User's Guide to AGMG

Yvan Notay [*]

Service de Métrologie Nucléaire
Université Libre de Bruxelles (C.P. 165/84)
50, Av. F.D. Roosevelt, B-1050 Brussels, Belgium.
email : yvan.notayt@agmg.eu, ynotay@ulb.ac.be

February 2008
Revised: March 2009, March 2010, July 2011, January & October 2012
March 2014, January 2017, March & August 2018, May 2019

Abstract

This manual gives an introduction to the use of AGMG. AGMG is available both as a software library for FORTRAN or C/C++ programs, and as an Octave/Matlab function. It solves systems of linear equations with the aggregation-based algebraic multigrid method described in [7] with further improvements from [5] and [8].

The software is expected to be efficient for large systems arising from the discretization of scalar second order elliptic PDEs. It may, however, be tested on any problem, as long as *all diagonal entries of the system matrix are positive.* It is indeed purely algebraic; that is, no information has to be supplied besides the system matrix and the right-hand-side.

The Octave/Matlab version accepts real and complex matrices, whereas the C/C++/FORTRAN library is available for double precision and double complex arithmetic. For this library, several level of parallelism are provided: multi-threading (multi-core acceleration of sequential programs), MPI-based, or hybrid mode (MPI+multi-threading).

See the web site `http://agmg.eu` for instructions to obtain a copy of the software and possible updgrade.

Key words. FORTRAN, C, Octave, Matlab, multigrid, linear systems, iterative methods, AMG, preconditioning, parallel computing, software.

AMS subject classification. 65F10

[*]Yvan Notay is Research Director of the Fonds de la Recherche Scientifique – FNRS.

Contents

1 Introduction

The AGMG library for FORTRAN or C/C++ program is written in FORTRAN 90. The main driver subroutine is `AGMG` for the sequential & multithreaded versions, and `AGMGPAR` (or `AGMGPARG4`, `AGMGPARG8`) for the MPI & hybbid version. Each driver is available in double precision (prefix `D`) and double complex (prefix `Z`) arithmetic. These subroutines are to be called from an application program in which are defined the system matrix and the right-hand-side of the linear system to be solved.

An Octave/Matlab function is also provided; that is, an Octave oct-file (agmg.oct) and a Matlab m-file (agmg.m) implementing the function *agmg*, which calls (sequential) AGMG from the Octave/Matlab environment. This guide is directed towards the use of the FORTRAN/C/C++ library; however, Section 2.2 on additional parameters, Section 2.5 on printed output and Sections 6–7 about some special usages apply to the Octave/Matlab version as well. Basic information can also be obtained by entering *help agmg* in the Octave/Matlab environment.

For a sample of performance in sequential and comparison with other solvers, see the paper [6] and the report [9] (`http://agmg.eu/numcompsolv.pdf`). For the parallel version, see [10].

1.1 How to use this guide

This guide is self contained, but does not describe methods and algorithms used in AGMG, for which we refer to [7, 5, 8]. We strongly recommend their reading for an enlightened use of the package.

On the other hand, the first lines of source codes for main driver routines are reproduced in Appendix, including the comment lines describing precisely each parameter. It is a good idea to have a look at the listing of a driver routine while reading the section of this guide describing its usage.

1.2 Release notes

This guide describes AGMG 3.x.y-aca (academic version) and AGMG 3.x.y-pro (professional version), with x$\geq$ 2.

New features from **release 3.3.2**:

- *Professional version only: new main drivers for the MPI and Hybrid versions (`AGMGPARG4` & `AGMGPARG8`) providing a much simpler interface based on global column numbers; see Section 4.1 for details.*

New features from **release 3.3.0**:

- *Professional version only: multithreading is available for both the standard and MPI versions. See, respectively, Section 3 and Section 5 for more details.*
- Options `ijob`=202 and `ijob`=212 have been added for faster execution when needing several successive solves with the same system matrix. See Section 2.2.1 for details.

New features from **release 3.2.1**:

- Improved treatment of singular systems; see Section 7.
- *Professional version only. Improved parallel performance (verified weak scalability up to 370,000 cores [10]).*
- *Professional version only. Compilation with the (provided) sequential MUMPS module can be replaced by a link with the Intel Math Kernel Library (MKL); see Section 1.3.*

New features from **release 3.2.0**:

- *Professional version only. A variant of the parallel version is provided that does not require the installation of the parallel MUMPS library. See Section 1.3 and the README file for details.*
- The smoother is now SOR with automatic estimation of the relaxation parameter ω, instead of Gauss-Seidel (which corresponds to SOR with $\omega = 1$). In many cases, AGMG still uses $\omega = 1$ as before, but this new feature provides additional robustness, especially in non symmetric cases.
- Default parameters for the parallel version have been modified, to improve scalability when using many processors. This may induce slight changes in the convergence, that should however not affect overall performances.
- Some compilation issues raised by the most recent versions of gfortran and ifort have been solved, as well as some compatibility issues with the latest releases of the MUMPS library.

New features from **release 3.1.2**:

- The printed output has been slightly changed: the work (number of floating points operations) is now reported in term of "work units per digit of accuracy", see Section 2.5 for details.

New features from **release 3.1.1**:

- A few bugs that sometimes prevented the compilation have been fixed; the code (especially MUMPS relates routines) has been also cleaned to avoid superfluous warning messages with some compilers.
- The parameters that control the aggregation are now automatically adapted in case of too slow coarsening.
- Some limited coverage of singular systems is now provided, see Section 7.

New features from **release 3.0** (the first three have no influence on the software usage):

- The aggregation algorithm is no longer the one described in [7]. It has been exchanged for the one in [5] in the symmetric case and for the one in [8] for general nonsymmetric matrices (Note that this is the user who tells if the problem is to be treated as a symmetric one or not via the parameter `nrest`, see Section 2.2.3 below).

  This is an "internal" change; i.e., it has no influence on the usage of the software. It makes the solver more robust and faster on average. We can however not exclude that in some cases AGMG 3.0 is significantly slower than previous versions. This may, in particular, happen for problems that are outside the standard scope of application. Feel free to contact the author if you have such an example. AGMG is an ongoing project and pointing out weaknesses of the new aggregation scheme can stimulate further progress.

- Another internal change is that the systems on the coarsest grid are now solved with the (sequential) MUMPS sparse direct solver [4]. Formerly, MUMPS was only used with the parallel version. Using it also in the sequential case allows to improve robustness; i.e. improved performances on "difficult" problems.

  The MUMPS library needs however not be installed when using the sequential version. We indeed provide an additional source file that contains needed subroutines from the MUMPS library. See Section 1.3 and the README file for details.

- The meaning of parameter `nrest` is slightly changed, as it influences also the aggregation process (see above). The rule however remains that `nrest` should be set to 1 only if the system matrix is symmetric positive definite; see Section 2.2.3 for details.
- It is now possible to solve a system with the transpose of the input matrix. This corresponds to further allowed values for parameter `ijob`; see Section 2.2.1 below for details.
- The meaning of the input matrix when using `ijob` not equal to its default is also changed; see Section 2.2.1 below for details.

- The naming schemes and the calling sequence of each driver routine remain unchanged from releases 2.x, but there are slight differences with respect to releases 1.x. Whereas with releases 2.x routines were provided for backward compatibility, these are no longer supported; i.e. application programs based on the naming scheme and calling sequence of releases 1.x need to be adapted to the new scheme.

1.3 Installation and external libraries

AGMG does not need to be installed as a library. For the sake of simplicity, source or object files are provided, which need just to be compiled or linked together with the application program in which AGMG routines are referenced. See the README file provided with the package for additional details and example of use.

AGMG requires LAPACK [1] and BLAS libraries. Most compilers come with an optimized implementation of these libraries, and it is strongly recommended to use it. If not available, we provide needed routines from LAPACK and BLAS together with the package. [1] Alternatively LAPACK may be downloaded from `http://www.netlib.org/lapack/` and BLAS may be downloaded from `http://www.netlib.org/blas/`

The sequential and the *professional* *multithreaded/MPI/hybbid* versions require also some subset of MUMPS (`http://graal.ens-lyon.fr/MUMPS/`). For convenience, we provide the needed (public domain) source files (one per arithmetic) together with AGMG (see the files header for a detailed copyright notice). Hence, MUMPS does not need to be installed as a library. To avoid mismatch with a standard implementation of the library, all MUMPS routines provided with AGMG have been renamed (the prefixes `d_`, `s_`, ..., have been exchanged for `dagmg_`, `sagmg_`, ... [2]).

Professional version only. When using the Intel compiler and linking with the Intel Math Kernel Library (MKL), the provided subset of MUMPS can be skipped, because needed functionalities are retrieved from the MKL (using an appropriate flag when compiling AGMG; see the README file for details).

On the other hand, the academic MPI version requires the installation of the parallel MUMPS library [4], which may be downloaded from `http://graal.ens-lyon.fr/MUMPS/`.

Professional version only. The parallel MUMPS library is not needed with the professional MPI/hybrid versions because they use a different algorithm, leading to better (or even much better) scalability above several hundreds of cores (depending on the hardware).

[1] These softwares are free of use, but copyrighted. It is required the following. *If you modify the source for a routine we ask that you change the name of the routine and comment the changes made to the original.* This, of course, applies to the routines redistributed together with AGMG.

[2] It means that if you have the sequential MUMPS library installed and that you would like to use it instead of the provided version, you need to edit the AGMG source file to restore the standard calling sequence to MUMPS

1.4 Error handling

Except the case of a number of allowed iterations insufficient to achieve the convergence criterion, any detected error is fatal and leads to a STOP statement, with a printed information that should allow to figure out what happened. Note that AGMG does not check the validity of all input parameters. For instance, **all diagonal entries of the supplied matrix must be positive**, but no explicit test is implemented. Feel free to contact support@agmg.eu for further explanations if errors persist after checking that the input parameters have been correctly defined.

2 Sequential version

2.1 Basic usage and matrix format

The application program has to call the main driver `AGMG` as follows.

FORTRAN Syntax:

```
call dagmg(n,a,ja,ia,f,x,ijob,iprint,nrest,iter,tol)
```

C/C++ Syntax[3]:

```
dagmg_(&n,a,ja,ia,f,x,&ijob,&iprint,&nrest,&iter,&tol);
```

PARAMETERS:

`n`:	INPUT	integer
`a`:	IN/OUT	double precision (array/pointer)
`ja`:	IN/OUT	integer (array/pointer)
`ia`:	IN/OUT	integer (array/pointer)
`f`:	IN/OUT	double precision (array/pointer)
`x`:	IN/OUT	double precision (array/pointer)
`ijob`:	INPUT	integer
`iprint`:	INPUT	integer
`nrest`:	INPUT	integer
`iter`:	IN/OUT	integer
`tol`:	INPUT	double precision

`n` is the order of the linear system, whose matrix is given in arrays `a`, `ja`, `ia`. The right hand side must be supplied in array `f` on input, and the computed solution is returned in `x` on output; optionally, `x` may contain an initial guess on input, see Section 2.2.1 below. `f` is also used as work space and thus modified on output. Additional parameters are described in Section 2.2.

