

# MLD2P4

## User's and Reference Guide

---

*A guide for the Multi-Level Domain Decomposition  
Parallel Preconditioners Package based on PSBLAS*

**Pasqua D'Ambra**  
ICAR-CNR, Naples, Italy

**Daniela di Serafino**  
Second University of Naples, Italy

**Salvatore Filippone**  
University of Rome "Tor Vergata", Italy

Software version: 1.2.1  
March 10, 2011



## Abstract

MLD2P4 (MULTI-LEVEL DOMAIN DECOMPOSITION PARALLEL PRECONDITIONERS PACKAGE BASED ON PSBLAS) is a package of parallel algebraic multi-level preconditioners. It implements various versions of one-level additive and of multi-level additive and hybrid Schwarz algorithms. 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, with a symmetric sparsity pattern.

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 conjunction with the Krylov solvers available in this framework. MLD2P4 enables the user to easily specify different aspects of a generic algebraic multilevel Schwarz preconditioner, thus allowing to search for the “best” preconditioner for the problem at hand.

The package has been designed employing object-based techniques, using Fortran 95, with interfaces to additional third party libraries such as UMFPACK, SuperLU and SuperLU\_Dist, that can be exploited in building multi-level preconditioners. The parallel implementation is based on a Single Program Multiple Data (SPMD) paradigm for distributed-memory architectures; the inter-process data 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

<b>Abstract</b>	<b>i</b>
<b>1 General overview</b>	<b>1</b>
<b>2 Code distribution</b>	<b>3</b>
<b>3 Configuring and building MLD2P4</b>	<b>4</b>
3.1 Prerequisites . . . . .	4
3.2 Optional third party libraries . . . . .	5
3.3 Configuration options . . . . .	5
3.4 Bug reporting . . . . .	8
3.5 Example and test programs . . . . .	8
<b>4 Multi-level Schwarz preconditioners</b>	<b>9</b>
4.1 Background . . . . .	9
4.2 Algebraic multi-level Schwarz preconditioners based on smoothed aggregation . . . . .	10
<b>5 Getting started</b>	<b>15</b>
5.1 Examples . . . . .	17
<b>6 User interface</b>	<b>20</b>
6.1 Subroutine mld_precinit . . . . .	21
6.2 Subroutine mld_precset . . . . .	22
6.3 Subroutine mld_precbld . . . . .	27
6.4 Subroutine mld_precaply . . . . .	28
6.5 Subroutine mld_precfree . . . . .	29
6.6 Subroutine mld_precdescr . . . . .	30
<b>7 Error handling</b>	<b>31</b>
<b>A License</b>	<b>32</b>
<b>References</b>	<b>33</b>



## 1 General overview

The MULTI-LEVEL DOMAIN DECOMPOSITION PARALLEL PRECONDITIONERS PACKAGE BASED ON PSBLAS (MLD2P4) provides *multi-level Schwarz preconditioners* [24], 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 with a symmetric sparsity pattern. These preconditioners have the following general features:

- both *additive and hybrid multilevel* variants are implemented, i.e. variants that are additive among the levels and inside each level, and variants that are multiplicative among the levels and additive inside each level; the basic Additive Schwarz (AS) preconditioners are obtained by considering only one level;
- a *purely algebraic* approach is used to generate a sequence of coarse-level corrections to a basic AS preconditioner, 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 [2, 28].

The package is written in *Fortran 95*, following an *object-based approach* 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 for distributed-memory architectures. Single and double precision implementations of MLD2P4 are available for both the real and the complex case, that 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]. PSBLAS is a library originally developed to address the parallel implementation of iterative solvers for sparse linear system, by providing basic linear algebra operators and data management facilities for distributed sparse matrices; it also includes parallel Krylov solvers, built on the top of the basic PSBLAS kernels. The preconditioners available in MLD2P4 can be used with these Krylov solvers. 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 and extensibility in the development of the preconditioner package. On the other hand, the implementation of MLD2P4 has led to some revisions and extensions of the PSBLAS kernels, leading to the recent PSBLAS 2.0 version [17]. The inter-process communication required by MLD2P4 is encapsulated into the PSBLAS routines, except few cases where MPI [25] is explicitly called. 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 and easy-to-use 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. On the other hand, the routines of the middle and lower layer can be used and extended by expert users to build new versions of multi-level Schwarz preconditioners. We provide here a description of the upper-layer routines, but not of the medium-layer ones.

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 description of multi-level Schwarz preconditioners based on smoothed aggregation is provided in Section 4, to help the users in choosing among the different preconditioners implemented in MLD2P4. The basics for building and applying the preconditioners with the Krylov solvers implemented in PSBLAS are reported in Section 5, where fragments from the Fortran 95 codes of a few sample programs are also shown. A reference guide for the upper-layer routines of MLD2P4, which are the user interface, is provided in Section 6. The error handling mechanism used by the package is briefly described in Section 7. 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.

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.

### 3 Configuring and building MLD2P4

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 95, with some interfaces to external libraries in C; the Fortran compiler must support the Fortran 95 standard plus the extension TR15581, which enhances the usability of `ALLOCATABLE` variables. Most modern Fortran compilers support this language level. In particular, this is supported by the GNU Fortran compiler as of version 4.2.0; however we recommend to use the latest available release (4.4.2 at the time of this writing). 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** [13, 14, 20] Many vendors provide optimized versions of the Basic Linear Algebra Subprograms; if no vendor version is available for a given platform, the ATLAS software (<http://math-atlas.sourceforge.net/>) may be employed. The reference BLAS from Netlib (<http://www.netlib.org/blas>) are meant to define the standard behaviour of the BLAS interface, so they are not optimized for any specific platform, 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 the UMFPACK or SuperLU third party libraries.

**MPI** [19, 25] A version of MPI is available on most high-performance computing systems; only version 1.1 is required.

**BLACS** [15] The Basic Linear Algebra Communication Subprograms are available in source form from <http://www.netlib.org/blacs>; some vendors include them in their parallel computing support libraries.

