MLD2P4
User's and Reference Guide

A guide for the MultiLevel Domain Decomposition Parallel Preconditioners Package based on PSBLAS

Pasqua D'Ambra
IAC-CNR, Naples, Italy

Daniela di Serafino
University of Campania "Luigi Vanvitelli", Caserta, Italy

Salvatore Filippone
Cranfield University, Cranfield, United Kingdom

Software version: 2.2
July 31, 2018

This page intentionally left blank

Abstract

MLD2P4 (MultiLevel Domain Decomposition Parallel Preconditioners Package based on PSBLAS) is a package of parallel algebraic multilevel preconditioners. The first release of MLD2P4 made available multilevel additive and hybrid Schwarz preconditioners, as well as one-level additive Schwarz preconditioners. The package has been extended to include further multilevel cycles and smoothers widely used in multigrid methods. In the multilevel case, a purely algebraic approach is applied to generate coarse-level corrections, so that no geometric background is needed concerning the matrix to be preconditioned. The matrix is assumed to be square, real or complex.

MLD2P4 has been designed to provide scalable and easy-to-use preconditioners in the context of the PSBLAS (Parallel Sparse Basic Linear Algebra Subprograms) computational framework and can be used in conjuction with the Krylov solvers available in this framework. MLD2P4 enables the user to easily specify different features of an algebraic multilevel preconditioner, thus allowing to search for the "best" preconditioner for the problem at hand.

The package employs object-oriented design techniques in Fortran 2003, with interfaces to additional third party libraries such as MUMPS, UMFPACK, SuperLU, and SuperLU_Dist, which can be exploited in building multilevel preconditioners. The parallel implementation is based on a Single Program Multiple Data (SPMD) paradigm; the inter-process communication is based on MPI and is managed mainly through PSBLAS.

This guide provides a brief description of the functionalities and the user interface of MLD2P4.

This page intentionally left blank

Contents

This page intentionally left blank

1 General Overview

The MultiLevel Domain Decomposition Parallel Preconditioners Package based on PSBLAS (MLD2P4) provides parallel Algebraic MultiGrid (AMG) and Domain Decomposition preconditioners (see, e.g., [3, 24, 22]), to be used in the iterative solution of linear systems,

$$Ax = b, \tag{1}$$

where A is a square, real or complex, sparse matrix. The name of the package comes from its original implementation, containing multilevel additive and hybrid Schwarz preconditioners, as well as one-level additive Schwarz preconditioners. The current version extends the original plan by including multilevel cycles and smoothers widely used in multigrid methods.

The multilevel preconditioners implemented in MLD2P4 are obtained by combining AMG cycles with smoothers and coarsest-level solvers. The V-, W-, and K-cycles [3, 20] are available, which allow to define almost all the preconditioners in the package, including the multilevel hybrid Schwarz ones; a specific cycle is implemented to obtain multilevel additive Schwarz preconditioners. The Jacobi, hybrid forward/backward Gauss-Seidel, block-Jacobi, and additive Schwarz methods are available as smoothers. An algebraic approach is used to generate a hierarchy of coarse-level matrices and operators, without explicitly using any information on the geometry of the original problem, e.g., the discretization of a PDE. To this end, the smoothed aggregation technique [2, 26] is applied. Either exact or approximate solvers can be used on the coarsest-level system. Specifically, different sparse LU factorizations from external packages, and native incomplete LU factorizations and Jacobi, hybrid Gauss-Seidel, and block-Jacobi solvers are available. All smoothers can be also exploited as one-level preconditioners.

MLD2P4 is written in Fortran 2003, following an object-oriented design through the exploitation of features such as abstract data type creation, type extension, functional overloading, and dynamic memory management. The parallel implementation is based on a Single Program Multiple Data (SPMD) paradigm. Single and double precision implementations of MLD2P4 are available for both the real and the complex case, which can be used through a single interface.

MLD2P4 has been designed to implement scalable and easy-to-use multilevel preconditioners in the context of the PSBLAS (Parallel Sparse BLAS) computational framework [15, 14]. PSBLAS provides basic linear algebra operators and data management facilities for distributed sparse matrices, as well as parallel Krylov solvers which can be used with the MLD2P4 preconditioners. The choice of PSBLAS has been mainly motivated by the need of having a portable and efficient software infrastructure implementing "de facto" standard parallel sparse linear algebra kernels, to pursue goals such as performance, portability, modularity ed extensibility in the development of the preconditioner package. On the other hand, the implementation of MLD2P4 has led to some revisions and extentions of the original PSBLAS kernels. The inter-process comunication required by MLD2P4 is encapsulated in the PSBLAS routines; therefore, MLD2P4 can be run on any parallel machine where PSBLAS implementations are

available.

MLD2P4 has a layered and modular software architecture where three main layers can be identified. The lower layer consists of the PSBLAS kernels, the middle one implements the construction and application phases of the preconditioners, and the upper one provides a uniform interface to all the preconditioners. This architecture allows for different levels of use of the package: few black-box routines at the upper layer allow all users to easily build and apply any preconditioner available in MLD2P4; facilities are also available allowing expert users to extend the set of smoothers and solvers for building new versions of the preconditioners (see Section 7).

We note that the user interface of MLD2P4 2.1 has been extended with respect to the previous versions in order to separate the construction of the multilevel hierarchy from the construction of the smoothers and solvers, and to allow for more flexibility at each level. The software architecture described in [8] has significantly evolved too, in order to fully exploit the Fortran 2003 features implemented in PSBLAS 3. However, compatibility with previous versions has been preserved.

This guide is organized as follows. General information on the distribution of the source code is reported in Section 2, while details on the configuration and installation of the package are given in Section 3. A short description of the preconditioners implemented in MLD2P4 is provided in Section 4, to help the users in choosing among them. The basics for building and applying the preconditioners with the Krylov solvers implemented in PSBLAS are reported in Section 5, where the Fortran codes of a few sample programs are also shown. A reference guide for the user interface routines is provided in Section 6. Information on the extension of the package through the addition of new smoothers and solvers is reported in Section 7. The error handling mechanism used by the package is briefly described in Section 8. The copyright terms concerning the distribution and modification of MLD2P4 are reported in Appendix A.

2 Code Distribution

MLD2P4 is available from the web site

`https://github.com/sfilippone/mld2p4-2`

where contact points for further information can be also found.

The software is available under a modified BSD license, as specified in Appendix A; please note that some of the optional third party libraries may be licensed under a different and more stringent license, most notably the GPL, and this should be taken into account when treating derived works.

The library defines a version string with the constant

`mld_version_string_`

whose current value is `2.1.0`.

Contributors

Contributors to version 2:

- Salvatore Filippone, Cranfield University, UK;
- Pasqua D'Ambra, IAC-CNR, Naples, IT;
- Daniela di Serafino, University of Campania "L. Vanvitelli", Caserta, IT;
- Ambra Abdullahi Hassan, University of Rome "Tor Vergata", IT.

Contributors to version 1:

- Salvatore Filippone;
- Pasqua D'Ambra;
- Daniela di Serafino;
- Alfredo Buttari, CNRS-IRIT, Toulouse, F.

3 Configuring and Building MLD2P4

In order to build MLD2P4 it is necessary to set up a Makefile with appropriate system-dependent variables; this is done by means of the `configure` script. The distribution also includes the autoconf and automake sources employed to generate the script, but usually this is not needed to build the software.

MLD2P4 is implemented almost entirely in Fortran 2003, with some interfaces to external libraries in C; the Fortran compiler must support the Fortran 2003 standard plus the extension `MOLD=` feature, which enhances the usability of `ALLOCATE`. Many compilers do this; in particular, this is supported by the GNU Fortran compiler, for which we recommend to use at least version 4.8. The software defines data types and interfaces for real and complex data, in both single and double precision.

Building MLD2P4 requires some base libraries (see Section 3.1); interfaces to optional third-party libraries, which extend the functionalities of MLD2P4 (see Section 3.2), are also available. Many Linux distributions (e.g., Ubuntu, Fedora, CentOS) provide precompiled packages for the prerequisite and optional software. In many cases these packages are split between a runtime part and a "developer" part; in order to build MLD2P4 you need both. A description of the base and optional software used by MLD2P4 is given in the next sections.

3.1 Prerequisites

The following base libraries are needed:

BLAS [11, 12, 18] Many vendors provide optimized versions of BLAS; if no vendor version is available for a given platform, the ATLAS software (`math-atlas.sourceforge.net`) may be employed. The reference BLAS from Netlib (`www.netlib.org/blas`) are meant to define the standard behaviour of the BLAS interface, so they are not optimized for any particular plaftorm, and should only be used as a last resort. Note that BLAS computations form a relatively small part of the MLD2P4/PSBLAS computations; they are however critical when using preconditioners based on MUMPS, UMFPACK or SuperLU third party libraries. Note that UMFPACK requires a full LAPACK library; our experience is that configuring ATLAS for building full LAPACK does not work in the correct way. Our advice is first to download the LAPACK tarfile from `www.netlib.org/lapack` and install it independently of ATLAS. In this case, you need to modify the OPTS and NOOPT definitions for including -fPIC compilation option in the make.inc file of the LAPACK library.

MPI [17, 23] A version of MPI is available on most high-performance computing systems.

PSBLAS [13, 15] Parallel Sparse BLAS (PSBLAS) is available from `github.com/sfilippone/psblas3`; version 3.5.0 (or later) is required. Indeed, all the prerequisites listed so far are also prerequisites of PSBLAS.

Please note that the four previous libraries must have Fortran interfaces compatible with MLD2P4; usually this means that they should all be built with the same compiler as MLD2P4.

