MLD2P4
User's and Reference Guide

A guide for the Multi-Level 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: 2.1
March 28, 2017

Abstract

MLD2P4 (Multi-Level Domain Decomposition Parallel Preconditioners Package based on PSBLAS) is a package of parallel algebraic multi-level preconditioners. The first release made available various versions of one-level additive and multi-level additive and hybrid Schwarz preconditioners. The package has been extended to include further multi-level cycles and smoothers widely used in multigrid methods. In the multi-level 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 multi-level 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 multi-level 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.

Contents

Abstract i

1 General Overview 1

2 Code Distribution 3

3 Configuring and Building MLD2P4 4
- 3.1 Prerequisites . . . 4
- 3.2 Optional third party libraries . . . 4
- 3.3 Configuration options . . . 5
- 3.4 Bug reporting . . . 9
- 3.5 Example and test programs . . . 9

4 Multi-level Domain Decomposition Background 10
- 4.1 Multi-level Schwarz Preconditioners . . . 11
- 4.2 Smoothed Aggregation . . . 13

5 Getting Started 16
- 5.1 Examples . . . 17

6 User Interface 21
- 6.1 Subroutine init . . . 22
- 6.2 Subroutine set . . . 23
- 6.3 Subroutine bld . . . 33
- 6.4 Subroutine hierarchy_bld . . . 34
- 6.5 Subroutine smoothers_bld . . . 35
- 6.6 Subroutine apply . . . 36
- 6.7 Subroutine free . . . 37
- 6.8 Subroutine descr . . . 38

7 Adding smoothers and solvers to MLD2P4 39

8 Error Handling 40

A License 41

References 43

1 General Overview

The Multi-Level Domain Decomposition Parallel Preconditioners Package based on PSBLAS (MLD2P4) provides parallel Algebraic MultiGrid (AMG) and domain decomposition preconditioners, designed to provide scalable and easy-to-use preconditioners multi-level Schwarz preconditioners [25, 23], to be used in the iterative solutions of sparse linear systems:

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

where A is a square, real or complex, sparse matrix. Multi-level preconditioners can be obtained by combining several AMG cycles (V, W, K) with different smoothers (Jacobi, hybrid forward/backward Gauss-Seidel, block-Jacobi, additive Schwarz methods). 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. The smoothed aggregation technique is applied as algebraic coarsening strategy [1, 27]. Either exact or approximate solvers are available to solve the coarsest-level system. Specifically, different versions of sparse LU factorizations from external packages, and native incomplete LU factorizations and iterative block-Jacobi solvers can be used. 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, 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 [18, 17]. PSBLAS provides basic linear algebra operators and data management facilities for distributed sparse matrices, as well as parallel Krylov solvers which can be coupled 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 into the PSBLAS routines, except few cases where MPI [24] is explicitly called **É ancora cosi???**. Therefore, MLD2P4 can be run on any parallel machine where PSBLAS and MPI 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 non-expert users to easily build any preconditioner available in MLD2P4 and

to apply it within a PSBLAS Krylov solver; **facilities are also available that allow more expert users to extend the set of smoothers and solvers for building new versions of preconditioners.**

We note that the user interface of MLD2P4 2.1 (**Perche 2.1 e non 2.0???...Ricordarsi di cambiare il configure**) has been extended with respect to the previous versions in order to separate the construction of the multi-level 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 upper-layer routines of MLD2P4, that are the user interface, is provided in Section 6. 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

`http://www.mld2p4.it`

where contact points for further information can be also found. **Passiamo subito a GitHub?**

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`

3 Configuring and Building MLD2P4

In order to build MLD2P4 it is necessary to set up a Makefile with appropriate values for your system; 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.

3.1 Prerequisites

The following base libraries are needed:

BLAS [12, 13, 20] 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/lapac` 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 [19, 24] A version of MPI is available on most high-performance computing systems.

PSBLAS [16, 18] Parallel Sparse BLAS (PSBLAS) is available from `www.ce.uniroma2.it/psblas`; version 3.4.0 (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 multi-level 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. 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 [10] 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 version 5.0.1.

SuperLU [11] 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 version 4.3 and 5.0. If you installed BLAS from ATLAS, remember to define the BLASLIB variable in the make.inc file.

SuperLU_Dist [21] 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 version 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

CONTROLLARE HELP DEL CONFIGURE: Versione MLD2P4, Versione PSBLAS, Influential Environmental Variables???

