FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

Multigrid HowTo: An Open Source Algebraic Multigrid Solver in C++

T. Preclik, H. Köstler

Lehrstuhlbericht 09-2

Multigrid HowTo (Part II): An Open Source Algebraic Multigrid Solver in C++

T. Preclik*, H. Köstler†

Abstract

This report is a continuation of the technical report 08-3 *A simple Multigrid solver in C++ in less than 200 lines of code*. It explains in detail an open source algebraic multigrid implementation and presents some simple test results.

1 Introduction

Multigrid is a general approach for finding a solution by using several levels or resolutions. The advantages of multigrid solvers compared to other solvers are that multigrid methods are applicable to a wide range of problems and can reach an asymptotically optimal complexity of $\mathcal{O}(N)$, where N is the number of unknowns in the system. For good introductions and a comprehensive overview on multigrid methods, we, e.g., refer to [2, 15, 16, 17], for details on efficient multigrid implementations see [3, 4, 5, 6, 9, 10, 13, 14, 7].

We distinguish different types of multigrid techniques depending on the meaning of a level. The *structured* or *geometric multigrid (MG)* as discussed in [8] identifies each level with a (structured) grid. In this report, we consider *algebraic multigrid (AMG)* [1, 12], which is applicable to general non-uniform or unstructured grids arising, e.g., from finite element discretizations on complex domains. AMG works directly on the algebraic system of equations and one level is just a matrix of a given size. Note that the adjacency graph of the matrix reveals the (unstructured) grid.

In section 2 we give a short overview of the multigrid algorithm, section 3 discusses implementation issues, and in section 4 we give some examples on how to use the open source implementation by solving a simple Poisson problem.

*tobias.preclik@informatik.uni-erlangen.de

†harald.koestler@informatik.uni-erlangen.de

2 Algebraic Multigrid

We use AMG to solve a linear system

$$A^h u^h = f^h \,, \quad \sum_{j \in \Omega^h} a_{ij}^h u_j^h = f_i^h \,, i \in \Omega^h \tag{1}$$

with system matrix $A^h \in \mathbb{R}^{N \times N}$, unknown vector $u^h \in \mathbb{R}^N$ and right-hand side (RHS) vector $f^h \in \mathbb{R}^N$ on a discrete (unstructured) grid Ω^h.

Basically the AMG solver can be split up into several processes: The coarsening begins by constructing a C/F-splitting, which means a classification of all points into coarse-grid points C and points existing only on the fine-grid F. After such a splitting has been performed an interpolation operator P has to be constructed. The restriction operator $R = P^T$ is just the transpose of the interpolation operator. Several different strategies exist, which offer interpolation properties of varying accuracy and complexities. The construction of the interpolation operator is followed by the computation of the coarse-grid system matrix, which is based on the Galerkin product RAP. The coarsening process is repeatedly applied to the coarse-systems until the system size is reduced to a threshold where a direct solver becomes efficient. Now that the hierarchy of system matrices is set up, the solution phase can start, either by the means of a full multigrid startup or by ordinary multigrid cycles. AMG cycles do not differ from geometric multigrid cycles in principle.

One multigrid iteration, here the so-called *V-cycle*, is summarized in Alg. 1.

Algorithm 1 Recursive V-cycle: $u_h^{(k+1)} = V_h(u_h^{(k)}, A^h, f^h, \nu_1, \nu_2)$

if coarsest level **then**
  solve $A^h u^h = f^h$ exactly or by many smoothing iterations
else
  $\bar{u}_h^{(k)} = \mathcal{S}_h^{\nu_1}(u_h^{(k)}, A^h, f^h)$ *// pre-smoothing*
  $r^h = f^h - A^h \bar{u}_h^{(k)}$ *// compute residual*
  $r^H = R r^h$ *// restrict residual*
  $e^H = V_H(0, A^H, r^H, \nu_1, \nu_2)$ *// recursion*
  $e^h = P e^H$ *// interpolate error*
  $\tilde{u}_h^{(k)} = \bar{u}_h^{(k)} + e^h$ *// coarse grid correction*
  $u_h^{(k+1)} = \mathcal{S}_h^{\nu_2}(\tilde{u}_h^{(k)}, A^h, f^h)$ *// post-smoothing*
end if