3.2 Optional third party libraries

We provide interfaces to the following third-party software libraries; note that these are optional, but if you enable them some defaults for multilevel preconditioners may change to reflect their presence.

UMFPACK [9] A sparse LU factorization package included in the SuiteSparse library, available from `faculty.cse.tamu.edu/davis/suitesparse.html`; it provides sequential factorization and triangular system solution for double precision real and complex data. We tested version 4.5.4 of SuiteSparse. Note that for configuring SuiteSparse you should provide the right path to the BLAS and LAPACK libraries in the `SuiteSparse_config/SuiteSparse_config.mk` file.

MUMPS [1] A sparse LU factorization package available from `mumps.enseeiht.fr`; it provides sequential and parallel factorizations and triangular system solution for single and double precision, real and complex data. We tested versions 4.10.0 and 5.0.1.

SuperLU [10] A sparse LU factorization package available from `crd.lbl.gov/~xiaoye/SuperLU/`; it provides sequential factorization and triangular system solution for single and double precision, real and complex data. We tested versions 4.3 and 5.0. If you installed BLAS from ATLAS, remember to define the BLASLIB variable in the make.inc file.

SuperLU_Dist [19] A sparse LU factorization package available from the same site as SuperLU; it provides parallel factorization and triangular system solution for double precision real and complex data. We tested versions 3.3 and 4.2. If you installed BLAS from ATLAS, remember to define the BLASLIB variable in the make.inc file and to add the `-std=c99` option to the C compiler options. Note that this library requires the ParMETIS library for parallel graph partitioning and fill-reducing matrix ordering, available from `glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.

3.3 Configuration options

In order to build MLD2P4, the first step is to use the `configure` script in the main directory to generate the necessary makefile.

As a minimal example consider the following:

```
./configure --with-psblas=PSB-INSTALL-DIR
```

which assumes that the various MPI compilers and support libraries are available in the standard directories on the system, and specifies only the PSBLAS install directory (note that the latter directory must be specified with an *absolute* path). The full set of options may be looked at by issuing the command `./configure --help`, which produces:

```
'configure' configures MLD2P4 2.1.1 to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE.  See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:
  -h, --help              display this help and exit
      --help=short        display options specific to this package
      --help=recursive    display the short help of all the included packages
  -V, --version           display version information and exit
  -q, --quiet, --silent   do not print 'checking ...' messages
      --cache-file=FILE   cache test results in FILE [disabled]
  -C, --config-cache      alias for '--cache-file=config.cache'
  -n, --no-create         do not create output files
      --srcdir=DIR        find the sources in DIR [configure dir or '..']

Installation directories:
  --prefix=PREFIX         install architecture-independent files in PREFIX
                          [/usr/local]
  --exec-prefix=EPREFIX   install architecture-dependent files in EPREFIX
                          [PREFIX]

By default, 'make install' will install all the files in
'/usr/local/bin', '/usr/local/lib' etc.  You can specify
an installation prefix other than '/usr/local' using '--prefix',
for instance '--prefix=$HOME'.

For better control, use the options below.

Fine tuning of the installation directories:
  --bindir=DIR            user executables [EPREFIX/bin]
  --sbindir=DIR           system admin executables [EPREFIX/sbin]
  --libexecdir=DIR        program executables [EPREFIX/libexec]
  --sysconfdir=DIR        read-only single-machine data [PREFIX/etc]
```

```
  --sharedstatedir=DIR    modifiable architecture-independent data [PREFIX/com]
  --localstatedir=DIR     modifiable single-machine data [PREFIX/var]
  --libdir=DIR            object code libraries [EPREFIX/lib]
  --includedir=DIR        C header files [PREFIX/include]
  --oldincludedir=DIR     C header files for non-gcc [/usr/include]
  --datarootdir=DIR       read-only arch.-independent data root [PREFIX/share]
  --datadir=DIR           read-only architecture-independent data [DATAROOTDIR]
  --infodir=DIR           info documentation [DATAROOTDIR/info]
  --localedir=DIR         locale-dependent data [DATAROOTDIR/locale]
  --mandir=DIR            man documentation [DATAROOTDIR/man]
  --docdir=DIR            documentation root [DATAROOTDIR/doc/mld2p4]
  --htmldir=DIR           html documentation [DOCDIR]
  --dvidir=DIR            dvi documentation [DOCDIR]
  --pdfdir=DIR            pdf documentation [DOCDIR]
  --psdir=DIR             ps documentation [DOCDIR]

Program names:
  --program-prefix=PREFIX            prepend PREFIX to installed program names
  --program-suffix=SUFFIX            append SUFFIX to installed program names
  --program-transform-name=PROGRAM   run sed PROGRAM on installed program names

Optional Features:
  --disable-option-checking  ignore unrecognized --enable/--with options
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --enable-silent-rules   less verbose build output (undo: "make V=1")
  --disable-silent-rules  verbose build output (undo: "make V=0")
  --enable-dependency-tracking
                          do not reject slow dependency extractors
  --disable-dependency-tracking
                          speeds up one-time build
  --enable-serial         Specify whether to enable a fake mpi library to run
                          in serial mode.
  --enable-long-integers  Specify usage of 64 bits integers.

Optional Packages:
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=no)
  --with-psblas=DIR       The install directory for PSBLAS, for example,
                          --with-psblas=/opt/packages/psblas-3.5
  --with-psblas-incdir=DIR
                          Specify the directory for PSBLAS C includes.
  --with-psblas-moddir=DIR
                          Specify the directory for PSBLAS Fortran modules.
```

```
--with-psblas-libdir=DIR
                        Specify the directory for PSBLAS library.
--with-ccopt            additional [CCOPT] flags to be added: will prepend
                        to [CCOPT]
--with-fcopt            additional [FCOPT] flags to be added: will prepend
                        to [FCOPT]
--with-libs             List additional link flags here. For example,
                        --with-libs=-lspecial_system_lib or
                        --with-libs=-L/path/to/libs
--with-clibs            additional [CLIBS] flags to be added: will prepend
                        to [CLIBS]
--with-flibs            additional [FLIBS] flags to be added: will prepend
                        to [FLIBS]
--with-library-path     additional [LIBRARYPATH] flags to be added: will
                        prepend to [LIBRARYPATH]
--with-include-path     additional [INCLUDEPATH] flags to be added: will
                        prepend to [INCLUDEPATH]
--with-module-path      additional [MODULE_PATH] flags to be added: will
                        prepend to [MODULE_PATH]
--with-extra-libs       List additional link flags here. For example,
                        --with-extra-libs=-lspecial_system_lib or
                        --with-extra-libs=-L/path/to/libs
--with-blas=<lib>       use BLAS library <lib>
--with-blasdir=<dir>    search for BLAS library in <dir>
--with-lapack=<lib>     use LAPACK library <lib>
--with-mumps=LIBNAME    Specify the libname for MUMPS. Default: autodetect
                        with minimum "-lmumps_common -lpord"
--with-mumpsdir=DIR     Specify the directory for MUMPS library and
                        includes. Note: you will need to add auxiliary
                        libraries with --extra-libs; this depends on how
                        MUMPS was configured and installed, at a minimum you
                        will need SCALAPACK and BLAS
--with-mumpsincdir=DIR  Specify the directory for MUMPS includes.
--with-mumpsmoddir=DIR  Specify the directory for MUMPS Fortran modules.
--with-mumpslibdir=DIR  Specify the directory for MUMPS library.
--with-umfpack=LIBNAME  Specify the library name for UMFPACK and its support
                        libraries. Default: "-lumfpack -lamd"
--with-umfpackdir=DIR   Specify the directory for UMFPACK library and
                        includes.
--with-umfpackincdir=DIR
                        Specify the directory for UMFPACK includes.
--with-umfpacklibdir=DIR
                        Specify the directory for UMFPACK library.
--with-superlu=LIBNAME  Specify the library name for SUPERLU library.
```

```
                          Default: "-lsuperlu"
  --with-superludir=DIR   Specify the directory for SUPERLU library and
                          includes.
  --with-superluincdir=DIR
                          Specify the directory for SUPERLU includes.
  --with-superlulibdir=DIR
                          Specify the directory for SUPERLU library.
  --with-superludist=LIBNAME
                          Specify the libname for SUPERLUDIST library.
                          Requires you also specify SuperLU. Default:
                          "-lsuperlu_dist"
  --with-superludistdir=DIR
                          Specify the directory for SUPERLUDIST library and
                          includes.
  --with-superludistincdir=DIR
                          Specify the directory for SUPERLUDIST includes.
  --with-superludistlibdir=DIR
                          Specify the directory for SUPERLUDIST library.

Some influential environment variables:
  FC          Fortran compiler command
  FCFLAGS     Fortran compiler flags
  LDFLAGS     linker flags, e.g. -L<lib dir> if you have libraries in a
              nonstandard directory <lib dir>
  LIBS        libraries to pass to the linker, e.g. -l<library>
  CC          C compiler command
  CFLAGS      C compiler flags
  CPPFLAGS    (Objective) C/C++ preprocessor flags, e.g. -I<include dir> if
              you have headers in a nonstandard directory <include dir>
  MPICC       MPI C compiler command
  MPIFC       MPI Fortran compiler command
  CPP         C preprocessor

Use these variables to override the choices made by ‘configure’ or to help
it to find libraries and programs with nonstandard names/locations.

Report bugs to <https://github.com/sfilippone/mld2p4-2/issues>.
```

For instance, if a user has built and installed PSBLAS 3.5 under the `/opt` directory and is using the SuiteSparse package (which includes UMFPACK), then MLD2P4 might be configured with:

```
 ./configure --with-psblas=/opt/psblas-3.5/ \
 --with-umfpackincdir=/usr/include/suitesparse/
```

Once the configure script has completed execution, it will have generated the file `Make.inc` which will then be used by all Makefiles in the directory tree; this file will be copied in the install directory under the name `Make.inc.MLD2P4`.