The required matrix format is the "compressed sparse row" (CSR) format described, e.g., in [12]. With this format, nonzero matrix entries (numerical values) are stored row-wise in `a`, whereas `ja` carries the corresponding column indices; entries in `ia` indicate then where every row starts. That is, nonzero entries (numerical values) and column indices of row i are located in `a`(k), `ja`(k) for $k =$`ia`(i),...,`ia`$(i$+1)-1. `ia` must have length (at least) `n`+1,

[3]Remember that AGMG is written in FORTRAN 90; using `dagmg_` as function name in C/C++ programs means adding an underscore to the FORTRAN name and staying with lowercase. This works with GNU and Intel compilers on Linux OS and OS X. Then, to properly link with GNU compilers: either link with *gfortran* even if the main is in C, or link with *gcc/g++*, but reference *gfortran* and *m* libraires through the option "-lgfortran -lm". To properly link with Intel compilers: link with *ifort* using the option "-nofor-main". Other changes may be needed on Windows systems or with other compilers; for intel compilers on Windows, see `https://software.intel.com/en-us/node/678441`

and `ia`(`n`+1) must be defined in such a way that the above rule also works for i =`n`; that is, the last valid entry in arrays `a` and `ja` must correspond to index k =`ia`(`n`+1)-1.

Remark for C users. *Observe that AGMG uses 1-based indexing; that is, indexing of arrays start at 1. Therefore,* ***ia***(i) *as mentioned above is* i^{th} *entry associated with the pointer* ***ia****, accessed as* ***ia***$[i\text{-}1]$ *in C. Further,* ***ia****(1) (i.e.,* ***ia***$[0]$ *in C) should be set equal to 1 to correctly point to the first entry associated with pointers* ***a*** *and* ***ja****, and the next entries defined accordingly. Finally, the array* ***ja*** *contains the "true" column numbers (according the usual mathematical representation of matrices), and not these numbers minus 1, as one would be inclined to use in 0-based indexing systems.*

By way of illustration, consider the matrix

$$A = \begin{pmatrix} 10 & & & -1 & \\ -2 & 11 & & -3 & \\ & -4 & 12 & & -5 \\ & & & 13 & \\ & -8 & -9 & & 14 \end{pmatrix} .$$

The compressed sparse row format of A is given (non uniquely) by the following three vectors.

$$\begin{aligned} \texttt{a} &= [\ 10.0 \quad -1.0 \mid -2.0 \quad 11.0 \quad -3.0 \mid -4.0 \quad 12.0 \quad -5.0 \mid 13.0 \mid 14.0 \quad -9.0 \quad -8.0\ ] \\ \texttt{ja} &= [\ 1 \quad 4 \mid 1 \quad 2 \quad 4 \mid 2 \quad 3 \quad 5 \mid 4 \mid 5 \quad 3 \quad 2\ ] \\ \texttt{ia} &= [\ 1 \quad 3 \quad 6 \quad 9 \quad 10 \quad 13\ ] \end{aligned}$$

As the example illustrates (see how is stored the last row of A), entries in a same row may occur in any order.

AGMG performs a partial reordering inside each row, so that, on output, `a`, `ja` and `ia` are possibly different than on input; they nevertheless still represent the same matrix. (That is why `a`, `ja` and `ia` are declared as IN/OUT parameters.)

Note that the SPARSKIT package [11] (`http://www-users.cs.umn.edu/~saad`) contains subroutines to convert various sparse matrix formats to CSR format.

2.2 Additional parameters: `ijob` , `iprint` , `nrest` , `iter` and `tol`

2.2.1 `ijob`: INPUT, integer

`ijob` has default value 0. With this value, the linear system will be solved in the usual way with the zero vector as initial approximation; this implies a setup phase followed by an iterative solution phase [7]. Non-default values allow to tell `AGMG` that an initial approximation is given in `x`, and/or to call `AGMG` for setup only, for solve only, or for just one application of the multigrid preconditioner (that is, a call to Algorithm 3.2 in [7] at top level, to be exploited in a more complex fashion by the calling program).

Details are as follows.

`ijob`	usage or remark
0	performs setup + solve + memory release, no initial guess
10	performs setup + solve + memory release, initial guess in `x`
1	performs setup only (preprocessing: prepares all parameters for subsequent solves)
2	solves only with the preconditioner based on previous setup, no initial guess; the system matrix may differ from the one provided for set up
12	solves only with the preconditioner based on previous setup, initial guess in `x`; the system matrix may differ from the one provided for set up
202	solves only with the preconditioner based on previous setup and the same matrix as the one provided for set up; no initial guess
212	solves only with the preconditioner based on previous setup and the same matrix as the one provided for set up; initial guess in `x`
3	the vector returned in `x` is not the solution of the linear system, but the result of the action of the multigrid preconditioner on the right hand side in `f`
-1	erases the setup and releases internal memory
100,110,101,102,112	same as, respectively, 0,10,1,2,12 but use the TRANSPOSE of the input matrix *Not available for multithreaded, MPI and hybrid versions*
2,3,12,102,112,202,212	require that one has previously called AGMG with `ijob`=1 or `ijob`=101
1,101	the input matrix and the built preconditioner are entirely kept in internal memory; hence:
202,212,3	the input matrix is not accessed
2,12,102,112	the input matrix is used only to perform matrix vector product within the main iterative loop

Thus, upon calls with `ijob`=2,12,102,112, the input matrix may differ from the input matrix that was supplied upon the previous call with `ijob`=1 or IJOB `ijob`=101; then AGMG attempts to solve a linear system with the "new" matrix using the preconditioner set up for the "old" one. The same remark applies to `ijob`$\geq$ 100 or not: the value `ijob`=1 or 101 determines whether the preconditioner set up and stored in internal memory is based on the matrix or its transpose; the value `ijob`=2,12 or 102,112 is used to determine whether the linear system to be solved is with the matrix or its transpose, independently of the set up. Hence one may set up a preconditioner for a matrix and use it for its transpose.

These functionalities (set up a preconditioner and use it for another matrix) are provided for the sake of generality but should be used with care; in general, set up is fast with AGMG and hence it is recommended to rerun it even if the matrix changes only slightly.

In the more standard case where one needs successive solves with a same matrix, it is recommended to use `ijob`=202 or 212 (in which case the input matrix is not accessed) instead of `ijob`=2 or 12 (in which case the input matrix has to be kept equal to the one provided for set up). This slightly faster and helps to save memory since the arrays that store the input matrix may be deallocated after set up.

Octave/Matlab: the usage is the same except that one should not specify whether an initial approximation is present in `x`. This is told via the presence or the absence of the argument. Hence, for instance, `ijob`=0 gathers the functionality of both `ijob`=0 and `ijob`=10.

2.2.2 `iprint`: INPUT, integer

`iprint` is the unit number where information (and possible error) messages are to be printed. A nonpositive number suppresses all messages, except the error messages which will then be printed on standard output.

Warning messages about insufficient convergence (in the allowed maximum number of iterations) are further suppressed supplying a negative number; this is useful if one intends to perform solves with a fixed number of iterations without caring for the achieved residual reduction.

2.2.3 `nrest`: INPUT, integer

`nrest` is the restart parameter for GCR [2, 13] (an alternative implementation of GMRES [12]), which is the default main iteration routine [7]. A nonpositive value is converted to 10 (suggested default).

If `nrest`=1, Flexible CG is used instead of GCR (when `ijob`=0,10,2, 12,100,110,102,112, 202,212) and also (`ijob`=0,1,100,101) some simplifications are performed during the set up based on the assumption that the input matrix is symmetric (there is then no more difference between `ijob`=1 and `ijob`=101). This is recommended if and only if the matrix is symmetric and positive definite.

2.2.4 `iter`: IN/OUT, integer

`iter` is the maximal number of iterations on input, and the effective number of iterations on output. Since it is both an input *and an output* parameter, it is important not to forget to reinitialize it between successive calls.

2.2.5 `tol`: INPUT, double precision

`tol` is the tolerance on the relative residual norm; that is, iterations will be pursued (within the allowed limit) until the residual norm is below `tol` times the norm of the input right-hand-side.

2.3 Common errors

- Be careful that `iter` is also output parameter. Hence declaring it as a constant in the calling program entails a fatal error.
- Don't forget to reinitialize `iter` between successive calls to AGMG.
- Remember taht **all diagonal entries of the supplied matrix must be positive**. In case of doubt, check the sign of the diagonal entries. For instance, some discretization software do not follow the standard and produce matrices with all diagonal entries negative; to correctly use AGMG, it is then needed to change the sign of all rows of equations.
- Don't forget that the right hand side `f` is overwritten on output.

2.4 Example

The source file of the following example is provided with the package.

Listing 1: source code of sequential Example (FORTRAN 90)

```fortran
      program example_seq
!
!  Solves the discrete Laplacian on the unit square by simple call to agmg.
!  The right-hand-side is such that the exact solution is the vector of all 1.
!
         implicit none
         real (kind(0d0)),allocatable :: a(:),f(:),x(:)
         integer,allocatable :: ja(:),ia(:)
         integer :: n,iter,iprint,nhinv
         real (kind(0d0)) :: tol
!
!        set inverse of the mesh size (feel free to change)
         nhinv=500
!
!        maximal number of iterations
         iter=50
!
!        tolerance on relative residual norm
         tol=1.e-6
!
!        unit number for output messages: 6 => standard output
         iprint=6
!
!        generate the matrix in required format (CSR)
!
!          first allocate the vectors with correct size
           n=(nhinv-1)**2
           allocate (a(5*n),ja(5*n),ia(n+1),f(n),x(n))
!          next call subroutine to set entries
           call uni2d(nhinv-1,f,a,ja,ia)
!
!        call agmg
!          argument 5 (ijob)  is 0 because we want a complete solve
!          argument 7 (nrest) is 1 because we want to use flexible CG
!                             (the matrix is symmetric positive definite)
!
         call dagmg(n,a,ja,ia,f,x,0,iprint,1,iter,tol)
!
      end program example_seq
```

The same example is also provided in C language.

Listing 2: source code of sequential Example (C)

```
      #include <stdlib.h>
      #include <stdio.h>
      void dagmg_(int*,double*,int*,int*,double*,double*,int*,int*,int*,int*,dou
      void uni2d(int,double*,double*,int*,int*);

      int main(void) {
/*
   Solves the discrete Laplacian on the unit square by simple call to agmg.
   The right-hand-side is such that the exact solution is the vector of all 1.
*/
        double *a,*f,*x;
        int n,*ja,*ia;
        int zero=0,one=1;

/*      set inverse of the mesh size (feel free to change) */
        int nhinv=500;
/*      maximal number of iterations */
        int iter=50;
/*      tolerance on relative residual norm */
        double tol=1.e-6;
/*      unit number for output messages: 6 => standard output */
        int iprint=6;

/*      generate the matrix in required format (CSR) */

/*        first allocate the vectors with correct size */
            n=(nhinv-1)*(nhinv-1);
            ia=malloc((n + 1) * sizeof(int));
            ja=malloc(5*n * sizeof(int));
            a=malloc(5*n * sizeof(double));
            f=malloc(n * sizeof(double));
            x=malloc(n * sizeof(double));
/*        next call subroutine to set entries */
            uni2d(nhinv-1,f,a,ja,ia);

/*      call agmg
         argument 5 (ijob)  is 0 because we want a complete solve
         argument 7 (nrest) is 1 because we want to use flexible CG
                            (the matrix is symmetric positive definite) */

        dagmg_(&n,a,ja,ia,f,x,&zero,&iprint,&one,&iter,&tol);

      }
```