To build MLD2P4 the first step is to use the `configure` script in the main directory to generate the necessary makefile(s).

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 2.0 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]

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-serial         Specify whether to enable a fake mpi library to run
                          in serial mode.

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-3.3
  --with-psblas-incdir=DIR
                          Specify the directory for PSBLAS includes.
  --with-psblas-libdir=DIR
                          Specify the directory for PSBLAS library.
  --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-mumps=LIBNAME    Specify the libname for MUMPS. Default: "-lsmumps
                          -ldmumps -lcmumps -lzmumps -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-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    C/C++/Objective C preprocessor flags, e.g. -I<include dir> if
              you have headers in a nonstandard directory <include dir>
  CPP         C preprocessor
  MPICC       MPI C compiler command
  F77         Fortran 77 compiler command
  FFLAGS      Fortran 77 compiler flags
  MPIF77      MPI Fortran 77 compiler command
  MPIFC       MPI Fortran compiler command

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 <bugreport@mld2p4.it>.
```

For instance, if a user has built and installed PSBLAS 3.4 under the `/opt` directory and

is using the SuiteSparse package (which includes UMFPACK), then MLD2P4 might be configured with:

```
./configure --with-psblas=/opt/psblas-3.4/ \
--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 -lzmumps -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 let us know at (DECIDERE A CHI FARE IL BUG REPORTING) `bugreport@mld2p4.it` ; be aware that the amount of information needed to reproduce a problem in a parallel program may vary quite a lot.

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.

`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 library.

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` instead generate matrices in full parallel mode from the discretization of a sample PDE.

4 Multi-level Domain Decomposition Background

Domain Decomposition (DD) preconditioners, coupled with Krylov iterative solvers, are widely used in the parallel solution of large and sparse linear systems. These preconditioners are based on the divide and conquer technique: the matrix to be preconditioned is divided into submatrices, a "local" linear system involving each submatrix is (approximately) solved, and the local solutions are used to build a preconditioner for the whole original matrix. This process often corresponds to dividing a physical domain associated to the original matrix into subdomains, e.g. in a PDE discretization, to (approximately) solving the subproblems corresponding to the subdomains and to building an approximate solution of the original problem from the local solutions [6, 7, 23].

Additive Schwarz preconditioners are DD preconditioners using overlapping submatrices, i.e. with some common rows, to couple the local information related to the submatrices (see, e.g., [23]). The main motivation for choosing Additive Schwarz preconditioners is their intrinsic parallelism. A drawback of these preconditioners is that the number of iterations of the preconditioned solvers generally grows with the number of submatrices. This may be a serious limitation on parallel computers, since the number of submatrices usually matches the number of available processors. Optimal convergence rates, i.e. iteration numbers independent of the number of submatrices, can be obtained by correcting the preconditioner through a suitable approximation of the original linear system in a coarse space, which globally couples the information related to the single submatrices.

Two-level Schwarz preconditioners are obtained by combining basic (one-level) Schwarz preconditioners with a coarse-level correction. In this context, the one-level preconditioner is often called 'smoother'. Different two-level preconditioners are obtained by varying the choice of the smoother and of the coarse-level correction, and the way they are combined [23]. The same reasoning can be applied starting from the coarse-level system, i.e. a coarse-space correction can be built from this system, thus obtaining *multi-level* preconditioners.

It is worth noting that optimal preconditioners do not necessarily correspond to minimum execution times. Indeed, to obtain effective multi-level preconditioners a tradeoff between optimality of convergence and the cost of building and applying the coarse-space corrections must be achieved. The choice of the number of levels, i.e. of the coarse-space corrections, also affects the effectiveness of the preconditioners. One more goal is to get convergence rates as less sensitive as possible to variations in the matrix coefficients.

Two main approaches can be used to build coarse-space corrections. The geometric approach applies coarsening strategies based on the knowledge of some physical grid associated to the matrix and requires the user to define grid transfer operators from the fine to the coarse levels and vice versa. This may result difficult for complex geometries; furthermore, suitable one-level preconditioners may be required to get efficient interplay between fine and coarse levels, e.g. when matrices with highly varying coefficients are considered. The algebraic approach builds coarse-space corrections using only matrix information. It performs a fully automatic coarsening and enforces the

interplay between the fine and coarse levels by suitably choosing the coarse space and the coarse-to-fine interpolation [25].