To use the MUMPS solver package, the user has to add the appropriate options to the configure script; by default we are looking for the libraries `-ldmumps -lsmumps -lzmumps -lcmumps -mumps_common -lpord`. MUMPS often uses additional packages such as ScaLAPACK, ParMETIS, SCOTCH, as well as enabling OpenMP; in such cases it is necessary to add linker options with the `--with-extra-libs` configure option.

To build the library the user will now enter

```
make
```

followed (optionally) by

```
make install
```

3.4 Bug reporting

If you find any bugs in our codes, please report them through our issues page on

https://github.com/sfilippone/mld2p4-2/issues

To enable us to track the bug, please provide a log from the failing application, the test conditions, and ideally a self-contained test program reproducing the issue.

3.5 Example and test programs

The package contains the `examples` and `tests` directories; both of them are further divided into `fileread` and `pdegen` subdirectories. Their purpose is as follows:

`examples` contains a set of simple example programs with a predefined choice of preconditioners, selectable via integer values. These are intended to get an acquaintance with the multilevel preconditioners available in MLD2P4.

`tests` contains a set of more sophisticated examples that will allow the user, via the input files in the `runs` subdirectories, to experiment with the full range of preconditioners implemented in the package.

The `fileread` directories contain sample programs that read sparse matrices from files, according to the Matrix Market or the Harwell-Boeing storage format; the `pdegen` programs generate matrices in full parallel mode from the discretization of a sample partial differential equation.

4 Multigrid Background

Multigrid preconditioners, coupled with Krylov iterative solvers, are widely used in the parallel solution of large and sparse linear systems, because of their optimality in the solution of linear systems arising from the discretization of scalar elliptic Partial Differential Equations (PDEs) on regular grids. Optimality, also known as algorithmic scalability, is the property of having a computational cost per iteration that depends linearly on the problem size, and a convergence rate that is independent of the problem size.

Multigrid preconditioners are based on a recursive application of a two-grid process consisting of smoother iterations and a coarse-space (or coarse-level) correction. The smoothers may be either basic iterative methods, such as the Jacobi and Gauss-Seidel ones, or more complex subspace-correction methods, such as the Schwarz ones. The coarse-space correction consists of solving, in an appropriately chosen coarse space, the residual equation associated with the approximate solution computed by the smoother, and of using the solution of this equation to correct the previous approximation. The transfer of information between the original (fine) space and the coarse one is performed by using suitable restriction and prolongation operators. The construction of the coarse space and the corresponding transfer operators is carried out by applying a so-called coarsening algorithm to the system matrix. Two main approaches can be used to perform coarsening: the geometric approach, which exploits the knowledge of some physical grid associated with the matrix and requires the user to define transfer operators from the fine to the coarse level and vice versa, and the algebraic approach, which builds the coarse-space correction and the associate transfer operators using only matrix information. The first approach may be difficult when the system comes from discretizations on complex geometries; furthermore, ad hoc one-level smoothers may be required to get an efficient interplay between fine and coarse levels, e.g., when matrices with highly varying coefficients are considered. The second approach performs a fully automatic coarsening and enforces the interplay between fine and coarse level by suitably choosing the coarse space and the coarse-to-fine interpolation (see, e.g., [3, 24, 22] for details.)

MLD2P4 uses a pure algebraic approach, based on the smoothed aggregation algorithm [2, 26], for building the sequence of coarse matrices and transfer operators, starting from the original one. A decoupled version of this algorithm is implemented, where the smoothed aggregation is applied locally to each submatrix [25]. A brief description of the AMG preconditioners implemented in MLD2P4 is given in Sections 4.1-4.3. For further details the reader is referred to [4, 5, 7, 8].

We note that optimal multigrid preconditioners do not necessarily correspond to minimum execution times in a parallel setting. Indeed, to obtain effective parallel multigrid preconditioners, a tradeoff between the optimality and the cost of building and applying the smoothers and the coarse-space corrections must be achieved. Effective parallel preconditioners require algorithmic scalability to be coupled with implementation scalability, i.e., a computational cost per iteration which remains (almost) constant as the number of parallel processors increases.

4.1 AMG preconditioners

In order to describe the AMG preconditioners available in MLD2P4, we consider a linear system

$$Ax = b, \tag{2}$$

where $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ is a nonsingular sparse matrix; for ease of presentation we assume A has a symmetric sparsity pattern.

Let us consider as finest index space the set of row (column) indices of A, i.e., $\Omega = \{1, 2, \ldots, n\}$. Any algebraic multilevel preconditioners implemented in MLD2P4 generates a hierarchy of index spaces and a corresponding hierarchy of matrices,

$$\Omega^1 \equiv \Omega \supset \Omega^2 \supset \ldots \supset \Omega^{nlev}, \quad A^1 \equiv A, A^2, \ldots, A^{nlev},$$

by using the information contained in A, without assuming any knowledge of the geometry of the problem from which A originates. A vector space $\mathbb{R}^{n_k}$ is associated with Ω^k, where n_k is the size of Ω^k. For all $k < nlev$, a restriction operator and a prolongation one are built, which connect two levels k and $k+1$:

$$P^k \in \mathbb{R}^{n_k \times n_{k+1}}, \quad R^k \in \mathbb{R}^{n_{k+1} \times n_k};$$

the matrix A^{k+1} is computed by using the previous operators according to the Galerkin approach, i.e.,

$$A^{k+1} = R^k A^k P^k.$$

In the current implementation of MLD2P4 we have $R^k = (P^k)^T$ A smoother with iteration matrix M^k is set up at each level $k < nlev$, and a solver is set up at the coarsest level, so that they are ready for application (for example, setting up a solver based on the LU factorization means computing and storing the L and U factors). The construction of the hierarchy of AMG components described so far corresponds to the so-called build phase of the preconditioner.

The components produced in the build phase may be combined in several ways to obtain different multilevel preconditioners; this is done in the application phase, i.e., in the computation of a vector of type $w = B^{-1}v$, where B denotes the preconditioner, usually within an iteration of a Krylov solver [21]. An example of such a combination, known as V-cycle, is given in Figure 1. In this case, a single iteration of the same smoother is used before and after the the recursive call to the V-cycle (i.e., in the pre-smoothing and post-smoothing phases); however, different choices can be performed. Other cycles can be defined; in MLD2P4, we implemented the standard V-cycle and W-cycle [3], and a version of the K-cycle described in [20].

4.2 Smoothed Aggregation

In order to define the prolongator P^k, used to compute the coarse-level matrix A^{k+1}, MLD2P4 uses the smoothed aggregation algorithm described in [2, 26]. The basic idea of this algorithm is to build a coarse set of indices Ω^{k+1} by suitably grouping the indices of Ω^k into disjoint subsets (aggregates), and to define the coarse-to-fine space transfer

```
procedure V-cycle(k, A^k, b^k, u^k)
  if (k ≠ nlev) then
    u^k = u^k + M^k (b^k − A^k u^k)
    b^{k+1} = R^{k+1} (b^k − A^k u^k)
    u^{k+1} = V-cycle(k + 1, A^{k+1}, b^{k+1}, 0)
    u^k = u^k + P^{k+1} u^{k+1}
    u^k = u^k + M^k (b^k − A^k u^k)
  else
    u^k = (A^k)^{-1} b^k
  endif
  return u^k
end
```

Figure 1: Application phase of a V-cycle preconditioner.

operator P^k by applying a suitable smoother to a simple piecewise constant prolongation operator, with the aim of improving the quality of the coarse-space correction.

Three main steps can be identified in the smoothed aggregation procedure:

1. aggregation of the indices of Ω^k to obtain Ω^{k+1};
2. construction of the prolongator P^k;
3. application of P^k and $R^k = (P^k)^T$ to build A^{k+1}.

In order to perform the coarsening step, the smoothed aggregation algorithm described in [26] is used. In this algorithm, each index $j \in \Omega^{k+1}$ corresponds to an aggregate Ω_j^k of Ω^k, consisting of a suitably chosen index $i \in \Omega^k$ and indices that are (usually) contained in a strongly-coupled neighborood of i, i.e.,

$$\Omega_j^k \subset \mathcal{N}_i^k(\theta) = \left\{ r \in \Omega^k : |a_{ir}^k| > \theta \sqrt{|a_{ii}^k a_{rr}^k|} \right\} \cup \{i\}\,, \tag{3}$$

for a given threshold $\theta \in [0, 1]$ (see [26] for the details). Since this algorithm has a sequential nature, a decoupled version of it is applied, where each processor independently executes the algorithm on the set of indices assigned to it in the initial data distribution. This version is embarrassingly parallel, since it does not require any data communication. On the other hand, it may produce some nonuniform aggregates and is strongly dependent on the number of processors and on the initial partitioning of the matrix A. Nevertheless, this parallel algorithm has been chosen for MLD2P4, since it has been shown to produce good results in practice [5, 7, 25].