2.5 Printed output

The above example (in either language) produces the following output.

```
****ENTERING AGMG **********************************************************

****        Number of unknowns:      249001
****                 Nonzeros :     1243009 (per row:   4.99)

****SETUP: Coarsening by multiple pairwise aggregation
****  Rmk: Setup performed assuming the matrix symmetric
****      Quality threshold (BlockD):  8.00 ;  Strong diag. dom. trs: 1.29
****        Maximal number of passes:  2  ; Target coarsening factor: 4.00
****                     Threshold for rows with large pos. offdiag.: 0.45

****                     Level:           2
****        Number of variables:       62000          (reduction ratio: 4.02)
****                  Nonzeros:      309006 (per row: 5.0; red. ratio: 4.02)

****                     Level:           3
****        Number of variables:       15375          (reduction ratio: 4.03)
****                  Nonzeros:       76381 (per row: 5.0; red. ratio: 4.05)

****                     Level:           4
****        Number of variables:        3721          (reduction ratio: 4.13)
****                  Nonzeros:       18361 (per row: 4.9; red. ratio: 4.16)

****                     Level:           5
****        Number of variables:         899          (reduction ratio: 4.14)
****                  Nonzeros:        4377 (per row: 4.9; red. ratio: 4.19)

****                 Grid complexity:     1.33
****             Operator complexity:     1.33
****  Theoretical Weighted complexity:     1.92 (K-cycle at each level)
****    Effective Weighted complexity:     1.92 (V-cycle enforced where needed)

****       Setup time (Elapsed):     1.02E-01 seconds

****SOLUTION: flexible conjugate gradient iterations (FCG(1))
****    AMG preconditioner with 1 Gauss-Seidel pre- and post-smoothing sweeps
****        at each level
****  Iter=    0       Resid= 0.45E+02       Relat. res.= 0.10E+01
****  Iter=    1       Resid= 0.14E+02       Relat. res.= 0.32E+00
```

```
****  Iter=    2        Resid= 0.23E+01        Relat. res.= 0.51E-01
****  Iter=    3        Resid= 0.87E+00        Relat. res.= 0.19E-01
****  Iter=    4        Resid= 0.14E+00        Relat. res.= 0.31E-02
****  Iter=    5        Resid= 0.35E-01        Relat. res.= 0.78E-03
****  Iter=    6        Resid= 0.66E-02        Relat. res.= 0.15E-03
****  Iter=    7        Resid= 0.23E-02        Relat. res.= 0.52E-04
****  Iter=    8        Resid= 0.54E-03        Relat. res.= 0.12E-04
****  Iter=    9        Resid= 0.13E-03        Relat. res.= 0.28E-05
****  Iter=   10        Resid= 0.33E-04        Relat. res.= 0.74E-06
****  - Convergence reached in   10 iterations -

****     level 2   #call=    10   #cycle=    20   mean=   2.00    max=  2
****     level 3   #call=    20   #cycle=    40   mean=   2.00    max=  2
****     level 4   #call=    40   #cycle=    80   mean=   2.00    max=  2

****       Number of work units:    11.37 per digit of accuracy (*)
****    Solution time (Elapsed):     1.49E-01 seconds

*** (*) 1 work unit represents the cost of 1 (fine grid) residual evaluation
****LEAVING AGMG * (MEMORY RELEASED) *****************************************
```

Note that each output line issued by the package starts with `****`.

AGMG first indicates the size of the matrix and the number of nonzero entries. One then enters the setup phase, and the name of the coarsening algorithm is recalled, together with the basic parameters used. Note that these parameters need not be defined by the user: AGMG always use default values. These, however, can be changed by expert users, see Section 2.6 below.

The quality threshold is the threshold used to accept or not a tentative aggregate when applying the coarsening algorithms from [5, 8]; `BlockD` indicates that the algorithm from [5] is used (quality for block diagonal smoother), whereas `Jacobi` is printed instead when the algorithm from [8] is used (quality for Jacobi smoother). The strong diagonal dominance threshold is the threshold used to keep outside aggregation rows and columns that are strongly diagonally dominant; by default, it is set automatically according to the quality threshold as indicated in [5, 8]. The maximal number of passes and the target coarsening factor are the two remaining parameters described in these papers. In addition, nodes having large positive offdiagonal elements in their row or column are transferred unaggregated to the coarse grid, and AGMG print the related threshold.

How the coarsening proceeds is then reported level by level.

To summarize setup, AGMG then reports on “complexities”. The grid complexity is the sum over all levels of the number of variables divided by the matrix size; the operator complexity is the complexity relative to the number of nonzero entries; that is, it is the sum over all levels of the number of nonzero entries divided by the number of nonzero entries in the input matrix (see [7, eq. (4.1)]). The theoretical weighted complexity reflects the cost of the preconditioner when two inner iterations are performed are each level; see [5, page 15] (with $\gamma = 2$) for a precise definition. The effective weighted complexity corrects the theoretical weighted complexity by taking into account that V-cycle is enforced at some levels according to the strategy described in [7, Section 3]. This allows to better control the complexity in cases where the coarsening is slow. In most cases, the coarsening is fast enough and both weighted complexities will be equal, but, when they differ, only the effective weighted complexity reflects the cost of the preconditioner as defined in AGMG.

Eventually, AGMG reports on the time elapsed during this setup phase (wall clock time).

Next one enters the solution phase. AGMG informs about the used iterative method (as defined via input parameter `nrest`), the used smoother, and the number of smoothing steps (which may also be tuned by expert users). How the convergence proceeds is then reported iteration by iteration, with an indication of both the residual norm and the relative residual norm (i.e., the residual norm divided by the norm of the right hand side). Note that values reported for "Iter=0" correspond to initial values (nothing done yet). When the iterative method is GCR, AGMG also reports on how many restarts have been performed.

Upon completion, AGMG reports, for each intermediate level, statistics about inner iterations; "#call" is the number of times one entered this level; "#cycle" is the cumulated number of inner iterations; "mean" and "max" are, respectively, the average and the maximal number of inner iteration performed on each call. If V-cycle formulation is enforced at some level (see [7, Section 3]), one will have #cycle=#call and mean=max=1.

Finally, the cost of this solution phase is reported, in term of "work units", one work unit being number of floating point operations needed for one residual evaluation (at top level: computation of $\mathbf{b} - A\,\mathbf{x}$ for some $\mathbf{b}\,,\,\mathbf{x}$). AGMG reports the number of needed work units per digit of accuracy; that is, how many digit of accuracy have been gained is computed as $d = \log_{10}(\|\mathbf{r}_0\|/\|\mathbf{r}_f\|)$ (where $\mathbf{r}_0$ and $\mathbf{r}_f$ are respectively the initial and final residual vectors), and the total number of work units for the solution phase is divided by d to get the mean work needed per digit of accuracy. The code also reports the elapsed time (wall clock).

2.6 Fine tuning

Some parameters referenced in [7, 5, 8] are assigned default values within the code, for the sake of simplicity. However, expert users may tune these values, to improve performances and/or if they experiment convergence problems. This can be done by editing the module `dagmg_mem`, at the top of the source file.

3 Multithreaded version

Professional version only

From the user viewpoint, there is no difference between the sequential and multithreaded versions, with the slight exception that the multithreaded version cannot work directly with the transpose of the input matrix. Hence, `ijob`= 100,110,101,102,112 are not permitted.

Naming convention and input parameters are thus exactly as described in the previous section.

Which version is used (sequential or multithreaded) is determined by the object file the application program is linked with: those with name containing *_mth* provide the multithreaded version, those without provide the purely sequential version.

The number of used threads is equal to the standard one for OpenMP, according the environment; i.e., it may be adjusted (if needed) by setting the environment variable OMP_NUM_THREADS. A specific rule for AGMG can be defined by setting the environment variable AGMG_NUM_THREADS; according the hardware and the type of problems, the best number of threads may indeed be occasionally below the number of available cores (enter at shell prompt: *export AGMG_NUM_THREADS=k*, where *k* is the desired number of threads).

If the parameter `iprint` is set to a positive number, AGMG will report in the first output lines about the number of threads actually used (thus confirming that one has correctly linked with the multithreaded version).

4 MPI version

We assume that the previous section about the sequential version has been carefully read. Many features are common between both versions, such as most input parameters and the CSR matrix format (Section 2.1) and the meaning of output lines (Section 2.5). This information will not be repeated here, where the focus is on the specific features of the MPI version.

We also assume that the reader is somewhat familiar with the MPI standard.

The MPI implementation is based on a partitioning of the rows of the system matrix, and assumes that this partitioning has been applied before calling AGMGPAR. Thus, if the matrix has been generated sequentially, some partitioning tool like METIS [3] should be called before going to AGMGPAR.

The partitioning of the rows induces, inside AGMG, a partitioning of the variables. Part of the local work is proportional to the number of nonzero entries in the local rows, and part of it is proportional to the number of local variables (that is, the number of local rows). The provided partitioning should therefore aim at a good load balancing of both these quantities. Besides, the algorithm scalability (with respect to the number of processors) will be best when minimizing the sum of absolute values of offdiagonal entries connecting rows assigned to different processors or MPI ranks, referred to below as *tasks*.

4.1 Basic usage and matrix format

To work, the MPI version needs that several instances of the application program have been launched in parallel and have initialized a MPI framework, with a valid communicator. Let `MPI_COMM` be this communicator (by default, it is `MPI_COMM_WORLD`). All instances or *tasks* sharing this MPI communicator must call simultaneously the main driver, which is declined in three different flavors.

First driver: `AGMGPAR`

FORTRAN Syntax:

```
call dagmgpar(n,a,ja,ia,f,x,ijob,iprint,nrest,iter,tol,MPI_COMM,  &
              listrank,ifirstlistrank)
```

C/C++ Syntax [4]:

```
dagmg_(&n,a,ja,ia,f,x,&ijob,&iprint,&nrest,&iter,&tol,&MPI_COMM,
       listrank,&ifirstlistrank);
```

[4] See footnote page 9.

NEW PARAMETERS (see Section 2.1 for the other ones):

`MPI_COMM`:	INPUT	integer
`listrank`:	IN/OUT	integer (array/pointer)
`ifirstlistrank`:	IN/OUT	integer

Second driver: `AGMGPARG4`

FORTRAN Syntax:

```
call dagmgpar4(n,a,ja,ia,f,x,ijob,iprint,nrest,iter,tol,MPI_COMM)
```

C/C++ Syntax[5]:

```
dagmgpar4_(&n,a,ja,ia,f,x,&ijob,&iprint,&nrest,&iter,&tol,&MPI_COMM);
```

NEW PARAMETERS (see Section 2.1 for the other ones):

`MPI_COMM`:	INPUT	integer

Third driver: `AGMGPARG8`

FORTRAN Syntax:

```
call dagmgpar8(n,a,ja,ia,f,x,ijob,iprint,nrest,iter,tol,MPI_COMM)
```

C/C++ Syntax[6]:

```
dagmgpar8_(&n,a,ja,ia,f,x,&ijob,&iprint,&nrest,&iter,&tol,&MPI_COMM);
```

NEW PARAMETERS (see Section 2.1 for the other ones):

`MPI_COMM`:	INPUT	integer

In all three cases, each task should supply only a portion of the matrix rows in arrays `a`, `ja`, `ia`, and `n` is the number of these local rows. EACH ROW SHOULD BE REFERENCED IN ONE AND ONLY ONE TASK.

The parameters `f` and `x` have the same meaning as in the sequential case, except that each task needs to receive and will return only the portion of these vectors corresponding to local rows.

[5]See footnote page 9.
[6]See footnote page 9.

Regarding the additional parameters `ijob`, `iprint`, `nrest`, `iter` and `tol`, they have exactly the same meaning as in the sequential case; see Section 2.2. The only difference, already mentioned there, is that the MPI version cannot work directly with the transpose of the input matrix. Hence `ijob`=100,101,102,110,112 are not permitted.