**PSBLAS** [17, 18] Parallel Sparse BLAS is available from <http://www.ce.uniroma2.it/psblas>; version 2.3.1 (or later) is required. Indeed, all the prerequisites listed so far are also prerequisites of PSBLAS. To build the MLD2P4 library it is necessary to get access to the source PSBLAS directory employed to build the version under use; after the MLD2P4 build process completes, only the compiled form of the PSBLAS library is necessary to build user applications.

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** [11] A sparse direct factorization package available from <http://www.cise.ufl.edu/research/sparse/umfpack/>; provides serial factorization and triangular system solution for double precision real and complex data. We have tested versions 4.4 and 5.1.

**SuperLU** [12] A sparse direct factorization package available from <http://crd.lbl.gov/~xiaoye/SuperLU/>; provides serial factorization and triangular system solution for single and double precision, real and complex data. We have tested versions 3.0 and 3.1.

**SuperLU\_Dist** [21] A sparse direct factorization package available from the same site as SuperLU; provides parallel factorization and triangular system solution for double precision real and complex data. We have tested version 2.1.

### 3.3 Configuration options

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=/opt/packages/psblas-2.3.1
```

which assumes that the various MPI compilers and support libraries are available in the standard directories on the system, and specifies only the PSBLAS build 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 1.2 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:

<code>-h, --help</code>	display this help and exit
<code>--help=short</code>	display options specific to this package
<code>--help=recursive</code>	display the short help of all the included packages
<code>-V, --version</code>	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 Packages:

```

--with-PACKAGE[=ARG]   use PACKAGE [ARG=yes]
--without-PACKAGE      do not use PACKAGE (same as --with-PACKAGE=no)
--with-psblas          The source directory for PSBLAS, for example,

```

```

--with-psblas=/opt/packages/psblas-2.3.1
--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-umfpack=LIBNAME Specify the library name for UMFPACK library.
                    Default: "-lumfpack -lamd"
--with-umfpackdir=DIR Specify the root directory for UMFPACK and AMD
                    installation
--with-superlu=LIBNAME Specify the library name for SUPERLU library.
                    Default: "-lslu"
--with-superludir=DIR Specify the directory for SUPERLU library and
                    includes.
--with-superludist=LIBNAME
                    Specify the libname for SUPERLUDIST library.
                    Requires you also specify SuperLU. Default: "-lslud"
--with-superludistdir=DIR
                    Specify the directory for SUPERLUDIST library and
                    includes.

```

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

```

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>.

Thus, a sample build specifying only the location of the BLACS and the use of UMF-PACK from U. Florida Sparse Suite might be:

```
./configure --with-psblas=/home/user/psblas-2.3.1/ \
--with-blacs=-lmpiblacs --with-umfpackdir=/usr/local/SparseSuite/
```

Note that most options are taken up from the PSBLAS configure output and hence need not be specified. Once the configure script has completed the execution, it will have generated the file `Make.inc` which will then be used by all Makefiles in the directory tree.

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 `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 Schwarz preconditioners

### 4.1 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 [8, 9, 24].

*Additive Schwarz* (AS) 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., [9, 24]). 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. a number of iterations independent of the number of submatrices, can be obtained by correcting the preconditioner using a suitable approximation of the original matrix in a coarse space, to globally couple the information related to the single submatrices. This process is called *coarse-level correction* and requires the solution of a linear system involving the coarse matrix. The combination of basic (one-level) Schwarz preconditioners with a coarse-level correction leads to the so-called *two-level Schwarz* preconditioners. In this context, the one-level preconditioner is often called ‘smoother’. The smoother and the coarse-level correction can be applied recursively to the coarse-level system, thus obtaining *multi-level* preconditioners. Different multi-level preconditioners are obtained by varying the choice of the smoother and of the coarse-level correction, and the way they are combined [24].

It is worth noting that optimal preconditioners do not necessarily correspond to minimum execution times. In order to obtain effective multi-level preconditioners, a tradeoff between optimality of convergence and the cost of building and applying the coarse-level corrections must be achieved. The choice of the number of levels also affects the performance 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-level 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-level corrections using only matrix in-

formation. 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 [26].

MLD2P4 uses a pure algebraic approach, based on the *smoothed aggregation* algorithm [2, 28]. A decoupled version of this algorithm is implemented, where the smoothed aggregation is applied locally to each submatrix [27]. In the next subsection 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 [3, 4, 5, 10, 24].

## 4.2 Algebraic multi-level Schwarz preconditioners based on smoothed aggregation

We outline a general framework for building and applying algebraic multilevel preconditioners, which encompasses all the multi-level preconditioners available in MLD2P4.

Given the linear system (1), where  $A = (a_{ij}) \in \mathfrak{R}^{n \times n}$  is a nonsingular sparse matrix with a symmetric nonzero pattern, we assume as finest index space the set of row (column) indices of  $A$ ,  $\Omega = \{1, 2, \dots, n\}$ . An algebraic multilevel method generates a hierarchy of index spaces and a corresponding hierarchy of matrices,

$$\Omega^1 \equiv \Omega \supset \Omega^2 \supset \dots \supset \Omega^{nlev}, \quad A^1 \equiv A, A^2, \dots, A^{nlev}, \quad (2)$$

by using the information contained in  $A$ , without assuming any knowledge of the geometry of the problem from which  $A$  originates. A vector space  $\mathfrak{R}^{n_k}$  is associated with  $\Omega^k$ , where  $n_k$  is the size of  $\Omega^k$ . In the following we use the term *contiguous* for two index spaces, vector spaces or associated matrices that correspond to subsequent levels  $k$  and  $k + 1$  of the hierarchy.

