the number of iterations does

not increase with the problem size. In parallel, this is clearly the downside of the approach, see the discussion in Section 5. Note that the other AMG methods avoid this drawback but have their own issues, all related to the more complex operations needed to transfer from fine to coarse and coarse to fine, and the related definition of the coarse grid matrices; see, e.g., [5,13,41].

To be complete, note that "exactly two iterations" has to be understood as "at most two iterations" for sequential AGMG and the version used to produce the results in Table 2. These indeed bypass the second iteration when the decrease of the residual norm after the first one is below a given threshold; see [27] for details. However, for the newly developed parallel version discussed in Sections 5 and 6, this option has been abandoned: (1) due to larger aggregates than in sequential, the criterion was almost never satisfied; (2) fixing in advance the number of iterations allows an enhanced implementation with only one global communication for all two iterations.

3. Problems specification, reported data and tested architectures

We have two test problems, which are both discrete versions of the Poisson equation solved in a three-dimensional parallelepiped ((1.2) with $D \equiv 1$, $c \equiv 0$, and $d = 3$). In all cases, we use homogeneous Dirichlet boundary conditions on the back, left and bottom boundaries, and homogeneous Neumann boundary conditions on the front, right and top boundaries of the parallelepiped.

In the first example, the parallelepiped is the unit cube $(0,1) \times (0,1) \times (0,1)$, and we consider 7-point finite difference discretization with uniform mesh size h in all directions; the right side is $f = 1$ in the central zone $(\frac{1}{4}, \frac{3}{4}) \times (\frac{1}{4}, \frac{3}{4}) \times (\frac{1}{4}, \frac{3}{4})$, and $f = 0$ elsewhere. In parallel, the p processes (where p is the number of MPI ranks that are launched) are arranged in a $p_x \times p_y \times p_z$ Cartesian grid as close as possible to a cubic grid, with $p = p_x\, p_y\, p_z$. A parallelepipedal portion of the domain (and the corresponding discrete unknowns) is then assigned accordingly to each process. In weak scalability tests, the mesh is refined as the number of processes increases, in such a way that the number of unknowns per process remains roughly constant. The system matrix has approximately 7 nonzero entries per row, whereas, in parallel, performing a matrix vector multiplication requires that each process communicates with 6 neighbors.

As second example we consider a system (1.1) with matrix A corresponding to the finite elements discretization using third order (P3) Lagrangian bases on a uniform symplectic mesh of size h in all directions; the right hand side is given by $(\mathbf{b})_i = \sin(\|\mathbf{r}\|^2)$, with $\mathbf{r} = (r_x, r_y, r_z)$ being the location of the ith unknown. In parallel, the p processes are again arranged in a $p_x \times p_y \times p_z$ Cartesian grid, which, however, here induces a slight change in the problem's geometry: the parallelepiped Ω is tuned in such a way that each process deals with a cubic subdomain even if p_x, p_y and p_z are not equal to each other; if $p_x = p_y = p_z$, then Ω is the unit cube. Here, the matrix is relatively denser and has approximately 44 nonzero entries per row, whereas performing a matrix vector multiplication requires that each process communicates with 26 neighbors.

In all cases, the reported data correspond to an iterative solution with the zero vector as initial approximation; the stopping criterion is the decrease of the relative residual error by a factor of 10^{-6} for the first problem and by 10^{-7} for the second. The number given as "#p" is always the number of processes (or launched MPI ranks); in most cases, it coincides with the number of cores available on the set of used computing nodes. All times reported are wall clock elapsed times observed on the root process (MPI rank 0). We distinguish the *setup time* ($\mathcal{T}_{\text{su}}$) and the *solution time* ($\mathcal{T}_{\text{solve}}$), the *total time* ($\mathcal{T}_{\text{tot}}$) being the sum of both. The setup time is the time needed to build the preconditioner; that is, essentially, the time to form the aggregates and compute the related coarse grid matrix at the successive levels of the hierarchy. On the other hand, the solution time is the time more specifically spent during FCG iterations. Sometimes we also report the *bottom level* time ($\mathcal{T}_{\text{bl}}$), which is the part of the solution time that is spent more specifically in solving the linear systems at the bottom level.

We have performed numerical tests on the following architectures (the first one being used only for some preliminary tests).

Intel cluster: two Intel XEON L5420 processors at 2.50 GHz and 16 GB RAM memory per computing node, with InfiniBand (half bandwidth) interconnect (2009).

Intel Farm (CURIE at CEA, France[3]): two eight-cores Intel Sandy Bridge EP (E5-2680) at 2.7 GHz and 64 GB RAM per computing node, with InfiniBand QDR Full Fat Tree interconnect (2012). Up to 5040 nodes (80,640 cores), 20th in top 500 supercomputer list of November 2013 with 1.4 Petaflop/s on the Linpack benchmark.

IBM BG/Q (JUQUEEN at Juelich, Germany[4]): one IBM PowerPC A2 at 1.6 GHz with 16 cores and 16 GB RAM per computing node, 5D Torus interconnect (2012). Up to 28,672 nodes (458,752 cores), 8th in top 500 supercomputer list of November 2013 with 5.0 Petaflop/s on the Linpack benchmark.

Cray XE6 (HERMIT at HLRS, Stuttgart, Germany[5]): two AMD Opteron(tm) 6276 (Interlagos) processors with 16 cores each, and 32 GB RAM per computing node, High Speed Network CRAY Gemini (2011). Up to 3552 nodes (113,664 cores), 39th in top 500 supercomputer list of November 2013 with 0.8 Petaflop/s on the Linpack benchmark.

[3] http://www-hpc.cea.fr/fr/complexe/tgcc-curie.htm.
[4] http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JUQUEEN.
[5] https://wickie.hlrs.de/platforms/index.php/CRAY_XE6_Hardware_and_Architecture.