Important remark: whereas `ijob`, `nrest`, `iter` and `tol` **should be assigned the same value on all tasks**, `iprint` *may be rank dependent*; in fact, in case of massive parallelism, it is even strongly advised against using positive `iprint` on more than just a few tasks. (In general, setting `iprint` to positive only for the Task with rank 0 provides enough information.)

4.1.1 `AGMGPARG4` and `AGMGPARG8`

Professional version only

These flavors of the driver assume that a global ordering of the unknowns has been set up in the calling program. Then, each task has to receive a contiguous block of rows according to this ordering, and, moreover, these blocks of rows should be attributed to the tasks in increasing order of their MPI rank: the Task (with rank) 0 holds the first block of rows, the Task (with rank) 1 the next one, etc.

Then, the matrix supplied in arguments `a`, `ja`, `ia` on input is just the submatrix corresponding to the locally assigned block of rows (and all columns: each supplied row has to contain the whole set of its entries, regardless whether column indexes point to local rows or not). This means that `ja` contains the global column indexes, but that `ia` is set up as if one would have defined a matrix with rows ranging from `1` to `n`: the i^{th} entry in `ia` points to the beginning of the i^{th} local row in `a` and `ja`, regardless the global index this row may have.

As in the sequential case, entries in a same row may occur in any order, whereas AGMG will perform a partial reordering inside each row, implying that, on output, `a` and `ja` are possibly different than on input; they nevertheless still represent the same matrix.

By way of illustration, consider the same matrix as in Section 2.1 partitioned into 3 tasks. We assume that Task 0 receives rows 1 & 2, Task 1 rows 3 & 4, and Task 2 row 5 (which is consistent with the requirements stated above):

$$A = \left(\begin{array}{cc|cc|c} 10 & & & -1 & \\ -2 & 11 & & -3 & \\ \hline & -4 & 12 & & -5 \\ & & & 13 & \\ \hline & -8 & -9 & & 14 \end{array} \right)$$

This yields (for instance, since the ordering inside each row is arbitrary):

TASK 0

$$\begin{aligned} \texttt{n} &= 2 \\ \texttt{a} &= \left[\begin{array}{cc|ccc} 10.0 & -1.0 & -2.0 & 11.0 & -3.0 \end{array}\right] \\ \texttt{ja} &= \left[\begin{array}{cc|ccc} 1 & 4 & 1 & 2 & 4 \end{array}\right] \\ \texttt{ia} &= \left[\begin{array}{ccc} 1 & 3 & 6 \end{array}\right] \end{aligned}$$

TASK 1

$$\begin{aligned} \texttt{n} &= 2 \\ \texttt{a} &= \left[\begin{array}{ccc|c} -4.0 & 12.0 & -5.0 & 13.0 \end{array}\right] \\ \texttt{ja} &= \left[\begin{array}{ccc|c} 2 & 3 & 5 & 4 \end{array}\right] \\ \texttt{ia} &= \left[\begin{array}{ccc} 1 & 4 & 5 \end{array}\right] \end{aligned}$$

TASK 2

$$\begin{aligned} \texttt{n} &= 1 \\ \texttt{a} &= \left[\begin{array}{ccc} 14.0 & -9.0 & -8.0 \end{array}\right] \\ \texttt{ja} &= \left[\begin{array}{ccc} 5 & 3 & 2 \end{array}\right] \\ \texttt{ia} &= \left[\begin{array}{cc} 1 & 4 \end{array}\right] \end{aligned}$$

`AGMGPARG4` OR `AGMGPARG8`. The only difference between both drivers is the declaration of `ja`. With `AGMGPARG4`, `ja` has the default integer size for the given architecture and compiler (in general, this means 32-bit integers). With `AGMGPARG8`, `ja` is declared as `INTEGER(SELECTED_INT_KIND(15))` (in general, this means 64-bit integers). This allows one to use matrices whose global size exceeds the standard integer range. Observe that only `ja` is affected by this modified declaration: all other parameters refer to "local" quantities and hence can be stored in standard integer format. This makes `AGMGPARG8` significantly less memory consuming that `AGMGPARG4` compiled with a flag turning the default integer size to 64-bit.

4.1.2 `AGMGPAR`

This flavor of the driver does not refer to any global ordering of the unknowns. Instead, one has to define on each task some LOCAL ORDERING of the unknowns, such that local rows (and thus local variables) have numbers $1, \ldots, \texttt{n}$, while nonlocal variable are given arbitrary indexes larger than `n` (see below).

NONLOCAL CONNECTIONS. Offdiagonal entries present in the local rows but connecting with nonlocal variables are to be referenced in the usual way; however, the corresponding column indices must be larger than `n`. The matrix supplied to AGMGPAR is thus formally a rectangular matrix with `n` rows, and an entry A_{ij} (with $1 \leq i \leq \texttt{n}$) corresponds to a local connection if $j \leq \texttt{n}$ and to an external connection if $j > \texttt{n}$.

IMPORTANT RESTRICTION. **The global matrix must be structurally symmetric with respect to nonlocal connections**. That is, if A_{ij} corresponds to an external connection, the local row corresponding to j, whatever the task to which it is assigned, should also contain an offdiagonal entry (with, possibly, a numerical value equal to zero) referencing (an external variable corresponding to) i.

CONSISTENCY OF LOCAL ORDERINGS. Besides the condition that they are larger than `n`, indexes of nonlocal variable may be chosen arbitrarily, providing that their ordering is consistent with the local ordering on their "home" task (the task to which the corresponding row is assigned). That is, if A_{ij} and A_{kl} are both present and such that j, $l > \texttt{n}$, and if further j and l have same home task (as specified in `listrank`, see below), then one should have $j < l$ if and only if, on their home task, the variable corresponding to j has lower index than the variable corresponding to l.

These constraints on the input matrix should not be difficult to meet in practice. Thanks to them, AGMG may set up the parallel solution process with minimal additional information. In fact, AGMG has only to know what is the rank of the "home" task of each referenced non-local variable. This information is supplied in input vector `listrank`.

Let $j_{\min}$ and $j_{\max}$ be, respectively, the smallest and the largest index of a non-local variable referenced in `ja` ($\texttt{n} < j_{\min} \leq j_{\max}$). Only entries `listrank`($j$) for $j_{\min} \leq j \leq j_{\max}$ will be referenced and need to be defined. If j is effectively present in `ja`, `listrank`(j) should be equal to the rank of the "home" task of (the row corresponding to) j; otherwise, `listrank`(j) should be equal to an arbitrary ***negative*** integer.

`listrank` is declared in AGMGPAR as `listrank(ifirstlistrank:*)`.
Setting `ifirstlistrank`= $j_{\min}$ or `ifirstlistrank`=`n`+1 allows to save on memory, since `listrank`(i) is never referenced for $i < j_{\min}$ (hence in particular for $i \leq \texttt{n}$).

By way of illustration, consider the same matrix as in Section 2.1 partitioned into 3 tasks, Task 0 receiving rows 1 & 2, Task 1 rows 3 & 4, and Task 2 row 5. Firstly, one has to add a few entries into the structure to enforce the structural symmetry with respect to non-local connections:

$$A = \left(\begin{array}{cc|cc|c} 10 & & & -1 & \\ -2 & 11 & & -3 & \\ \hline & -4 & 12 & & -5 \\ & & & 13 & \\ \hline & -8 & -9 & & 14 \end{array}\right) \to \left(\begin{array}{cc|cc|c} 10 & & & -1 & \\ -2 & 11 & 0 & -3 & 0 \\ \hline & -4 & 12 & & -5 \\ 0 & 0 & & 13 & \\ \hline & -8 & -9 & & 14 \end{array}\right).$$

Then A is given (non uniquely) by the following variables and vectors.

TASK 0

$$\begin{aligned} \texttt{n} &= 2 \\ \texttt{a} &= \left[\begin{array}{cc|ccccc} 10.0 & -1.0 & -2.0 & 11.0 & -3.0 & 0.0 & 0.0 \end{array}\right] \\ \texttt{ja} &= \left[\begin{array}{cc|ccccc} 1 & 4 & 1 & 2 & 4 & 3 & 5 \end{array}\right] \\ \texttt{ia} &= \left[\begin{array}{ccc} 1 & 3 & 8 \end{array}\right] \\ \texttt{listrank} &= \left[\begin{array}{ccccc} * & * & 1 & 1 & 2 \end{array}\right] \end{aligned}$$

TASK 1

$$\begin{aligned} \texttt{n} &= 2 \\ \texttt{a} &= \left[\begin{array}{ccc|ccc} -4.0 & 12.0 & -5.0 & 13.0 & 0.0 & 0.0 \end{array}\right] \\ \texttt{ja} &= \left[\begin{array}{ccc|ccc} 7 & 1 & 5 & 2 & 6 & 7 \end{array}\right] \\ \texttt{ia} &= \left[\begin{array}{ccc} 1 & 4 & 7 \end{array}\right] \\ \texttt{listrank} &= \left[\begin{array}{ccccccc} * & * & * & * & 2 & 0 & 0 \end{array}\right] \end{aligned}$$

TASK 2

$$\begin{aligned} \texttt{n} &= 1 \\ \texttt{a} &= \left[\begin{array}{ccc} 14.0 & -9.0 & -8.0 \end{array}\right] \\ \texttt{ja} &= \left[\begin{array}{ccc} 1 & 5 & 3 \end{array}\right] \\ \texttt{ia} &= \left[\begin{array}{cc} 1 & 4 \end{array}\right] \\ \texttt{listrank} &= \left[\begin{array}{ccccc} * & * & 0 & -1 & 1 \end{array}\right] \end{aligned}$$

Note that these vectors, in particular the numbering of nonlocal variables, were not constructed in a logical way, but rather to illustrate the flexibility and also the requirements of the format. One sees for instance with TASK 2 that the numbering of nonlocal variables may be quite arbitrary as long as "holes" in `listrank` are filled with negative numbers.

COMMON ERRORS. Be careful that `ifirstlistrank` is also output parameter. Hence declaring it as a constant in the calling program entails a fatal error. On the other hand, don't forget to initialize "holes" in `listrank` with negative numbers.

4.2 Example

The source file of the following example is provided with the package. The matrix corresponding to the five-point approximation of the Laplacian on the unit square is partitioned according a strip partitioning of the domain, with internal boundaries parallel to the x direction. Note that this strip partitioning is not optimal and has been chosen for the sake of simplicity.

If IRANK>0, nodes at the bottom line have a non-local connection with Task IRANK-1, and if IRANK<NPROC-1, nodes at the top line have a non-local connection with Task IRANK+1. Observe that these non-local variables are given indexes >n in subroutine uni2dstrip, and that listrank is set up accordingly. Note also that with this simple example the consistency of local orderings arises in a natural way.

Note also that each task will print its output in a different file. This is recommended.