The prolongator P^k is built starting from a tentative prolongator $\bar{P}^k \in \mathbb{R}^{n_k \times n_{k+1}}$, defined as

$$\bar{P}^k = (\bar{p}_{ij}^k), \quad \bar{p}_{ij}^k = \begin{cases} 1 & \text{if } i \in \Omega_j^k, \\ 0 & \text{otherwise,} \end{cases} \tag{4}$$

where Ω_j^k is the aggregate of Ω^k corresponding to the index $j \in \Omega^{k+1}$. P^k is obtained by applying to $\bar{P}^k$ a smoother $S^k \in \mathbb{R}^{n_k \times n_k}$:

$$P^k = S^k \bar{P}^k,$$

in order to remove nonsmooth components from the range of the prolongator, and hence to improve the convergence properties of the multilevel method [2, 24]. A simple choice for S^k is the damped Jacobi smoother:

$$S^k = I - \omega^k (D^k)^{-1} A_F^k,$$

where D^k is the diagonal matrix with the same diagonal entries as A^k, $A_F^k = (\bar{a}_{ij}^k)$ is the filtered matrix defined as

$$\bar{a}_{ij}^k = \begin{cases} a_{ij}^k & \text{if } j \in \mathcal{N}_i^k(\theta), \\ 0 & \text{otherwise,} \end{cases} \quad (j \neq i), \qquad \bar{a}_{ii}^k = a_{ii}^k - \sum_{j \neq i} (a_{ij}^k - \bar{a}_{ij}^k), \tag{5}$$

and ω^k is an approximation of $4/(3\rho^k)$, where ρ^k is the spectral radius of $(D^k)^{-1} A_F^k$ [2]. In MLD2P4 this approximation is obtained by using $\|A_F^k\|_\infty$ as an estimate of ρ^k. Note that for systems coming from uniformly elliptic problems, filtering the matrix A^k has little or no effect, and A^k can be used instead of A_F^k. The latter choice is the default in MLD2P4.

4.3 Smoothers and coarsest-level solvers

The smoothers implemented in MLD2P4 include the Jacobi and block-Jacobi methods, a hybrid version of the forward and backward Gauss-Seidel methods, and the additive Schwarz (AS) ones (see, e.g., [21, 22]).

The hybrid Gauss-Seidel version is considered because the original Gauss-Seidel method is inherently sequential. At each iteration of the hybrid version, each parallel process uses the most recent values of its own local variables and the values of the non-local variables computed at the previous iteration, obtained by exchanging data with other processes before the beginning of the current iteration.

In the AS methods, the index space Ω^k is divided into m_k subsets Ω_i^k of size $n_{k,i}$, possibly overlapping. For each i we consider the restriction operator $R_i^k \in \mathbb{R}^{n_{k,i} \times n_k}$ that maps a vector x^k to the vector x_i^k made of the components of x^k with indices in Ω_i^k, and the prolongation operator $P_i^k = (R_i^k)^T$. These operators are then used to build $A_i^k = R_i^k A^k P_i^k$, which is the restriction of A^k to the index space Ω_i^k. The classical AS preconditioner M_{AS}^k is defined as

$$(M_{AS}^k)^{-1} = \sum_{i=1}^{m_k} P_i^k (A_i^k)^{-1} R_i^k,$$

where A_i^k is supposed to be nonsingular. We observe that an approximate inverse of A_i^k is usually considered instead of $(A_i^k)^{-1}$. The setup of M_{AS}^k during the multilevel build phase involves

- the definition of the index subspaces Ω_i^k and of the corresponding operators R_i^k (and P_i^k);
- the computation of the submatrices A_i^k;
- the computation of their inverses (usually approximated through some form of incomplete factorization).

The computation of $z^k = M_{AS}^k w^k$, with $w^k \in \mathbb{R}^{n_k}$, during the multilevel application phase, requires

- the restriction of w^k to the subspaces $\mathbb{R}^{n_{k,i}}$, i.e. $w_i^k = R_i^k w^k$;
- the computation of the vectors $z_i^k = (A_i^k)^{-1} w_i^k$;
- the prolongation and the sum of the previous vectors, i.e. $z^k = \sum_{i=1}^{m_k} P_i^k z_i^k$.

Variants of the classical AS method, which use modifications of the restriction and prolongation operators, are also implemented in MLD2P4. Among them, the Restricted AS (RAS) preconditioner usually outperforms the classical AS preconditioner in terms of convergence rate and of computation and communication time on parallel distributed-memory computers, and is therefore the most widely used among the AS preconditioners [6].

Direct solvers based on sparse LU factorizations, implemented in the third-party libraries reported in Section 3.2, can be applied as coarsest-level solvers by MLD2P4. Native inexact solvers based on incomplete LU factorizations, as well as Jacobi, hybrid (forward) Gauss-Seidel, and block Jacobi preconditioners are also available. Direct solvers usually lead to more effective preconditioners in terms of algorithmic scalability; however, this does not guarantee parallel efficiency.

5 Getting Started

We describe the basics for building and applying MLD2P4 one-level and multilevel (i.e., AMG) preconditioners with the Krylov solvers included in PSBLAS [13]. The following steps are required:

1. *Declare the preconditioner data structure.* It is a derived data type, `mld_`x`prec_type`, where x may be `s`, `d`, `c` or `z`, according to the basic data type of the sparse matrix (`s` = real single precision; `d` = real double precision; `c` = complex single precision; `z` = complex double precision). This data structure is accessed by the user only through the MLD2P4 routines, following an object-oriented approach.

2. *Allocate and initialize the preconditioner data structure, according to a preconditioner type chosen by the user.* This is performed by the routine `init`, which also sets defaults for each preconditioner type selected by the user. The preconditioner types and the defaults associated with them are given in Table 1, where the strings used by `init` to identify the preconditioner types are also given. Note that these strings are valid also if uppercase letters are substituted by corresponding lowercase ones.

3. *Modify the selected preconditioner type, by properly setting preconditioner parameters.* This is performed by the routine `set`. This routine must be called only if the user wants to modify the default values of the parameters associated with the selected preconditioner type, to obtain a variant of that preconditioner. Examples of use of `set` are given in Section 5.1; a complete list of all the preconditioner parameters and their allowed and default values is provided in Section 6, Tables 2-8.

4. *Build the preconditioner for a given matrix.* If the selected preconditioner is multilevel, then two steps must be performed, as specified next.

   4.1 *Build the aggregation hierarchy for a given matrix.* This is performed by the routine `hierarchy_build`.

   4.2 *Build the preconditioner for a given matrix.* This is performed by the routine `smoothers_build`.

   If the selected preconditioner is one-level, it is built in a single step, performed by the routine `bld`.

5. *Apply the preconditioner at each iteration of a Krylov solver.* This is performed by the method `apply`. When using the PSBLAS Krylov solvers, this step is completely transparent to the user, since `apply` is called by the PSBLAS routine implementing the Krylov solver (`psb_krylov`).

6. *Free the preconditioner data structure.* This is performed by the routine `free`. This step is complementary to step 1 and should be performed when the preconditioner is no more used.

All the previous routines are available as methods of the preconditioner object. A detailed description of them is given in Section 6. Examples showing the basic use of MLD2P4 are reported in Section 5.1.

TYPE	STRING	DEFAULT PRECONDITIONER
No preconditioner	`'NONE'`	Considered to use the PSBLAS Krylov solvers with no preconditioner.
Diagonal	`'DIAG'` or `'JACOBI'`	Diagonal preconditioner. For any zero diagonal entry of the matrix to be preconditioned, the corresponding entry of the preconditioner is set to 1.
Gauss-Seidel	`'GS'`	Hybrid Gauss-Seidel (forward), that is, global block Jacobi with Gauss-Seidel as local solver.
Symmetrized Gauss-Seidel	`'FBGS'`	Symmetrized hybrid Gauss-Seidel,that is, forward Gauss-Seidel followed by backward Gauss-Seidel.
Block Jacobi	`'BJAC'`	Block-Jacobi with ILU(0) on the local blocks.
Additive Schwarz	`'AS'`	Additive Schwarz (AS), with overlap 1 and ILU(0) on the local blocks.
Multilevel	`'ML'`	V-cycle with one hybrid forward Gauss-Seidel (GS) sweep as pre-smoother and one hybrid backward GS sweep as post-smoother, basic smoothed aggregation as coarsening algorithm, and LU (plus triangular solve) as coarsest-level solver. See the default values in Tables 2-8 for further details of the preconditioner.

Table 1: Preconditioner types, corresponding strings and default choices.

Note that the module `mld_prec_mod`, containing the definition of the preconditioner data type and the interfaces to the routines of MLD2P4, must be used in any program calling such routines. The modules `psb_base_mod`, for the sparse matrix and communication descriptor data types, and `psb_krylov_mod`, for interfacing with the Krylov solvers, must be also used (see Section 5.1).

Remark 1. Coarsest-level solvers based on the LU factorization, such as those implemented in UMFPACK, MUMPS, SuperLU, and SuperLU_Dist, usually lead to smaller numbers of preconditioned Krylov iterations than inexact solvers, when the linear system comes from a standard discretization of basic scalar elliptic PDE problems. However, this does not necessarily correspond to the smallest execution time on parallel computers.

5.1 Examples

The code reported in Figure 2 shows how to set and apply the default multilevel preconditioner available in the real double precision version of MLD2P4 (see Table 1). This preconditioner is chosen by simply specifying `'ML'` as the second argument of `P%init` (a call to `P%set` is not needed) and is applied with the CG solver provided by PSBLAS (the matrix of the system to be solved is assumed to be positive definite). As previously observed, the modules `psb_base_mod`, `mld_prec_mod` and `psb_krylov_mod` must be used by the example program.

The part of the code concerning the reading and assembling of the sparse matrix and the right-hand side vector, performed through the PSBLAS routines for sparse matrix and vector management, is not reported here for brevity; the statements concerning the deallocation of the PSBLAS data structure are neglected too. The complete code can be found in the example program file `mld_dexample_ml.f90`, in the directory `examples/fileread` of the MLD2P4 implementation (see Section 3.5). A sample test problem along with the relevant input data is available in `examples/fileread/runs`. For details on the use of the PSBLAS routines, see the PSBLAS User's Guide [13].