Table 1
Sequential results on Intel cluster.

h^{-1}	$\frac{n}{10^6}$	AGMG sequential				AGMG parallel				*hypre*			
		it.	$\mathcal{T}_{su}$	$\mathcal{T}_{sol}$	$\mathcal{T}_{tot}$	it.	$\mathcal{T}_{su}$	$\mathcal{T}_{sol}$	$\mathcal{T}_{tot}$	it.	$\mathcal{T}_{su}$	$\mathcal{T}_{sol}$	$\mathcal{T}_{tot}$
		Finite Difference on Intel cluster											
60	0.22	10	0.5	0.5	1.0	10	0.2	0.4	0.6	13	1.0	1.1	2.1
120	1.73	11	2.3	5.0	7.3	10	3.0	3.3	6.3	15	9.0	11.2	20.1
200	8.00	11	13.9	24.8	38.7	10	17.4	15.6	33.1	17	44.6	61.2	105.8
300	27.00	11	50.7	86.4	137.1	10	63.3	54.6	117.9	19	162.2	241.1	403.3
		Finite elements (P3) on Intel cluster											
13	0.06	16	0.3	1.1	1.3	19	0.4	0.9	1.3	32	0.4	2.2	2.6
21	0.25	17	1.2	5.2	6.4	20	1.4	4.4	5.9	40	1.9	12.1	13.9
33	0.97	18	4.9	22.9	27.8	22	5.9	20.0	25.8	43	7.6	54.3	61.9
52	3.79	18	19.6	95.8	115.4	23	23.6	84.9	108.5	63	30.8	321.8	352.6

Table 2
Results with a direct solver at bottom level (AGMG 3.2.0).

#p	$\frac{n}{10^6}$	it.	$\mathcal{T}_{su}$	$\mathcal{T}_{sol}$ ($\mathcal{T}_{bl}$)	$\mathcal{T}_{tot}$
	Finite Difference on Intel cluster *(1 process per node)*				
1	27.0	10	63.3	54.6 (0.5)	117.9
2	53.2	11	62.8	62.4 (0.4)	125.2
4	107.9	11	69.8	72.2 (0.6)	142.0
8	216.0	11	72.8	75.1 (0.7)	147.9
16	432.1	11	69.9	79.2 (1.7)	149.1
32	862.8	11	72.7	76.4 (1.2)	149.1
	Finite Difference on IBM BG/Q *(16 processes per node)*				
16	43.6	11	27.4	26.6 (1.8)	54.0
64	175.6	11	28.0	29.5 (4.2)	57.5
512	1404.9	11	30.2	59.1 (33.6)	89.3

4. Results with a direct solver at bottom level

The release 3.2.0 of the AGMG software [25] implements the algorithm sketched in the previous section, calling an external sparse direct solver to solve the linear systems at the bottom level. The number of levels is then calculated in such a way that the needed LU factorization of the coarsest grid matrix requires only a negligible amount of time relative to fine grid operations. The used sparse direct solver is MUMPS [20], which has both a sequential and a parallel version.

The sequential performance of AGMG is documented in several publications [23,24,27,28]. However, as written in Section 2, the parallel version uses slightly different parameters. Before testing its scalability, it is then important to assess the impact of these modifications on the sequential performance. This is done in Table 1, where we also give the results obtained with the *hypre* software package [19], using the parameters recommended in [41, p. 282] for the parallel solution of 3D problems.

One sees that AGMG is not much affected by the change of the default parameters. The setup time somehow increases, but this is more than compensated by a decrease of the solution time. This situation is in fact typical for 3D problems.[6] On the other hand, AGMG appears roughly three times faster than *hypre*. This is in line with the more detailed comparison developed in [24] (see also [8,10]), which further displays the sensitivity of *hypre* to the many available options and parameters: tuning these at best for the problem at hand allows, in the sequential case, to reduce the penalty to a factor of about two.

It is worth noting that both sequential and parallel AGMG are scalable in the sense that the number of iterations is nearly constant whereas the time needed only slightly deviates from a linear growth with the number of unknowns.

We next give in Table 2 the results obtained with AGMG in parallel (again, the release 3.2.0 that uses parallel MUMPS to solve the bottom level systems). The results on Intel cluster may be seen as an update of the numbers published in [27], where the same architecture was tested with an earlier version of the package (not using the improved aggregation scheme developed since then [23]). One sees that good weak scalability is achieved despite the relatively slow (compared with today standard) communication network. Opposite to this, the results obtained on IBM BG display the limit of the approach on many cores architectures. It is worth noting that the observed increase of the solution time is entirely due to the time needed to solve the bottom level systems (reported in brackets). This observation is the starting point of the enhancements presented in the next section.

[6] In 2D cases, one often observes the same increase of the setup time (due to the additional pairwise aggregation pass), while the solution time is essentially unaffected, reason for which the parameters used in parallel are not the default in sequential as well.

5. Algorithm redesign

Any parallel multigrid or algebraic multigrid solver needs a carefully designed bottom level solver [11]. Typically, only few unknowns per process are left on the coarsest grid, making it difficult, if not impossible, to achieve good scalability for that part of the algorithm. On top of that, even few unknowns per process may lead to non-trivial system sizes on massively parallel computers with, say, more than hundred thousands of cores.

Now, with classical V-cycle implementations, only a tiny portion of the overall computational work is done at the bottom level. Hence, even if that part scales poorly, there is an important margin before this affects significantly the overall scalability of the method. Then, a number of approaches are possible. Some keep a non-trivial number of unknowns at the bottom level and use a highly parallel iterative solver like the Chebyshev semi-iterative method [3] or the unpreconditioned conjugate gradient method [16]; it is also possible to make gradually idle the processes which cannot be used efficiently given the size of the coarse grid matrix at hand [36, Section 6.3.2]; in the context of AMG methods, an efficient implementation of this latter approach may be hard to obtain [6], which may motivate a simpler variant, where the whole coarsest grid system is gathered on a single process, then solved directly with a (sequential) sparse direct solver, the solution being scattered next to all processes (or, somehow equivalently, the sequential solve is repeated in parallel on all processes).

The K-cycle used by AGMG (see Section 2) induces however an additional difficulty: more we use levels, more we need to call the bottom level solver *per fine grid iteration*; see Fig. 1. In fact, if we use ℓ levels, the algorithm needs exactly $2^{\ell-2}$ bottom level solutions per iteration; the grow is thus exponential. Therefore, in the context where the number of levels grows with the problem size so as to keep the bottom level reasonably small, none of the approaches mentioned above is likely successful on petascale computers.