By nested iteration, the multigrid algorithm can be extended to *Full Multigrid (FMG)* shown in Fig. 1. This means that we combine an unidirectional multilevel approach with a V-cycle and start on each level l with an interpolated initial guess computed on level $l-1$. Since ultimately only a small number of relaxation steps independent of the number of unknowns must be performed on each level, multigrid can reach an asymptotically optimal complexity $\mathcal{O}(N)$ [15]. The meaning of the signs and arrows in Fig. 1 is explained in the following:

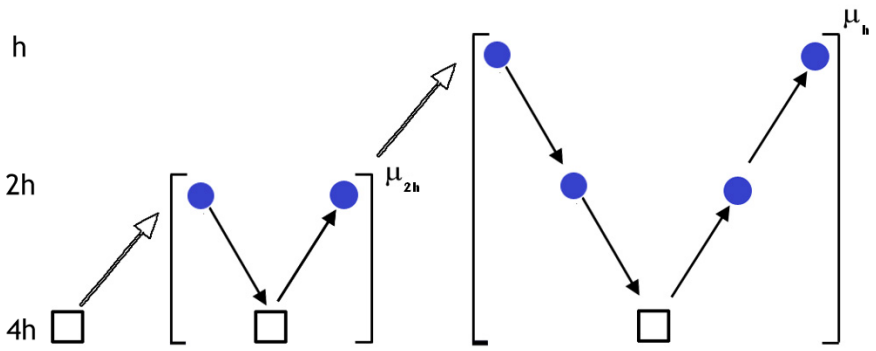

Figure 1: FMG algorithm (with V-cycles).

- exact solution on the coarsest grid: □
- prolongation of a coarse grid solution from coarse to fine grid: ↗
- a small number (typically 1-3) of iteration steps, e.g., of GS method, called smoothing steps, on a fine grid applied to the current solution: ●
- restriction of fine grid residual to coarse grid: ↘
- prolongation of coarse grid correction to fine grid and adding it to the corresponding previous fine grid approximation: ↗
- the corresponding procedure is applied μ_h times, e.g., once for a V-cycle.

3 Implementation

Apart from the AMG code several accompanying components are necessary. First of all some linear algebra framework must be present. Our code comes with basic vector and matrix data structures. The vector class *VectorN* supports primarily dynamic resizing and arithmetic operations. For matrices in contrast two data structures exist: *MatrixNxN* provides operations for working on dense matrices which are completely stored in memory in a row-major order[1]. In contrast *MatrixCRS* stores the matrix in a compressed-row storage format again in a row-major order but this time omitting zero values. Each entry must additionally be annotated by its column index. This storage format is especially suited for sparse matrices without any additional information about the sparsity structure. The CRS format is used intensively in the AMG solver whereas the full matrix storage is needed in the Gaussian elimination algorithm. The latter is typically used as a direct solver for the coarsest system in each cycle. This particular implementation, encapsulated by the class *GaussianElimination*, employs a partial pivoting strategy for improving the numerical stability of the algorithm.

[1]Entries of matrix rows are stored consecutively in memory.

The actual AMG solver is implemented in the *AmgSolver* class. The direct coarse-system solver is specified as a template parameter, which is by default the Gaussian elimination but can be exchanged if desired by other solvers providing an appropriate solver interface. The solution process is started by calling *solve()*, which expects a linear system of equations *Lse* as an argument. The method drives the solution process by triggering the initial setup and the subsequent multigrid cycles. The setup phase starts out by repeatedly coarsening the system. Several coarsening strategies are available and can be chosen beforehand using *setCoarseningStrategy()*. In particular three strategies exist: *coarsening_amg1r5*, *coarsening_direct* and *coarsening_standard*. In the following we will elaborate on the direct coarsening for illustration purposes. The corresponding coarsening method is *coarsenByDirectInterpolation()*. First off it constructs a C/F splitting by calling *splitPoints()*, which implements the standard splitting shared by all the coarsening strategies. The splitting is based on the concept of *strong dependency*. A variable i strongly depends on (or is strongly coupled to) another variable j if the absolute value of a_{ij} is comparable in magnitude to the largest absolute off-diagonal value in the i-th row with the same sign:

$$a_{ij} \leq \theta^{-} \min_{k \neq i} a_{ik}, \quad a_{ij} \neq 0 \tag{2a}$$

$$a_{ij} \geq \theta^{+} \max_{k \neq i} a_{ik}, \quad a_{ij} \neq 0 \tag{2b}$$

Eq. (2a) corresponds to a strong negative coupling and Eq. (2b) to a strong positive coupling of variable i to j. θ qualitatively controls the number of strong couplings separately for each sign. These coarsening parameters can be chosen beforehand by calling *setCoarseningParameters()*. The resulting strong dependency relations define a directed subgraph S of the adjacency graph of the matrix A. Though each edge in the adjacency graph contains a corresponding backward edge due to the symmetry of A, the subgraph S is no longer symmetric because if variable i strongly depends on j, j does not necessarily strongly depend on i according to the above criterions. The subgraph S is important later on and is stored using adjacency lists, which hold for each variable i the adjacent variables j in a list denoted by S_i. That means S_i contains variables, i strongly depends on. As the subgraph S is constructed, its transpose S^T is constructed too by reversing all edges in S. S^T thus reflects the relation of strong influence since if i strongly depends on j, then j strongly influences i. After these preliminary tasks *splitPoints()* proceeds by continuously assigning points to the set of coarse points C. Points which are well suited for interpolation are always preferred. The interpolation quality of a variable i is initially the number of variables strongly influenced by i which equals $|S_i^T|$. As points are added to C the quality measures are adapted. Assuming point i is added to C, then all (undecided) points which are strongly influenced by i are now determined to reside on the fine-grid and are added to F. The reasoning is that those points now have already at least one well suited interpolatory point and should therefore not become

coarse-grid points in order to minimize the number of coarse-grid variables. However one interpolatory point is barely satisfying and therefore the algorithm increments the qualities of the strongly influencing (undecided) points which are neighbors of the points just added to F. Our implementation relies on a priority queue for queueing the points with their interpolatory quality as priority. To support the quality adaptation, the priority queue must provide methods to increment the priorities of queued items. The splitting algorithm proceeds until no more undecided points are left in the queue.

Afterwards the control flow returns to the coarsening method *coarsenByDirectInterpolation()* and continues by constructing the interpolation operator $P \in \mathbb{R}^{|F \cup C| \times |C|}$, which will be used to interpolate the fine-grid errors from the coarse-grid errors. Variables i which are present on the coarse-grid ($i \in C$) will be injected to the fine-grid and the corresponding row of the interpolation operator is a unit vector. Variables i which exist on the fine-grid only ($i \in F$) are interpolated from their interpolatory points $P_i = S_i \cap C$ according to the following interpolation formula:

$$\begin{aligned} e_i &= \sum_{j \in P_i} p_{ij} e_j, \quad p_{ij} = \begin{cases} -\alpha_i a_{ij}/a_{ii} & \text{if } a_{ij} < 0 \\ -\beta_i a_{ij}/a_{ii} & \text{if } a_{ij} > 0 \end{cases} \\ \alpha_i &= \frac{\sum_{j \neq i, a_{ij} < 0} a_{ij}}{\sum_{j \in P_i, a_{ij} < 0} a_{ij}}, \quad \beta_i = \frac{\sum_{j \neq i, a_{ij} > 0} a_{ij}}{\sum_{j \in P_i, a_{ij} > 0} a_{ij}} \end{aligned} \tag{3}$$

P is constructed as a CRS matrix in the first place. Afterwards the final coarse system matrix is computed using the Galerkin product $P^T AP$. The linear algebra operations for transposing and multiplying sparse matrices are provided by the *MatrixCRS* class. Therewith the main setup phase is completed and the mutligrid solution process can start.

Back in the coarsening method *coarsenByDirectInterpolation()* optionally a full multigrid startup is executed, which can be controlled by calling *setStartupExecution()* beforehand.