The setup and application of the default multilevel preconditioner for the real single precision and the complex, single and double precision, versions are obtained with straightforward modifications of the previous example (see Section 6 for details). If these versions are installed, the corresponding codes are available in `examples/fileread/`.

Different versions of the multilevel preconditioner can be obtained by changing the default values of the preconditioner parameters. The code reported in Figure 3 shows how to set a V-cycle preconditioner which applies 1 block-Jacobi sweep as pre- and post-smoother, and solves the coarsest-level system with 8 block-Jacobi sweeps. Note that the ILU(0) factorization (plus triangular solve) is used as local solver for the block-Jacobi sweeps, since this is the default associated with block-Jacobi and set by `P%init`. Furthermore, specifying block-Jacobi as coarsest-level solver implies that the coarsest-level matrix is distributed among the processes. Figure 4 shows how to set a W-cycle preconditioner which applies 2 hybrid Gauss-Seidel sweeps as pre- and post-smoother, and solves the coarsest-level system with the multifrontal LU factorization implemented in MUMPS. It is specified that the coarsest-level matrix is distributed, since MUMPS can be used on both replicated and distributed matrices, and by default it is used on replicated ones. The code fragments shown in Figures 3 and 4 are included in the example program file `mld_dexample_ml.f90` too.

Finally, Figure 5 shows the setup of a one-level additive Schwarz preconditioner, i.e., RAS with overlap 2. Note also that a Krylov method different from CG must be used to solve the preconditioned system, since the preconditione in nonsymmetric. The corresponding example program is available in the file `mld_dexample_1lev.f90`.

For all the previous preconditioners, example programs where the sparse matrix and the right-hand side are generated by discretizing a PDE with Dirichlet boundary conditions are also available in the directory `examples/pdegen`.

```
  use psb_base_mod
  use mld_prec_mod
  use psb_krylov_mod
... ...
!
! sparse matrix
  type(psb_dspmat_type) :: A
! sparse matrix descriptor
  type(psb_desc_type)   :: desc_A
! preconditioner
  type(mld_dprec_type)  :: P
! right-hand side and solution vectors
  type(psb_d_vect_type) :: b, x
... ...
!
! initialize the parallel environment
  call psb_init(ictxt)
  call psb_info(ictxt,iam,np)
... ...
!
! read and assemble the spd matrix A and the right-hand side b
! using PSBLAS routines for sparse matrix / vector management
... ...
!
! initialize the default multilevel preconditioner, i.e. V-cycle
! with basic smoothed aggregation, 1 hybrid forward/backward
! GS sweep as pre/post-smoother and UMFPACK as coarsest-level
! solver
  call P%init('ML',info)
!
! build the preconditioner
  call P%hierarchy_build(A,desc_A,info)
  call P%smoothers_build(A,desc_A,info)

!
! set the solver parameters and the initial guess
  ... ...
!
! solve Ax=b with preconditioned CG
  call psb_krylov('CG',A,P,b,x,tol,desc_A,info)
  ... ...
!
! deallocate the preconditioner
  call P%free(info)
!
! deallocate other data structures
  ... ...
!
! exit the parallel environment
  call psb_exit(ictxt)
  stop
```

Figure 2: setup and application of the default multilevel preconditioner (example 1).

```
... ...
! build a V-cycle preconditioner with 1 block-Jacobi sweep (with
! ILU(0) on the blocks) as pre- and post-smoother, and 8  block-Jacobi
! sweeps (with ILU(0) on the blocks) as coarsest-level solver
  call P%init('ML',info)
  call_P%set('SMOOTHER_TYPE','BJAC',info)
  call P%set('COARSE_SOLVE','BJAC',info)
  call P%set('COARSE_SWEEPS',8,info)
  call P%hierarchy_build(A,desc_A,info)
  call P%smoothers_build(A,desc_A,info)
... ...
```

Figure 3: setup of a multilevel preconditioner

```
... ...
! build a W-cycle preconditioner with 2 hybrid Gauss-Seidel sweeps
! as pre- and post-smoother, a distributed coarsest
! matrix, and MUMPS as coarsest-level solver
  call P%init('ML',info)
  call P%set('ML_CYCLE','WCYCLE',info)
  call P%set('SMOOTHER_TYPE','FBGS',info)
  call P%set('SMOOTHER_SWEEPS',2,info)
  call P%set('COARSE_SOLVE','MUMPS',info)
  call P%set('COARSE_MAT','DIST',info)
  call P%hierarchy_build(A,desc_A,info)
  call P%smoothers_build(A,desc_A,info)
... ...
```

Figure 4: setup of a multilevel preconditioner

```
... ...
! set RAS with overlap 2 and ILU(0) on the local blocks
  call P%init('AS',info)
  call P%set('SUB_OVR',2,info)
  call P%bld(A,desc_A,info)
... ...
! solve Ax=b with preconditioned BiCGSTAB
  call psb_krylov('BICGSTAB',A,P,b,x,tol,desc_A,info)
```

Figure 5: setup of a one-level Schwarz preconditioner.

6 User Interface

The basic user interface of MLD2P4 consists of eight methods. The six methods `init`, `set`, `build`, `hierarchy_build`, `smoothers_build` and `apply` encapsulate all the functionalities for the setup and the application of any multilevel and one-level preconditioner implemented in the package. The method `free` deallocates the preconditioner data structure, while `descr` prints a description of the preconditioner setup by the user. For backward compatibility, methods are also accessible as stand-alone subroutines.

For each method, the same user interface is overloaded with respect to the real/complex case and the single/double precision; arguments with appropriate data types must be passed to the method, i.e.,

- the sparse matrix data structure, containing the matrix to be preconditioned, must be of type `psb_`x`spmat_type` with x = `s` for real single precision, x = `d` for real double precision, x = `c` for complex single precision, x = `z` for complex double precision;
- the preconditioner data structure must be of type `mld_`x`prec_type`, with x = `s`, `d`, `c`, `z`, according to the sparse matrix data structure;
- the arrays containing the vectors v and w involved in the preconditioner application $w = B^{-1}v$ must be of type `psb_`x`vect_type` with x = `s`, `d`, `c`, `z`, in a manner completely analogous to the sparse matrix type;
- real parameters defining the preconditioner must be declared according to the precision of the sparse matrix and preconditioner data structures (see Section 6.2).

A description of each method is given in the remainder of this section.

6.1 Method init

```
call p%init(ptype,info)
```

This method allocates and initializes the preconditioner `p`, according to the preconditioner type chosen by the user.

Arguments

`ptype` `character(len=*), intent(in)`.
The type of preconditioner. Its values are specified in Table 1.
Note that the strings are case insensitive.

`info` `integer, intent(out)`.
Error code. If no error, 0 is returned. See Section 8 for details.

For compatibility with the previous versions of MLD2P4, this method can be also invoked as follows:

```
call mld_precinit(p,ptype,info)
```

6.2 Method set

```
call p%set(what,val,info [,ilev, ilmax, pos, idx])
```

This method sets the parameters defining the preconditioner `p`. More precisely, the parameter identified by `what` is assigned the value contained in `val`.

Arguments

`what` `character(len=*)`.
The parameter to be set. It can be specified through its name; the string is case-insensitive. See Tables 2-8.

`val` `integer` *or* `character(len=*)` *or* `real(psb_spk_)` *or* `real(psb_dpk_)`, `intent(in)`.
The value of the parameter to be set. The list of allowed values and the corresponding data types is given in Tables 2-8. When the value is of type `character(len=*)`, it is also treated as case insensitive.

`info` `integer, intent(out)`.
Error code. If no error, 0 is returned. See Section 8 for details.

`ilev` `integer, optional, intent(in)`.
For the multilevel preconditioner, the level at which the preconditioner parameter has to be set. The levels are numbered in increasing order starting from the finest one, i.e., level 1 is the finest level. If `ilev` is not present, the parameter identified by `what` is set at all the appropriate levels (see Tables 2-8).

`ilmax` `integer, optional, intent(in)`.
For the multilevel preconditioner, when both `ilev` and `ilmax` are present, the settings are applied at all levels `ilev:ilmax`. When `ilev` is present but `ilmax` is not, then the default is `ilmax=ilev`. The levels are numbered in increasing order starting from the finest one, i.e., level 1 is the finest level.

`pos` `charater(len=*), optional, intent(in)`.
Whether the other arguments apply only to the pre-smoother (`'PRE'`) or to the post-smoother (`'POST'`). If `pos` is not present, the other arguments are applied to both smoothers. If the preconditioner is one-level or the parameter identified by `what` does not concern the smoothers, `pos` is ignored.

`idx` `integer, optional, intent(in)`.
An auxiliary input argument that can be passed to the underlying objects.

For compatibility with the previous versions of MLD2P4, this method can be also invoked as follows:

```
call mld_precset(p,what,val,info)
```

However, in this case the optional arguments `ilev`, `ilmax`, `pos` and `idx` cannot be used.

A variety of preconditioners can be obtained by a suitable setting of the preconditioner parameters. These parameters can be logically divided into four groups, i.e., parameters defining

1. the type of multilevel cycle and how many cycles must be applied;
2. the aggregation algorithm;
3. the coarse-space correction at the coarsest level (for multilevel preconditioners only);
4. the smoother of the multilevel preconditioners, or the one-level preconditioner.

A list of the parameters that can be set, along with their allowed and default values, is given in Tables 2-8. For a description of the meaning of the parameters, please refer also to Section 4.