Of course, it is possible to combine aggregation with the V-cycle, but, as mentioned in Section 2, then one looses algorithm scalability, because the number of iterations will in general grow with the number of levels. Hence, the comparison with purely sequential AGMG would become less favorable, whereas the weak scalability would suffer if the number of levels is increased with the global size of the system (so as to maintain the size of the coarsest grid roughly constant). This is for instance seen in [6], where an aggregation-based method is considered that is somehow similar to AGMG except that the V-cycle is used. In the weak scalability tests reported in [6, Table 6], one may observe that the total solution time is increased by a factor of about 5 when going from 1 to 262,000 cores. Most of this increase comes from the setup time, because the sophisticated approach used on the coarsest levels does not scale that well during that phase. But a significant part of the loss of scalability further comes from the increase of the number of iterations.

Using the K-cycle is thus in some sense the price to pay to have an optimal multilevel method with the appealing features associated with aggregation schemes, including the inherent parallelization of restriction and prolongation operations (see Section 2).

We therefore considered another strategy: limit the number of levels to a few, whatever the global problem size. This can be achieved by using very large aggregates to obtain a much faster reduction of the number of unknowns from one level to the next. Alternatively, one may run only few steps of "standard" aggregation, and then develop a bottom level solver able to cope with the still relatively large systems that are left at the bottom level. Here we opt for the latter approach, because the former would again imply a deterioration of the convergence rate. The challenge is of course to design a "sufficiently good" bottom level solver. It has to be highly parallelizable, and, in particular, avoid the above mentioned problem associated with the K-cycle. On the other hand, it does not need to be as fast as AGMG from a purely sequential viewpoint. Indeed, even after only three aggregation steps (like in Fig. 1), the coarsest grid system will typically have something like $8^3 \approx 500$ times less unknowns than the fine grid system. Hence we may use a somehow suboptimal solver from the "operation count" viewpoint without a significant impact on the overall computing time.

These considerations lead us to the following design choices for the new bottom level solver. Firstly, the solver is now an *iterative* method. Note that the solution at the bottom level does not need to be computed very accurately. According to our numerical experiments, a stopping criterion based on a relative tolerance of 0.4 is sufficient to ensure that the number of outer iterations (at fine grid level) is essentially the same as when using a direct bottom level solver. This is in agreement with the observations in [36, Section 6.3.2] (made for a slightly different W-cycle).

Now, to maintain scalability while maximizing the parallel performance, we consider as bottom level solver a two-level scheme with coarse grid obtained by very aggressive aggregation: all "local" unknowns on a same process are grouped in a single aggregate. Thus, see Fig. 2, the bottom level solver is the PCG method with a preconditioner of the same type as the one sketched in Algorithm 2, with a coarse grid matrix that has exactly as many unknowns as there are processes (i.e., MPI ranks).

However, the convergence properties of such a bottom level solver suffer from this very aggressive aggregation, compared with the aggregation in standard AGMG. To maintain the number of iterations needed to achieve the prescribed tolerance to only a few (typically 3–5), we consider using a better smoother than hybrid forward or backward Gauss–Seidel. Staying with the idea that the smoother should be purely parallel, one option is block Jacobi; that is, based on the exact inversion on each process of the local diagonal block. The method is then similar to a two-level non-overlapping additive Schwarz scheme [34]. However, for large 3D problems, the needed exact factorization of these matrix blocks is too costly from both memory and computational time viewpoints. The idea is then to replace, for each diagonal block, its exact inversion by the approximation resulting from one application of the *sequential* AGMG preconditioner. The convergence happens to be nearly

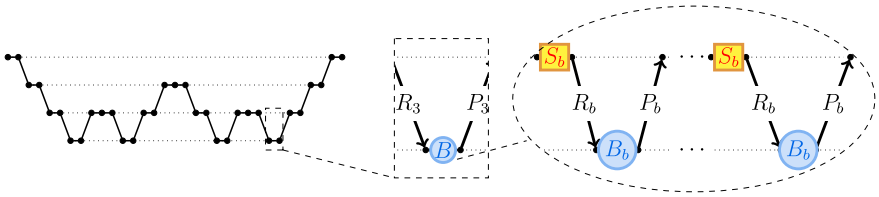

Fig. 2. Iterative two-level scheme as the bottom level solver: S_b is the bottom level smoother; R_b is the restriction to the very coarse grid; B_b is the very coarse grid solver; P_b is prolongation from the very coarse grid to the bottom level.

identical to that obtained with the smoother based on exact inversion, whereas computational and memory requirements are modest and proportional to the number of local unknowns at the bottom level.

Next, with such an excellent smoother we do not need in fact two smoothing iterations per preconditioning step as in Algorithm 2 (steps 1 and 7). We thus use only one step, in combination with a deflation based implementation of the PCG method [22], to avoid inconsistencies with standard PCG (that requires a symmetric preconditioner). Further, working out at a technical level, it turns out that the idea from [33] already used in Algorithm 1 can be further extended so as to group in a single global communication step both the finalization of the inner products needed by PCG and the gathering of the data needed for the very coarse grid correction.

Regarding the coarse grid system of the bottom level solver (i.e., as written above, the very coarse grid system with exactly one unknown per process), one option is to solve it sequentially with a sparse direct solver, each process repeating the solution to avoid a further communication to scatter the computed solution. This works fine up to 1000–10,000 cores, but, beyond, the time needed to solve these systems becomes a performance bottleneck. The idea is then to exchange these sequential solves for parallel ones, but using only a restricted set of processes. More precisely, we found that if there are p processes and thus p unknowns on the very coarse grid, using about $p/512$ processes to solve the systems in parallel is about optimal. Thus, typically, 512 subgroups are formed with $p/512$ processes each, and a parallel solution is performed within each subgroup. This way we also avoid a further global communication: the solution computed in parallel has only to be scattered inside the subgroup. Now, what method to use for this parallel solution? AGMG again, which is thus used recursively. However, to avoid an endless recursion, we make sure that, for this secondary call, the very coarse grid solver uses a sequential direct method; using a direct solver is harmless here since the number of processes has been reduced by a factor of 512.

Schematically, the work flow for each iteration of the bottom level solver is thus as follows.

1. Smoothing (S_b in Fig. 2): apply *sequential* AGMG to the local part of the residual.
2. Compute the local part of the needed inner products, and the local component of the right hand side vector for the very coarse grid correction (this latter: local part of R_b in Fig. 2).
3. Gather/Scatter data: inner products and right hand side of the very coarse grid system (this latter: global part of R_b in Fig. 2).
4. Solve the very coarse grid system (B_b in Fig. 2): depending on the number of unknowns, call either *parallel* AGMG within subgroups, or a sequential sparse direct solver.
5. Vector operations and residual update.