For each  $k < nlev$ , the method builds two maps between two contiguous vector spaces, i.e. a *prolongation* and a *restriction* operator

$$P^k : \mathfrak{R}^{n_{k+1}} \longrightarrow \mathfrak{R}^{n_k}, \quad R^k : \mathfrak{R}^{n_k} \longrightarrow \mathfrak{R}^{n_{k+1}}; \quad (3)$$

it is common to choose  $R^k = (P^k)^T$ . The matrix  $A^{k+1}$  is computed by exploiting the previous maps, usually according to the *Galerkin approach*, i.e.

$$A^{k+1} = R^k A^k P^k. \quad (4)$$

A smoother  $S^k$  is set, which provides an approximation of the inverse of the matrix  $A^k$ . At the coarsest level, a direct solver is generally considered to obtain the inverse of  $A^{nlev}$ . The process just described corresponds to the so-called *build phase* of the preconditioner and is sketched in Figure 1.

The components produced in the build phase may be combined in several ways to obtain different multilevel preconditioners (see [24, Chapters 2 and 3]); this is done in the *application phase*, i.e. in the computation of a vector of type  $w = M^{-1}v$ , where  $M$  denotes the preconditioner, usually within an iteration of a Krylov solver. Two basic types of preconditioners can be identified, i.e. the *additive* and the *multiplicative*

```

 $A^1 \leftarrow A, \Omega^1 \leftarrow \Omega$ 
for  $k = 1, nlev - 1$  do
  generate  $\Omega^{k+1}$  from  $\Omega^k$ 
  define  $P^k$  (and  $R^k = (P^k)^T$ )
  compute  $A^{k+1} = R^k A^k P^k$ 
  set up  $S^{k+1}$ 
end for

```

Figure 1: Build phase of a multilevel preconditioner.

```

 $v^1 \leftarrow v$ 
! Apply restrictor at all levels but the coarsest
for  $k = 1, nlev - 1$  do
   $v^{k+1} \leftarrow R^k v^k$ 
end for
! Apply smoother at all levels
for  $k = 1, nlev$  do
   $y^k \leftarrow S^k v^k$ 
end for
! Apply prolongator and update
! at all levels but the coarsest
for  $k = nlev - 1, 1, -1$  do
   $y^k \leftarrow y^k + P^k y^{k+1}$ 
end for
 $w \leftarrow y^1$ 

```

Figure 2: Application phase of an additive multilevel preconditioner.

one. In the additive case, at each level the smoother and the coarse-level correction are applied to the restriction of the vector  $v$  to that level, and the resulting vectors are added; this leads to the algorithm shown in Figure 2. In the multiplicative case, at each level the smoother and the coarse-level correction are applied in turn, the former on a vector resulting from the application of the latter and/or vice versa. An example of such a combination, where the smoother is applied before and after the coarse-level correction, is given in Figure 3; this is known as *symmetrized multiplicative multilevel preconditioner* or, more generally, *V-cycle*. *Multiplicative multi-level preconditioners with pre-smoothing only* are obtained by neglecting the last three statements in the second loop, while multiplicative multi-level ones *with post-smoothing only* correspond to setting  $y^k = 0$  in the first loop, i.e. to performing only the restriction  $v^{k+1} \leftarrow R^k v^k$  in the first loop and to changing into  $y^k \leftarrow P^k y^{k+1}$  the first statement of the second loop.

At each level  $k < nlev$ , MLD2P4 implements, as smoothers, domain decomposition AS preconditioners [6, 9, 24]. Therefore, the multilevel preconditioners based on the

```

 $v^1 \leftarrow v$ 
! Apply smoother and restrictor
! at all levels but the coarsest
for  $k = 1, nlev - 1$  do
   $y^k \leftarrow S^k v^k$ 
   $r^k \leftarrow v^k - A^k y^k$ 
   $v^{k+1} \leftarrow R^k r^k$ 
end for
! Apply smoother at coarsest level
 $y^{nlev} = S^{nlev} v^{nlev}$ 
! At all levels but the coarsest, interpolate  $y^k$ 
! update the residual, apply the smoother and update
 $y^k$ 
for  $k = nlev - 1, 1, -1$  do
   $y^k \leftarrow y^k + P^k y^{k+1}$ 
   $r^k \leftarrow v^k - A^k y^k$ 
   $r^k \leftarrow S^k r^k$ 
   $y^k \leftarrow y^k + r^k$ 
end for
 $w \leftarrow y^1$ 

```

Figure 3: Application phase of a symmetrized multiplicative multilevel preconditioner.

multiplicative framework are referred to as *hybrid*, i.e. multiplicative among the levels and additive inside any single level. The point point-Jacobi smoother is available too; furthermore, multiple sweeps of each smoother can be performed.

In the AS methods the index space  $\Omega^k$  is divided into  $m_k$  subsets  $\Omega_i^k$  of size  $n_{k,i}$ . The subsets may have an overlap  $\delta$ , which is defined using the adjacency graph  $(\Omega^k, E^k)$  of the matrix  $A^k$ , where  $\Omega^k$  is the vertex set and  $E^k = \{(i, j) : a_{ij}^k \neq 0\}$  is the edge set. Two vertices are called adjacent if there is an edge connecting them. For any integer  $\delta > 0$ , a  $\delta$ -overlap partition of  $\Omega^k$  is defined recursively as follows. Given a 0-overlap partition of  $\Omega^k$ , i.e. a set of  $m_k$  disjoint nonempty sets of  $\Omega_i^k(0) \subset \Omega^k$  such that  $\cup_{i=1}^{m_k} \Omega_i^k(0) = \Omega^k$ , a  $\delta$ -overlap partition of  $\Omega^k$  is obtained by considering the sets  $\Omega_i^k(\delta) \supset \Omega_i^k(\delta - 1)$  obtained by including the vertices that are adjacent to any vertex in  $\Omega_i^k(\delta - 1)$ . In MLD2P4 the number of sets  $\Omega_i^k$  matches the number of available processors and each processor is assigned one of these set. The 0-overlap partition at the finest level is determined by the (general row-block) distribution of the matrix to be preconditioned (see [17]).

