Consiglio Nazionale delle Ricerche Istituto di Calcolo e Reti ad Alte Prestazioni

MLD2P4: a Package of Parallel Multilevel Algebraic Domain Decomposition Preconditioners in Fortran 95

P. D'Ambra – D. di Serafino – S. Filippone

RT-ICAR-NA-09-01 marzo 2009

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
Sede di Napoli, Via P. Castellino 111, 80131 Napoli, URL: www.na.icar.cnr.it

Consiglio Nazionale delle Ricerche Istituto di Calcolo e Reti ad Alte Prestazioni

MLD2P4: a Package of Parallel Multilevel Algebraic Domain Decomposition Preconditioners in Fortran 95

P. D'Ambra[1] – D. di Serafino[2] – S. Filippone[3]

Rapporto Tecnico N.: RT-ICAR-NA-09-01

Data: marzo 2009

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sede di Napoli, Via P. Castellino 111, 80131 Napoli - pasqua.dambra@na.icar.cnr.it
[2] Dipartimento di Matematica, Seconda Università di Napoli, Via Vivaldi 43, 81100 Napoli - daniela.diserafino@unina2.it
[3] Dipartimento di Ingegneria Meccanica, Università di Roma "Tor Vergata", Via del Politecnico 1, 00133 Roma - salvatore.filippone@uniroma2.it

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR)
Sede di Napoli, Via P. Castellino 111, 80131 Napoli, URL: www.na.icar.cnr.it

MLD2P4: a Package of Parallel Multilevel Algebraic Domain Decomposition Preconditioners in Fortran 95*

PASQUA D'AMBRA
ICAR-CNR, Naples

DANIELA DI SERAFINO
Second University of Naples

SALVATORE FILIPPONE
University of Rome "Tor Vergata"

Abstract

Domain decomposition ideas have long been an essential tool for the solution of PDEs on parallel computers. In recent years many research efforts have been focused on employing recursively domain decomposition methods to obtain multilevel preconditioners to be used with Krylov solvers. In this context, we developed MLD2P4 (MultiLevel Domain Decomposition Parallel Preconditioners Package based on PSBLAS), a package of parallel multilevel preconditioners that combines Additive Schwarz domain decomposition methods with a smoothed aggregation technique to build a hierarchy of coarse-level corrections in an algebraic way. The design of MLD2P4 was guided by objectives such as extensibility, flexibility, performance, portability and ease of use. They were achieved by following an object-oriented approach while using the Fortran 95 language, as well as by employing the PSBLAS library as basic framework. In this paper we present MLD2P4 focusing on its design principles, software architecture and use.

1 Introduction

It is well known that multigrid and, more generally, multilevel methods are optimal for the solution of linear systems arising from the discretization of elliptic Partial Differential Equations (PDEs); here optimality is defined as the ability to solve such systems employing a number of operations that scales

*Preprint submitted to ACM Transactions on Mathematical Software (2009)

linearly with the number of unknowns. Since the pioneering work of Achi Brandt [4], multilevel methods have therefore been widely investigated and successfully applied not only to linear systems arising from PDEs, but also to a large variety of linear and nonlinear problems [5, 30]. In addition, multilevel domain decomposition methods are well suited for modern computing systems, from large-scale distributed systems to current multi-core processors. The algorithmic scalability of multilevel methods, i.e. their capability of keeping the number of iterations to get a fixed accuracy constant as the number of subdomains increases, is a key feature to obtain scalable software as the problem size and the number of processors increase. Furthermore, the instrinsic hierarchy of multilevel methods allows to fit the hierarchy of memories and of types of parallelism of emergent high-performance architectures, through a suitable combination of the number of levels and of the size of the single-level numerical problem.

Many efforts have been devoted to the development of parallel software implementing multilevel domain decomposition solvers and preconditioners, with the multiple objectives of parallel efficiency, flexibility, extensibility, general applicability and ease of use. In many cases this is carried out by employing concepts and tools of object-oriented programming [2, 28, 27], sometimes sacrificing runtime efficiency and easy interfacing with Fortran legacy codes.