6. Results on petascale computers

Before considering the concurrent use of many computing nodes, the first point to address is whether the parallelization scheme is efficient on a single node. In the right columns of Table 3, we report the results of weak scaling experiments, using an increasing number of processes on a single node of some modern multicore architectures. At first sight, these results suggest a mixed answer, the computing time increasing significantly as we use more cores. It is however interesting to compare the numbers in the right columns with those in the left columns. There, we test the impact on the computing time of running simultaneously several instances of the sequential program. The resulting time therefore simulates an ideal situation where the impact of the communication may be neglected because the local systems are solved independently of each other. One sees that this “ideal” parallelization is not more efficient than that of AGMG.

The above observations are explained by hardware features. In particular, shared memory is efficient from the communication viewpoint, as most MPI implementations actually replace the communications with simple “write to memory”. However, there is some downside: caches and access channels are also at least partly shared, meaning that memory traffic slows down when more threads run concurrently. And it is a well-known fact that, in computations dealing with sparse matrices, one hardly achieves a small percentage of machine peak performance, because the real bottleneck is not CPU speed but memory access.

This analysis is further confirmed by checking the time needed to solve *sequentially* the largest systems considered in Table 3. In Table 1, we have seen that sequential AGMG requires a time nearly proportional to the number of unknowns. Accordingly, we would expect that, on Intel Farm, something like $16 \times 26.8 = 428.8$ seconds would be needed to solve a system 16 times larger than the system solved in 26.8 seconds. Instead, AGMG solves that system in 614.1 seconds. As

Table 3
Results with enhanced AGMG (iterative coarse grid solver) on a single multicore node.

#p	AGMG sequential (#p *concurrent run*)					AGMG parallel (*weak scaling rule*)				
	$\frac{n}{10^6}$	it.	$\mathcal{T}_{su}$	$\mathcal{T}_{sol}$	$\mathcal{T}_{tot}$	$\frac{n}{10^6}$	it.	$\mathcal{T}_{su}$	$\mathcal{T}_{sol}$	$\mathcal{T}_{tot}$
	Finite Difference on Intel Farm									
1	1×11.2	11	10.9	15.9	26.8	11.2	10	14.0	10.2	24.2
8	8×11.2	11	16.4	28.4	44.8	89.9	12	22.1	25.0	47.1
16	16×11.2	11	15.7	28.4	44.1	179.4	12	20.1	24.8	44.9
	Finite Difference on IBM BG/Q									
1	1×2.7	11	12.6	30.1	42.7	2.7	10	15.4	19.6	35.0
8	8×2.7	11	19.3	31.6	50.9	22.0	11	26.2	24.0	50.2
16	16×2.7	11	20.1	33.6	53.8	43.6	11	27.1	26.1	53.3
	Finite Difference on Cray XE6									
1	1×2.7	11	3.0	7.1	10.1	2.7	10	3.8	4.5	8.3
8	8×2.7	11	3.4	7.6	11.0	22.0	11	4.8	5.7	10.4
32	32×2.7	11	5.5	16.8	22.3	87.5	12	7.3	15.1	22.4

Table 4
Results with enhanced AGMG (iterative coarse grid solver); the number of cores is equal to the number of processes ("#p").

#p	$\frac{n}{10^6}$	it.	$\mathcal{T}_{su}$	$\mathcal{T}_{sol}$ ($\mathcal{T}_{bl}$)	$\mathcal{T}_{tot}$
	Finite Difference on Intel Farm				
16	179	12	20.1	24.8 (0.9)	44.9
64	719	12	22.4	25.7 (1.0)	48.1
512	5755	12	22.6	26.7 (1.7)	49.3
4096	46,037	12	22.9	29.0 (3.4)	51.9
13,824	155,374	12	22.7	33.7 (8.0)	56.4
32,768	368,293	12	23.3	41.9 (15.4)	65.2
	Finite Difference on IBM BG/Q				
16	44	11	27.1	26.1 (1.1)	53.2
64	176	11	28.0	26.6 (1.3)	54.6
512	1405	12	28.3	29.4 (1.7)	57.7
4096	11,239	12	28.3	31.3 (3.4)	59.6
32,768	89,915	12	28.5	34.8 (6.9)	63.3
110,592	303,464	13	28.7	52.9 (22.5)	81.6
373,248	1,024,193	13	29.3	81.2 (50.5)	110.5
	Finite Difference on Cray XE6				
32	88	12	7.3	15.1 (0.4)	22.4
128	349	12	7.5	15.5 (0.6)	23.1
512	1405	12	7.9	16.0 (0.7)	23.9
4096	11,239	12	8.2	17.0 (1.4)	25.1
32,768	89,915	12	8.6	19.7 (4.1)	28.3
110,592	303,464	13	8.9	29.4 (12.5)	38.3

there is no doubt that the number of arithmetic operations is still proportional to the number of unknowns, this has to be explained again by hardware features. In particular, the cache organization is more sophisticated on modern architectures, but this mostly benefits to computations that use only part of the memory of the computing node.

It also follows from this last observation that the *strong scalability* is in fact better than the weak scalability analyzed in Table 3: as written above, the time needed to solve sequentially on Intel Farm the system with 179 millions unknowns is 614.1 seconds, whereas, as reported in Table 3, using 16 processes we need 44.9 seconds. The speed up is thus $614.1/44.9 = 13.7$, which is not far from the ideal speed up of 16.

Hence, to conclude, although the situation depicted in Table 3 is not ideal, issues are not related to the parallelization model of AGMG, but rather to the suboptimal CPU usage of computer programs dealing with sparse matrices, and any competitor to AGMG would have to face similar difficulties.

We now report in Tables 4 and 5 the weak scalability results obtained when running on many computing nodes, keeping the number of processes per node equal to the number of available cores. The weak scalability curve is not perfectly flat, and the time needed on a single node can incur an increase by a factor up to two. Despite the improvements discussed in the preceding section, this is still the bottom level solver which is the major source of bottleneck.

Nevertheless, the results are satisfactory: in fact, the time is first essentially constant, and then increases but moderately, and a significant increase is only observed at a somehow extreme scale. For instance, the factor of two is seen on IBM BG/Q

Table 5
Results with enhanced AGMG (iterative coarse grid solver); the number of cores is equal to the number of processes ("#p").

