Nonsymmetric preconditioning for conjugate gradient and steepest descent methods

Bouwmeester, H.; Dougherty, A.; Knyazev, A.

TR2015-113 June 2015

Abstract

We analyze a possibility of turning off post-smoothing (relaxation) in geometric multigrid when used as a preconditioner in preconditioned conjugate gradient (PCG) linear and eigenvalue solvers for the 3D Laplacian. The geometric Semicoarsening Multigrid (SMG) method is provided by the hypre parallel software package. We solve linear systems using two variants (standard and flexible) of PCG and preconditioned steepest descent (PSD) methods. The eigenvalue problems are solved using the locally optimal block preconditioned conjugate gradient (LOBPCG) method available in hypre through BLOPEX software. We observe that turning off the post-smoothing in SMG dramatically slows down the standard PCG-SMG. For flexible PCG and LOBPCG, our numerical tests show that removing the post-smoothing results in overall 40-50 percent acceleration, due to the high costs of smoothing and relatively insignificant decrease in convergence speed. We demonstrate that PSD-SMG and flexible PCG-SMG converge similarly if SMG post-smoothing is off. A theoretical justification is provided.

International Conference on Computational Science (ICCS)

Nonsymmetric preconditioning for conjugate gradient and steepest descent methods[*]

Henricus Bouwmeester[1], Andrew Dougherty[1], and Andrew V. Knyazev[2]

[1] University of Colorado Denver, USA
Henricus.Bouwmeester@ucdenver.edu and Andrew.Dougherty@ucdenver.edu
[2] Mitsubishi Electric Research Laboratories, Cambridge, MA, USA
Andrew.Knyazev@merl.com

Abstract
We analyze a possibility of turning off post-smoothing (relaxation) in geometric multigrid when used as a preconditioner in preconditioned conjugate gradient (PCG) linear and eigenvalue solvers for the 3D Laplacian. The geometric Semicoarsening Multigrid (SMG) method is provided by the *hypre* parallel software package. We solve linear systems using two variants (standard and flexible) of PCG and preconditioned steepest descent (PSD) methods. The eigenvalue problems are solved using the locally optimal block preconditioned conjugate gradient (LOBPCG) method available in *hypre* through BLOPEX software. We observe that turning off the post-smoothing in SMG dramatically slows down the standard PCG-SMG. For flexible PCG and LOBPCG, our numerical results show that post-smoothing can be avoided, resulting in overall acceleration, due to the high costs of smoothing and relatively insignificant decrease in convergence speed. We demonstrate for linear systems that PSD-SMG and flexible PCG-SMG converge similarly if SMG post-smoothing is off. A theoretical justification is provided.

Keywords: linear equations, eigenvalue, iterative, multigrid, smoothing, preconditioning, conjugate gradient, steepest descent, convergence, parallel, hypre, BLOPEX, LOBPCG

1 Introduction

Smoothing (relaxation) and coarse-grid correction are the two cornerstones of multigrid technique. In algebraic multigrid, where only the system matrix is (possibly implicitly) available, smoothing is more fundamental since it is often used to construct the coarse grid problem. In geometric multigrid, the coarse grid is generated by taking into account the geometry of the fine grid, in addition to the chosen smoothing procedure. If full multigrid is used as a stand-alone solver, proper smoothing is absolutely necessary for convergence. If multigrid is used as a preconditioner in an iterative method, one is tempted to check what happens if smoothing is turned partially off.

[*]A preliminary version posted at http://arxiv.org/abs/1212.6680 is already cited three times

For symmetric positive definite (SPD) linear systems, the preconditioner is typically required to be also a fixed linear SPD operator, to preserve the symmetry of the preconditioned system. In the multigrid context, the preconditioner symmetry is achieved by using balanced pre- and post-smoothing, and by properly choosing the restriction and prolongation pair. In order to get a fixed linear preconditioner, one avoids using nonlinear smoothing, restriction, prolongation, or coarse solves. The positive definiteness is obtained by performing enough (in practice, even one may be enough), and an equal number of, pre- and post-smoothing steps; see, e.g., [6].

If smoothing is unbalanced, e.g., there is one step of pre-smoothing, but no post-smoothing, the multigrid preconditioner becomes nonsymmetric. Traditional assumptions of the standard convergence theory of iterative solvers are no longer valid, and convergence behavior may be unpredictable. The main goal of this paper is to describe our numerical experience experimenting with the influence of unbalanced smoothing in practical geometric multigrid preconditioning, specifically, the Semicoarsening Multigrid (SMG) method, see [15], provided by the parallel software package *hypre* [1].