We present a package of parallel multilevel domain decomposition preconditioners developed following an object-based design, translated into Fortran 95 in order to preserve runtime efficiency with a resonable programming effort and to allow immediate interfacing with Fortran codes. This package, named *MLD2P4 (MultiLevel Domain Decomposition Parallel Preconditioners Package based on PSBLAS)*, implements a suite of parallel algebraic multilevel Schwarz preconditioners, based on smoothed aggregation, on top of the PSBLAS linear algebra package [24]; the preconditioners can be used with Krylov solvers available in the PSBLAS framework for the solution of real or complex linear systems. The software architecture of MLD2P4 is the result of a modular approach naturally inspired by the intrinsic hierarchy of multilevel methods: data structures and methods of increasing complexity are obtained by combining simpler components where different levels of parallelism can be exploited to get high performance on a given problem with the parallel machine at hand. In addition to the key objectives of efficiency, flexibility and extensibility, the issues of portability and ease of use have been central in the design of MLD2P4. "De-facto" standards for sparse basic linear algebra operations and for data communications are the basic layers of the architecture; few simple APIs are available to the users

to exploit the MLD2P4 functionalities at different levels of expertise. Furthermore, single and double precision implementations of the package for both real and complex matrices are available, that are accessed through the same, overloaded, interface. Previous work concerning the use of MLD2P4 preconditioners on model problems and CFD applications has shown the good performance of our software on various parallel computers [8, 13, 1].

The paper is organized as follows. In Section 2 we outline the algebraic multilevel Schwarz preconditioners based on smoothed aggregation, in order to provide a numerical background for the description of MLD2P4. In Section 3 we discuss the main principles and choices driving the design of our package. In Section 4 we present the software architecture of MLD2P4, describing its main components and how they are combined, and providing also implementation details. In Section 5 we show the basics for using the MLD2P4 preconditioners with the Krylov solvers available in PSBLAS. In Section 6 we present related software packages. Finally, in Section 7 we report some conclusions and future work.

2 Background: algebraic multilevel Schwarz preconditioners

Let us consider a linear system

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

where $A = (a_{ij}) \in \Re^{n \times n}$ is a nonsingular sparse matrix; for ease of presentation we assume A is real, but the results are valid for the complex case as well. Multilevel methods are often used to build preconditioners for the matrix A, which are coupled with Krylov methods to solve system (1). Generally speaking, a multilevel method provides an approximate inverse of A by suitably combining approximate inverses of a hierarchy of matrices which represent A in increasingly coarser spaces. This is achieved by recursively applying two main processes: *smoothing*, which provides an approximate inverse of a single matrix in the hierarchy, and *coarse-space correction*, which computes a correction to the approximate inverse by transferring suitable information from the current space to the next coarser one and vice versa, and by computing, through smoothing, an approximate inverse of the coarse matrix (see, e.g., [35, 37, 22]).

We outline a general framework for building and applying multilevel preconditioners, focusing on the *algebraic* approach, which is the one we employ. Let us assume as finest index space the set of row (column) indices of

$A^1 = A$, $\Omega^1 = \Omega$
set up S^1
for $k = 1, nlev - 1$ **do**
  generate Ω^{k+1} from Ω^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}
endfor

Figure 1: Build phase of a multilevel preconditioner.

A, $\Omega = \{1, 2, \ldots, n\}$. An algebraic multilevel method generates a hierarchy of index spaces and a corresponding hierarchy of matrices,

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

by using the information contained in A, without assuming any knowledge of the geometry of the problem from which A originates. A vector space $\Re^{n_k}$ is associated to Ω^k, where n_k is the size of Ω^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

$$P^k : \Re^{n_{k+1}} \longrightarrow \Re^{n_k}, \quad R^k : \Re^{n_k} \longrightarrow \Re^{n_{k+1}};$$

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.

$$A^{k+1} = R^k A^k P^k. \tag{2}$$

A smoother S^k is set up (in a sense that will be clarified later), and is used to compute products of type $(\tilde{A}^k)^{-1} v$, where $(\tilde{A}^k)^{-1}$ is an approximate inverse of A^k; therefore, in the sequel we identify $(S^k)^{-1}$ with $(\tilde{A}^k)^{-1}$. 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.