#p	$\frac{n}{10^6}$	it.	$\mathcal{T}_{su}$	$\mathcal{T}_{solve}$ ($\mathcal{T}_{bl}$)	$\mathcal{T}_{tot}$
	Finite elements (P3) on Intel Farm				
16	37	24	10.2	42.9 (6.6)	53.1
64	147	25	10.2	43.1 (7.1)	53.3
512	1178	25	10.3	47.1 (9.0)	57.4
1728	3974	26	10.3	49.6 (11.3)	59.9
4096	9421	26	10.4	52.1 (12.4)	62.5
13,824	31,795	27	10.8	65.8 (24.1)	76.6
32,768	75,365	27	11.9	74.7 (33.3)	86.6
	Finite elements (P3) on IBM BG/Q				
16	9	23	18.1	50.0 (6.0)	68.1
64	38	24	18.5	53.0 (7.3)	71.5
512	303	25	19.1	56.0 (8.4)	75.1
4096	2428	25	18.8	54.4 (12.4)	73.2
32,768	19,422	26	19.4	64.4 (20.6)	83.8
	Finite elements (P3) on Cray XE6				
64	38	24	5.4	23.9 (2.7)	29.3
512	303	25	7.2	26.2 (3.5)	33.4
4096	2428	25	9.9	28.3 (5.2)	38.2
32,768	19,422	26	9.4	34.7 (9.8)	44.1
110,592	65,548	26	11.0	44.7 (20.1)	55.7

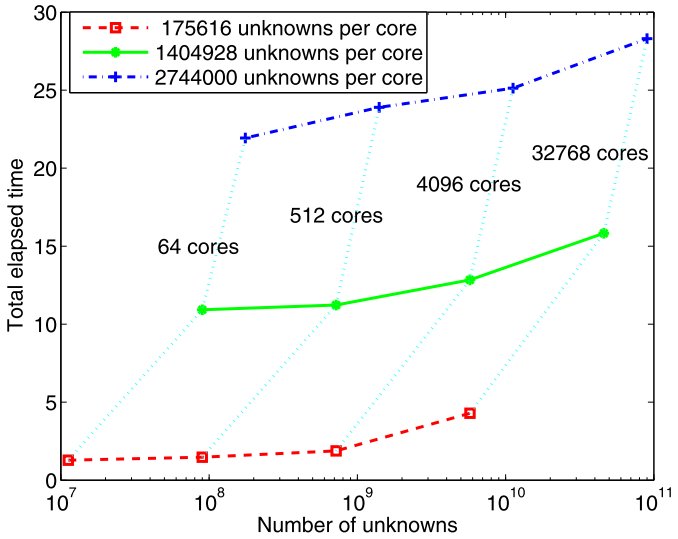

Fig. 3. Total elapsed time on Cray XE6 as a function of the system size, for different size per core; the number of processes is equal to the number of cores (Finite Difference).

when using more than 370,000 cores, that is, more than 80% of the machine ranked eighth in the top 500 supercomputer list of November 2013.

More importantly, when considering scalability results, one should never forget that their relevance depends on the quality of the sequential code one starts from. For instance, the factor of two mentioned above has to be put in perspective with the factor of three with respect to *hypre* observed in Table 1 (and on the many more tests reported in [24]). It is also clear that our parallelization strategy for aggregation-based AMG is more efficient than the one developed in [6].

On the other hand, it is also interesting to check the behavior of the solver for different problem sizes per core. This is done in Fig. 3, where one sees that the weak scalability remains nice even when there are significantly less unknowns per core. Consequently, the parallelization can also be used to compute the solution of moderately large linear systems in just a few seconds.

Finally, as seen in Tables 4 and 5, the bottom level solver remains the main source of performance bottleneck despite the significant improvements over the previous version. In Figs. 4–6 we give, in regard of the total elapsed time, the detail on how the time is spent during operations associated with this bottom level solver: "Computation" refers to the time spent with purely local computation; "Neighbor comm." refers to the time spent waiting for data from neighbor processes, needed to achieve Matvecs; "Global comm." refers to the time taken by the global communications (one per inner iteration, which,

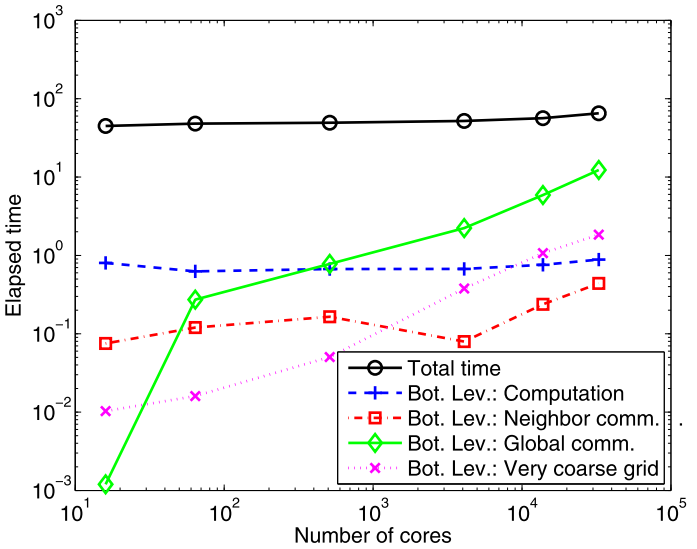

Fig. 4. Total time and partial times spent by the bottom level solver on Intel Farm as a function of the number of cores, for a fixed size of 11.2 millions of unknowns per core (Finite Difference).