We numerically analyze the possibility of turning off post-smoothing in geometric multigrid when used as a preconditioner in iterative linear and eigenvalue solvers for the 3D Laplacian in *hypre*. We solve linear systems using two variants (standard and flexible, e.g., [8]) of the preconditioned conjugate gradient (PCG) and preconditioned steepest descent (PSD) methods. The standard PCG is already coded in *hypre*. We have written the codes of flexible PCG and PSD by modifying the *hypre* standard PCG function. The eigenvalue problems are solved using the locally optimal block preconditioned conjugate gradient (LOBPCG) method, readily available in *hypre* through BLOPEX [3].

We observe that turning off the post-smoothing in SMG dramatically slows down the standard PCG-SMG. However, for the flexible PCG and LOBPCG, our numerical tests show that post-smoothing can be avoided. In the latter case, turning off the post-smoothing in SMG results in overall acceleration, due to the high costs of smoothing and relatively insignificant decrease in convergence speed. Our observations are also generally applicable for algebraic multigrid preconditioning for graph Laplacians, appearing, e.g., in computational photography problems, as well as in 3D mesh processing tasks [13].

A different case of non-standard preconditioning, specifically, *variable preconditioning*, in PCG is considered in our earlier work [10]. There, we also find a dramatic difference in convergence speed between the standard and flexible version of PCG. The better convergence behavior of the flexible PCG is explained in [10] by its *local optimality*, which guarantees its convergence with at least the speed of PSD. Our numerical tests there show that, in fact, the convergence of PSD and the flexible PCG is practically very close. We perform the same comparison here, and obtain a similar result. We demonstrate for linear systems that PSD-SMG converges almost as fast as the flexible PCG-SMG if SMG post-smoothing is off in both methods. Our numerical experiments are executed in both a strictly shared memory environment and in a distributed memory environment, demonstrating that the effect of acceleration is does not depend on the memory interconnection speed.

While our numerical tests all use multigrid preconditioning, see also [7], our theory predicts that non-symmetric preconditioning for symmetric positive definite linear systems and eigenvalue problems could be efficient in general, e.g., for domain decomposition and inexact factorization preconditioners.

The rest of the paper is organized as follows. We formally describe the PSD and PCG methods used here for testing, and explain their differences. We briefly discuss the SMG preconditioning in *hypre* and present our numerical results for linear systems. One section is dedicated to eigenvalue problems. Our last section contains the relevant theory.

2 PSD and PCG methods

For a general exposition of PSD and PCG, let SPD matrices A and T, and vectors b and x_0 be given, and denote $r_k = b - Ax_k$. Algorithm 1 is described in [10]

Algorithm 1: PSD and PCG methods

```
1  for $k = 0, 1, \ldots$ do
2      $s_k = Tr_k$
3      if $k = 0$ then
4          $p_0 = s_0$
5      else
6          $p_k = s_k + \beta_k p_{k-1}$ (where $\beta_k$ is either (1) or (2) for all iterations)
7      end
8      $\alpha_k = \frac{(s_k, r_k)}{(p_k, Ap_k)}$
9      $x_{k+1} = x_k + \alpha_k p_k$
10     $r_{k+1} = r_k - \alpha_k Ap_k$
11 end
```

Various methods are obtained by using different formulas for the scalar β_k. We set $\beta_k = 0$ for PSD,

$$\beta_k = \frac{(s_k, r_k)}{(s_{k-1}, r_{k-1})} \tag{1}$$

for the standard PCG, or

$$\beta_k = \frac{(s_k, r_k - r_{k-1})}{(s_{k-1}, r_{k-1})} \tag{2}$$

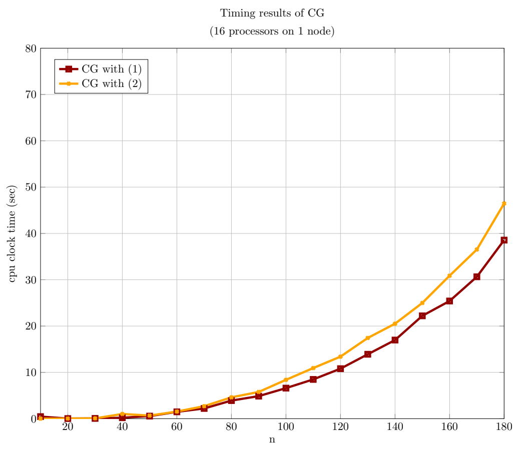
for the flexible PCG.