MLD2P4 uses a pure algebraic approach for building the sequence of coarse matrices starting from the original matrix. The algebraic approach is based on the *smoothed aggregation* algorithm [1, 27]. A decoupled version of this algorithm is implemented, where the smoothed aggregation is applied locally to each submatrix [26]. In the next two subsections we provide a brief description of the multi-level Schwarz preconditioners and of the smoothed aggregation technique as implemented in MLD2P4. For further details the reader is referred to [2, 3, 4, 8, 23].

4.1 Multi-level Schwarz Preconditioners

The Multilevel preconditioners implemented in MLD2P4 are obtained by combining AS preconditioners with coarse-space corrections; therefore we first provide a sketch of the AS preconditioners.

Given the linear system (1), where $A = (a_{ij}) \in \Re^{n\times n}$ is a nonsingular sparse matrix with a symmetric nonzero pattern, let $G = (W, E)$ be the adjacency graph of A, where $W = \{1, 2, \ldots, n\}$ and $E = \{(i, j) : a_{ij} \neq 0\}$ are the vertex set and the edge set of G, respectively. Two vertices are called adjacent if there is an edge connecting them. For any integer $\delta > 0$, a δ-overlap partition of W can be defined recursively as follows. Given a 0-overlap (or non-overlapping) partition of W, i.e. a set of m disjoint nonempty sets $W_i^0 \subset W$ such that $\cup_{i=1}^m W_i^0 = W$, a δ-overlap partition of W is obtained by considering the sets $W_i^\delta \supset W_i^{\delta-1}$ obtained by including the vertices that are adjacent to any vertex in $W_i^{\delta-1}$.

Let n_i^δ be the size of W_i^δ and $R_i^\delta \in \Re^{n_i^\delta \times n}$ the restriction operator that maps a vector $v \in \Re^n$ onto the vector $v_i^\delta \in \Re^{n_i^\delta}$ containing the components of v corresponding to the vertices in W_i^δ. The transpose of R_i^δ is a prolongation operator from $\Re^{n_i^\delta}$ to $\Re^n$. The matrix $A_i^\delta = R_i^\delta A (R_i^\delta)^T \in \Re^{n_i^\delta \times n_i^\delta}$ can be considered as a restriction of A corresponding to the set W_i^δ.

The *classical one-level AS* preconditioner is defined by

$$M_{AS}^{-1} = \sum_{i=1}^{m} (R_i^\delta)^T (A_i^\delta)^{-1} R_i^\delta,$$

where A_i^δ is assumed to be nonsingular. Its application to a vector $v \in \Re^n$ within a Krylov solver requires the following three steps:

1. restriction of v as $v_i = R_i^\delta v$, $i = 1, \ldots, m$;
2. solution of the linear systems $A_i^\delta w_i = v_i$, $i = 1, \ldots, m$;
3. prolongation and sum of the w_i's, i.e. $w = \sum_{i=1}^m (R_i^\delta)^T w_i$.

Note that the linear systems at step 2 are usually solved approximately, e.g. using incomplete LU factorizations such as ILU(p), MILU(p) and ILU(p, t) [22, Chapter 10].

A variant of the classical AS preconditioner that outperforms it in terms of convergence rate and of computation and communication time on parallel distributed-memory computers is the so-called *Restricted AS (RAS)* preconditioner [5, 15]. It is obtained by zeroing the components of w_i corresponding to the overlapping vertices when applying the prolongation. Therefore, RAS differs from classical AS by the prolongation operators, which are substituted by $(\tilde{R}_i^0)^T \in \Re^{n_i^\delta \times n}$, where $\tilde{R}_i^0$ is obtained by zeroing the rows of R_i^δ corresponding to the vertices in $W_i^\delta \backslash W_i^0$:

$$M_{RAS}^{-1} = \sum_{i=1}^{m} (\tilde{R}_i^0)^T (A_i^\delta)^{-1} R_i^\delta.$$

Analogously, the AS variant called *AS with Harmonic extension (ASH)* is defined by

$$M_{ASH}^{-1} = \sum_{i=1}^{m} (R_i^\delta)^T (A_i^\delta)^{-1} \tilde{R}_i^0.$$

We note that for $\delta = 0$ the three variants of the AS preconditioner are all equal to the block-Jacobi preconditioner.