For each  $i$  we consider the restriction operator  $R_i^k : \mathfrak{R}^{n_k} \longrightarrow \mathfrak{R}^{n_{k,i}}$  that maps a vector  $v^k$  to the vector  $v_i^k$  made of the components of  $v^k$  with indices in  $\Omega_i^k$ , 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 a restriction of  $A^k$  to the index space  $\Omega_i^k$ . The *classical AS* preconditioner is

defined as

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

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 application of  $S_{AS}^k$  to a vector  $w^k \in \mathfrak{R}^{n_k}$ , i.e. the computation of  $z^k = S_{AS}^k w^k$ , requires

- the restriction of  $w^k$  to the subspaces  $\mathfrak{R}^{n_{k,i}}$ , 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 [7, 16]. Direct solvers based on the LU factorization as well as approximate solvers based on the ILU factorization or on the point- and block-Jacobi iterative methods are implemented as smoothers at the coarsest level.

In MLD2P4 the hierarchy of index spaces (see (2)) and the corresponding mapping operators (see (3)) are built by applying the *smoothed aggregation* technique [2, 22, 28]. The basic idea is to build the coarse set of indices  $\Omega^{k+1}$  by suitably grouping the indices of  $\Omega^k$  into disjoint subsets (aggregates) and to define a simple “tentative” prolongator  $\tilde{P}^k$  whose range should contain the near null space of  $A^k$ ; the final interpolation operator  $P^k$  is formed by applying a suitable smoother to  $\tilde{P}^k$ , in order to obtain low energy coarse basis functions and hence good convergence rates.

To build the aggregates we have implemented the coarsening algorithm sketched in [5]. According to [28], a modification of this algorithm has been actually considered, in which each aggregate  $N_r^k$  is made of vertices of  $\Omega^k$  that are *strongly coupled* to a certain root vertex  $r \in \Omega^k$ , i.e.

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

for a given  $\theta \in [0, 1]$ . Since this algorithm has a sequential nature, a *decoupled* version of it has been considered, where each processor  $i$  independently applies the algorithm to the set of vertices  $\Omega_i^k(0)$  assigned to it according to the initial data distribution [27]. 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 [1, 4, 5, 27].

The tentative prolongator  $\tilde{P}^k$  is piecewise constant interpolation operator, defined as

$$\tilde{P}^k = (\tilde{p}_{ij}^k), \quad \tilde{p}_{ij}^k = \begin{cases} 1 & \text{if } i \in N_j^k \\ 0 & \text{otherwise} \end{cases} . \quad (5)$$

Then  $P^k$  is computed as

$$P^k = S\tilde{P}^k, \quad (6)$$

where

$$S = I - \omega D^{-1}A \quad (7)$$

is the damped Jacobi smoother and the value of  $\omega$  can be chosen using some estimate of the spectral radius of  $D^{-1}A$  [2].

## 5 Getting started

We describe the basics for building and applying MLD2P4 one-level and multi-level Schwarz preconditioners with the Krylov solvers included in PSBLAS [17]. The following steps are required:

1. *Declare the preconditioner data structure.* It is a derived data type, `mld_xprec_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 `mld_precinit`, which also sets defaults for each preconditioner type selected by the user. The defaults associated to each preconditioner type are given in Table 1, where the strings used by `mld_precinit` 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 `mld_precset`. This routine must be called only if the user wants to modify the default values of the parameters associated to the selected preconditioner type, to obtain a variant of the preconditioner. Examples of use of `mld_precset` 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 2-5.
4. *Build the preconditioner for a given matrix.* This is performed by the routine `mld_precbld`.
5. *Apply the preconditioner at each iteration of a Krylov solver.* This is performed by the routine `mld_precaply`. When using the PSBLAS Krylov solvers, this step is completely transparent to the user, since `mld_precaply` is called by the PSBLAS routine implementing the Krylov solver (`psb_krylov`).
6. *Free the preconditioner data structure.* This is performed by the routine `mld_precfree`. This step is complementary to step 1 and should be performed when the preconditioner is no more used.

A detailed description of the above routines is given in Section 6. Examples showing the basic use of MLD2P4 are reported in Section 5.1.

Note that the Fortran 95 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).

TYPE	STRING	DEFAULT PRECONDITIONER
No preconditioner	'NOPREC'	Considered only to use the PSBLAS Krylov solvers with no preconditioner.
Point-Jacobi	'JACOBI', 'DIAG'	Point-Jacobi, i.e. diagonal, preconditioner (see note).
Block-Jacobi	'BJAC'	Block-Jacobi preconditioner, with ILU(0) on the local blocks.
Additive Schwarz	'AS'	Restricted Additive Schwarz (RAS) preconditioner, with overlap 1 and ILU(0) on the local blocks.
Multilevel	'ML'	Multi-level hybrid preconditioner (additive inside each level and multiplicative through the levels) with pre- and post-smoothing, i.e. symmetrized multiplicative. Number of levels: 2. Smoother: RAS (1 sweep) with overlap 1 and ILU(0) on the local blocks. Aggregation: decoupled smoothed aggregation with threshold $\theta = 0$ . Coarsest matrix: distributed among the processors. Coarsest-level solver: 4 sweeps of the block-Jacobi solver, with LU or ILU(0) factorization of the blocks (UMFPACK for the double precision versions and SuperLU for the single precision ones, if the packages have been installed; ILU(0), otherwise).
<b>Note:</b> The string 'DIAG' has been considered too, to ensure the compatibility of MLD2P4 with codes written to use the point-Jacobi preconditioner available in PSBLAS (see Remark 3, p. 19).		

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

**Remark 1.** The coarsest-level solver used by the default two-level preconditioner has been chosen by taking into account that, on parallel machines, it often leads to the smallest execution time when applied to linear systems coming from finite-difference discretizations of basic elliptic PDE problems, considered as standard tests for multi-level Schwarz preconditioners [4, 5]. However, this solver does not necessarily correspond to the smallest number of iterations of the preconditioned Krylov method, which is usually obtained by applying a direct solver to the coarsest-level system, e.g. based on the LU factorization (see Section 6 for the coarsest-level solvers available in MLD2P4).