We note that in using (2), we are merely subtracting one term, (s_k, r_{k-1}), in the numerator of (1), which appears in the standard CG algorithm. If T is a fixed SPD matrix, this term actually vanishes; see, e.g., [10]. By using (2) in a computer code, it is required that an extra vector be allocated to either calculate $r_k - r_{k-1}$ or store $-\alpha_k Ap_k$, compared to (1). The associated costs may be noticeable for large problems solved on parallel computers. Next, we numerically evaluate the extra costs by comparing the standard and flexible PCG with no preconditioning for a variety of problem sizes.

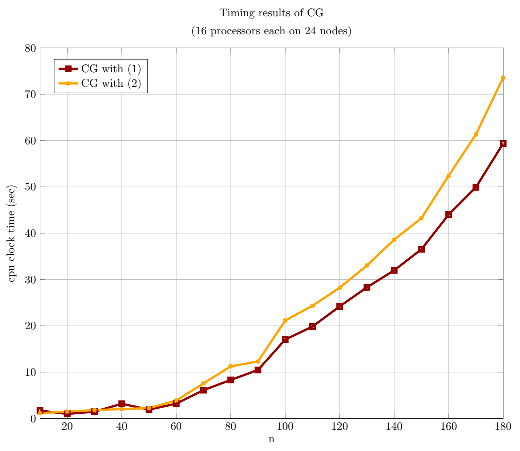
Our model problem used for all calculations in the present paper is for the three-dimensional negative Laplacian in a brick with homogeneous Dirichlet boundary conditions approximated by the standard finite difference scheme using the 7-point stencil with the grid size one in all three directions. The initial approximation is (pseudo)random.

In Figure 1, for the CG method without preconditioning we see a 20-25% cost overhead incurred due to the extra storage and calculation for the flexible variant, (2), relative to the standard variant, (1). Note that in each instance of the problem, the number of iterations of the CG method is the same regardless if either (1) or (2) is used for β_k.

From Figure 1 one observes that the distributed memory (DM) case is not perfectly scalable, as there is about a 50% increase when the size of the problem is increased 24 times, proportionally to the number of processors. The same comparison actually holds for all other (a) and (b) figures up to Figure 5.



(a) Shared memory

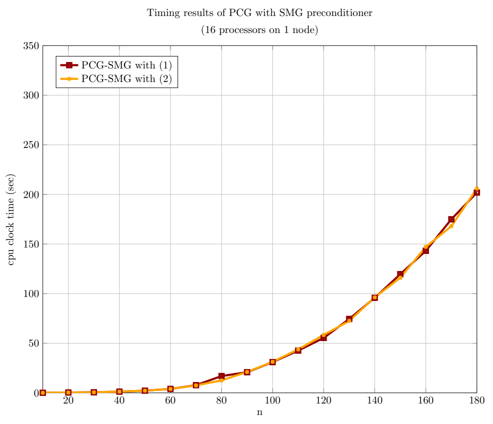


(b) Distributed memory

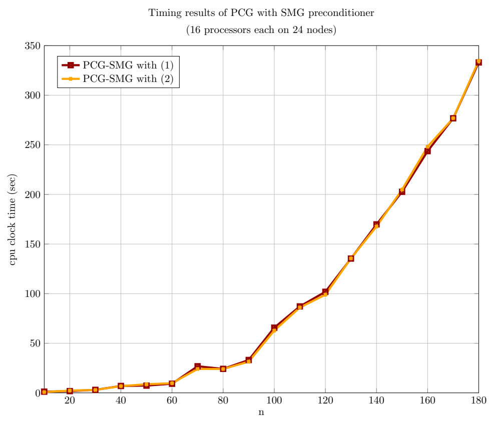
Figure 1: Timing of unpreconditioned CG using (1) and (2).

3 Preconditioning with SMG

In order to help overcome the slow convergence of the CG method, it is customary to introduce preconditioning. Here, we use the SMG solver as a preconditioner, provided by *hypre*. The SMG solver/preconditioner uses plane-relaxation as a smoother at each level in the V-cycle [1].