Listing 3: source code of MPI Example (FORTRAN 90)

```
      program example_par
!
!  Solves the discrete Laplacian on the unit square by simple call to agmg.
!  The right-hand-side is such that the exact solution is the vector of all 1.
!  Uses a strip partitioning of the domain, with internal boundaries parallel
!       to the x direction.
!
       implicit none
       include 'mpif.h'
       real (kind(0d0)),allocatable :: a(:),f(:),x(:)
       integer,allocatable :: ja(:),ia(:),listrank(:)
       integer :: n,iter,iprint,nhinv,NPROC,IRANK,mx,my,ifirstlistrank,ierr
       real (kind(0d0)) :: tol
       character*10 filename
!
!      set inverse of the mesh size (feel free to change)
       nhinv=1000
!
!      maximal number of iterations
       iter=50
!
!      tolerance on relative residual norm
       tol=1.e-6
!
!      Initialize MPI
!
       call MPI_INIT(ierr)
       call MPI_COMM_SIZE(MPI_COMM_WORLD,NPROC,ierr)
       call MPI_COMM_RANK(MPI_COMM_WORLD,IRANK,ierr)
```

```
!
!       unit number for output messages (alternative: iprint=10+IRANK)
        iprint=10
        filename(1:8)='res.out_'
        write (filename(9:10),'(i2.2)') IRANK        ! processor dependent
        open(iprint,file=filename,form='formatted')
!
!       calculate local grid size
!
        mx=nhinv-1
        my=(nhinv-1)/NPROC
        if (IRANK < mod(nhinv-1,NPROC)) my=my+1
!
!       generate the matrix in required format (CSR)
!
!         first allocate the vectors with correct size
            n=mx*my
            allocate (a(5*n),ja(5*n),ia(n+1),f(n),x(n),listrank(2*mx))
!           external nodes connected with local ones on top and bottom
!           internal boundaries will receive numbers [n+1,...,n+2*mx]
            ifirstlistrank=n+1
!         next call subroutine to set entries
!          before, initialize listrank to zero so that entries
!          that do not correspond to a nonlocal variable present
!          in ja are anyway properly defined
            listrank(1:2*mx)=0
            call uni2dstrip(mx,my,f,a,ja,ia,IRANK,NPROC,listrank,ifirstlistrank)
!
!       call agmg
!         argument 5 (ijob)  is 0 because we want a complete solve
!         argument 7 (nrest) is 1 because we want to use flexible CG
!                            (the matrix is symmetric positive definite)
!
        call dagmgpar(n,a,ja,ia,f,x,0,iprint,1,iter,tol,          &
                      MPI_COMM_WORLD,listrank,ifirstlistrank)
!
        call MPI_FINALIZE(ierr)
!
      end program example_par
!------------------------------------------------------------------------
      subroutine uni2dstrip(mx,my,f,a,ja,ia,IRANK,NPROC,listrank,ifirstlistrank)
!
! Fill a matrix in CSR format corresponding to a constant coefficient
! five-point stencil on a rectangular grid
! Bottom boundary is an internal boundary if IRANK > 0, and
!     top boundary is an internal boundary if IRANK < NPROC-1
!
        implicit none
        real (kind(0d0)) :: f(*),a(*)
        integer :: mx,my,ia(*),ja(*),ifirstlistrank,listrank(ifirstlistrank:*)
        integer :: IRANK,NPROC,k,l,i,j
        real (kind(0d0)), parameter :: zero=0.0d0,cx=-1.0d0,cy=-1.0d0, cd=4.0d0
        !
        k=0
        l=0
        ia(1)=1
        do i=1,my
           do j=1,mx
              k=k+1
              l=l+1
              a(l)=cd
              ja(l)=k
              f(k)=zero
              if(j < mx) then
                 l=l+1
                 a(l)=cx
                 ja(l)=k+1
              else
                 f(k)=f(k)-cx
              end if
              if(i < my) then
                 l=l+1
                 a(l)=cy
                 ja(l)=k+mx
              else if (IRANK == NPROC-1) then
                 f(k)=f(k)-cy                  !real boundary
              else
                 l=l+1                 !internal boundary (top)
                 a(l)=cy               ! these external nodes are given the
                 ja(l)=k+mx            !   numbers [mx*my+1,...,mx*(my+1)]
                                       !Thus: listrank(mx*my+1:mx*(my+1))=IRANK+1
                listrank(k+mx)=IRANK+1
              end if
              if(j > 1) then
                 l=l+1
                 a(l)=cx
                 ja(l)=k-1
              else
                 f(k)=f(k)-cx
              end if
        if(i >  1) then
           l=l+1
           a(l)=cy
           ja(l)=k-mx
        else if (IRANK == 0) then
           f(k)=f(k)-cy             !real boundary
        else
           l=l+1            !internal boundary (bottom)
           a(l)=cy          ! these external nodes are given the
           ja(l)=k+mx*(my+1)! numbers [mx*(my+1)+1,...,mx*(my+2)]
                            !Thus: listrank(mx*(my+1)+1:mx*(my+2))=IRANK-1
           listrank(k+mx*(my+1))=IRANK-1
        end if
        ia(k+1)=l+1
     end do
  end do
  !
  return
end subroutine uni2dstrip
```

4.3 Printed output

Running the above example with 4 tasks, Task 0 produces the following output.

```
  0*ENTERING AGMG **********************************************************

**** Global number of unknowns:      998001
  0*      Number of local rows:      249750
**** Global number of nonzeros:     4986009 (per row:   5.00)
  0*    Nonzeros in local rows:     1247251 (per row:   4.99)

  0*SETUP: Coarsening by multiple pairwise aggregation
****  Rmk: Setup performed assuming the matrix symmetric
****       Quality threshold (BlockD): 10.00 ;  Strong diag. dom. trs: 1.22
****         Maximal number of passes:  3  ; Target coarsening factor: 8.00
****                     Threshold for rows with large pos. offdiag.: 0.45

  0*                      Level:           2
**** Global number of variables:      124563          (reduction ratio: 8.01)
  0*       Number of local rows:       31249          (reduction ratio: 7.99)
****  Global number of nonzeros:      622693 (per row: 5.0; red. ratio: 8.01)
  0*     Nonzeros in local rows:      155996 (per row: 5.0; red. ratio: 8.00)

  0*                      Level:           3
**** Global number of variables:       15704          (reduction ratio: 7.93)
  0*       Number of local rows:        3860          (reduction ratio: 8.10)
****  Global number of nonzeros:       79406 (per row: 5.1; red. ratio: 7.84)
  0*     Nonzeros in local rows:       19390 (per row: 5.0; red. ratio: 8.05)

  0*                      Level:           4
**** Global number of variables:        1963          (reduction ratio: 8.00)
  0*       Number of local rows:         465          (reduction ratio: 8.30)
****  Global number of nonzeros:       10035 (per row: 5.1; red. ratio: 7.91)
  0*     Nonzeros in local rows:        2294 (per row: 4.9; red. ratio: 8.45)

****           Global grid complexity:     1.14
****       Global Operator complexity:     1.14
  0*            Local grid complexity:     1.14
  0*        Local Operator complexity:     1.14
****  Theoretical Weighted complexity:     1.33 (K-cycle at each level)
****    Effective Weighted complexity:     1.33 (V-cycle enforced where needed)

  0*       Setup time (Elapsed):     1.37E-01 seconds
  0*SOLUTION: flexible conjugate gradient iterations (FCG(1))
****     AMG preconditioner with 1 Gauss-Seidel pre- and post-smoothing sweeps
****         at each level
****  Iter=    0       Resid= 0.63E+02       Relat. res.= 0.10E+01
****  Iter=    1       Resid= 0.20E+02       Relat. res.= 0.32E+00
****  Iter=    2       Resid= 0.54E+01       Relat. res.= 0.86E-01
****  Iter=    3       Resid= 0.20E+01       Relat. res.= 0.31E-01
****  Iter=    4       Resid= 0.54E+00       Relat. res.= 0.85E-02
****  Iter=    5       Resid= 0.30E+00       Relat. res.= 0.47E-02
****  Iter=    6       Resid= 0.84E-01       Relat. res.= 0.13E-02
****  Iter=    7       Resid= 0.39E-01       Relat. res.= 0.62E-03
****  Iter=    8       Resid= 0.11E-01       Relat. res.= 0.18E-03
****  Iter=    9       Resid= 0.40E-02       Relat. res.= 0.63E-04
****  Iter=   10       Resid= 0.20E-02       Relat. res.= 0.31E-04
****  Iter=   11       Resid= 0.77E-03       Relat. res.= 0.12E-04
****  Iter=   12       Resid= 0.28E-03       Relat. res.= 0.45E-05
****  Iter=   13       Resid= 0.11E-03       Relat. res.= 0.18E-05
****  Iter=   14       Resid= 0.41E-04       Relat. res.= 0.64E-06
****  - Convergence reached in   14 iterations -

****     level 2   #call=    14   #cycle=    28   mean=   2.00    max=  2
****     level 3   #call=    28   #cycle=    56   mean=   2.00    max=  2

  0*       Number of work units:    11.51 per digit of accuracy (*)
  0*    Solution time (Elapsed):     4.02E-01 seconds

  0 (*) 1 work unit represents the cost of 1 (fine grid) residual evaluation
  0*LEAVING AGMG * (MEMORY RELEASED) ******************************************
```

For some items, both a local and a global quantity are now given. All output lines starting with `****` correspond to global information and are printed only by the Task with rank 0. Other lines starts with `xxx*`, where "xxx" is the task rank. They are printed by each task, based on local computation. For instance, reported number of work units for the solution phase is based on local the number of local floating point operations relative to the local cost of one residual evaluation. If one has about the same number on each task, it means that the initial load balancing (whether good or bad) has been well preserved in AGMG.

4.4 Tuning the MPI version

As already written in Section 2.6, default parameters defined in the module `dagmgpar_mem` at the very top of the source file are easy to change. Perhaps such tuning is more useful in the parallel case than in the sequential one. Here performances not only depend on the application, but also on the computer; e.g., the parameters providing smallest average solution time may depend on the communication speed.

Compared with the sequential version, by default, the number of pairwise aggregation passes `npass` has been increased from 2 to 3. This favors more aggressive coarsening, in general at the price of some increase of the setup time. Hence, on average, this slightly increases the sequential time of roughly 10 %. But this turns out to be often beneficial in parallel, because this reduces the time spent on small grids where communications are relatively more costly and more difficult to overlap with computation. If, however, you experience convergence difficulties, it may be wise to restore the default `npass`=2. On the other hand, you may try to obtain even more aggressive coarsening by increasing also `kaptg_blocdia` and/or `kaptg_dampJac`, after having checked effects on the convergence speed (they are application dependent).

Eventually, in the parallel case, it is worth checking how the solver works on the coarsest grid. Large set up time may indicate that this latter is too large. This can be reduced by decreasing `maxcoarsesize` and `maxcoarsesizeslow`. It is also possible to tune their effect according to the number of processors in subroutine `dagmgpar_init` (just below `agmgpar_mem`). Conversely, increasing the size of the coarsest system may help reduce communications as long as the coarsest solver remains efficient, which depends not only of this size, but also on the number of processors and on the communication speed. *Professional version only: with the professional version, the coarsest grid solver is a special iterative solver, which is insensitive to the global coarsest grid size; parameters that define the target accuracy and the maximal number of iterations at this level can also be tuned.*

5 Hybrid version

Professional version only

The hybrid version adds multithreading on top of the MPI version; that is, each MPI rank will run in multithread mode.

From the user viewpoint, there is no difference between the MPI and hybrid versions.

Drivers, naming convention and input parameters are thus exactly as described in the previous section.

Which version is used (MPI or hybrid) is determined by object file the application program is linked with: those with name containing *_hyb* provide the hybrid version, those containing *_mpi* provide the pure MPI version.

The number of used threads for each MPI rank is equal to the standard one for OpenMP, according to the environment; i.e., it may be adjusted (if needed) by setting the environment variable OMP_NUM_THREADS. A specific rule for AGMG can be defined by setting the environment variable AGMG_NUM_THREADS; according the hardware and the type of problems, the best number of threads may indeed be occasionally below the number of available cores (enter at shell prompt: *export AGMG_NUM_THREADS=k*, where *k* is the desired number of threads).