As already observed, the convergence rate of the one-level Schwarz preconditioned iterative solvers deteriorates as the number m of partitions of W increases [7, 23]. To reduce the dependency of the number of iterations on the degree of parallelism we may introduce a global coupling among the overlapping partitions by defining a coarse-space approximation A_C of the matrix A. In a pure algebraic setting, A_C is usually built with the Galerkin approach. Given a set W_C of *coarse vertices*, with size n_C, and a suitable restriction operator $R_C \in \Re^{n_C \times n}$, A_C is defined as

$$A_C = R_C A R_C^T$$

and the coarse-level correction matrix to be combined with a generic one-level AS preconditioner M_{1L} is obtained as

$$M_C^{-1} = R_C^T A_C^{-1} R_C,$$

where A_C is assumed to be nonsingular. The application of M_C^{-1} to a vector v corresponds to a restriction, a solution and a prolongation step; the solution step, involving the matrix A_C, may be carried out also approximately.

The combination of M_C and M_{1L} may be performed in either an additive or a multiplicative framework. In the former case, the *two-level additive* Schwarz preconditioner is obtained:

$$M_{2LA}^{-1} = M_C^{-1} + M_{1L}^{-1}.$$

Applying M_{2L-A}^{-1} to a vector v within a Krylov solver corresponds to applying M_C^{-1} and M_{1L}^{-1} to v independently and then summing up the results.

In the multiplicative case, the combination can be performed by first applying the smoother M_{1L}^{-1} and then the coarse-level correction operator M_C^{-1}:

$$\begin{aligned} w &= M_{1L}^{-1} v, \\ z &= w + M_C^{-1}(v - Aw); \end{aligned}$$

this corresponds to the following *two-level hybrid pre-smoothed* Schwarz preconditioner:

$$M_{2LH-PRE}^{-1} = M_C^{-1} + \left(I - M_C^{-1}A\right) M_{1L}^{-1}.$$

On the other hand, by applying the smoother after the coarse-level correction, i.e. by computing

$$\begin{aligned} w &= M_C^{-1}v, \\ z &= w + M_{1L}^{-1}(v - Aw), \end{aligned}$$

the *two-level hybrid post-smoothed* Schwarz preconditioner is obtained:

$$M_{2LH-POST}^{-1} = M_{1L}^{-1} + \left(I - M_{1L}^{-1}A\right) M_C^{-1}.$$

One more variant of two-level hybrid preconditioner is obtained by applying the smoother before and after the coarse-level correction. In this case, the preconditioner is symmetric if A, M_{1L} and M_C are symmetric.

As previously noted, on parallel computers the number of submatrices usually matches the number of available processors. When the size of the system to be preconditioned is very large, the use of many processors, i.e. of many small submatrices, often leads to a large coarse-level system, whose solution may be computationally expensive. On the other hand, the use of few processors often leads to local sumatrices that are too expensive to be processed on single processors, because of memory and/or computing requirements. Therefore, it seems natural to use a recursive approach, in which the coarse-level correction is re-applied starting from the current coarse-level system. The corresponding preconditioners, called *multi-level* preconditioners, can significantly reduce the computational cost of preconditioning with respect to the two-level case (see [23, Chapter 3]). Additive and hybrid multilevel preconditioners are obtained as direct extensions of the two-level counterparts. For a detailed descrition of them, the reader is referred to [23, Chapter 3]. The algorithm for the application of a multi-level hybrid post-smoothed preconditioner M to a vector v, i.e. for the computation of $w = M^{-1}v$, is reported, for example, in Figure 1. Here the number of levels is denoted by $nlev$ and the levels are numbered in increasing order starting from the finest one, i.e. the finest level is level 1; the coarse matrix and the corresponding basic preconditioner at each level l are denoted by A_l and M_l, respectively, with $A_1 = A$, while the related restriction operator is denoted by R_l.

4.2 Smoothed Aggregation

In order to define the restriction operator R_C, which is used to compute the coarse-level matrix A_C, MLD2P4 uses the *smoothed aggregation* algorithm described in [1, 27]. The basic idea of this algorithm is to build a coarse set of vertices W_C by suitably grouping the vertices of W into disjoint subsets (aggregates), and to define the coarse-to-fine space transfer operator R_C^T by applying a suitable smoother to a simple piecewise constant prolongation operator, to improve the quality of the coarse-space correction.