(a) Shared memory



(b) Distributed memory

Figure 2: Cost for storage and calculation in PCG-SMG with (1) or (2)

Although using (2) introduces some extra overhead, in the case of the SMG preconditioning, this is negligible as can be seen in Figure 2, since the SMG preconditioning, first, is relatively expensive computationally and, second, makes the PCG converge much faster, compared to the non-preconditioned case, The PSD method performs a bit worse (only 2-3 more iterations) than either variant of PCG, which clearly indicates the high quality of the SMG preconditioner for these problems.

Using the same number of pre- and post-relaxation steps creates a symmetric preconditioner, in this test, an SPD preconditioner. Thus both PCG-SMG, (1) and (2), generate identical (up to round-off errors) iterative approximations.

If no post-relaxation is performed within the SMG preconditioner, the convergence of the standard PCG method, i.e., with (1), is significantly slowed down, almost to the level where preconditioning is useless. In PCG-SMG using (2) without post-relaxation of the SMG preconditioner, we still achieve a very good convergence rate, as well as for PSD-SMG.

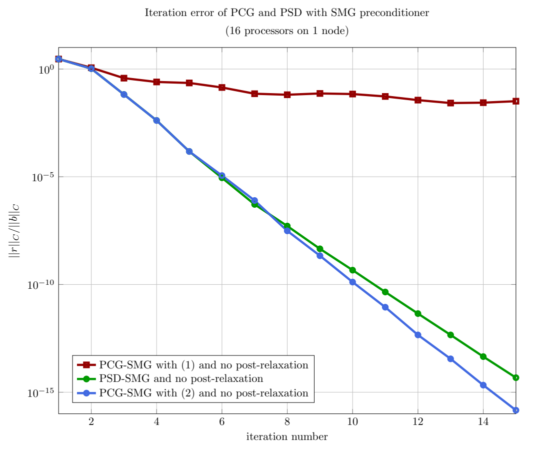


Figure 3: Iteration Error for PCG-SMG and PSD-SMG

The convergence behavior in Figure 3 is similar to that observed in [10]. There, a variable SPD preconditioner makes the standard PCG, i.e., using (1), almost stall, while the convergence rates of PSD and flexible PCG, i.e., with (2), are good and close to each other.

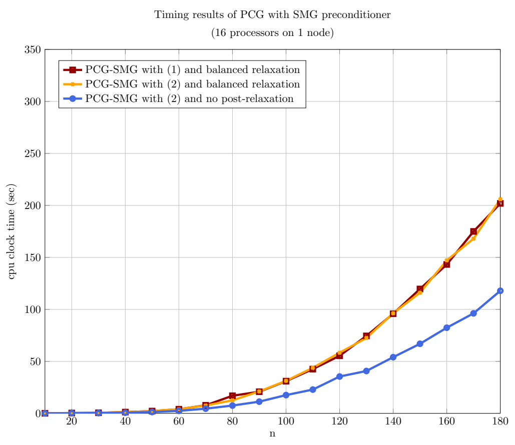
However, the SMG preconditioner is fixed and linear, according to its description in [15] and our numerical verification, so the theoretical explanation of such a convergence behavior in [10] is not directly applicable here. Moreover, turning off the post-relaxation smoothing in a multigrid preconditioner makes it nonsymmetric—the case not theoretically covered in [10], where the assumption is always made that the preconditioner is SPD.

We add the data for the absence of post-relaxation (option *-v 1 0*) in the SMG preconditioner to Figure 2 to obtain Figure 4. The number of iterations of the solver does increase a bit for both (2) and for the PSD method with no post-relaxation in the SMG preconditioner, but not enough to outweigh the cost savings of not using the post-relaxation in the SMG preconditioner. The overall improvement is 43% for all problems tested.

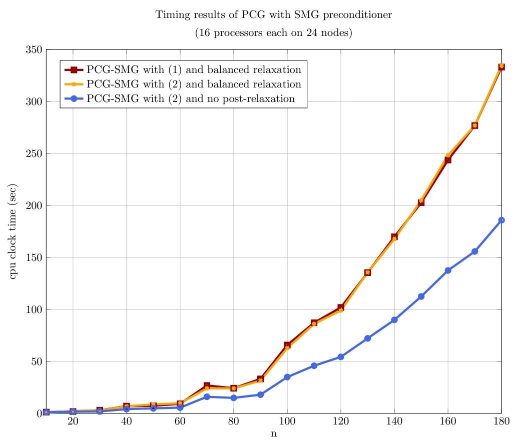
4 Preconditioning of LOBPCG with SMG

The LOBPCG method [9] computes the m smallest eigenvalues of a linear operator and is implemented within *hypre* through BLOPEX; see [3]. We conclude our numerical experiments with a comparison of the use of balanced and unbalanced relaxation in the SMG preconditioner for the LOBPCG method with $m = 1$.