If the parameter `iprint` is set to a positive number, AGMG will report in the first output lines about the number of threads actually used (thus confirming that one has correctly linked with the hybrid version).

Note that, on each MPI rank, the number of threads should not exceed the number of local rows `n`; otherwise AGMG will exit with an error message. (Such usage makes little sense but may occur during test phase.)

Note also that AGMG will properly work only if the same number of threads is used on all MPI ranks. When, for any reason (heterogeneous platform, dynamic behavior according to the processor load), the default number of threads might be different on different MPI ranks, it is then mandatory to enforce a common number of threads by defining either the OMP_NUM_THREADS or the AGMG_NUM_THREADS environment variable. The rationale for this is as follows. On the one hand, this simplifies a bit the implementation. On the other hand, we don't think that, to date, there is a real demand more flexibility. If you think otherwise and feel that this limits the practical usage of AGMG, please share with us your experience and arguments.

6 Running several instances of AGMG

In some contexts, it is interesting to have simultaneously in memory several instances of the preconditioner. For instance, when a preconditioner for a matrix in block form is defined considering separately the different blocks, and when one would like to use AGMG for several of these blocks.

Such usage of AGMG is not foreseen with the present version. However, there is a simple workaround. In the sequential case, copy the source file, say, dagmg.f90 to dagmg1.f90. Then, edit dagmg1.f90, and make a global replacement of all instances of `dagmg` (lowercase) by `dagmg1`, *but not* `DAGMG_MUMSP` (uppercase), which should remain unchanged. Repeat this as many times as needed to generate dagmg2.f90, dagmg3.f90, etc. Then, compile all these source files together with the application program and dagmg_mumps.f90. The application program can call dagmg1(), dagm2(), dagm3(), . . . , defining several instances of the preconditioner that will not interact with each other.

You can proceed similarly with any arithmetic and also the parallel versions. In the latter case, replace all instances of `dagmgpar` by `dagmgpar1` (no exception).

Octave: the same result is obtained as follows. Copy the file agmg.cc to, say, agmg1.cc. Edit agmg1.cc and replace the line

```
DEFUN_DLD (agmg, args, nargout,
```

with the line

```
DEFUN_DLD (agmg1, args, nargout,
```

(i.e., in that line, replace "agmg" by "agmg1": the new filename without extension). Then, edit the m-file make_agmg_octfile.m and replace agmg.cc by agmg1.cc (the new filename). Run that script in an Octave session. This will produce a file agmg1.oct, defining a function *agmg1* identical to *agmg*, but which can be called independently. You may repeat this process as many times as you want to generate functions *agmg2*, *agmg3*, etc.

Matlab: the same result is obtained as follows. Copy the three files agmg.m, dmtlagmg.mex??? and zmtlagmg.mex??? (where ??? depends upon your OS and architecture) to, say, agmg1.m, dmtlagmg1.mex??? and zmtlagmg1.mex???. Edit the m-file agmg1.m, and replace the 3 calls to `dmtlagmg` by calls to `dmtlagmg1`, and the 3 calls to `zmtlagmg` by calls to `zmtlagmg1`. This defines a function *agmg1* identical to *agmg*, but which can be called independently. You may repeat this process as many times as you want to generate functions *agmg2*, *agmg3*, etc.

7 Solving singular systems

Singular systems can be solved directly. Further, this is done automatically, without needing to tell the software that the provided system is singular. However, this usage should be considered with care and a few limitations apply. Basically, it works smoothly when the system is compatible and when the left null space is spanned by the vector of all ones (i.e., the range of the input matrix is the set of vectors orthogonal to the constant vector and the right hand side belongs to this set). Please note that failures in other cases are generally not fatal and hence not detected if the solution is not checked appropriately; for instance, AGMG may seemingly work well but returns a solution vector which is highly dominated by null space components.

To handle such cases, note that AGMG can often be efficiently applied to solve the near singular system obtained by deleting one row and one column in a linear system originally singular (and compatible). This approach is not recommended in general because the truncated system obtained in this way tends to be ill-conditioned and hence no very easy to solve with iterative methods in general. However, it is sensible to use this approach with AGMG because one of the features of the method in AGMG is precisely its capability to solve efficiently ill-conditioned linear systems.

A Listing of DAGMG (sequential version)

```
  SUBROUTINE dagmg( n,a,ja,ia,f,x,ijob,iprint,nrest,iter,tol )
    INTEGER     :: n,ia(n+1),ja(*),ijob,iprint,nrest,iter
    REAL (kind(0.0d0)) :: a(*),f(n),x(n)
    REAL (kind(0.0d0)) :: tol
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  Arguments
!  =========
!
!  N         (input) INTEGER.
!            The dimension of the matrix.
!
!  A         (input/output) REAL (kind(0.0e0)). Numerical values of the matrix.
!  IA        (input/output) INTEGER. Pointers for every row.
!  JA        (input/output) INTEGER. Column indices.
!
!            Significant only if IJOB==0,1,2,10,12,100,101,102,110,112
!
!            Detailed description of the matrix format
!
!                On input, IA(I), I=1,...,N, refers to the physical start
!                of row I. That is, the entries of row I are located
!                in A(K), where K=IA(I),...,IA(I+1)-1. JA(K) carries the
!                associated column indices. IA(N+1) must be defined in such
!                a way that the above rule also works for I=N (that is,
!                the last valid entry in arrays A,JA should correspond to
!                index K=IA(N+1)-1).
!
!            AGMG ASSUMES THAT ALL DIAGONAL ENTRIES ARE POSITIVE
!
!                That is, AGMG assumes that, for K=IA(I),...,IA(I+1)-1,
!                some of JA(K) is equal to I with corresponding A(K)>0
!                (value of the diagonal element, which must be positive).
!
!                A,IA,JA are "output" parameters because on exit the
!                entries of each row may occur in a different order
!                (The matrix is mathematically the same, but stored in
!                different way).
!
!
!  F         (input/output) REAL (kind(0.0e0)).
!            On input, the right hand side vector f.
!            Overwritten on output.
!            Significant only if IJOB is not equal to 1 or 101.
```

```
!
!  X        (input/output) REAL (kind(0.0e0)).
!           On input and if IJOB== 10, 12, 110, 112, 212: initial guess
!           On output, the computed solution
!                (IJOB==3: result of the application of the preconditioner)
!
! IJOB      (input) INTEGER. Tells AGMG what has to be done.
!            0: performs setup + solve + memory release, no initial guess
!           10: performs setup + solve + memory release, initial guess in x(1:n)
!            1: performs setup only
!               (preprocessing: prepares all parameters for subsequent solves)
!            2: solves only with the preconditioner based on previous setup,
!               using the system matrix provided in A, JA, IA; no initial guess
!           12: solves only with the preconditioner based on previous setup,
!               using the system matrix provided in A, JA, IA;
!               initial guess in x(1:n)
!          202: solves only with the preconditioner based on previous setup and
!               the same matrix as the one provided for set up;  no initial guess
!               using the system matrix provided during previous setup
!          212: solves only with the preconditioner based on previous setup and
!               the same matrix as the one provided for set up;
!               initial guess in x(1:n)
!            3: the vector returned in x(1:n) is not the solution of the linear
!                   system, but the result of the action of the multigrid
!                   preconditioner to the right hand side in f(1:n)
!           -1: erases the setup and releases internal memory
!
!   IJOB == 100,110,101,102,112: same as, respectively, IJOB==0,10,1,2,12
!        but, use the TRANSPOSE of the input matrix in A, JA, IA.
!
!   !!! IJOB==2,3,12,102,112,202,212 require that one has previously called
!        AGMG with IJOB==1 or IJOB==101
!
!   !!! (change with respect to versions 2.x) !!!
!        The preconditioner defined when calling AGMG
!          with IJOB==1 or IJOB==101 is entirely kept in internal memory.
!        Hence the arrays A, JA and IA are not accessed upon subsequent calls
!          with IJOB==3 and IJOB==202,212
!        Upon subsequent calls with IJOB==2,12,102,112: a matrix needs to
!             be supplied in arrays A, JA, IA. It will be used to
!             perform matrix vector product within the main iterative
!             loop (and only for this).
!             Hence the system is solved with this matrix which
!             may differ from the matrix in A, JA, IA that was supplied
!             upon the previous call with IJOB==1 or IJOB==101;
!              then AGMG attempts to solve a linear system with the "new"
!              matrix (supplied when IJOB==2,12,102 or 112) using the
!              preconditioner set up for the "old" one (supplied when
!              IJOB==1 or 101).
!          The same remarks apply to IJOB >= 100 or not: the value IJOB==1
!              or 101 determines whether the preconditioner set up and stored
!              in internal memory is based on the matrix or its transpose;
!              the value IJOB==2,12 or 102,112 is used to determine whether
!              the linear system to be solved is with the matrix or its
!              transpose, independently of the set up.
!              Hence one may set up a preconditioner for a matrix and use it
!              for its transpose.
!        These functionalities (set up a preconditioner and use it for another
!              matrix) are provided for the sake of generality but should be
!              used with care; in general, set up is fast with AGMG and hence
!              it is recommended to rerun it even if the matrix changes only
!              slightly.
!        In the more standard case where one needs successive solves with a
!        same matrix, it recommended to use IJOB==202 or 212 (in which case
!        the arrays A, JA, IA are not accessed) instead of IJOB==2 or 12
!        (in which case the input matrix has to be kept equal to the one
!        provided for set up). This  slightly faster and helps to save memory
!        since the arrays that store the input matrix may be deallocated after
!        set up.
!
! IPRINT    (input) INTEGER.
!                Indicates the unit number where information is to be printed
!                (N.B.: 5 is converted to 6). If nonpositive, only error
!                messages are printed on standard output. Warning messages about
!                insufficient convergence (in the allowed  maximum number of
!                iterations) are further suppressed supplying a negative number.
!
! NREST     (input) INTEGER.
!               Restart parameter for GCR (an implementation of GMRES)
!               Nonpositive values are converted to NREST=10 (default)
!
! !!   If NREST==1, Flexible CG is used instead of GCR (when IJOB==0,10,2,
!               12,100,110,102,112) and also (IJOB==0,1,100,101) performs some
!               simplification during the set up based on the assumption
!               that the matrix supplied in A, JA, IA is symmetric (there is
!               then no more difference between IJOB==1 and IJOB==101).
!
! !!!  NREST=1 Should be used if and only if the matrix is really SYMMETRIC
! !!!           (and positive definite).
!
```

```
!   ITER      (input/output) INTEGER.
!             On input, the maximum number of iterations. Should be positive.
!             On output, actual number of iterations.
!               If this number of iteration was insufficient to meet convergence
!               criterion, ITER will be returned negative and equal to the
!               opposite of the number of iterations performed.
!             Significant only if IJOB==0, 2, 10, 12, 100, 102, 110, 112, 202, 212
!
!   TOL       (input) REAL (kind(0.0e0))
!             The tolerance on residual norm. Iterations are stopped whenever
!                  || A*x-f || <= TOL* || f ||
!             Should be positive and less than 1.0
!             Significant only if IJOB==0, 2, 10, 12, 100, 102, 110, 112, 202, 212
!
!
!!!!! Remark !!!! Except insufficient number of iterations to achieve
!                  convergence (characterized by a negative value returned
!                  in ITER), all other detected errors are fatal and lead
!                  to a STOP statement.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

B Listing of DAGMG (multithreaded version)