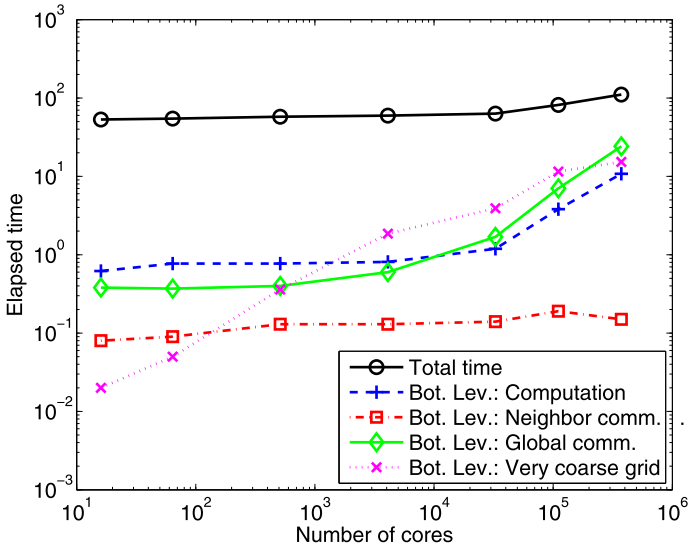

Fig. 5. Total time and partial times spent by the bottom level solver on IBM BG/Q as a function of the number of cores, for a fixed size of 2.75 millions of unknowns per core (Finite Difference).

as written above, gathers inner products needed by the conjugate gradient method and the exchange of data needed by the very coarse grid correction); "Very coarse grid" refers to the time taken by the solution of the linear systems on the very coarse grid with one unknown per core.

One sees that, for the largest tested configurations, the global communications represent in all cases the most time consuming part of the bottom level solver operations. Therefore, for the approach developed here to remain viable at larger scale (say, exascale), one needs to accompany the growth in the number of cores with the improvement of the communication network, in such a way that performing global communications remains affordable.

On IBM BG/Q and Cray XE the solution of the very coarse grid systems also takes a significant time. Note, however, that this part of the code has not been fully optimized; e.g., the number of cores per subgroup has not been finely tuned.

Finally, one may wonder why the purely sequential part of the algorithm appears also not completely scalable. Actually, this is an effect inherited from the trick used to gather all global communications in a single MPI reduce operation per inner iteration. Technically, this operation has to be an "Allgather", meaning that all partial inner products are gathered on each process in a long vector. The finalization of the inner products requires then, as part of the local computations, to sum all these partial components. This seems harmless, but on IBM BG/Q and Cray XE, the number of unknowns at bottom level is roughly $2.75 \times 10^6/500 \approx 5500$, whereas the number of components to sum is equal to the number of cores and may thus

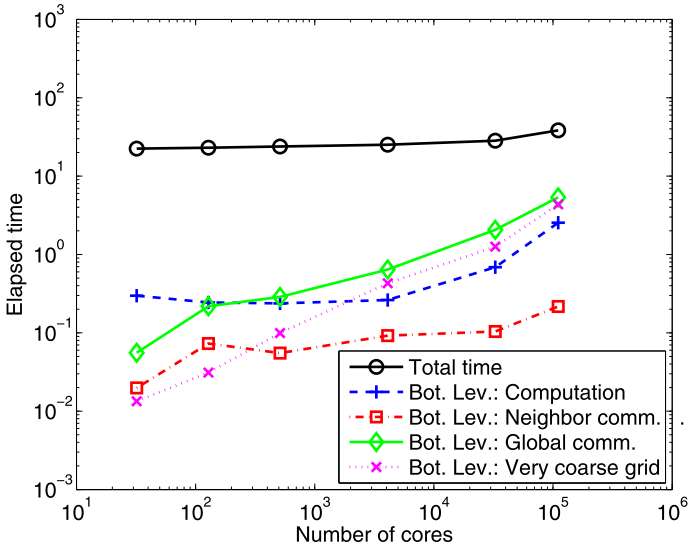

Fig. 6. Total time and partial times spent by the bottom level solver on Cray XE6 as a function of the number of cores, for a fixed size of 2.75 millions of unknowns per core (Finite Difference).

exceed 10^5. Here, a better implementation would parallelize the sum in subgroups, similarly to the parallelization scheme used to solve the very coarse grid systems.

7. Conclusions

We considered the AGMG software to solve discrete Poisson-like problems on massively parallel computers. AGMG implements the flexible conjugate gradient iterative method with an aggregation-based algebraic multigrid preconditioning. While aggregation itself is fairly well adapted to parallel computation, it necessitates the use of the K-cycle (instead of the classical V-cycle), which has been found a source of performance bottleneck because of the many bottom level solves required per iteration step. This issue has been addressed by drastically limiting the number of levels and therefore the number of bottom level solves per iteration. Because the bottom level systems are then much larger than usual with multigrid, this required the development of an appropriate (and highly parallel) bottom level solver. We were successful in this task by considering a two-level method that combines very aggressive aggregation with a smoother based on *sequential* AGMG applied ignoring matrix entries that connect unknowns on different processors.

The numerical experiments on some of today largest computers show excellent weak scalability even for several hundred thousands of cores.

Acknowledgement

We acknowledge PRACE for awarding us access to resources CURIE (Intel Farm at CEA, France), JUQUEEN (IBM BG/Q at Juelich, Germany) and HERMIT (Cray XE6 at HLRS, Stuttgart, Germany).

Appendix A

Here we show that Algorithm 1 is mathematically equivalent to the FCG method introduced in [26]; more particularly, to the FCG(1) variant which is the cheapest and the closest to the standard PCG algorithm (compared with this latter, it requires only one more inner product computation per iteration). We first recall the algorithm given in [26].

Algorithm 3 (FCG(1) method from [26] to solve $A\,\mathbf{u} = \mathbf{b}$)

1. Initialization: select initial approximation $\tilde{\mathbf{u}}_0$ and set $\tilde{\mathbf{r}}_0 = \mathbf{b} - A\,\tilde{\mathbf{u}}_0$
2. **for** $k = 0, 1, \ldots$ until convergence
3. $\quad \tilde{\mathbf{v}}_k = \mathcal{B}(\tilde{\mathbf{r}}_k)$
4. $\quad$ **if** $k > 0$
5. $\quad\quad$ **then**
6. $\quad\quad\quad \tilde{\mathbf{d}}_k = \tilde{\mathbf{v}}_k - \frac{\tilde{\mathbf{v}}_k^T A \tilde{\mathbf{d}}_{k-1}}{\tilde{\mathbf{d}}_{k-1}^T A \tilde{\mathbf{d}}_{k-1}} \tilde{\mathbf{d}}_{k-1}$
7. $\quad\quad$ **else**
8. $\quad\quad\quad \tilde{\mathbf{d}}_k = \tilde{\mathbf{v}}_k$

9. $\tilde{\mathbf{u}}_{k+1} = \tilde{\mathbf{u}}_k + \frac{\tilde{\mathbf{d}}_k^T \tilde{\mathbf{r}}_k}{\tilde{\mathbf{d}}_k^T A \tilde{\mathbf{d}}_k} \tilde{\mathbf{d}}_k$

10. $\tilde{\mathbf{r}}_{k+1} = \tilde{\mathbf{r}}_k - \frac{\tilde{\mathbf{d}}_k^T \tilde{\mathbf{r}}_k}{\tilde{\mathbf{d}}_k^T A \tilde{\mathbf{d}}_k} A\tilde{\mathbf{d}}_k$

The following result then holds.