Remark 2. A smoother is usually obtained by combining two objects: a smoother (`SMOOTHER_TYPE`) and a local solver (`SUB_SOLVE`), as specified in Tables 7-8. For example, the block-Jacobi smoother using ILU(0) on the blocks is obtained by combining the block-Jacobi smoother object with the ILU(0) solver object. Similarly, the hybrid Gauss-Seidel smoother (see Note in Table 7) is obtained by combining the block-Jacobi smoother object with a single sweep of the Gauss-Seidel solver object, while the point-Jacobi smoother is the result of combining the block-Jacobi smoother object with a single sweep of the pointwise-Jacobi solver object. However, for simplicity, shortcuts are provided to set point-Jacobi, hybrid (forward) Gauss-Seidel, and hybrid backward Gauss-Seidel, i.e., the previous smoothers can be defined by setting only `SMOOTHER_TYPE` to appropriate values (see Tables 7), i.e., without setting `SUB_SOLVE` too.

The smoother and solver objects are arranged in a hierarchical manner. When specifying a smoother object, its parameters, including the local solver, are set to their default values, and when a solver object is specified, its defaults are also set, overriding in both cases any previous settings even if explicitly specified. Therefore if the user sets a smoother, and wishes to use a solver different from the default one, the call to set the solver must come *after* the call to set the smoother.

Similar considerations apply to the point-Jacobi, Gauss-Seidel and block-Jacobi coarsest-level solvers, and shortcuts are available in this case too (see Table 5).

Remark 3. In general, a coarsest-level solver cannot be used with both the replicated and distributed coarsest-matrix layout; therefore, setting the solver after the layout may change the layout. Similarly, setting the layout after the solver may change the solver.

More precisely, UMFPACK and SuperLU require the coarsest-level matrix to be replicated, while SuperLU_Dist requires it to be distributed. In these cases, setting the coarsest-level solver implies that the layout is redefined according to the solver,

ovverriding any previous settings. MUMPS, point-Jacobi, hybrid Gauss-Seidel and block-Jacobi can be applied to replicated and distributed matrices, thus their choice does not modify any previously specified layout. It is worth noting that, when the matrix is replicated, the point-Jacobi, hybrid Gauss-Seidel and block-Jacobi solvers reduce to the corresponding local solver objects (see Remark 2). For the point-Jacobi and Gauss-Seidel solvers, these objects correspond to a *single* point-Jacobi sweep and a *single* Gauss-Seidel sweep, respectively, which are very poor solvers.

On the other hand, the distributed layout can be used with any solver but UMFPACK and SuperLU; therefore, if any of these two solvers has already been selected, the coarsest-level solver is changed to block-Jacobi, with the previously chosen solver applied to the local blocks. Likewise, the replicated layout can be used with any solver but SuperLu_Dist; therefore, if SuperLu_Dist has been previously set, the coarsest-level solver is changed to the default sequential solver.

Remark 4. The argument `idx` can be used to allow finer control for those solvers; for instance, by specifying the keyword `MUMPS_IPAR_ENTRY` and an appropriate value for `idx`, it is possible to set any entry in the MUMPS integer control array. See also Sec. 7.

what	DATA TYPE	val	DEFAULT	COMMENTS
'ML_CYCLE'	character(len=*)	'VCYCLE' 'WCYCLE' 'KCYCLE' 'MULT' 'ADD'	'VCYCLE'	Multilevel cycle: V-cycle, W-cycle, K-cycle, hybrid Multiplicative Schwarz, and Additive Schwarz. Note that hybrid Multiplicative Schwarz is equivalent to V-cycle and is included for compatibility with previous versions of MLD2P4.
'OUTER_SWEEPS'	integer	Any integer number ≥ 1	1	Number of multilevel cycles.

Table 2: Parameters defining the multilevel cycle and the number of cycles to be applied.

what	DATA TYPE	val	DEFAULT	COMMENTS
'MIN_COARSE_SIZE'	integer	Any number > 0	$\lfloor 40\sqrt[3]{n} \rfloor$, where n is the dimension of the matrix at the finest level	Coarse size threshold. The aggregation stops if the global number of variables of the computed coarsest matrix is lower than or equal to this threshold (see Note).
'MIN_CR_RATIO'	real	Any number > 1	1.5	Minimum coarsening ratio. The aggregation stops if the ratio between the matrix dimensions at two consecutive levels is lower than or equal to this threshold (see Note).
'MAX_LEVS'	integer	Any integer number > 1	20	Maximum number of levels. The aggregation stops if the number of levels reaches this value (see Note).
'PAR_AGGR_ALG'	character(len=*)	'DEC', 'SYMDEC'	'DEC'	Parallel aggregation algorithm. Currently, only the decoupled aggregation (DEC) is available; the SYMDEC option applies decoupled aggregation to the sparsity pattern of $A + A^T$.
'AGGR_TYPE'	character(len=*)	'SOC1'	'SOC1', 'SOC2'	Type of aggregation algorithm: currently, we implement to measures of strength of connection, the one by Vaněk, Mandel and Brezina [26], and the one by Gratton et al [16].
'AGGR_PROL'	character(len=*)	'SMOOTHED', 'UNSMOOTHED'	'SMOOTHED'	Prolongator used by the aggregation algorithm: smoothed or unsmoothed (i.e., tentative prolongator).
Note. The aggregation algorithm stops when at least one of the following criteria is met: the coarse size threshold, the minimum coarsening ratio, or the maximum number of levels is reached. Therefore, the actual number of levels may be smaller than the specified maximum number of levels.				

Table 3: Parameters defining the aggregation algorithm.

what	DATA TYPE	val	DEFAULT	COMMENTS
'AGGR_ORD'	character(len=*)	'NATURAL' 'DEGREE'	'NATURAL'	Initial ordering of indices for the aggregation algorithm: either natural ordering or sorted by descending degrees of the nodes in the matrix graph.
'AGGR_THRESH'	real(*kind_parameter*)	Any real number $\in [0, 1]$	0.01	The threshold θ in the aggregation algorithm, see (3) in Section 4.2. See also the note at the bottom of this table.
'AGGR_FILTER'	character(len=*)	'FILTER' 'NOFILTER'	'NOFILTER'	Matrix used in computing the smoothed prolongator: filtered or unfiltered (see (5) in Section 4.2).
Note. Different thresholds at different levels, such as those used in [26, Section 5.1], can be easily set by invoking the routine set with the parameter ilev.				

Table 4: Parameters defining the aggregation algorithm (continued).

what	DATA TYPE	val	DEFAULT	COMMENTS
'COARSE_MAT'	character(len=*)	'DIST' 'REPL'	'REPL'	Coarsest matrix layout: distributed among the processes or replicated on each of them.
'COARSE_SOLVE'	character(len=*)	'MUMPS' 'UMF' 'SLU' 'SLUDIST' 'JACOBI' 'GS' 'BJAC'	See Note.	Solver used at the coarsest level: sequential LU from MUMPS, UMFPACK, or SuperLU (plus triangular solve); distributed LU from MUMPS or SuperLU_Dist (plus triangular solve); point-Jacobi, hybrid Gauss-Seidel or block-Jacobi. Note that UMF and SLU require the coarsest matrix to be replicated, SLUDIST, JACOBI, GS and BJAC require it to be distributed, and MUMPS can be used with either a replicated or a distributed matrix. When any of the previous solvers is specified, the matrix layout is set to a default value which allows the use of the solver (see Remark 3, p. 24). Note also that UMFPACK and SuperLU_Dist are available only in double precision.
'COARSE_SUBSOLVE'	character(len=*)	'ILU' 'ILUT' 'MILU' 'MUMPS' 'SLU' 'UMF'	See Note.	Solver for the diagonal blocks of the coarse matrix, in case the block Jacobi solver is chosen as coarsest-level solver: ILU(p), ILU(p,t), MILU(p), LU from MUMPS, SuperLU or UMFPACK (plus triangular solve). Note that UMFPACK and SuperLU_Dist are available only in double precision.
Note. Defaults for COARSE_SOLVE and COARSE_SUBSOLVE are chosen in the following order: single precision version – MUMPS if installed, then SLU if installed, ILU otherwise; double precision version – UMF if installed, then MUMPS if installed, then SLU if installed, ILU otherwise.				

Table 5: Parameters defining the coarse-space correction at the coarsest level.

what	DATA TYPE	val	DEFAULT	COMMENTS
'COARSE_SWEEPS'	integer	Any integer number > 0	10	Number of sweeps when JACOBI, GS or BJAC is chosen as coarsest-level solver.
'COARSE_FILLIN'	integer	Any integer number ≥ 0	0	Fill-in level p of the ILU factorizations.
'COARSE_ILUTHRS'	real(*kind_parameter*)	Any real number ≥ 0	0	Drop tolerance t in the ILU(p,t) factorization.

Table 6: Parameters defining the coarse-space correction at the coarsest level (continued).

what	DATA TYPE	val	DEFAULT	COMMENTS
'SMOOTHER_TYPE'	character(len=*)	'JACOBI' 'GS' 'BGS' 'BJAC' 'AS'	'FBGS'	Type of smoother used in the multilevel preconditioner: point-Jacobi, hybrid (forward) Gauss-Seidel, hybrid backward Gauss-Seidel, block-Jacobi, and Additive Schwarz. It is ignored by one-level preconditioners.
'SUB_SOLVE'	character(len=*)	'JACOBI' 'GS' 'BGS' 'ILU' 'ILUT' 'MILU' 'MUMPS' 'SLU' 'UMF'	GS and BGS for pre- and post-smoothers of multilevel preconditioners, respectively ILU for block-Jacobi and Additive Schwarz one-level preconditioners	The local solver to be used with the smoother or one-level preconditioner (see Remark 2, page 24): point-Jacobi, hybrid (forward) Gauss-Seidel, hybrid backward Gauss-Seidel, ILU(p), ILU(p,t), MILU(p), LU from MUMPS, SuperLU or UMFPACK (plus triangular solve). See Note for details on hybrid Gauss-Seidel.
'SMOOTHER_SWEEPS'	integer	Any integer number ≥ 0	1	Number of sweeps of the smoother or one-level preconditioner. In the multilevel case, no pre-smother or post-smoother is used if this parameter is set to 0 together with pos='PRE' or pos='POST', respectively.
'SUB_OVR'	integer	Any integer number ≥ 0	1	Number of overlap layers, for Additive Schwarz only.