```
  SUBROUTINE dagmg( n,a,ja,ia,f,x,ijob,iprint,nrest,iter,tol )
    INTEGER    :: n,ia(n+1),ja(*),ijob,iprint,nrest,iter
    REAL (kind(0.0d0)) :: a(*),f(n),x(n)
    REAL (kind(0.0d0)) :: tol
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  Arguments
!  =========
!
!  N        (input) INTEGER.
!           The dimension of the matrix.
!
!  A        (input/output) REAL (kind(0.0e0)). Numerical values of the matrix.
!  IA       (input/output) INTEGER. Pointers for every row.
!  JA       (input/output) INTEGER. Column indices.
!
!           Significant only if IJOB==0,1,2,10,12
!
!           Detailed description of the matrix format
!
!               On input, IA(I), I=1,...,N, refers to the physical start
!               of row I. That is, the entries of row I are located
!               in A(K), where K=IA(I),...,IA(I+1)-1. JA(K) carries the
!               associated column indices. IA(N+1) must be defined in such
!               a way that the above rule also works for I=N (that is,
!               the last valid entry in arrays A,JA should correspond to
!               index K=IA(N+1)-1).
!
!           AGMG ASSUMES THAT ALL DIAGONAL ENTRIES ARE POSITIVE
!
!               That is, AGMG assumes that, for K=IA(I),...,IA(I+1)-1,
!               some of JA(K) is equal to I with corresponding A(K)>0
!               (value of the diagonal element, which must be positive).
!
!               A,IA,JA are "output" parameters because on exit the
!               entries of each row may occur in a different order
!               (The matrix is mathematically the same, but stored in
!               different way).
!
!
!  F        (input/output) REAL (kind(0.0e0)).
!           On input, the right hand side vector f.
!           Overwritten on output.
!           Significant only if IJOB is not equal to 1 or 101.
```

```
!
!  X        (input/output) REAL (kind(0.0e0)).
!           On input and if IJOB==10, 12, 212: initial guess
!           On output, the computed solution
!               (IJOB==3: result of the application of the preconditioner)
!
! IJOB      (input) INTEGER. Tells AGMG what has to be done.
!            0: performs setup + solve + memory release, no initial guess
!           10: performs setup + solve + memory release, initial guess in x(1:n)
!            1: performs setup only
!              (preprocessing: prepares all parameters for subsequent solves)
!            2: solves only with the preconditioner based on previous setup,
!              using the system matrix provided in A, JA, IA; no initial guess
!           12: solves only with the preconditioner based on previous setup,
!              using the system matrix provided in A, JA, IA;
!              initial guess in x(1:n)
!          202: solves only with the preconditioner based on previous setup and
!              the same matrix as the one provided for set up;  no initial guess
!              using the system matrix provided during previous setup
!          212: solves only with the preconditioner based on previous setup and
!              the same matrix as the one provided for set up;
!              initial guess in x(1:n)
!            3: the vector returned in x(1:n) is not the solution of the linear
!                  system, but the result of the action of the multigrid
!                  preconditioner to the right hand side in f(1:n)
!           -1: erases the setup and releases internal memory
!
!   !!! IJOB==2,3,12,202,212 require that one has previously called
!       AGMG with IJOB==1
!
!   !!! (change with respect to versions 2.x) !!!
!       The preconditioner defined when calling AGMG
!         with IJOB==1 is entirely kept in internal memory.
!       Hence the arrays A, JA and IA are not accessed upon subsequent calls
!         with IJOB==3 and IJOB==202,212
!       Upon subsequent calls with IJOB==2, 12:  a matrix needs to
!            be supplied in arrays A, JA, IA. It will be used to
!            perform matrix vector product within the main iterative
!            loop (and only for this).
!            Hence the system is solved with this matrix which
!            may differ from the matrix in A, JA, IA that was supplied
!            upon the previous call with IJOB==1;
!            then AGMG attempts to solve a linear system with the "new"
!            matrix (supplied when IJOB==2,12) using the preconditioner
!            set up for the "old" one (supplied when IJOB==1).
!       These functionalities (set up a preconditioner and use it for another
!            matrix) are provided for the sake of generality but should be
!            used with care; in general, set up is fast with AGMG and hence
!            it is recommended to rerun it even if the matrix changes only
!            slightly.
!       In the more standard case where one needs successive solves with a
!       same matrix, it recommended to use IJOB==202 or 212 (in which case
!       the arrays A, JA, IA are not accessed) instead of IJOB==2 or 12
!       (in which case the input matrix has to be kept equal to the one
!       provided for set up). This  slightly faster and helps to save memory
!       since the arrays that store the input matrix may be deallocated after
!       set up.
!
! IPRINT    (input) INTEGER.
!              Indicates the unit number where information is to be printed
!              (N.B.: 5 is converted to 6). If nonpositive, only error
!              messages are printed on standard output. Warning messages about
!              insufficient convergence (in the allowed  maximum number of
!              iterations) are further suppressed supplying a negative number.
!
! NREST     (input) INTEGER.
!             Restart parameter for GCR (an implementation of GMRES)
!             Nonpositive values are converted to NREST=10 (default)
!
! !!   If NREST==1, Flexible CG is used instead of GCR (when IJOB==0,10,2, 12)
!             and also (IJOB==0,1) performs some simplification during
!             the set up based on the assumption that the matrix
!             supplied in A, JA, IA is symmetric.
!
! !!!  NREST=1 Should be used if and only if the matrix is really SYMMETRIC
! !!!           (and positive definite).
!
!  ITER     (input/output) INTEGER.
!           On input, the maximum number of iterations. Should be positive.
!           On output, actual number of iterations.
!             If this number of iteration was insufficient to meet convergence
!             criterion, ITER will be returned negative and equal to the
!             opposite of the number of iterations performed.
!           Significant only if IJOB==0, 2, 10, 12, 202, 212
!
!  TOL      (input) REAL (kind(0.0e0))
!           The tolerance on residual norm. Iterations are stopped whenever
!                || A*x-f || <= TOL* || f ||
!           Should be positive and less than 1.0
!           Significant only if IJOB==0, 2, 10, 12, 202, 212
```

```
!
!
! Remarks for the multithreaded (OpenMP) version:
!
!      *  The number of threads to be used in OpenMP parallel regions
!         can be specified in the AGMG_NUM_THREADS environment variable
!         If this variable is not set, AGMG will use the default
!         provided by the working environment
!
!      *  Multithreaded and non-multithreaded (sequential or pure MPI)
!         versions are 100% compatible except that IJOB=100,101,110,112
!         (working with the transpose) is not allowed with the
!         multithreaded version
!         Which version is actually used depends on which object has
!         been loaded during link phase
!
!!!!! Remark !!!! Except insufficient number of iterations to achieve
!                 convergence (characterized by a negative value returned
!                 in ITER), all other detected errors are fatal and lead
!                 to a STOP statement.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

C Listing of DAGMGPAR (MPI version)

```
  SUBROUTINE dagmgpar( n,a,ja,ia,f,x,ijob,iprint,nrest,iter,tol &
                      ,MPI_COMM_AGMG,listrank,ifirstlistrank )
    INTEGER     :: n,ia(n+1),ja(*),ijob,iprint,nrest,iter
    REAL (kind(0.0d0)) :: a(*),f(n),x(n)
    REAL (kind(0.0d0)) :: tol
    INTEGER     :: MPI_COMM_AGMG,ifirstlistrank
    INTEGER     :: listrank(ifirstlistrank:*)
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  Arguments
!  =========
!
!  N        (input) INTEGER.
!           The number of rows in the local part of the matrix.
!
!  A        (input/output) REAL (kind(0.0e0)). Numerical values of the matrix
!                                               (local rows).
!  IA       (input/output) INTEGER. Pointers for every local row.
!  JA       (input/output) INTEGER. Column indices (w.r.t. local numbering).
!
!           Significant only if IJOB==0,1,2,10,12
!
!           Detailed description of the matrix format
!
!               On input, IA(I), I=1,...,N, refers to the physical start
!               of row I. That is, the entries of row I are located
!               in A(K), where K=IA(I),...,IA(I+1)-1. JA(K) carries the
!               associated column indices. IA(N+1) must be defined in such
!               a way that the above rule also works for I=N (that is,
!               the last valid entry in arrays A,JA should correspond to
!               index K=IA(N+1)-1).
!
!          AGMG ASSUMES THAT ALL DIAGONAL ENTRIES ARE POSITIVE
!
!               That is, AGMG assumes that, for K=IA(I),...,IA(I+1)-1,
!               some of JA(K) is equal to I with corresponding A(K)>0
!               (value of the diagonal element, which must be positive).
!
!               A,IA,JA are "output" parameters because on exit the
!               entries of each row may occur in a different order
!               (The matrix is mathematically the same, but stored in
!               different way).
!
!
```

```
!   F         (input/output) REAL (kind(0.0e0)).
!             On input, the right hand side vector f.
!             Overwritten on output.
!             Significant only if IJOB is not equal to 1 or 101.
!
!   X         (input/output) REAL (kind(0.0e0)).
!             On input and if IJOB==10, 12, 212: initial guess
!             On output, the computed solution
!                 (IJOB==3: result of the application of the preconditioner)
!
! IJOB        (input) INTEGER. Tells AGMG what has to be done.
!              0: performs setup + solve + memory release, no initial guess
!             10: performs setup + solve + memory release, initial guess in x(1:n)
!              1: performs setup only
!                 (preprocessing: prepares all parameters for subsequent solves)
!              2: solves only with the preconditioner based on previous setup,
!                 using the system matrix provided in A, JA, IA; no initial guess
!             12: solves only with the preconditioner based on previous setup,
!                 using the system matrix provided in A, JA, IA;
!                 initial guess in x(1:n)
!            202: solves only with the preconditioner based on previous setup and
!                 the same matrix as the one provided for set up;  no initial guess
!                 using the system matrix provided during previous setup
!            212: solves only with the preconditioner based on previous setup and
!                 the same matrix as the one provided for set up;
!                 initial guess in x(1:n)
!              3: the vector returned in x(1:n) is not the solution of the linear
!                     system, but the result of the action of the multigrid
!                     preconditioner to the right hand side in f(1:n)
!             -1: erases the setup and releases internal memory
!
!    !!! IJOB==2,3,12,202,212 require that one has previously called
!        AGMG with IJOB==1
!
!    !!! (change with respect to versions 2.x) !!!
!        The preconditioner defined when calling AGMG
!          with IJOB==1 is entirely kept in internal memory.
!        Hence the arrays A, JA and IA are not accessed upon subsequent calls
!          with IJOB==3 and IJOB==202,212
!        Upon subsequent calls with IJOB==2, 12:  a matrix needs to
!             be supplied in arrays A, JA, IA. It will be used to
!             perform matrix vector product within the main iterative
!             loop (and only for this).
!             Hence the system is solved with this matrix which
!             may differ from the matrix in A, JA, IA that was supplied
!               upon the previous call with IJOB==1;
!               then AGMG attempts to solve a linear system with the "new"
!               matrix (supplied when IJOB==2,12) using the preconditioner
!               set up for the "old" one (supplied when IJOB==1).
!         These functionalities (set up a preconditioner and use it for another
!               matrix) are provided for the sake of generality but should be
!               used with care; in general, set up is fast with AGMG and hence
!               it is recommended to rerun it even if the matrix changes only
!               slightly.
!         In the more standard case where one needs successive solves with a
!         same matrix, it recommended to use IJOB==202 or 212 (in which case
!         the arrays A, JA, IA are not accessed) instead of IJOB==2 or 12
!         (in which case the input matrix has to be kept equal to the one
!         provided for set up). This  slightly faster and helps to save memory
!         since the arrays that store the input matrix may be deallocated after
!         set up.
!
! IPRINT    (input) INTEGER.
!                  Indicates the unit number where information is to be printed
!                  (N.B.: 5 is converted to 6). If nonpositive, only error
!                  messages are printed on standard output. Warning messages about
!                  insufficient convergence (in the allowed  maximum number of
!                  iterations) are further suppressed supplying a negative number.
!
! NREST     (input) INTEGER.
!                 Restart parameter for GCR (an implementation of GMRES)
!                 Nonpositive values are converted to NREST=10 (default)
!
! !!   If NREST==1, Flexible CG is used instead of GCR (when IJOB==0,10,2, 12)
!                 and also (IJOB==0,1) performs some simplification during
!                 the set up based on the assumption that the matrix
!                 supplied in A, JA, IA is symmetric.
!
! !!!  NREST=1 Should be used if and only if the matrix is really SYMMETRIC
! !!!             (and positive definite).
!
!  ITER     (input/output) INTEGER.
!            On input, the maximum number of iterations. Should be positive.
!            On output, actual number of iterations.
!              If this number of iteration was insufficient to meet convergence
!              criterion, ITER will be returned negative and equal to the
!              opposite of the number of iterations performed.
!            Significant only if IJOB==0, 2, 10, 12, 202, 212
!
!  TOL      (input) REAL (kind(0.0e0))
!           The tolerance on residual norm. Iterations are stopped whenever
!                || A*x-f || <= TOL* || f ||
!           Should be positive and less than 1.0
!           Significant only if IJOB==0, 2, 10, 12, 202, 212
!
! MPI_COMM_AGMG (input) INTEGER
!                MPI communicator
!
! LISTRANK(IFIRSTLISTRANK:*) INTEGER (input/output)
!           Contains the rank of the tasks to which rows corresponding to
!           nonlocal variable referenced in (A,JA,IA) are assigned.
!           Let Jmin and Jmax be, respectively, the smallest and the largest
!           index of a nonlocal variable referenced in JA(1:IA(N+1)-1)
!           (Note that N < Jmin <= Jmax).
!           listrank(i) will be referenced if and only if  Jmin <= i <= Jmax,
!           and listrank(i) should then be equal to the rank of the "home"
!           task of i if i is effectively present in JA(1:IA(N+1)-1),
!           and equal to some arbitrary NEGATIVE integer otherwise.
!
!           listrank and ifirstlistrank may be modified on output, according
!           to the possible modification of the indexes of nonlocal variables;
!           that is, on output, listrank and ifirstlistrank still carry the
!           correct information about nonlocal variables, but for the
!           matrix as defined in (A,JA,IA) on output.
!
!
!!!!! Remark !!!! Except insufficient number of iterations to achieve
!                 convergence (characterized by a negative value returned
!                 in ITER), all other detected errors are fatal and lead
!                 to a STOP statement.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

D Listing of DAGMGPARG8 (MPI version)