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

$v_1 = v$;
for $l = 2, nlev$ **do**
  ! transfer v_{l-1} to the next coarser level
  $v_l = R_l v_{l-1}$
endfor
! apply the coarsest-level correction
$y_{nlev} = A_{nlev}^{-1} v_{nlev}$
for $l = nlev - 1, 1, -1$ **do**
  ! transfer y_{l+1} to the next finer level
  $y_l = R_{l+1}^T y_{l+1}$;
  ! compute the residual at the current level
  $r_l = v_l - A_l^{-1} y_l$;
  ! apply the basic Schwarz preconditioner to the residual
  $r_l = M_l^{-1} r_l$
  ! update y_l
  $y_l = y_l + r_l$
endfor
$w = y_1$;

Figure 1: Application of the multi-level hybrid post-smoothed preconditioner.

1. coarsening of the vertex set W, to obtain W_C;
2. construction of the prolongator R_C^T;
3. application of R_C and R_C^T to build A_C.

To perform the coarsening step, we have implemented the aggregation algorithm sketched in [4]. According to [27], a modification of this algorithm has been actually considered, in which each aggregate N_r is made of vertices of W that are *strongly coupled* to a certain root vertex $r \in W$, i.e.

$$N_r = \left\{ s \in W : |a_{rs}| > \theta \sqrt{|a_{rr} a_{ss}|} \right\} \cup \{r\},$$

for a given $\theta \in [0, 1]$. Since this algorithm has a sequential nature, a *decoupled* version of it has been chosen, where each processor i independently applies the algorithm to the set of vertices W_i^0 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 non-uniform aggregates near boundary vertices, i.e. near vertices adjacent to vertices in other processors, and is strongly dependent on the number of processors and on the initial partitioning of the matrix A. Nevertheless, this algorithm has been chosen for the implementation in MLD2P4, since it has been shown to produce good results in practice [3, 4, 26].

The prolongator $P_C = R_C^T$ is built starting from a *tentative prolongator* $P \in \Re^{n \times n_C}$, defined as

$$P = (p_{ij}), \quad p_{ij} = \begin{cases} 1 & \text{if } i \in V_C^j \\ 0 & \text{otherwise} \end{cases} . \tag{2}$$

P_C is obtained by applying to P a smoother $S \in \Re^{n \times n}$:

$$P_C = SP, \tag{3}$$

in order to remove oscillatory components from the range of the prolongator and hence to improve the convergence properties of the multi-level Schwarz method [1, 25]. A simple choice for S is the damped Jacobi smoother:

$$S = I - \omega D^{-1} A, \tag{4}$$

where the value of ω can be chosen using some estimate of the spectral radius of $D^{-1}A$ [1].

5 Getting Started

We describe the basics for building and applying MLD2P4 one-level and multi-level (i.e., AMG) preconditioners with the Krylov solvers included in PSBLAS [16]. 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_bld`.

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

   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 routine `aply`. When using the PSBLAS Krylov solvers, this step is completely transparent to the user, since `aply` 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	`'NOPREC'`	Considered only 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 he preconditioner is set to 1.
Block Jacobi	`'BJAC'`	Block-Jacobi with ILU(0) on the local blocks.
Additive Schwarz	`'AS'`	Restricted Additive Schwarz (RAS), 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 multi-level preconditioner available in the real double precision version of MLD2P4 (see Table 1). This preconditioner is chosen by simply specifying `'ML'` as 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 [16].

The setup and application of the default multi-level 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 multi-level 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 no pre-smoother and 2 Gauss-Seidel sweeps as 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. Note the use of the parameter `pos` to specify a property only for the pre-smoother or the post-smoother (see Section 6.2 for more details). Note also that a Krylov method different from CG must be used to solve the preconditioned system, since the preconditione in nonsymmetric. 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. 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 multi-level 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(P,'ML',info)
!
! build the preconditioner
  call P%hierarchy_bld(A,desc_A,P,info)
  call P%smoothers_bld(A,desc_A,P,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(P,info)
!
! deallocate other data structures
  ... ...
!
! exit the parallel environment
  call psb_exit(ictxt)
  stop
```

Figure 2: setup and application of the default multi-level 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(P,'ML',info)
  call_P%set(P,'SMOOTHER_TYPE','BJAC',info)
  call P%set(P,'COARSE_SOLVE','BJAC',info)
  call P%set(P,'COARSE_SWEEPS',8,info)
  call P%hierarchy_bld(A,desc_A,P,info)
  call P%smoothers_bld(A,desc_A,P,info)
... ...
```