In these tests, the matrix remains the same as in the previous section, i.e., corresponds to the three-dimensional negative Laplacian in a brick with homogeneous Dirichlet boundary conditions approximated by the standard finite difference scheme using the 7-point stencil with



(a) Shared memory



(b) Distributed memory

Figure 4: Cost comparison of relaxation in PCG-SMG using (1) or (2)

the grid size of one in all three directions. But in this section we compute the smallest eigenvalue and the corresponding eigenvector, rather than solve a linear system. The initial approximation to the eigenvector is (pseudo)random.

For this experiment, as seen in Figure 6, the number of LOBPCG iterations for the non-balanced SMG preconditioner is roughly 50% more than that for the balanced SMG preconditioner. However, the cost savings, which is about 30-50%, of not using the post-relaxation in the SMG preconditioner outweigh the higher number of iterations, as shown in Figure 5, and thus justify turning off the post-relaxation in this case. From Figure 5 one observes that the distributed memory case is not perfectly scalable, as there is about a 50% increase when the size of the problem is increased 24 times, proportionally to the number of processors. Finally, we note that the existing LOBPCG convergence theory in [9, 12] requires an SPD preconditioner T, and does not explain convergence if T is nonsymmetric.

5 Theoretical justification

For linear systems with SPD coefficient matrices, the use of nonsymmetric preconditioning has been justified, e.g., in [4, Section 12.3], [5, 14], and [16, Section 10.2]. The arguments used there are applicable for nonsymmetric and variable preconditioning.

Specifically, it is shown in [16, Section 10.2] that the flexible PCG, i.e., using (2), is locally optimal, i.e., on every step it converges not slower than PSD. The convergence rate bound for the PSD with nonsymmetric preconditioning established in [16, Section 10.2] is

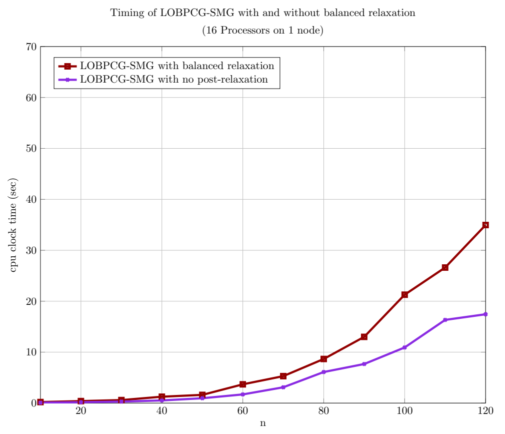
$$\|r_{k+1}\|_{A^{-1}} \le \delta \|r_k\|_{A^{-1}}, \tag{3}$$

under the assumption

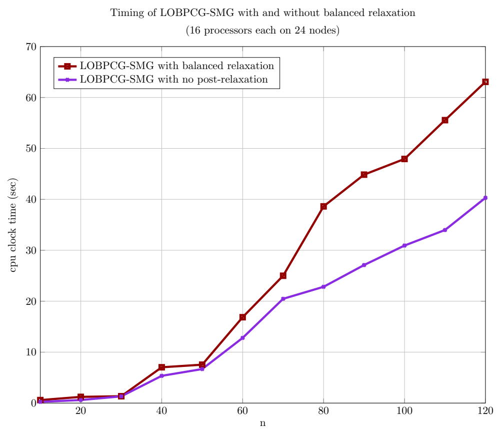
$$\|I - AT\|_{A^{-1}} \le \delta < 1, \tag{4}$$

where $\|\cdot\|_{A^{-1}}$ denotes the operator norm induced by the corresponding vector norm $\sqrt{x'A^{-1}x}$.

The key identity for PSD, that easily leads to bound (3), is presented in the following theorem.