```
  SUBROUTINE dagmgparG8( n,a,ja,ia,f,x,ijob,iprint,nrest,iter,tol &
                ,MPI_COMM_AGMG )
    INTEGER     :: n,ia(n+1),ijob,iprint,nrest,iter
    INTEGER (SELECTED_INT_KIND(15)) :: ja(*)
    REAL (kind(0.0d0)) :: a(*),f(n),x(n)
    REAL (kind(0.0d0)) :: tol
    INTEGER     :: MPI_COMM_AGMG
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!  Arguments
!  =========
!
!  N         (input) INTEGER.
!            The number of rows in the local part of the matrix.
!
!  A         (input/output) REAL (kind(0.0e0)). Numerical values of the matrix
!                                               (local rows).
!  IA        (input/output) INTEGER. Pointers for every local row.
!  JA        (input/output) INTEGER. Column indices (w.r.t. global numbering).
!
!            Significant only if IJOB==0,1,2,10,12
!
!            Detailed description of the matrix format
!
!                On input, IA(I), I=1,...,N, refers to the physical start
!                of row I. That is, the entries of row I are located
!                in A(K), where K=IA(I),...,IA(I+1)-1. JA(K) carries the
!                associated column indices. IA(N+1) must be defined in such
!                a way that the above rule also works for I=N (that is,
!                the last valid entry in arrays A,JA should correspond to
!                index K=IA(N+1)-1).
!
!           AGMG ASSUMES THAT ALL DIAGONAL ENTRIES ARE POSITIVE
!
!                That is, AGMG assumes that, for K=IA(I),...,IA(I+1)-1,
!                some of JA(K) is equal to I with corresponding A(K)>0
!                (value of the diagonal element, which must be positive).
!
!                A,IA,JA are "output" parameters because on exit the
!                entries of each row may occur in a different order
!                (The matrix is mathematically the same, but stored in
!                different way).
!
!
```

```
!   F         (input/output) REAL (kind(0.0e0)).
!             On input, the right hand side vector f.
!             Overwritten on output.
!             Significant only if IJOB is not equal to 1 or 101.
!
!   X         (input/output) REAL (kind(0.0e0)).
!             On input and if IJOB==10, 12, 212: initial guess
!             On output, the computed solution
!                 (IJOB==3: result of the application of the preconditioner)
!
! IJOB        (input) INTEGER. Tells AGMG what has to be done.
!             0: performs setup + solve + memory release, no initial guess
!            10: performs setup + solve + memory release, initial guess in x(1:n)
!             1: performs setup only
!                (preprocessing: prepares all parameters for subsequent solves)
!             2: solves only with the preconditioner based on previous setup,
!                using the system matrix provided in A, JA, IA; no initial guess
!            12: solves only with the preconditioner based on previous setup,
!                using the system matrix provided in A, JA, IA;
!                initial guess in x(1:n)
!           202: solves only with the preconditioner based on previous setup and
!                the same matrix as the one provided for set up;  no initial guess
!                using the system matrix provided during previous setup
!           212: solves only with the preconditioner based on previous setup and
!                the same matrix as the one provided for set up;
!                initial guess in x(1:n)
!             3: the vector returned in x(1:n) is not the solution of the linear
!                    system, but the result of the action of the multigrid
!                    preconditioner to the right hand side in f(1:n)
!            -1: erases the setup and releases internal memory
!
!    !!! IJOB==2,3,12,202,212 require that one has previously called
!        AGMG with IJOB==1
!
!    !!! (change with respect to versions 2.x) !!!
!        The preconditioner defined when calling AGMG
!          with IJOB==1 is entirely kept in internal memory.
!        Hence the arrays A, JA and IA are not accessed upon subsequent calls
!          with IJOB==3 and IJOB==202,212
!        Upon subsequent calls with IJOB==2, 12:  a matrix needs to
!             be supplied in arrays A, JA, IA. It will be used to
!             perform matrix vector product within the main iterative
!             loop (and only for this).
!             Hence the system is solved with this matrix which
!             may differ from the matrix in A, JA, IA that was supplied
!              upon the previous call with IJOB==1;
!              then AGMG attempts to solve a linear system with the "new"
!              matrix (supplied when IJOB==2,12) using the preconditioner
!              set up for the "old" one (supplied when IJOB==1).
!        These functionalities (set up a preconditioner and use it for another
!              matrix) are provided for the sake of generality but should be
!              used with care; in general, set up is fast with AGMG and hence
!              it is recommended to rerun it even if the matrix changes only
!              slightly.
!        In the more standard case where one needs successive solves with a
!        same matrix, it recommended to use IJOB==202 or 212 (in which case
!        the arrays A, JA, IA are not accessed) instead of IJOB==2 or 12
!        (in which case the input matrix has to be kept equal to the one
!        provided for set up). This  slightly faster and helps to save memory
!        since the arrays that store the input matrix may be deallocated after
!        set up.
!
! IPRINT    (input) INTEGER.
!                Indicates the unit number where information is to be printed
!                (N.B.: 5 is converted to 6). If nonpositive, only error
!                messages are printed on standard output. Warning messages about
!                insufficient convergence (in the allowed  maximum number of
!                iterations) are further suppressed supplying a negative number.
!
! NREST     (input) INTEGER.
!               Restart parameter for GCR (an implementation of GMRES)
!               Nonpositive values are converted to NREST=10 (default)
!
! !!   If NREST==1, Flexible CG is used instead of GCR (when IJOB==0,10,2, 12)
!               and also (IJOB==0,1) performs some simplification during
!               the set up based on the assumption that the matrix
!               supplied in A, JA, IA is symmetric.
!
! !!!  NREST=1 Should be used if and only if the matrix is really SYMMETRIC
! !!!           (and positive definite).
!
!  ITER     (input/output) INTEGER.
!           On input, the maximum number of iterations. Should be positive.
!           On output, actual number of iterations.
!             If this number of iteration was insufficient to meet convergence
!             criterion, ITER will be returned negative and equal to the
!             opposite of the number of iterations performed.
!           Significant only if IJOB==0, 2, 10, 12, 202, 212
!
!  TOL      (input) REAL (kind(0.0e0))
!           The tolerance on residual norm. Iterations are stopped whenever
!                || A*x-f || <= TOL* || f ||
!           Should be positive and less than 1.0
!           Significant only if IJOB==0, 2, 10, 12, 202, 212
!
! MPI_COMM_AGMG (input) INTEGER
!                MPI communicator
!
!
!!!!! Remark !!!! Except insufficient number of iterations to achieve
!                 convergence (characterized by a negative value returned
!                 in ITER), all other detected errors are fatal and lead
!                 to a STOP statement.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide, 3rd ed.*, SIAM, 1999.

[2] S. C. Eisenstat, H. C. Elman, and M. H. Schultz, *Variational iterative methods for nonsymmetric systems of linear equations*, SIAM J. Numer. Anal., 20 (1983), pp. 345–357.

[3] G. Karypis, *METIS software and documentation.* `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[4] *MUMPS software and documentation.* `http://graal.ens-lyon.fr/MUMPS/`.

[5] A. Napov and Y. Notay, *An algebraic multigrid method with guaranteed convergence rate*, SIAM J. Sci. Comput., 34 (2012), pp. A1079–A1109.

[6] A. Napov and Y. Notay, *Algebraic multigrid for moderate order finite elements*, SIAM J. Sci. Comput., 36 (2014), p. A1678A1707.

[7] Y. Notay, *An aggregation-based algebraic multigrid method*, Electron. Trans. Numer. Anal., 37 (2010), pp. 123–146.

[8] Y. Notay, *Aggregation-based algebraic multigrid for convection-diffusion equations*, SIAM J. Sci. Comput., 34 (2012), pp. A2288–A2316.

[9] Y. Notay, *Numerical comparison of solvers for linear systems from the discretization of scalar PDEs*, 2012. `http://agmg.eu/numcompsolv.pdf`.

[10] Y. Notay and A. Napov, *A massively parallel solver for discrete poisson-like problems*, J. Comput. Physics, 281 (2015), pp. 237–250.

[11] Y. Saad, *SPARSKIT: a basic tool kit for sparse matrix computations*, tech. rep., University of Minnesota, Minneapolis, 1994. `http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html`.

[12] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, PA, 2003. Second ed.

[13] H. A. Van der Vorst and C. Vuik, *GMRESR: a family of nested GMRES methods*, Numer. Linear Algebra Appl., 1 (1994), pp. 369–386.