**Remark 2.** The include path for MLD2P4 must override those for PSBLAS, i.e. the former must come first in the sequence passed to the compiler, as the MLD2P4 version of the Krylov solver interfaces must override that of PSBLAS. This will change in the future when the support for the `class` statement becomes widespread in Fortran compilers.

## 5.1 Examples

The code reported in Figure 4 shows how to set and apply the default multi-level preconditioner available in the real double precision version of MLD2P4 (see Table 1; we assume here that UMFPACK has been installed). This preconditioner is chosen by simply specifying 'ML' as second argument of `mld_precinit` (a call to `mld_precset` is not needed) and is applied with the BiCGSTAB solver provided by PSBLAS. 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_m1.f90`, in the directory `examples/fileread` of the MLD2P4 tree (see Section 3.5). For details on the use of the PSBLAS routines, see the PSBLAS User's Guide [17].

The setup and application of the default multi-level preconditioners 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 Fortran 95 codes are available in `examples/fileread/`.

Different versions of multi-level preconditioners can be obtained by changing the default values of the preconditioner parameters. The code reported in Figure 5 shows how to set a three-level hybrid Schwarz preconditioner with post-smoothing only, which uses as smoother block-Jacobi with ILU(0) on the local blocks, has a coarsest matrix replicated on the processors, and solves the coarsest-level system with the LU factorization from UMFPACK [11]. The number of levels is specified by using `mld_precinit`; the other preconditioner parameters are set by calling `mld_precset`. Note that the type of multilevel framework (i.e. multiplicative among the levels with post-smoothing only) is not specified since it is the default set by `mld_precinit`.

Figure 6 shows how to set a three-level additive Schwarz preconditioner, which uses as smoother RAS, with overlap 1 and ILU(0) on the blocks, and applies five block-Jacobi sweeps, with LU from UMFPACK on the blocks, as distributed coarsest-level solver. Again, `mld_precset` is used only to set non-default values of the parameters (see Tables 2-5). In both cases, the construction and the application of the preconditioner are carried out as for the default multi-level preconditioner. The code fragments shown in in Figures 5-6 are included in the example program file `mld_dexample_m1.f90` too.

Finally, Figure 7 shows the setup of a one-level additive Schwarz preconditioner, i.e.

```

    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
    real(kind(1.d0))      :: b(:), x(:)
... ..
!
! initialize the parallel environment
    call psb_init(ictxt)
    call psb_info(ictxt,iam,np)
... ..
!
! read and assemble the matrix A and the right-hand side b
! using PSBLAS routines for sparse matrix / vector management
... ..
!
! initialize the default multi-level preconditioner, i.e. two-level
! symmetrized multiplicative Schwarz, using 1 sweep of RAS (with overlap 1
! and ILU(0) on the blocks) as smoother and 4 block-Jacobi sweeps (with
! UMFPACK LU on the blocks) as distributed coarse-level solver
    call mld_precinit(P,'ML',info)
!
! build the preconditioner
    call mld_precbld(A,desc_A,P,info)
!
! set the solver parameters and the initial guess
    ... ..
!
! solve Ax=b with preconditioned BiCGSTAB
    call psb_krylov('BICGSTAB',A,P,b,x,tol,desc_A,info)
    ... ..
!
! deallocate the preconditioner
    call mld_precfree(P,info)
!
! deallocate other data structures
    ... ..
!
! exit the parallel environment
    call psb_exit(ictxt)
    stop

```

Figure 4: Setup and application of the default multi-level Schwarz preconditioner.

RAS with overlap 2. The corresponding example program is available in `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`.

**Remark 3.** Any PSBLAS-based program using the basic preconditioners implemented in PSBLAS, i.e. the point-Jacobi (diagonal) and block-Jacobi ones, can use the point-Jacobi and block-Jacobi preconditioners implemented in MLD2P4 without any change in the code. The PSBLAS-based program must be only recompiled and linked to the MLD2P4 library.

```

... ..
! set a three-level hybrid Schwarz preconditioner with post-smoothing
! only, which uses 1 sweep of block-Jacobi (with ILU(0) on the blocks)
! as smoother, a coarsest matrix replicated on the processors, and the
! LU factorization from UMFPACK as coarsest-level solver
call mld_precinit(P,'ML',info,nlev=3)
call mld_precset(P,mld_smoother_pos_,'POST',info)
call mld_precset(P,mld_smoother_type_,'BJAC',info)
call mld_precset(P,mld_coarse_mat_,'REPL',info)
call mld_precset(P,mld_coarse_solve_,'UMF',info)
... ..

```

Figure 5: Setup of a hybrid three-level Schwarz preconditioner.

```

... ..
! set a three-level additive Schwarz preconditioner, which uses
! 1 sweep of RAS (with overlap 1 and ILU(0) on the blocks) as
! smoother and 5 block-Jacobi sweeps (with UMFPACK LU on the
! blocks) as distributed coarsest-level solver
call mld_precinit(P,'ML',info,nlev=3)
call mld_precset(P,mld_ml_type_,'ADD',info)
call mld_precset(P,mld_coarse_sweeps_,5,info)
... ..

```

Figure 6: Setup of an additive three-level Schwarz preconditioner.

```

... ..
! set RAS with overlap 2 and ILU(0) on the local blocks
call mld_precinit(P,'AS',info)
call mld_precset(P,mld_sub_ovr_,2,info)
... ..

```

Figure 7: Setup of a one-level Schwarz preconditioner.

## 6 User interface

The basic user interface of MLD2P4 consists of six routines. The four routines `mld_precinit`, `mld_precset`, `mld_precbld` and `mld_precaply` encapsulate all the functionalities for the setup and the application of any one-level and multi-level preconditioner implemented in the package. The routine `mld_precfree` deallocates the preconditioner data structure, while `mld_precedescr` prints a description of the preconditioner setup by the user.