Lemma A.1. *Algorithms 1 and 3 are mathematically equivalent if A is symmetric and $\mathbf{u}_0 = \tilde{\mathbf{u}}_0$. More precisely, for any $k \geq 0$ one has then $\mathbf{v}_k = \tilde{\mathbf{v}}_k$, $\mathbf{d}_k = \tilde{\mathbf{d}}_k$, $\mathbf{q}_k = A\tilde{\mathbf{d}}_k$, $\mathbf{u}_{k+1} = \tilde{\mathbf{u}}_{k+1}$, $\mathbf{r}_{k+1} = \tilde{\mathbf{r}}_{k+1}$ and*

$$\rho_k = \tilde{\mathbf{d}}_k^T A \tilde{\mathbf{d}}_k, \tag{A.1}$$

$$\alpha_k = \mathbf{d}_k^T \mathbf{r}_k. \tag{A.2}$$

Proof. The proof is by induction. First, the statement is easily checked for $k = 0$. Next, assume that it holds for $k - 1$. Then the steps 3 of both algorithms imply $\mathbf{v}_k = \tilde{\mathbf{v}}_k$. Comparing for $k > 0$ steps 9, 10 of Algorithm 1 with step 6 of Algorithm 3, and using the equalities $\mathbf{q}_{k-1} = A\tilde{\mathbf{d}}_{k-1}$ and $\rho_{k-1} = \tilde{\mathbf{d}}_{k-1}^T A \tilde{\mathbf{d}}_{k-1}$ (induction on (A.1)), one obtains $\mathbf{d}_k = \tilde{\mathbf{d}}_k$. Then, steps 4, 10, 11 of Algorithm 1 together with $\mathbf{q}_{k-1} = A\tilde{\mathbf{d}}_{k-1}$ yield $\mathbf{q}_k = A\tilde{\mathbf{d}}_k$. Further, for $k > 0$ the equality (A.1) is obtained by noting that $\mathbf{d}_k^T A \mathbf{d}_{k-1} = \tilde{\mathbf{d}}_k^T A \tilde{\mathbf{d}}_{k-1} = 0$ as follows from step 6 of Algorithm 3, and hence, using step 10 of Algorithm 1,

$$\tilde{\mathbf{d}}_k^T A \tilde{\mathbf{d}}_k = \mathbf{d}_k^T A \mathbf{d}_k = \mathbf{d}_k^T A \mathbf{v}_k - \frac{\gamma_k}{\rho_{k-1}} \underbrace{\mathbf{d}_k^T A \mathbf{d}_{k-1}}_{0} = \underbrace{\mathbf{v}_k^T A \mathbf{v}_k}_{\beta_k} - \frac{\gamma_k}{\rho_{k-1}} \underbrace{\mathbf{d}_{k-1}^T A \mathbf{v}_k}_{\gamma_k} = \rho_k.$$

To show (A.2), we note that $\tilde{\mathbf{d}}_{k-1}^T \tilde{\mathbf{r}}_k = 0$ as follows from step 10 of Algorithm 3. Hence, for $k > 0$, there holds

$$\tilde{\mathbf{d}}_k^T \tilde{\mathbf{r}}_k = \tilde{\mathbf{v}}_k^T \tilde{\mathbf{r}}_k - \frac{\tilde{\mathbf{v}}_k^T A \tilde{\mathbf{d}}_{k-1}}{\tilde{\mathbf{d}}_{k-1}^T A \tilde{\mathbf{d}}_{k-1}} \underbrace{\tilde{\mathbf{d}}_{k-1}^T \tilde{\mathbf{r}}_k}_{0} = \mathbf{v}_k^T \mathbf{r}_k = \alpha_k.$$

Eventually, comparing for $k > 0$ the steps 17, 18 of Algorithm 1 with steps 9, 10 of Algorithm 3 and using (A.1), (A.2) as well as $\mathbf{q}_k = A\tilde{\mathbf{d}}_k$, one concludes that $\mathbf{u}_{k+1} = \tilde{\mathbf{u}}_{k+1}$ and $\mathbf{r}_{k+1} = \tilde{\mathbf{r}}_{k+1}$. □