Finally recursive multigrid cycles are performed until the residual drops below a threshold or a maximum numer of cycles has been reached. The code closely follows Alg. 1 and is implemented in the method *cycle()*. Pre- and postsmoothing steps can be adjusted by a call to *setSmoothingParameters()*. By default an ordinary Gauss-Seidel smoother is used, which relaxes each variable once per smoothing step. The smoother is implemented in *smooth()*.

4 Code Usage

As an example we solve the partial differential equation (PDE)

$$\Delta \boldsymbol{u} = -2((1-6x^2)y^2(1-y^2) + (1-6y^2)x^2(1-x^2) \quad \text{in } \Omega \tag{4a}$$

$$\boldsymbol{u} = 0 \quad \text{on } \partial\Omega \tag{4b}$$

with Dirichlet boundary conditions on a rectangular domain Ω. Eq. (4) is discretized by finite differences on a structured grid which leads to a linear system (1). $n = \frac{1}{h} - 1$ denotes the number of unknowns in x- and y-direction, and $N = n^2$ the problem size. In stencil notation the discretized Laplacian reads

$$A^h = \frac{1}{h^2} \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}. \tag{5}$$

The AMG solver already provides code to readily execute the solution of the PDE. The source code is released open-source under the GNU General Public License (GPL) version 3 and can be obtained from our website[2]. It was written in an attempt to transfer algebraic multigrid ideas to the solution of complementarity problems [11]. The source can be built by executing `make`. The package comes with a slim command-line tool, which can be used to specify the system matrix and right-hand side. The system can be set up manually and can be passed to the AMG solver via the command-line interface:

```
$ ./amgpp --matrix A.txt --rhs b.txt
```

The input format of the matrix begins with the number of rows, the number of columns and the total number of non-zero entries. All non-zero entries are listed in row-major order. Each entry consists of its row and column index followed by the entry's value. All quantities must be separated by white-space of any kind. The input format of the right-hand side is the size of the vector followed by all entries separated by white-space. However for the test problem described above the `--poisson` switch can be used.

```
$ ./amgpp --poisson 33
System size: 1024
Solved the LSE in 6 multigrid cycles.
```

To get a more verbose output set the switch `--log` to `info`. Information about the coarsening and the residual evolution on all grids in all multigrid cycles is printed. There are several more switches to control the AMG solver from the command-line. To solve the Poisson problem with 32 points in each direction, applying V(2, 2) cycles, use a strong dependency threshold $\theta^+ = 0.0$ for positive and $\theta^- = 0.4$ for negative matrix entries, execute a full multigrid startup and apply a direct coarsening strategy, type the following:

```
$ ./amgpp --poisson 33 --log info --nu 2 2 --sdep 0.0 0.4 --fmg
          --coarsening direct
...
                3.73084e-07 (residual on grid 2 after descent)
                2.22297e-08 (residual on grid 2 after postsmoothing)
                0.0343424 (convergence)
```

[2] `http://www10.informatik.uni-erlangen.de/~toprec/amgpp/`

```
        1.82221e-07 (residual on grid 1 after descent)
        8.77108e-09 (residual on grid 1 after postsmoothing)
        0.0370066 (convergence)
1.55314e-07 (residual on grid 0 after descent)
2.02745e-08 (residual on grid 0 after postsmoothing)
0.029187 (convergence)
Solved the LSE in 2 multigrid cycles.
```

If you want to increase the domain resolution, adapt the parameter of the `--poisson` switch. However keep in mind that the solver has to calculate the Galerkin product for each coarse system and that quickly becomes expensive. The Galerkin product can easily take more than 90% of the execution time.

5 Future Work

Besides general improvements like extending the command-line interface and polishing the solver interface, future work includes extending the AMG solver to more advanced interpolation schemes and more sophisticated coarse-grid variable selections.

References

[1] A. Brandt, "Algebraic multigrid theory: The symmetric case," *Applied Mathematics and Computation*, vol. 19, no. 1-4, pp. 23–56, 1986.

[2] W. Briggs, V. Henson, and S. McCormick, *A Multigrid Tutorial*, 2nd ed. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 2000.