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 `mld_xspmat_type` with  $x = \mathbf{s}$  for real single precision,  $x = \mathbf{d}$  for real double precision,  $x = \mathbf{c}$  for complex single precision,  $x = \mathbf{z}$  for complex double precision;
- the preconditioner data structure must be of type `mld_xprec_type`, with  $x = \mathbf{s}, \mathbf{d}, \mathbf{c}, \mathbf{z}$ , 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 `type(kind_parameter)`, with `type = real, complex` and `kind_parameter = kind(1.e0), kind(1.d0)`, according to the sparse matrix and preconditioner data structure; note that the PSBLAS module `psb_base_mod` provides the constants `psb_spk_ = kind(1.e0)` and `psb_dpk_ = kind(1.d0)`;
- 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 `mld_precinit`

```

      mld_precinit(p,ptype,info)
      mld_precinit(p,ptype,info,nlev)

```

This routine allocates and initializes the preconditioner data structure, according to the preconditioner type chosen by the user.

### Arguments

- p**            `type(mld_xprec_type), intent(inout)`.  
The preconditioner data structure. Note that *x* must be chosen according to the real/complex, single/double precision version of MLD2P4 under use.
- ptype**        `character(len=*)`, `intent(in)`.  
The type of preconditioner (see below).  
Note that the strings are case insensitive.
- info**         `integer, intent(out)`.  
Error code. If no error, 0 is returned. See Section 7 for details.
- nlev**         `integer, optional, intent(in)`.  
The number of levels of the multilevel preconditioner. If `nlev` is not present and `ptype='ML'`, `'ml'`, then `nlev=2` is assumed. Otherwise, `nlev` is ignored.

The available preconditioner types are those described in table 1, briefly repeated here: Legal inputs to this subroutine are interpreted depending on the *ptype* string as follows<sup>1</sup>:

**NOPREC** No Preconditioner

**JACOBI** Point-Jacobi (the string **DIAG** is also allowed);

**BJAC** Block-Jacobi;

**AS** Additive Schwarz;

**ML** Multi-Level Schwarz.

---

<sup>1</sup>The string is case-insensitive

## 6.2 Subroutine `mld_precset`

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

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

### Arguments

<code>p</code>	<code>type(mld_xprec_type), intent(inout).</code> The preconditioner data structure. Note that $x$ must be chosen according to the real/complex, single/double precision version of MLD2P4 under use.
<code>what</code>	<code>integer, intent(in).</code> The number identifying the parameter to be set. A mnemonic constant has been associated to each of these numbers, as reported in Tables 2-5.
<code>val</code>	<code>integer or character(len=*) or real(psb_spk_) or real(psb_dpk_), intent(in).</code> The value of the parameter to be set. The list of allowed values and the corresponding data types is given in Tables 2-5. When the value is of type <code>character(len=*)</code> , it is also treated as case insensitive.
<code>info</code>	<code>integer, intent(out).</code> Error code. If no error, 0 is returned. See Section 7 for details.

A variety of (one-level and multi-level) 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 preconditioner;
2. the one-level preconditioner used as smoother;
3. the aggregation algorithm;
4. the coarsest-level correction.

A list of the parameters that can be set, along with their allowed and default values, is given in Tables 2-5. For a detailed description of the meaning of the parameters, please refer to Section 4.1.

what	DATA TYPE	val	DEFAULT	COMMENTS
mld_ml_type_	character(len=*)	'ADD' 'MULT'	'MULT'	Basic multi-level framework: additive or multiplicative among the levels (always additive inside a level).
mld_smoother_type_	character(len=*)	'JACOBI' 'BJAC' 'AS'	'AS'	Basic one-level preconditioner (i.e. smoother): point-Jacobi, block-Jacobi, AS.
mld_smoother_pos_	character(len=*)	'PRE' 'POST' 'TWO_SIDE'	'TWO_SIDE'	"Position" of the smoother in the multilevel framework: pre-smoother, post-smoother, pre- and post-smoother. Ignored if mld_ml_type_ is set to 'ADD'.
mld_smoother_sweeps_	integer	Any int. num. > 0	1	Number of sweeps of the (pre-/post) smoother. When mld_ml_type_ is set to 'MULT', it sets the default value for both mld_smoother_sweeps_pre_ and mld_smoother_sweeps_post_.
mld_smoother_sweeps_pre_	integer	Any int. num. > 0	1	Number of sweeps of the pre-smoother. Used if mld_ml_type_ is set to 'MULT' and mld_smoother_pos_ is 'PRE' or 'TWO_SIDE'.
mld_smoother_sweeps_post_	integer	Any int. num. > 0	1	Number of sweeps of the post-smoother. Used if mld_ml_type_ is set to 'MULT' and mld_smoother_pos_ is 'POST' or 'TWO_SIDE'.

Table 2: Parameters defining the type of multi-level preconditioner.

what	DATA TYPE	val	DEFAULT	COMMENTS
mld_sub_ovr_	integer	any int. num. $\geq 0$	1	Number of overlap layers.
mld_sub_restr_	character(len=*)	'HALO' 'NONE'	'HALO'	Type of restriction operator: 'HALO' for taking into account the overlap, 'NONE' for neglecting it.
mld_sub_prol_	character(len=*)	'SUM' 'NONE'	'NONE'	Type of prolongation operator: 'SUM' for adding the contributions from the overlap, 'NONE' for neglecting them.
mld_sub_solve_	character(len=*)	'DSCALE' 'ILU' 'MILU' 'ILUT' 'UMF' 'SLU'	'ILU'	Local solver for block-Jacobi and AS smoothers: diagonal scale (point Jacobi), ILU( $p$ ), MILU( $p$ ), ILU( $p, t$ ), LU from UMFPACK, LU from SuperLU (plus triangular solve).
mld_sub_fillin_	integer	Any int. num. $\geq 0$	0	Fill-in level $p$ of the incomplete LU factorizations.
mld_sub_iluthrs_	real ( <i>kind=parameter</i> )	Any real num. $\geq 0$	0	Drop tolerance $t$ in the ILU( $p, t$ ) factorization.
mld_sub_ren_	character(len=*)	'RENUM_NONE' 'RENUM_GLOBAL'	'RENUM_NONE'	Row and column reordering of the local submatrices: no reordering, reordering according to the global numbering of the rows and columns of the whole matrix.