$v^1 = v$;

! for each level k but the coarsest one, apply the smoother
! and restrict the residual

for $k = 1, nlev - 1$ **do**

$y^k = S^k v^k$

$r^k = v^k - A^k y^k$

$v^{k+1} = R^k r^k$

endfor

! apply the smoother at the coarsest level

$y^{nlev} = S^{nlev} v^{nlev}$

! for each level k but the coarsest one, interpolate y^k,
! update the residual, apply the smoother and update y^k

for $k = nlev - 1, 1, -1$ **do**

$y^k = y^k + P^k y^{k+1}$

$r^k = v^k - A^k y^k$

$r^k = S^k r^k$

$y^k = y^k + r^k$

endfor

$w = y^1$

Figure 2: Application phase of a symmetrized multiplicative multilevel preconditioner.

The components produced in the build phase may be combined in several ways to obtain different multilevel preconditioners (see [35]); 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. An example of such a combination, known as *symmetrized multiplicative multilevel*, or more generally *V-cycle*, is given in Figure 2.

In MLD2P4 the hierarchy of index spaces and the corresponding mapping operators are built by applying a *smoothed aggregation* technique [40, 31, 6]. The basic idea is to build the coarse set of indices Ω^{k+1} by grouping the indices of Ω^k into disjoint subsets (aggregates) and to define a simple "tentative" prolongator P^k_{tent} 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 P^k_{tent}, in order to obtain low energy coarse basis functions and

hence good convergence rates.

The algoritm implemented in MLD2P4 builds each aggregate by grouping the indices that are strongly coupled to a certain "root" index, and computes

$$P^k = (I - \omega(D^k)^{-1}A^k)P^k_{tent}, \tag{3}$$

where P^k_{tent} is a piecewise constant interpolation operator, ω is a damping parameter and D^k is the diagonal part of A^k (see [13, 14]). In the build process we assume that the matrix A has a symmetric sparsity pattern.

The multilevel preconditioners implemented in MLD2P4 are of Schwarz type, since the smoother S^k used at each level $k < nlev$ is a domain decomposition *Additive Schwarz* (AS) preconditioner [12, 35, 10]. Therefore, the multilevel preconditioners based on a multiplicative framework are referred to as *hybrid*, i.e. multiplicative among the levels and additive inside any single level.

In the AS methods the index space Ω^k is divided into m_k subsets Ω^k_i of size $n_{k,i}$, possibly overlapping. For each i we consider the restriction operator $R^k_i : \Re^{n_k} \longrightarrow \Re^{n_{k,i}}$, mapping a vector x^k to the vector x^k_i made of the components of x^k with indices in Ω^k_i, and the prolongation operator $P^k_i = (R^k_i)^T$. The restriction and prolongation operators are then used to build $A^k_i = R^k_i A^k P^k_i$, which is the restriction of A^k to the index space Ω^k_i. The *classical AS* preconditioner is defined as

$$\left(S^k_{AS}\right)^{-1} = \sum_{i=1}^{m_k} P^k_i (A^k_i)^{-1} R^k_i,$$

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

- the definition of the index subspaces Ω^k_i and of the corresponding operators R^k_i and P^k_i;
- the computation of the submatrices A^k_i;
- the computation of their inverses (usually approximated through some form of incomplete factorization).

The computation of $z^k = S^k_{AS} w^k$, where $w^k \in \Re^{n_k}$, during the multilevel application phase, requires

- the restriction of w^k to the subspaces $\Re^{n_{k,i}}$, i.e. $w^k_i = R^k_i w^k$;

- 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 [11, 21].

Direct solvers based on the LU factorization as well as approximate solvers based on the ILU factorization or on the block-Jacobi iterative method are implemented as "smoothers" at the coarsest level. More details on this issue are given in Section 4.

3 Design principles

The basic design principle of MLD2P4 is the development of set of "objects" (data structures and routines operating on them) that can be combined with high flexibility by both the user and the developer. These objects should be as general as possible (to allow for maximal reuse), within performance and portability constraints. We briefly describe below how the basic design objectives of MLD2P4 have been accomplished.