Figure 3: setup of a multi-level preconditioner

```
... ...
! build a W-cycle preconditioner with 2 Gauss-Seidel sweeps as
! post-smoother (and no pre-smoother), a distributed coarsest
! matrix, and MUMPS as coarsest-level solver
  call P%init(P,'ML',info)
  call P%set('ML_TYPE','WCYCLE',info)
  call P%set('SMOOTHER_TYPE','GS',info)
  call P%set('SMOOTHER_SWEEPS',0,info,pos='PRE')
  call P%set('SMOOTHER_SWEEPS',2,info,pos='POST')
  call P%set('COARSE_SOLVE','MUMPS',info)
  call P%set('COARSE_MAT','DIST',info)
  call P%hierarchy_bld(A,desc_A,P,info)
  call P%smoothers_bld(A,desc_A,P,info)
... ...
! solve Ax=b with preconditioned CG
  call psb_krylov('BICGSTAB',A,P,b,x,tol,desc_A,info)
```

Figure 4: setup of a multi-level preconditioner

```
... ...
! set RAS with overlap 2 and ILU(0) on the local blocks
  call P%init(P,'AS',info)
  call P%set(P,'SUB_OVR',2,info)
  call P%bld(A,desc_A,P,info)
... ...
```

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

6 User Interface

The basic user interface of MLD2P4 consists of eight routines. The six routines `init`, `set`, `hierarchy_bld`, `smoothers_bld`, `bld`, and `apply` encapsulate all the functionalities for the setup and the application of any multi-level and one-level preconditioner implemented in the package. The routine `free` deallocates the preconditioner data structure, while `descr` prints a description of the preconditioner setup by the user.

All the routines are available as methods of the preconditioner object. For each routine, 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 routine, 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 $w = M^{-1}v$ 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 routine is given in the remainder of this section.

6.1 Subroutine init

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

This routine 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 routine can be also invoked as follows:

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

6.2 Subroutine set

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

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

Arguments

`what` `integer, intent(in)` *or* `character(len=*)`.
The parameter to be set. It can be specified by a predefined constant, or through its name; the string is case-insensitive. See also 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 multi-level 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 multi-level 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.

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

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

However, in this case the optional arguments `ilev`, `ilmax`, and `pos` 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 multi-level cycle and how many cycles must be applied;
2. the aggregation algorithm;
3. the coarse-space correction at the coarsest level (for multi-level preconditioners only);
4. the smoother of the multi-level 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 (`mld_smoother_type_`) and a local solver (`mld_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 `mld_smoother_type_` to appropriate values (see Tables 7), i.e., without setting `mld_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, and vice versa; therefore, setting the solver after the layout may change the layout, and setting the layout after the solver may change the solver, if the choices of the two parameters do not agree.

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.

what	DATA TYPE	val	DEFAULT	COMMENTS
mld_ml_cycle_ ML_CYCLE	character(len=*)	'VCYCLE' 'WCYCLE' 'KCYCLE' 'MULT' 'ADD'	'VCYCLE'	Multi-level 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.
mld_outer_sweeps_ OUTER_SWEEPS	integer	Any integer number ≥ 1	1	Number of multi-level cycles.

Table 2: Parameters defining the multi-level cycle and the number of cycles to be applied.