Table 3: Parameters defining the one-level preconditioner used as smoother.

what	DATA TYPE	val	DEFAULT	COMMENTS
mld_aggr_alg_	character(len=*)	'DEC'	'DEC'	Aggregation algorithm. Currently, only the decoupled aggregation is available.
mld_aggr_kind_	character(len=*)	'SMOOTHED', 'NONSMOOTHED'	'SMOOTHED'	Type of aggregation: smoothed, non-smoothed (i.e. using the tentative prolongator).
mld_aggr_thresh_	real(kind=parameter)	Any real num. $\in [0, 1]$	$0.08/2^{(l-1)}$	Threshold $\theta$ in the aggregation algorithm. Note that the default threshold varies at each level $l$ , where $l = 1$ corresponds to the fine level (see [28]). Conversely a value of theta chosen by the user is set to the same value at all levels.
mld_aggr_omega_alg_	character(len=*)	'EIG_EST', 'USER_CHOICE'	'EIG_EST'	How the damping parameter $\omega$ in the smoothed aggregation should be computed: either via an estimate of the spectral radius of $D^{-1}A$ , or explicitly specified by the user.
mld_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_	real(kind=parameter)	Any nonnegative real num.	$4/(3\rho(D^{-1}A))$	Damping parameter $\omega$ 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 $D^{-1}A$ .

Table 4: Parameters defining the aggregation algorithm.

what	DATA TYPE	val	DEFAULT	COMMENTS
<code>mld_coarse_mat_</code>	<code>character(len=*)</code>	<code>'DISTR'</code> <code>'REPL'</code>	<code>'DISTR'</code>	Coarsest matrix: distributed among the processors or replicated on each of them.
<code>mld_coarse_solve_</code>	<code>character(len=*)</code>	<code>'JACOBI'</code> <code>'BJAC'</code> <code>'UMF'</code> <code>'SLU'</code> <code>'ILU'</code> <code>'MILU'</code> <code>'ILUT'</code> <code>'SLUDIST'</code>	<code>'BJAC'</code>	Solver used at the coarsest level: point-Jacobi, block-Jacobi, sequential incomplete LU (three variants), sequential LU from UMFPACK, sequential LU from SuperLU, distributed LU from SuperLU-Dist. <code>'JACOBI'</code> , <code>'BJAC'</code> and <code>'SLUDIST'</code> requires the coarsest matrix to be distributed, while the others require it to be replicated.
<code>mld_coarse_subsolve_</code>	<code>character(len=*)</code>	<code>'ILU'</code> <code>'MILU'</code> <code>'ILUT'</code> <code>'UMF'</code> <code>'SLU'</code>	See note	Solver for the diagonal blocks of the coarse matrix, in case the point-Jacobi or block-Jacobi solver is chosen as coarsest-level solver: <code>ILU(p)</code> , <code>MILU(p)</code> , <code>ILU(p,t)</code> , LU from UMFPACK, LU from SuperLU (plus triangular solve).
<code>mld_coarse_sweeps_</code>	<code>integer</code>	Any int. num. $> 0$	4	Number of point-Jacobi or block-Jacobi sweeps of the coarsest-level solver.
<code>mld_coarse_fillin_</code>	<code>integer</code>	Any int. num. $\geq 0$	0	Fill-in level $p$ of the incomplete LU factorizations.
<code>mld_coarse_iltuthrs_</code>	<code>real(kind=parameter)</code>	Any real. num. $\geq 0$	0	Drop tolerance $t$ in the <code>ILU(p,t)</code> factorization.

**Note:** defaults for `mld_coarse_subsolve_` are chosen as  
- single precision version: `'SLU'` if installed, `'ILU'` otherwise,  
- double precision version: `'UMF'` if installed, else `'SLU'` if installed, `'ILU'` otherwise.

Table 5: Parameters defining the coarsest-level correction.

### 6.3 Subroutine `mld_precbld`

`mld_precbld(a,desc_a,p,info)`

This routine builds the preconditioner according to the requirements made by the user through the routines `mld_precinit` and `mld_precset`.

#### Arguments

- `a`            `type(psb_xspmat_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 [17].
- `desc_a`       `type(psb_desc_type), intent(in)`.  
The communication descriptor of `a`. See the PSBLAS User's Guide for details [17].
- `p`            `type(mld_xprec_type), intent(inout)`.  
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 7 for details.