Flexibility and extensibility The implementation of MLD2P4 is based on a modular approach. Objects of increasing complexity have been obtained by composing simpler ones; to this aim, object-oriented features of Fortran 95 for data abstraction and functional overloading have been exploited. The requirement of having very general and reusable objects has also naturally led to a layered software architecture, providing increasingly more specific functionalities in going from the bottom layer to the top one. In this framework, the implementation has been split between MLD2P4 and the Parallel Sparse BLAS (PSBLAS) package, on the top of which MLD2P4 has been built. The original implementation of PSBLAS (version 1.0) [25] provided parallel versions of most of the Sparse BLAS computational kernels proposed in [20] and auxiliary routines for the creation and management of distributed sparse matrices, to be used for building sparse iterative linear algebra solvers on distributed-memory parallel computers. In designing MLD2P4, whenever a basic data structure or method used by the multilevel preconditioners resulted amenable to a more general use,

we defined it in PSBLAS, thus leading to an extension of the latter package (version 2.0 and later, see [24]). More details on these issues are given in Section 4.

Portability MLD2P4 has been written in the Fortran 95 language standard coupled with the extensions of the ISO Technical Report TR 15581; coverage of this language level is nowadays universal among the compilers commonly in use, including the free Gnu Fortran compiler (as of version 4.2). The inter-processor communication has been implemented by using the message-passing environments BLACS (Basic Linear Algebra Communication Subprograms) [18], and MPI (Message Passing Interface) [36], which are considered "de-facto" standards.

Indeed, PSBLAS provides overloaded interfaces to the communication operators commonly employed in the context of sparse computations and in the context of mesh-based algorithms, as well as common send/receive operations and environment inquiry; thus, the PSBLAS user will rarely, if ever, need to invoke explicitly a BLACS or MPI primitive. The interface overloading allows the compiler to check aand prevent xcommon programming errors such as mismatch between data types and communication routines. The BLACS layer has been used internally because it provides a convenient model of communication in which the user is relieved from the buffer-handling chores that are often necessary for point-to-point communications; this layer provides a "fire and forget" model that relies on internal buffering, while sacrificing some performance for convenience. Direct access to the MPI layer is still possible when needed, and is indeed employed in some internal routines; for instance, the most critical communication operation, the PSBLAS halo data exchange [24], by default makes direct use of MPI send/receive operators, and many data setup routines in both PSBLAS and MLD2P4 call MPI global communication operators to provide maximum performance.

Effective memory management Fortran 95 allows dynamic memory allocation, which is a very useful feature when dealing with large-scale problems. Thanks to the TR 15581 extensions, arrays with the `ALLOCATABLE` attribute can be used, instead of `POINTER` variables, as dummy arguments and as components of derived data types. `ALLOCATABLE` arrays have two useful features:

- their association status is guaranteed to be consistent;

- they only point to contiguous memory (as opposed to pointers, which may implicitly specify a stride).

Therefore, `ALLOCATABLE` arrays are advantageous for memory management, since they avoid the generation of memory leaks in a user program (barring compiler bugs); furthermore, they enable a very simple memory release scheme based simply on the `DEALLOCATE` statement. Finally, `ALLOCATABLE` arrays tend to give better performance, because the memory contiguousness helps the compiler optimizers produce more efficient machine code and because they tend to avoid memory aliasing.

High performance This objective has been pursued at both the implementation and the algorithmic level, as explained below. The results obtained on various large-scale linear systems, arising from model problems as well as from real applications [8, 13, 1], show the effectiveness of our approach.

- Other object-oriented languages often encourage programming styles placing very substantial runtime requirements and great care must be exercised in their usage in connection with numerical codes where the execution time is an important feature. Conversely, Fortran 95 has been designed to enable compilers to generate highly optimized executable codes. Thus by choosing this language we are using many object-oriented features while maintaining a good runtime efficiency with limited programming effort.
- The internal representation of sparse matrices in PSBLAS is a key element of performance handling for two reasons:
  - the storage layout is not tied to a single representation method, thus it can easily adapt to different computing architectures; indeed releated research efforts are devoted to finding new storage schemes for multiple architectures [9];
  - the storage layout may change during program execution, providing optimal support to different phases of the computation; this is essentially an implementation of the State design pattern [26].
- It is well known that one-level Schwarz preconditioners are scalable with respect to a single iteration of the preconditioned solver,