what	DATA TYPE	val	DEFAULT	COMMENTS
mld_min_coarse_size_ 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).
mld_min_cr_ratio_ 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).
mld_max_levs_ 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).
mld_par_aggr_alg_ PAR_AGGR	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 $A + A^T$.
mld_aggr_type_ AGGR_TYPE	character(len=*)	'VMB'	'VMB'	Type of aggregation algorithm: currently, the scalar aggregation algorithm by Vaněk, Mandel and Brezina is implemented [27].
mld_aggr_prol_ 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 maximum 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
mld_aggr_ord_ 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.
mld_aggr_thresh_ AGGR_THRESH	real(*kind_parameter*)	Any real number $\in [0,1]$	0.05	The threshold θ in the aggregation algorithm (see Note).
mld_aggr_omega_alg_ AGGR_OMEGA_ALG	character(len=*)	'EIG_EST' 'USER_CHOICE'	'EIG_EST'	How the damping parameter ω in the smoothed aggregation is obtained: either via an estimate of the spectral radius of $D^{-1}A$, or explicily specified by the user.
mld_aggr_eig_ AGGR_EIG	character(len=*)	'A_NORMI'	'A_NORMI'	How to estimate the spectral radius of $D^{-1}A$. Currently only the infinity norm estimate is available.
mld_aggr_omega_val_ AGGR_OMEGA_VAL	real(*kind_parameter*)	Any real number > 0	$4/(3\rho(D^{-1}A))$	Damping parameter ω in the smoothed aggregation algorithm. It must be set by the user if USER_CHOICE was specified for mld_aggr_omega_alg_, otherwise it is computed by the library, using the selected estimate of the spectral radius $\rho(D^{-1}A)$ of $D^{-1}A$.
mld_aggr_filter_ AGGR_FILTER	character(len=*)	'FILTER' 'NOFILTER'	'NOFILTER'	Matrix used in computing the smoothed prolongator: filtered or unfiltered.
Note. Different thresholds at different levels, such as those used in [27, 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
mld_coarse_mat_ COARSE_MAT	character(len=*)	'DIST' 'REPL'	'REPL'	Coarsest matrix layout: distributed among the processes or replicated on each of them.
mld_coarse_solve_ COARSE_SOLVE	character(len=*)	'MUMPS' 'UMF' 'SLU' 'SLUDIST' 'JACOBI' 'GS' 'BJAC'	See Note 1	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 (see Note 2) 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, 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 value UMFPACK and SuperLU_Dist are available only in double precision.
mld_coarse_subsolve_ COARSE_SUBSOLVE	character(len=*)	'ILU' 'ILUT' 'MILU' 'MUMPS' 'SLU' 'UMF'	See Note 1	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 1. Defaults for mld_coarse_solve_ and mld_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. **Note 2.** The hybrid Gauss-Seidel method is between the Gauss-Seidel and Jacobi methods: at each iteration, the processes use the most recent values of their own local variables, and the values of the non-local variables computed at the previous iteration.				

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

what	DATA TYPE	val	DEFAULT	COMMENTS
mld_coarse_sweeps_ COARSE_SWEEPS	integer	Any integer number > 0	10	Number of sweeps when JACOBI, GS or BJAC is chosen as coarsest-level solver.
mld_coarse_fillin_ COARSE_FILLIN	integer	Any integer number ≥ 0	0	Fill-in level p of the ILU factorizations.
mld_coarse_iluthrs_ 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
mld_smoother_type_ SMOOTHER_TYPE	character(len=*)	'JACOBI' 'GS' 'BGS' 'BJAC' 'AS'	'FBGS'	Type of smoother used in the multi-level preconditioner: point-Jacobi, hybrid (forward) Gauss-Seidel, hybrid backward Gauss-Seidel, block-Jacobi, and Additive Schwarz. See Note for details on hybrix Gauss-Seidel. It is ignored by one-level preconditioners.
mld_sub_solve_ SUB_SOLVE	character(len=*)	'JACOBI' 'GS' 'BGS' 'ILU' 'ILUT' 'MILU' 'MUMPS' 'SLU' 'UMF'	GS and BGS for pre- and post-smoothers of multi-level 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.
mld_moother_sweeps_ SMOOTHER_SWEEPS	integer	Any integer number ≥ 0	1	Number of sweeps of the smoother or one-level preconditioner. In the multi-level case, no pre-smother or post-smoother is used if this parameter is set to 0 together with pos='PRE' or pos='POST, respectively.
mld_sub_ovr_ SUB_OVR	integer	Any integer number ≥ 0	1	Number of overlap layers, for Additive Schwarz only.
Note. The hybrid Gauss-Seidel method is between the Gauss-Seidel and Jacobi methods: at each iteration, the processes use the most recent values of their own local variables, and the values of the non-local variables computed at the previous iteration.				

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

what	DATA TYPE	val	DEFAULT	COMMENTS
mld_sub_restr_ 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.
mld_sub_prol_ 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.
mld_sub_fillin_ SUB_FILLIN	integer	Any integer number ≥ 0	0	Fill-in level p of the incomplete LU factorizations.
mld_sub_iluthrs_ SUB_ILUTHRS	real(*kind_parameter*)	Any real number ≥ 0	0	Drop tolerance t in the ILU(p,t) factorization.

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

6.3 Subroutine bld

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

This routine builds the one-level preconditioner `p` according to the requirements made by the user through the routines `init` and `set` (see Sections 6.4 and 6.5 for multi-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 [16].