[3] I. Christadler, H. Köstler, and U. Rüde, "Robust and efficient multigrid techniques for the optical flow problem using different regularizers," in *Proceedings of 18th Symposium Simulationstechnique ASIM 2005*, ser. Frontiers in Simulation, F. Hülsemann, M. Kowarschik, and U. Rüde, Eds., vol. 15. SCS Publishing House, Erlangen, Germany, 2005, pp. 341–346, preprint version published as Tech. Rep. 05-6.

[4] C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß, "Cache Optimization for Structured and Unstructured Grid Multigrid," *Electronic Transactions on Numerical Analysis (ETNA)*, vol. 10, pp. 21–40, 2000.

[5] T. Gradl, C. Freundl, H. Köstler, and U. Rüde, "Scalable Multigrid," in *High Performance Computing in Science and Engineering. Garching/Munich 2007*, S. Wagner, M. Steinmetz, A. Bode, and M. Brehm, Eds., LRZ, KONWIHR. Springer-Verlag, Berlin, Heidelberg, New York, 2008, pp. 475–483.

[6] F. Hülsemann, M. Kowarschik, M. Mohr, and U. Rüde, "Parallel geometric multigrid," in *Numerical Solution of Partial Differential Equations on Parallel Computers*, ser. Lecture Notes in Computational Science and Engineering, A. Bruaset and A. Tveito, Eds. Springer-Verlag, Berlin, Heidelberg, New York, 2005, vol. 51, ch. 5, pp. 165–208.

[7] H. Köstler, *A Multigrid Framework for Variational Approaches in Medical Image Processing and Computer Vision.* Verlag Dr. Hut, München, 2008.

[8] H. Köstler, "Multigrid howto: A simple multigrid solver in c++ in less than 200 lines of code," Department of Computer Science 10 (System Simulation), Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany, Tech. Rep. 08-3, 2008.

[9] H. Köstler, M. Stürmer, and U. Rüde, "A fast full multigrid solver for applications in image processing," *Numerical Linear Algebra with Applications*, vol. 15, no. 2–3, pp. 187–200, 2008.

[10] M. Kowarschik, C. Weiß, and U. Rüde, "DiMEPACK — A Cache–Optimized Multigrid Library," in *Proceedings of the Int. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*, H. Arabnia, Ed., vol. I. Las Vegas, NV, USA: CSREA Press, Irvine, CA, USA, 2001, pp. 425–430.

[11] T. Preclik, "Iterative rigid multibody dynamics," Diploma thesis, Friedrich-Alexander University Erlangen-Nuremberg, 2008. [Online]. Available: http://www10.informatik.uni-erlangen.de/Publications/Theses/2008/Preclik_DA08.pdf

[12] J. Ruge and K. Stüben, "Algebraic multigrid (AMG)," in *Multigrid Methods*, ser. Frontiers in Applied Mathematics, S. McCormick, Ed. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 1987, vol. 5, pp. 73–130.

[13] M. Stürmer, J. Treibig, and U. Rüde, "Optimizing a 3D multigrid algorithm for the IA-64 architecture," in *Proceedings of 18th Symposium Simulationstechnique ASIM 2006, Hannover*, ser. Frontiers in Simulation, M. Becker and H. Szczerbicka, Eds., vol. 16. SCS Publishing House, Erlangen, Germany, 2006, pp. 271–276.

[14] M. Stürmer, G. Wellein, G. Hager, H. Köstler, and U. Rüde, "Challenges and Potentials of Emerging Multicore Architectures," in *High Performance Computing in Science and Engineering. Garching/Munich 2007*, S. Wagner, M. Steinmetz, A. Bode, and M. Brehm, Eds., LRZ, KONWIHR. Springer-Verlag, Berlin, Heidelberg, New York, 2008, pp. 551–566.

[15] U. Trottenberg, C. Oosterlee, and A. Schüller, *Multigrid.* Academic Press, San Diego, CA, USA, 2001.

[16] P. Wesseling, *Multigrid Methods*. Edwards, Philadelphia, PA, USA, 2004.

[17] R. Wienands and W. Joppich, *Practical Fourier Analysis for Multigrid Methods*, ser. Numerical Insights. Chapmann and Hall/CRC Press, Boca Raton, Florida, USA, 2005, vol. 5.