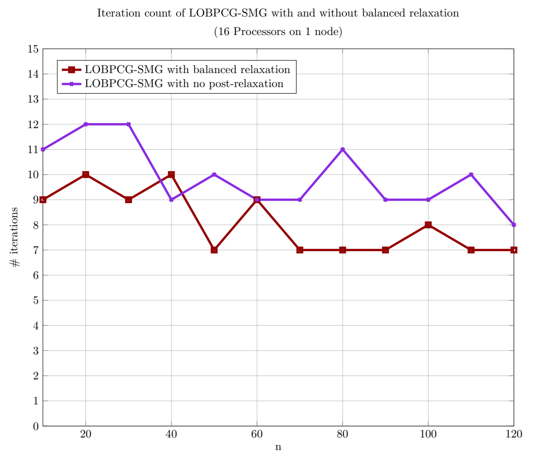


(a) Shared memory

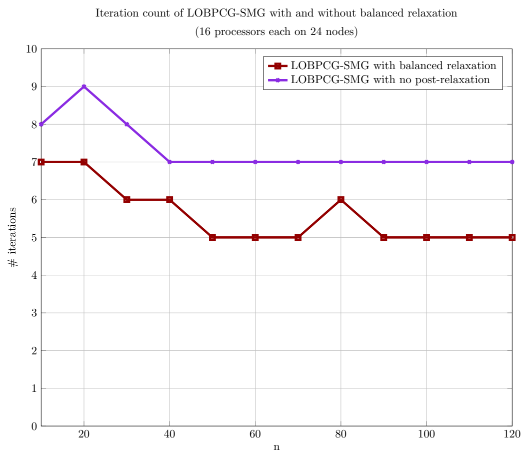


(b) Distributed memory

Figure 5: Cost for higher relaxation level for LOBPCG-SMG



(a) Shared memory



(b) Distributed memory

Figure 6: Iteration comparison of relaxation levels for LOBPCG-SMG

Theorem 1. *The identity holds,*

$$\|r_{k+1}\|_{A^{-1}}/\|r_k\|_{A^{-1}} = \sin\left(\angle_{A^{-1}}\{r_k, ATr_k\}\right), \tag{5}$$

where the right-hand side is defined via

$$\cos\left(\angle_{A^{-1}}\{r_k, ATr_k\}\right) = \frac{|(r_k)'Tr_k|}{\|r_k\|_{A^{-1}}\|ATr_k\|_{A^{-1}}}.$$

Proof. Identity (5) is actually proved, although not explicitly formulated, in the proof of [16, Theorem 10.2]. Alternatively, identity (5) is equivalent to

$$\|e_{k+1}\|_A/\|e_k\|_A = \sin\left(\angle_A\{e_k, TAe_k\}\right), \tag{6}$$

where $Ae_k = r_k$, which is the statement of [10, Lemma 4.1]. We note that [10] generally assumes that the preconditioner T is SPD, but this assumption is not actually used in the proof of [10, Lemma 4.1]. $\square$

Assumption (4) is very simple, but has one significant drawback—it does not allow arbitrary scaling of the preconditioner T, while the PCG and PSD methods are invariant with respect to scaling of T. The way around it is to scale the preconditioner T *before* assumption (4) is verified. We now illustrate such a scaling under an additional assumption that T is SPD, following [10]. We start with a theorem, connecting assumption (4) with its equivalent, and probably more traditional form.

Theorem 2. *Let the preconditioner T be SPD. Then assumption* (4) *is equivalent to*

$$\|I - TA\|_{T^{-1}} \leq \delta < 1. \tag{7}$$

Proof. Since T is SPD, on the one hand, the matrix product AT is also SPD, but with respect to the A^{-1} scalar product. This implies that assumption (4) is equivalent to the statement that $\Lambda(AT) \in [1-\delta, 1+\delta]$ with $\delta < 1$, where $\Lambda(\cdot)$ denotes the matrix spectrum. On the other hand, the matrix product TA is SPD as well, with respect to the T^{-1} scalar product. Thus, assumption (7) is equivalent to the statement that $\Lambda(TA) \in [1-\delta, 1+\delta]$. This means the equivalence of assumptions (4) and (7), since $\Lambda(AT) = \Lambda(TA)$. $\square$