`desc_a` `type(psb_desc_type), intent(in).`
The communication descriptor of `a`. See the PSBLAS User's Guide for details [16].

`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 routine can be also invoked as follows:

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

In this case, the routine can be used to build multi-level preconditioners too.

6.4 Subroutine hierarchy_bld

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

This routine builds the hierarchy of matrices and restriction/prolongation operators for the multi-level preconditioner `p`, according to the requirements made by the user through the routines `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 [16].

`desc_a` `type(psb_desc_type), intent(in)`.
The communication descriptor of `a`. See the PSBLAS User's Guide for details [16].

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

6.5 Subroutine smoothers_bld

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

This routine builds the smoothers and the coarsest-level solvers for the multi-level preconditioner `p`, according to the requirements made by the user through the routines `init` and `set`, and based on the aggregation hierarchy produced by a previous call to `hierarchy_bld` (see Section 6.4).

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 [16].

`desc_a` `type(psb_desc_type), intent(in).`
The communication descriptor of `a`. See the PSBLAS User's Guide for details [16].

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

6.6 Subroutine apply

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

This routine computes $y = op(M^{-1})\,x$, where M 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 routine `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 $op(M^{-1}) = M^{-1}$; if `trans` = `'T','t'` then $op(M^{-1}) = M^{-T}$ (transpose of M^{-1}); if `trans` = `'C','c'` then $op(M^{-1}) = M^{-C}$ (conjugate transpose of M^{-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 routine can be also invoked as follows:

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

6.7 Subroutine free

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

This routine 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 routine can be also invoked as follows:

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

6.8 Subroutine descr

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

This routine prints a description of the preconditioner `p` to the standard output or to a file. It must be called after `hierachy_bld` and `smoothers_bld`, or `bld`, 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 routine can be also invoked as follows:

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

7 Adding smoothers and solvers to MLD2P4

Da ampliare e completare - SALVATORE.

Completely new smoother and/or solver classes derived from the base objects in the library may be used without recompiling the library itself. Once the new smoother/solver class has been developed, the user can declare a variable of that new type in the application, and pass that variable to the `p%set(solver,info)` call; the new solver object is then dynamically included in the preconditioner structure.

If the user has developed a new type of smoother and/or solver by extending one of the base MLD2P4 types, and has declared a variable of the new type in the main program, it is possible to pass the new smoother/solver variable to the setup routine as follows:

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

In this way, the variable will act as a *mold* to which the preconditioner will conform, even though the MLD2P4 library is not modified, and thus has no direct knowledge about the new type.

`smootherclass(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.

8 Error Handling

The error handling in MLD2P4 is based on the PSBLAS (version 2) 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 [16].

A License

The MLD2P4 is freely distributable under the following copyright terms:

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] X. C. Cai, O. B. Widlund, *Domain Decomposition Algorithms for Indefinite Elliptic Problems*, SIAM Journal on Scientific and Statistical Computing, 13, 1, 1992, 243–258.

[7] T. Chan and T. Mathew, *Domain Decomposition Algorithms*, in A. Iserles, editor, Acta Numerica 1994, 61–143. Cambridge University Press.

[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.

[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] 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 SISC, volume 37, number 3, pages A1452-A1474. (See also `http://mumps.enseeiht.fr`)

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

[12] 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, 1990, 1–17.

[13] 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, 1988, 1–17.

[14] J. J. Dongarra and R. C. Whaley, *A User's Guide to the BLACS v. 1.1*, Lapack Working Note 94, Tech. Rep. UT-CS-95-281, University of Tennessee, March 1995 (updated May 1997).

[15] E. Efstathiou, J. G. Gander, *Why Restricted Additive Schwarz Converges Faster than Additive Schwarz*, BIT Numerical Mathematics, 43, 2003, 945–959.

[16] S. Filippone, A. Buttari, *PSBLAS-3.0 User's Guide. A Reference Guide for the Parallel Sparse BLAS Library*, 2012, available from `http://www.ce.uniroma2.it/psblas/`.

[17] Salvatore Filippone and Alfredo Buttari. *Object-Oriented Techniques for Sparse Matrix Computations in Fortran 2003.* ACM Trans. on Math Software, 38(4), 2012.

[18] 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.

[19] 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.

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

[21] 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.

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

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

[24] 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.

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

[26] 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.

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