References

[1] M. Adams, M. Brezina, J. Hu, R. Tuminaro, Parallel multigrid smoothing: polynomial versus Gauss–Seidel, J. Comput. Phys. 188 (2003) 593–610.
[2] A. Adelmann, P. Arbenz, Y. Ineichen, A fast parallel Poisson solver on irregular domains applied to beam dynamics simulations, J. Comput. Phys. 229 (2010) 4554–4566.
[3] A. Adelmann, P. Arbenz, Y. Ineichen, Improvements of a fast parallel Poisson solver on irregular domains, in: K. Jónasson (Ed.), Applied Parallel and Scientific Computing, in: Lect. Notes Comput. Sci., vol. 7133, Springer, Berlin, Heidelberg, 2012, pp. 65–74.
[4] A.H. Baker, R.D. Falgout, T.V. Kolev, U.M. Yang, Multigrid smoothers for ultraparallel computing, SIAM J. Sci. Comput. 33 (2011) 2864–2887.
[5] A.H. Baker, T. Gamblin, M. Schulz, U.M. Yang, Challenges of scaling algebraic multigrid across modern multicore architectures, in: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, IPDPS '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 275–286.
[6] M. Blatt, O. Ippisch, P. Bastian, A massively parallel algebraic multigrid preconditioner based on aggregation for elliptic problems with heterogeneous coefficients, http://arxiv.org/abs/1209.0960, 2013.
[7] A. Brandt, S.F. McCormick, J.W. Ruge, Algebraic multigrid (AMG) for sparse matrix equations, in: D.J. Evans (Ed.), Sparsity and Its Application, Cambridge University Press, Cambridge, 1984, pp. 257–284.
[8] M. Duarte, Z. Bonaventura, M. Massot, A. Bourdon, A numerical strategy to discretize and solve Poisson equation on dynamically adapted multiresolution grids for time-dependent streamer discharge simulation, http://hal.archives-ouvertes.fr/hal-00903307, 2014.
[9] H. Elman, V. Howle, J. Shadid, R. Shuttleworth, R. Tuminaro, A taxonomy and comparison of parallel block multi-level preconditioners for the incompressible Navier–Stokes equations, J. Comput. Phys. 227 (2008) 1790–1808.
[10] M. Emans, Performance of parallel AMG-preconditioners in CFD-codes for weakly compressible flows, Parallel Comput. 36 (2010) 326–338.
[11] M. Emans, Coarse-grid treatment in parallel AMG for coupled systems in CFD applications, J. Comput. Sci. 2 (2011) 365–376.
[12] M. Emans, Krylov-accelerated algebraic multigrid for semi-definite and nonsymmetric systems in computational fluid dynamics, Numer. Linear Algebra Appl. 19 (2012) 210–231.
[13] R.D. Falgout, J.B. Schroder, Non-Galerkin coarse grids for algebraic multigrid, Tech. Rep. LLNL-JRNL-641635, Lawrence Livermore National Laboratory, Center for Applied Scientific Computing, Livermore, CA, USA, 2013.
[14] H. Gahvari, A.H. Baker, M. Schulz, U.M. Yang, K.E. Jordan, W. Gropp, Modeling the performance of an algebraic multigrid cycle on HPC platforms, in: D.K. Lowenthal, B.R. Supinski, S.A. McKee (Eds.), Proceedings of the 25th International Conference on Supercomputing, 2011, Tucson, AZ, USA, May 31–June 04, 2011, ACM, 2011, pp. 172–181.
[15] M.W. Gee, C.M. Siefert, J. Hu, R.S. Tuminaro, M. Sala, Ml 5.0 smoothed aggregation user's guide, Tech. Rep. SAND2006-2649, Sandia National Laboratories, Albuquerque, NM, USA, 2006.
[16] B. Gmeiner, H. Köstler, M. Stürmer, U. Rüde, Parallel multigrid on hierarchical hybrid grids: a performance study on current high performance computing clusters, Concurr. Comput. 26 (2014) 217–240.
[17] V.E. Henson, U.M. Yang, BoomerAMG: a parallel algebraic multigrid solver and preconditioner, Appl. Numer. Math. 41 (2002) 155–177.
[18] C. Hirsch, Numerical Computation of Internal and External Flows, 2nd edition, Elsevier, Amsterdam, 2003.
[19] Hypre software and documentation, available at https://computation.llnl.gov/casc/linear_solvers/sls_hypre.html.
[20] MUMPS software and documentation, available online at http://graal.ens-lyon.fr/MUMPS/.

[21] A.C. Muresan, Y. Notay, Analysis of aggregation-based multigrid, SIAM J. Sci. Comput. 30 (2008) 1082–1103.
[22] R. Nabben, C. Vuik, A comparison of deflation and the balancing preconditioner, SIAM J. Sci. Comput. 27 (2006) 1742–1759.
[23] A. Napov, Y. Notay, An algebraic multigrid method with guaranteed convergence rate, SIAM J. Sci. Comput. 34 (2012) A1079–A1109.
[24] A. Napov, Y. Notay, Algebraic multigrid for moderate order finite elements, SIAM J. Sci. Comput. 36 (2014) A1678–A1707; preprint available online at http://homepages.ulb.ac.be/~ynotay.
[25] Y. Notay, AGMG software and documentation, see http://homepages.ulb.ac.be/~ynotay/AGMG.
[26] Y. Notay, Flexible conjugate gradients, SIAM J. Sci. Comput. 22 (2000) 1444–1460.
[27] Y. Notay, An aggregation-based algebraic multigrid method, Electron. Trans. Numer. Anal. 37 (2010) 123–146.
[28] Y. Notay, Aggregation-based algebraic multigrid for convection–diffusion equations, SIAM J. Sci. Comput. 34 (2012) A2288–A2316.
[29] Y. Notay, P.S. Vassilevski, Recursive Krylov-based multigrid cycles, Numer. Linear Algebra Appl. 15 (2008) 473–487.
[30] V. Novák, P. Kočí, F. Štěpánek, M. Marek, Integrated multiscale methodology for virtual prototyping of porous catalysts, Ind. Eng. Chem. Res. 50 (2011) 12904–12914.
[31] M. Pennacchio, V. Simoncini, Fast structured AMG preconditioning for the bidomain model in electrocardiology, SIAM J. Sci. Comput. 33 (2011) 721–745.
[32] J.W. Ruge, K. Stüben, Algebraic multigrid (AMG), in: S.F. McCormick (Ed.), Multigrid Methods, in: Front. Appl. Math., vol. 3, SIAM, Philadelphia, PA, 1987, pp. 73–130.
[33] Y. Saad, Practical use of polynomial preconditionings for the conjugate gradient method, SIAM J. Sci. Stat. Comput. 6 (1985) 865–881.
[34] B. Smith, P. Bjørstad, W. Gropp, Domain Decomposition, Cambridge University Press, Cambridge, 1996.
[35] K.C. Smith, T.S. Fisher, Conduction in jammed systems of tetrahedra, J. Heat Transf. 135 (2013), Article ID 081301, 7 pp.
[36] U. Trottenberg, C.W. Oosterlee, A. Schüller, Multigrid, Academic Press, London, 2001.
[37] R.S. Tuminaro, C. Tong, Parallel smoothed aggregation multigrid: aggregation strategies on massively parallel machines, in: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, Supercomputing '00, IEEE Computer Society, Washington, DC, USA, 2000.
[38] T. Unfer, An asynchronous framework for the simulation of the plasma/flow interaction, J. Comput. Phys. 236 (2013) 229–246.
[39] M. ur Rehman, C. Vuik, G. Segal, SIMPLE-type preconditioners for the Oseen problem, Int. J. Numer. Methods Fluids 61 (2008) 432–452.
[40] P. Vaněk, J. Mandel, M. Brezina, Algebraic multigrid based on smoothed aggregation for second and fourth order elliptic problems, Computing 56 (1996) 179–196.
[41] P.S. Vassilevski, U.M. Yang, Reducing communication in algebraic multigrid using additive variants, Numer. Linear Algebra Appl. 21 (2014) 275–296.