Let us now, without loss of generality, as in [11, p. 96] and [10, pp. 1268–1269], always scale the SPD preconditioner T in such a way that $\max\{\Lambda(TA)\} + \min\{\Lambda(TA)\} = 2$. Then we have $\delta = (\kappa(TA) - 1)/(\kappa(TA) + 1)$ and, vice versa, $\kappa(TA) = (1+\delta)/(1-\delta)$, where $\kappa(\cdot)$ denotes the matrix spectral condition number. The convergence rate bound (3) for the PSD with nonsymmetric preconditioning in this case turns into the standard PSD convergence rate bound for the case of SPD preconditioner T; see. e.g., [10, Bound (1.3)]. Moreover, [10, Theorem 5.1] shows that this convergence rate bound is sharp for PSD, and cannot be improved for flexible PCG, i.e., using (2), if the SPD preconditioner T changes on every iteration. The latter result naturally extends to the case of nonsymmetric preconditioning of [16, Section 10.2].

Compared to linear systems, eigenvalue problems are significantly more complex. Sharp convergence rate bounds for symmetric eigenvalue problems have been obtained in the last decade, and only for the simplest preconditioned method; see [11, 12] and references therein. A possibility of using nonsymmetric preconditioning for symmetric eigenvalue problems has not been considered before, to our knowledge. However, our check of arguments of [11] and preceding works, where a PSD convergence rate bound is proved assuming (4) and SPD preconditioning, reveals that the latter assumption, SPD, is actually never significantly used, and can be dropped without affecting the bound; see our idea in action in [2].

The arguments above lead us to a surprising determination that whether or not the preconditioner is SPD is of no importance for PSD convergence, given the same quality of preconditioning, measured by (4) after preconditioner prescaling. If the preconditioner is fixed SPD then the standard PCG is the method of choice. The cases, where the preconditioner is variable or nonsymmetric, are similar to each other—the standard non-flexible PCG, i.e., using (1), stalls, while the flexible PCG converges, due to its local optimality, but may not be much faster compared to PSD. This explains the numerical results using nonsymmetric preconditioning reported in this work, as related to results of [10] for variable SPD preconditioning.

6 Conclusion

Although the flexible PCG linear solver does require a bit more computational effort and storage as compared to the standard PCG, within the scope of preconditioning the extra effort can be worthwhile, if the preconditioner is not fixed SPD. Moreover, our numerical tests showed that the behavior is similar in both shared and distributed memory. Thus our conclusion is that the effect of acceleration is independent of the speed of the node interconnection. The use of geometric multigrid without post-relaxation is demonstrated to be surprisingly efficient as a preconditioner for locally optimal iterative methods, such as the flexible PCG for linear systems and LOBPCG for eigenvalue problems.

7 Acknowledgment

The authors would like to thank Rob Falgout, Van Henson, Panayot Vassilevski, and other members of the *hypre* team for their attention to our numerical results reported here and the Center for Computational Mathematics University of Colorado Denver for the use of the cluster. This work is partially supported by NSF awards CNS 0958354 and DMS 1115734.

References

[1] *HYPRE User's Manual.* Center for Applied Scientific Computing, Lawrence Livermore National Lab, 2012.

[2] Merico E Argentati, Andrew V Knyazev, Klaus Neymeyr, Evgueni E Ovtchinnikov, and Ming Zhou. Convergence theory for preconditioned eigenvalue solvers in a nutshell. *arXiv preprint arXiv:1412.5005*, 2014.

[3] Merico E. Argentati, Ilya Lashuk, Andrew V. Knyazev, and Evgenii E. Ovtchinnikov. Block locally optimal preconditioned eigenvalue Xolvers (BLOPEX) in HYPRE and PETSC. *SIAM J. Sci. Comput.*, 29(4):2224–2239, 2007.

[4] Owe Axelsson. *Iterative solution methods.* Cambridge University Press, Cambridge, 1994.

[5] Radim Blaheta. GPCG-generalized preconditioned CG method and its use with non-linear and non-symmetric displacement decomposition preconditioners. *Numer. Linear Algebra Appl.*, 9(6-7):527–550, 2002.

[6] James H. Bramble and Xuejun Zhang. The analysis of multigrid methods. In P.G. Ciarlet and J.L. Lions, editors, *Solution of Equation in R^n (Part 3), Techniques of Scientific Computing*, volume 7 of *Handbook of Numerical Analysis*, pages 173–415. Elsevier, 2000.

[7] Laurent Debreu, Neveu Emilie, Ehouarn Simon, François Xavier Le Dimet, Arthur Vidard, et al. Multigrid solvers and multigrid preconditioners for the solution of variational data assimilation problems. 2013.