Table 7: Parameters defining the smoother or the details of the one-level preconditioner.

what	DATA TYPE	val	DEFAULT	COMMENTS
'SUB_RESTR'	character(len=*)	'HALO' 'NONE'	'HALO'	Type of restriction operator, for Additive Schwarz only: HALO for taking into account the overlap, NONE for neglecting it. Note that HALO must be chosen for the classical Addditive Schwarz smoother and its RAS variant.
'SUB_PROL'	character(len=*)	'SUM' 'NONE'	'NONE'	Type of prolongation operator, for Additive Schwarz only: SUM for adding the contributions from the overlap, NONE for neglecting them. Note that SUM must be chosen for the classical Additive Schwarz smoother, and NONE for its RAS variant.
'SUB_FILLIN'	integer	Any integer number ≥ 0	0	Fill-in level p of the incomplete LU factorizations.
'SUB_ILUTHRS'	real(*kind_parameter*)	Any real number ≥ 0	0	Drop tolerance t in the ILU(p,t) factorization.
'MUMPS_IPAR_ENTRY'	integer	Any integer number	0	Set an entry in the MUMPS control array, as chosen via the idx optional argument.

Table 8: Parameters defining the smoother or the details of the one-level preconditioner (continued).

6.3 Method hierarchy_build

```
call p%hierarchy_build(a,desc_a,info)
```

This method builds the hierarchy of matrices and restriction/prolongation operators for the multilevel preconditioner `p`, according to the requirements made by the user through the methods `init` and `set`.

Arguments

`a` `type(psb_`x`spmat_type), intent(in).`
The sparse matrix structure containing the local part of the matrix to be preconditioned. Note that x must be chosen according to the real/complex, single/double precision version of MLD2P4 under use. See the PSBLAS User's Guide for details [13].

`desc_a` `type(psb_desc_type), intent(in).`
The communication descriptor of `a`. See the PSBLAS User's Guide for details [13].

`info` `integer, intent(out).`
Error code. If no error, 0 is returned. See Section 8 for details.

6.4 Method smoothers_build

```
call p%smoothers_build(a,desc_a,p,info[,amold,vmold,imold])
```

This method builds the smoothers and the coarsest-level solvers for the multilevel preconditioner `p`, according to the requirements made by the user through the methods `init` and `set`, and based on the aggregation hierarchy produced by a previous call to `hierarchy_build` (see Section 6.3).

Arguments

`a` `type(psb_`x`spmat_type), intent(in).`
The sparse matrix structure containing the local part of the matrix to be preconditioned. Note that x must be chosen according to the real/complex, single/double precision version of MLD2P4 under use. See the PSBLAS User's Guide for details [13].

`desc_a` `type(psb_desc_type), intent(in).`
The communication descriptor of `a`. See the PSBLAS User's Guide for details [13].

`info` `integer, intent(out).`
Error code. If no error, 0 is returned. See Section 8 for details.

`amold` `class(psb_`x`_base_sparse_mat), intent(in), optional.`
The desired dynamic type for internal matrix components; this allows e.g. running on GPUs; it needs not be the same on all processes. See the PSBLAS User's Guide for details [13].

`vmold` `class(psb_`x`_base_vect_type), intent(in), optional.`
The desired dynamic type for internal vector components; this allows e.g. running on GPUs.

`imold` `class(psb_i_base_vect_type), intent(in), optional.`
The desired dynamic type for internal integer vector components; this allows e.g. running on GPUs.

6.5 Method build

```
call p%build(a,desc_a,info[,amold,vmold,imold])
```

This method builds the preconditioner `p` according to the requirements made by the user through the methods `init` and `set` (see Sections 6.3 and 6.4 for multilevel preconditioners). It is mostly provided for backward compatibility; indeed, it is internally implemented by invoking the two previous methods `hierarchy_build` and `smoothers_build`, whose nomenclature would however be somewhat unnatural when dealing with simple one-level preconditioners.

Arguments

`a` `type(psb_`x`spmat_type), intent(in).`
The sparse matrix structure containing the local part of the matrix to be preconditioned. Note that x must be chosen according to the real/complex, single/double precision version of MLD2P4 under use. See the PSBLAS User's Guide for details [13].

`desc_a` `type(psb_desc_type), intent(in).`
The communication descriptor of `a`. See the PSBLAS User's Guide for details [13].

`info` `integer, intent(out).`
Error code. If no error, 0 is returned. See Section 8 for details.

`amold` `class(psb_`x`_base_sparse_mat), intent(in), optional.`
The desired dynamic type for internal matrix components; this allows e.g. running on GPUs; it needs not be the same on all processes. See the PSBLAS User's Guide for details [13].

`vmold` `class(psb_`x`_base_vect_type), intent(in), optional.`
The desired dynamic type for internal vector components; this allows e.g. running on GPUs.

`imold` `class(psb_i_base_vect_type), intent(in), optional.`
The desired dynamic type for internal integer vector components; this allows e.g. running on GPUs.

For compatibility with the previous versions of MLD2P4, this method can be also invoked as follows:

```
call mld_precbld(p,what,val,info[,amold,vmold,imold])
```

The method can be used to build multilevel preconditioners too.

6.6 Method apply

```
call p%apply(x,y,desc_a,info [,trans,work])
```

This method computes $y = op(B^{-1})\,x$, where B is a previously built preconditioner, stored into `p`, and *op* denotes the preconditioner itself or its transpose, according to the value of `trans`. Note that, when MLD2P4 is used with a Krylov solver from PSBLAS, `p%apply` is called within the PSBLAS method `psb_krylov` and hence it is completely transparent to the user.

Arguments

`x`	*type*(*kind_parameter*), `dimension(:), intent(in)`. The local part of the vector x. Note that *type* and *kind_parameter* must be chosen according to the real/complex, single/double precision version of MLD2P4 under use.
`y`	*type*(*kind_parameter*), `dimension(:), intent(out)`. The local part of the vector y. Note that *type* and *kind_parameter* must be chosen according to the real/complex, single/double precision version of MLD2P4 under use.
`desc_a`	`type(psb_desc_type), intent(in)`. The communication descriptor associated to the matrix to be preconditioned.
`info`	`integer, intent(out)`. Error code. If no error, 0 is returned. See Section 8 for details.
`trans`	`character(len=1), optional, intent(in)`. If `trans` = `'N','n'` then $op(B^{-1}) = B^{-1}$; if `trans` = `'T','t'` then $op(B^{-1}) = B^{-T}$ (transpose of B^{-1}); if `trans` = `'C','c'` then $op(B^{-1}) = B^{-C}$ (conjugate transpose of B^{-1}).
`work`	*type*(*kind_parameter*), `dimension(:), optional, target`. Workspace. Its size should be at least `4 * psb_cd_get_local_cols(desc_a)` (see the PSBLAS User's Guide). Note that *type* and *kind_parameter* must be chosen according to the real/complex, single/double precision version of MLD2P4 under use.

For compatibility with the previous versions of MLD2P4, this method can be also invoked as follows:

```
call mld_precaply(p,what,val,info)
```

6.7 Method free

```
call p%free(p,info)
```

This method deallocates the preconditioner data structure `p`.

Arguments

`info` `integer, intent(out).`
Error code. If no error, 0 is returned. See Section 8 for details.

For compatibility with the previous versions of MLD2P4, this method can be also invoked as follows:

```
call mld_precfree(p,info)
```

6.8 Method descr

```
call p%descr(info, [iout])
```

This method prints a description of the preconditioner `p` to the standard output or to a file. It must be called after `hierachy_build` and `smoothers_build`, or `build`, have been called.

Arguments

`info` `integer, intent(out).`
Error code. If no error, 0 is returned. See Section 8 for details.

`iout` `integer, intent(in), optional.`
The id of the file where the preconditioner description will be printed; the default is the standard output.

For compatibility with the previous versions of MLD2P4, this method can be also invoked as follows:

```
call mld_precdescr(p,info [,iout])
```

6.9 Auxiliary Methods

Various functionalities are implemented as additional methods of the preconditioner object.

6.9.1 Method: dump

```
call p%dump(info[,istart,iend,prefix,head,ac,rp,smoother,solver,global_num])
```

Dump on file.

Arguments

`info` `integer, intent(out).`
Error code. If no error, 0 is returned. See Section 8 for details.

`amold` `class(psb_`x`_base_sparse_mat), intent(in), optional.`
The desired dynamic type for internal matrix components; this allows e.g. running on GPUs; it needs not be the same on all processes. See the PSBLAS User's Guide for details [13].

6.9.2 Method: clone

```
call p%clone(pout,info)
```

Create a (deep) copy of the preconditioner object.

Arguments

`pout` `type(mld_`x`prec_type), intent(out).`
The copy of the preconditioner data structure. Note that x must be chosen according to the real/complex, single/double precision version of MLD2P4 under use.

`info` `integer, intent(out).`
Error code. If no error, 0 is returned. See Section 8 for details.

6.9.3 Method: sizeof

```
sz =  p%sizeof()
```

Return memory footprint in bytes.