## 6.4 Subroutine `mld_precaply`

```

      mld_precaply(p,x,y,desc_a,info)
      mld_precaply(p,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, `mld_precaply` is called within the PSBLAS routine `psb_krylov` and hence it is completely transparent to the user.

### Arguments

- |                     |                                                                                                                                                                                                                                                                                                                                                    |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>p</code>      | <code>type(mld_xprec_type), intent(inout).</code><br>The preconditioner data structure, containing the local part of $M$ . Note that $x$ must be chosen according to the real/complex, single/double precision version of MLD2P4 under use.                                                                                                        |
| <code>x</code>      | <code>type(kind_parameter), dimension(:), intent(in).</code><br>The local part of the vector $x$ . Note that <code>type</code> and <code>kind_parameter</code> must be chosen according to the real/complex, single/double precision version of MLD2P4 under use.                                                                                  |
| <code>y</code>      | <code>type(kind_parameter), dimension(:), intent(out).</code><br>The local part of the vector $y$ . Note that <code>type</code> and <code>kind_parameter</code> must be chosen according to the real/complex, single/double precision version of MLD2P4 under use.                                                                                 |
| <code>desc_a</code> | <code>type(psb_desc_type), intent(in).</code><br>The communication descriptor associated to the matrix to be preconditioned.                                                                                                                                                                                                                       |
| <code>info</code>   | <code>integer, intent(out).</code><br>Error code. If no error, 0 is returned. See Section 7 for details.                                                                                                                                                                                                                                           |
| <code>trans</code>  | <code>character(len=1), optional, intent(in).</code><br>If <code>trans = 'N', 'n'</code> then $op(M^{-1}) = M^{-1}$ ; if <code>trans = 'T', 't'</code> then $op(M^{-1}) = M^{-T}$ (transpose of $M^{-1}$ ); if <code>trans = 'C', 'c'</code> then $op(M^{-1}) = M^{-C}$ (conjugate transpose of $M^{-1}$ ).                                        |
| <code>work</code>   | <code>type(kind_parameter), dimension(:), optional, target.</code><br>Workspace. Its size should be at least $4 * psb_cd\_get\_local\_cols(desc\_a)$ (see the PSBLAS User's Guide). Note that <code>type</code> and <code>kind_parameter</code> must be chosen according to the real/complex, single/double precision version of MLD2P4 under use. |

## 6.5 Subroutine `mld_precfree`

`mld_precfree(p, info)`

This routine deallocates the preconditioner data structure.

### Arguments

- |                   |                                                                                                                                                                                                          |
|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>p</code>    | <code>type(mld_xprec_type), intent(inout).</code><br>The preconditioner data structure. Note that $x$ must be chosen according to the real/complex, single/double precision version of MLD2P4 under use. |
| <code>info</code> | <code>integer, intent(out).</code><br>Error code. If no error, 0 is returned. See Section 7 for details.                                                                                                 |

## 6.6 Subroutine `mld_precdescr`

```
mld_precdescr(p,info)
mld_precdescr(p,info,iout)
```

This routine prints a description of the preconditioner to the standard output or to a file. It must be called after `mld_precbld` has been called.

### Arguments

- |                   |                                                                                                                                                                                                       |
|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>p</code>    | <code>type(mld_xprec_type), intent(in).</code><br>The preconditioner data structure. Note that $x$ must be chosen according to the real/complex, single/double precision version of MLD2P4 under use. |
| <code>info</code> | <code>integer, intent(out).</code><br>Error code. If no error, 0 is returned. See Section 7 for details.                                                                                              |
| <code>iout</code> | <code>integer, intent(in), optional.</code><br>The id of the file where the preconditioner description will be printed; the default is the standard output.                                           |

## 7 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 [17].

## A License

The MLD2P4 is freely distributable under the following copyright terms:

MLD2P4 version 1.2  
MultiLevel Domain Decomposition Parallel Preconditioners Package  
based on PSBLAS (Parallel Sparse BLAS version 2.3.1)

(C) Copyright 2008, 2009

Salvatore Filippone University of Rome Tor Vergata  
Alfredo Buttari University of Rome Tor Vergata  
Pasqua D'Ambra ICAR-CNR, Naples  
Daniela di Serafino Second University of Naples

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name of the MLD2P4 group or the names of its contributors may not be used to endorse or promote products derived from this software without specific written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS 'AS IS' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE MLD2P4 GROUP OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## References

- [1] A. Aproxitola, P. D'Ambra, F. Denaro, D. di Serafino, S. Filippone, *Scalable Algebraic Multilevel Preconditioners with Application to CFD*, to appear in Proceedings of the International Conference on Parallel Computational Fluid Dynamics, Lecture Notes in Computational Science and Engineering, Springer.
- [2] M. Brezina, P. Vaněk, *A Black-Box Iterative Solver Based on a Two-Level Schwarz Method*, Computing, 63, 1999, 233–263.
- [3] 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.
- [4] 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.
- [5] P. D'Ambra, S. Filippone, D. di Serafino, *On the Development of PSBLAS-based Parallel Two-level Schwarz Preconditioners*, Applied Numerical Mathematics, Elsevier, 57, 11-12, 2007, 1181-1196.
- [6] X. C. Cai and Y. Saad, *Overlapping Domain Decomposition Algorithms for General Sparse Matrices*, Numerical Linear Algebra with Applications, 3, 3, 1996, pp. 221–237, .
- [7] 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.
- [8] 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.
- [9] T. Chan and T. Mathew, *Domain Decomposition Algorithms*, in A. Iserles, editor, Acta Numerica 1994, 61–143. Cambridge University Press.
- [10] P. D'Ambra, D. di Serafino, S. Filippone, *MLD2P4: a Package of Parallel Multilevel Algebraic Domain Decomposition Preconditioners in Fortran 95*, ICAR-CNR Technical Report RT-ICAR-NA-09-01, 2009.
- [11] 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/>)

- [12] 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.
- [13] 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.
- [14] 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.
- [15] 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).
- [16] E. Efstathiou, J. G. Gander, *Why Restricted Additive Schwarz Converges Faster than Additive Schwarz*, *BIT Numerical Mathematics*, 43, 2003, 945–959.
- [17] S. Filippone, A. Buttari, *PSBLAS-2.3 User's Guide. A Reference Guide for the Parallel Sparse BLAS Library*, 2008, available from <http://www.ce.uniroma2.it/psblas/>.
- [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] J. Mandel, M. Brezina, P. Vaněk, *Energy Optimization of Algebraic Multigrid Bases*, *Computing*, 62,3, 1999, 205–228.
- [23] Y. Saad, *Iterative methods for sparse linear systems*, 2nd edition, SIAM, 2003
- [24] B. Smith, P. Bjorstad, W. Gropp, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge University Press, 1996.

- [25] 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.
- [26] K. Stüben, *Algebraic Multigrid (AMG): an Introduction with Applications*, in A. Schüller, U. Trottenberg, C. Oosterlee, editors, *Multigrid*, Academic Press, 2000.
- [27] 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.
- [28] P. Vaněk, J. Mandel and M. Brezina, *Algebraic Multigrid by Smoothed Aggregation for Second and Fourth Order Elliptic Problems*, *Computing*, 56, 1996, 179-196.