[8] Gene H. Golub and Qiang Ye. Inexact preconditioned conjugate gradient method with inner-outer iteration. *SIAM J. Sci. Comput.*, 21(4):1305–1320, 1999.

[9] Andrew V. Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM J. Sci. Comput.*, 23(2):517–541, 2001.

[10] Andrew V. Knyazev and Ilya Lashuk. Steepest descent and conjugate gradient methods with variable preconditioning. *SIAM J. Matrix Anal. Appl.*, 29(4):1267–1280, 2007.

[11] Andrew V. Knyazev and Klaus Neymeyr. A geometric theory for preconditioned inverse iteration. III. A short and sharp convergence estimate for generalized eigenvalue problems. *Linear Algebra Appl.*, 358:95–114, 2003.

[12] Andrew V. Knyazev and Klaus Neymeyr. Gradient flow approach to geometric convergence analysis of preconditioned eigensolvers. *SIAM J. Matrix Anal. Appl.*, 31(2):621–628, 2009.

[13] Dilip Krishnan, Raanan Fattal, and Richard Szeliski. Efficient Preconditioning of Laplacian Matrices for Computer Graphics. In *ACM Transactions on Graphics (Proc. SIGGRAPH 2013)*, volume 32(4). ACM SIGRAPH, July 2013.

[14] Yvan Notay. Flexible conjugate gradients. *SIAM J. Sci. Comput.*, 22(4):1444–1460, 2000.

[15] Steve Schaffer. A semicoarsening multigrid method for elliptic partial differential equations with highly discontinuous and anisotropic coefficients. *SIAM J. Sci. Comput.*, 20(1):228–242, 1998.

[16] Panayot S. Vassilevski. *Multilevel block factorization preconditioners.* Springer, New York, 2008.

A Hypre drivers used for testing

We call a code *struct* provided in *hypre* to test SMG, with different command-line options. To generate the data to compare iterative methods, we execute the following command,

```
mpiexec −np 16 ./struct −n $n $n $n −solver 19
```

for the shared memory experiments and

```
mpiexec −np 384 ./struct −n $n $n $n −solver 19
```

for the distributed memory experiments, where $n runs from 10 to 180, and determines the number of grid points in each of the three directions *per processor*. The size of the brick here is $np-times-$n-by-$n-by-$n, i.e., the brick gets longer in the first direction with the increase in the number of cores. For example, using the largest value $n=180, the maximum problem size we solve is 16x180-by-180-by-180=93,312,000 unknowns for $np=16 and 384x180-by-180-by-180=2,239,488,000 unknowns for $np=384. The option *-solver 19* tells the driver *struct* to use no preconditioning. The MPI option *-np 16* means that we run on 16 cores and we restrict to using only one node for the shared memory whereas for distributed memory we use 16 cores on 24 nodes with the MPI option *-np 384*. In fact, all our tests in this paper are performed on either 16 cores on one node or 16 cores on each of the 24 nodes, so in the rest of the paper we always omit the “mpiexec -np 16(384)” part of the execution command for brevity.

The number of pre- and post-relaxation smoothing steps is controlled by a command line parameter in the *struct* test driver. The data in Figures 2 is obtained by

```
./struct −n $n $n $n −solver 10 −v 1 1
```

in which the *-solver 10* option refers to the SMG preconditioning, and the number of pre- and post-relaxation smoothing steps is specified by the *-v* flag—one step each of pre- and post-relaxation in this call. The command used for no post-relaxation comparison in Figure 3 is:

```
./struct −n 80 80 80 −solver 10 −v 1 0
```

in which the unbalanced relaxation is specified by the *-v* flag and the grid is 1280x80x80. In Figure 3, we compare convergence of PCG-SMG using (1) and (2) and PSD-SMG, where the SMG preconditioner has no post-relaxation. We generate the data for Figures 5 and 6 with

```
./struct −n $n $n $n −solver 10 −lobpcg −v 1 0
./struct −n $n $n $n −solver 10 −lobpcg −v 1 1
```

where $n runs from 10 to 120. We also use the *-P* option in *struct* to make the brick more even-sided. In the shared memory experiment we use *-P* 4 2 2 which creates a $4n$-by-$2n$-by-$2n$ brick and in the distrubuted case we use *-P* 8 8 6, which creates an $8n$-by-$8n$-by-$6n$ brick.