6.9.4 Method: allocate_wrk

```
call p%allocate_wrk(info[, vmold])
```

Allocate internal work vectors. Each application of the preconditioner uses a number of work vectors which are allocated internally as necessary; therefore allocation and deallocation of memory occurs multiple times during the execution of a Krylov method. In most cases this strategy is perfectly acceptable, but on some platforms, most notably GPUs, memory allocation is a slow operation, and the default behaviour would lead to a slowdown. This method allows to trade space for time by preallocating the internal workspace outside of the invocation of a Krylov method. When using GPUs or other specialized devices, the `vmold` argument is also necessary to ensure the internal work vectors are of the appropriate dynamic type to exploit the accelerator hardware; when allocation occurs internally this is taken care of based on the dynamic type of the `x` argument to the `apply` method.

Arguments

`info` `integer, intent(out).`
Error code. If no error, 0 is returned. See Section 8 for details.

`vmold` `class(psb_`x`_base_vect_type), intent(in), optional.`
The desired dynamic type for internal vector components; this allows e.g. running on GPUs.

6.9.5 Method: free_wrk

```
call p%free_wrk(info)
```

Deallocate internal work vectors.

Arguments

`info` `integer, intent(out).`
Error code. If no error, 0 is returned. See Section 8 for details.

7 Adding new smoother and solver objects to MLD2P4

Developers can add completely new smoother and/or solver classes derived from the base objects in the library (see Remark 2 in Section 6.2), without recompiling the library itself.

To do so, it is necessary first to select the base type to be extended. In our experience, it is quite likely that the new application needs only the definition of a "solver" object, which is almost always acting only on the local part of the distributed matrix. The parallel actions required to connect the various solver objects are most often already provided by the block-Jacobi or the additive Schwarz smoothers. To define a new solver, the developer will then have to define its components and methods, perhaps taking one of the predefined solvers as a starting point, if possible.

Once the new smoother/solver class has been developed, to use it in the context of the multilevel preconditioners it is necessary to:

- declare in the application program a variable of the new type;
- pass that variable as the argument to the `set` routine as in the following:

```
call p%set(smoother,info [,ilev,ilmax,pos])
 call p%set(solver,info [,ilev,ilmax,pos])
```

- link the code implementing the various methods into the application executable.

The new solver object is then dynamically included in the preconditioner structure, and acts as a *mold* to which the preconditioner will conform, even though the MLD2P4 library has not been modified to account for this new development.

It is possible to define new values for the keyword `WHAT` in the `set` routine; if the library code does not recognize a keyword, it passes it down the composition hierarchy (levels containing smoothers containing in turn solvers), so that it can be eventually caught by the new solver. By the same token, any keyword/value pair that does not pertain to a given smoother should be passed down to the contained solver, and any keyword/value pair that does not pertain to a given solver is by default ignored.

An example is provided in the source code distribution under the folder `tests/newslv`. In this example we are implementing a new incomplete factorization variant (which is simply the ILU(0) factorization under a new name). Because of the specifics of this case, it is possible to reuse the basic structure of the ILU solver, with its L/D/U components and the methods needed to apply the solver; only a few methods, such as the description and most importantly the build, need to be ovverridden (rewritten).

The interfaces for the calls shown above are defined using

`smoother` `class(mld_x_base_smoother_type)`
: The user-defined new smoother to be employed in the preconditioner.

`solver` `class(mld_x_base_solver_type)`
: The user-defined new solver to be employed in the preconditioner.

The other arguments are defined in the way described in Sec. 6.2. As an example, in the `tests/newslv` code we define a new object of type `mld_d_tlu_solver_type`, and we pass it as follows:

```
  ! sparse matrix and preconditioner
  type(psb_dspmat_type) :: a
  type(mld_dprec_type)  :: prec
  type(mld_d_tlu_solver_type) :: tlusv

.......
  !
  !  prepare the preconditioner: an ML with defaults, but with TLU solver at
  !  intermediate levels. All other parameters are at default values.
  !
  call prec%init('ML',       info)
  call prec%hierarchy_build(a,desc_a,info)
  nlv = prec%get_nlevs()
  call prec%set(tlusv,   info,ilev=1,ilmax=max(1,nlv-1))
  call prec%smoothers_build(a,desc_a,info)
```

8 Error Handling

The error handling in MLD2P4 is based on the PSBLAS error handling. Error conditions are signaled via an integer argument `info`; whenever an error condition is detected, an error trace stack is built by the library up to the top-level, user-callable routine. This routine will then decide, according to the user preferences, whether the error should be handled by terminating the program or by returning the error condition to the user code, which will then take action, and whether an error message should be printed. These options may be set by using the PSBLAS error handling routines; for further details see the PSBLAS User's Guide [13].

A License

The MLD2P4 is freely distributable under the following copyright terms:

References

[1] P. R. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J. L'Excellent, C. Weisbecker, *Improving multifrontal methods by means of block low-rank representations*, SIAM Journal on Scientific Computing, volume 37 (3), 2015, A1452–A1474. See also `http://mumps.enseeiht.fr`.

[2] M. Brezina, P. Vaněk, *A Black-Box Iterative Solver Based on a Two-Level Schwarz Method*, Computing, 63, 1999, 233–263.

[3] W. L. Briggs, V. E. Henson, S. F. McCormick, *A Multigrid Tutorial, Second Edition*, SIAM, 2000.

[4] A. Buttari, P. D'Ambra, D. di Serafino, S. Filippone, *Extending PSBLAS to Build Parallel Schwarz Preconditioners*, in J. Dongarra, K. Madsen, J. Wasniewski, editors, Proceedings of PARA 04 Workshop on State of the Art in Scientific Computing, Lecture Notes in Computer Science, Springer, 2005, 593–602.

[5] A. Buttari, P. D'Ambra, D. di Serafino, S. Filippone, *2LEV-D2P4: a package of high-performance preconditioners for scientific and engineering applications*, Applicable Algebra in Engineering, Communications and Computing, 18 (3) 2007, 223–239.

[6] X. C. Cai, M. Sarkis, *A Restricted Additive Schwarz Preconditioner for General Sparse Linear Systems*, SIAM Journal on Scientific Computing, 21 (2), 1999, 792–797.

[7] P. D'Ambra, S. Filippone, D. di Serafino, *On the Development of PSBLAS-based Parallel Two-level Schwarz Preconditioners*, Applied Numerical Mathematics, Elsevier Science, 57 (11-12), 2007, 1181-1196.

[8] P. D'Ambra, D. di Serafino, S. Filippone, *MLD2P4: a Package of Parallel Multilevel Algebraic Domain Decomposition Preconditioners in Fortran 95*, ACM Trans. Math. Softw., 37(3), 2010, art. 30.

[9] T. A. Davis, *Algorithm 832: UMFPACK - an Unsymmetric-pattern Multifrontal Method with a Column Pre-ordering Strategy*, ACM Transactions on Mathematical Software, 30, 2004, 196–199. (See also `http://www.cise.ufl.edu/~davis/`)

[10] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, J. W. H. Liu, *A supernodal approach to sparse partial pivoting*, SIAM Journal on Matrix Analysis and Applications, 20 (3), 1999, 720–755.

[11] J. J. Dongarra, J. Du Croz, I. S. Duff, S. Hammarling, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 16 (1) 1990, 1–17.

[12] J. J. Dongarra, J. Du Croz, S. Hammarling, R. J. Hanson, *An extended set of FORTRAN Basic Linear Algebra Subprograms*, ACM Transactions on Mathematical Software, 14 (1) 1988, 1–17.

[13] S. Filippone, A. Buttari, *PSBLAS 3.5.0 User's Guide. A Reference Guide for the Parallel Sparse BLAS Library*, 2012, available from `https://github.com/sfilippone/psblas3/tree/master/docs`.

[14] S. Filippone, A. Buttari, *Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003*. ACM Transactions on on Mathematical Software, 38 (4), 2012, art. 23.

[15] S. Filippone, M. Colajanni, *PSBLAS: A Library for Parallel Linear Algebra Computation on Sparse Matrices*, ACM Transactions on Mathematical Software, 26 (4), 2000, 527–550.

[16] S. Gratton, P. Henon, P. Jiranek and X. Vasseur, *Reducing complexity of algebraic multigrid by aggregation*, Numerical Lin. Algebra with Applications, 2016, 23:501-518

[17] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, M. Snir, *MPI: The Complete Reference. Volume 2 - The MPI-2 Extensions*, MIT Press, 1998.

[18] C. L. Lawson, R. J. Hanson, D. Kincaid, F. T. Krogh, *Basic Linear Algebra Subprograms for FORTRAN usage*, ACM Transactions on Mathematical Software, 5 (3), 1979, 308–323.

[19] X. S. Li, J. W. Demmel, *SuperLU_DIST: A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric Linear Systems*, ACM Transactions on Mathematical Software, 29 (2), 2003, 110–140.

[20] Y. Notay, P. S. Vassilevski, *Recursive Krylov-based multigrid cycles*, Numerical Linear Algebra with Applications, 15 (5), 2008, 473–487.

[21] Y. Saad, *Iterative methods for sparse linear systems*, 2nd edition, SIAM, 2003.

[22] B. Smith, P. Bjorstad, W. Gropp, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge University Press, 1996.

[23] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, *MPI: The Complete Reference. Volume 1 - The MPI Core*, second edition, MIT Press, 1998.

[24] K. Stüben, *An Introduction to Algebraic Multigrid*, in A. Schüller, U. Trottenberg, C. Oosterlee, Multigrid, Academic Press, 2001.

[25] R. S. Tuminaro, C. Tong, *Parallel Smoothed Aggregation Multigrid: Aggregation Strategies on Massively Parallel Machines*, in J. Donnelley, editor, Proceedings of SuperComputing 2000, Dallas, 2000.

[26] P. Vaněk, J. Mandel, M. Brezina, *Algebraic Multigrid by Smoothed Aggregation for Second and Fourth Order Elliptic Problems*, Computing, 56 (3) 1996, 179–196.