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

A R T I C L E   I N F O

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

A B S T R A C T

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/© 2014 Elsevier Inc. All rights reserved.

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.