but they yield a convergence rate that deteriorates as the number of submatrices increases. The use of a coarse-space correction allows to obtain optimal preconditioners, i.e. such that the number of iterations is bounded independently of the number of the submatrices; however it may substantially affect the parallel performance because of the cost of solving, at each iteration, the coarse-level system. A possible remedy is the approximate solution of the coarse-level system, which is generally less expensive, but also less effective in terms of convergence rate; this can be obtained by recursively applying the coarse-level correction in a multilevel framework. In practice, finding a good combination of the number of levels and of the solver to be applied at the coarsest level is a key point in achieving high performance in a parallel environment. The choice of these two features is generally dependent on the characteristics of the linear system to be solved and on the characteristics of the parallel computer. Therefore, MLD2P4 has been designed to allow the user to experiment with different solutions, in order to find the "best" preconditioner for the problem at hand. It provides different coarsest-level solvers, i.e. sparse distributed and sequential LU solvers, as well as distributed block-Jacobi ones, with ILU or LU factorizations of the blocks; furthermore, no limitation on the number of levels is imposed, except the obvious one that no more levels are considered if the coarsest one is such that the aggregation algorithm applied to it does not change the index space.

Ease of use The MLD2P4 functionalities can be accessed through a set of uniform and easy-to-use APIs, that allow for different levels of exploitation of the package: non-expert users can easily select default preconditioners, while expert users can choose among various types of multilevel preconditioners and fine tune their application parameters. Note also that the use of Fortran 95 allows easy interfacing of MLD2P4 with Fortran 77 legacy codes; the PSBLAS facilities for sparse matrix data management can be used to build the distributed matrix data structure to which the MLD2P4 has to be applied.

4 Software architecture

The software architecture of MLD2P4 is the result of a modular approach based on object-oriented design principles, where data structures and "meth-

ods" of increasing complexity are obtained by combining simpler components. This hierarchical view nicely reflects the hierarchical nature of the underlying multilevel methods. A description of the fundamental components of MLD2P4 and of their interactions within the software package follows. We note that the final software architecture of MLD2P4 described here has evolved considerably from the original two-level version of the package described in [8].

4.1 Basic components

Following the description of the multilevel preconditioners in Section 2, we identify three basic components:

- *matrices*,
- *index spaces*, and their associated vector spaces
- *maps* between pairs of vector spaces.

Furthermore, we distinguish between *intra-level* maps, which act within a single level of the multilevel hierachy, and *inter-level* maps, which transfer information between contiguous levels.

At each level k, the matrix A^k is distributed among the processors in a general row-block fashion, with possibly overlapping rows, and the related vectors are distributed accordingly. Two main PSBLAS data structures are used to hold the information concerning A^k: the *sparse matrix* `psb_Tspmat_type` and the *communication descriptor* `psb_desc_type` (note that `T` denotes `s`, `d`, `c`, `z`, according to the real/complex, single/double precision version of MLD2P4).[1] The sparse matrix contains the nonzero entries of the matrix rows assigned to the local processor and the corresponding row and column indices, according to some representation format, that may be changed during the computation to achieve efficiency in the operations involving the matrix. The commmunication descriptor includes information needed for handling communication operations pertaining to the implementation of basic sparse matrix computations, such as matrix-vector products. It is logically associated with the sparsity pattern of the matrix as distributed on the parallel machine, and stores the local index space Ω_i^k, which identifies the submatrix assigned to each processor, as well as the indices of the adjacent matrix rows owned by other processors (halo indices) and

[1] Actually, at the coarsest level A^k may be also replicated on each processor, as explained in Section 4.3, but this is managed through the same data structures.

other information about the global matrix. We note that while the interface to the communication descriptor remained stable with respect to older versions of PSBLAS, its internal implementation has been largely rewritten in PSBLAS 2.3, to provide improved performance for very large index spaces, corresponding to order of millions of matrix rows/columns. For details on the PSBLAS data structures, the reader may refer to [24]. Note that, since the intra-level maps P_i^k and R_i^k closely follow the sparsity pattern of the associated matrix at level k, they can rely on the existing communication descriptors to hold their communication requirements.

In order to store all the information related to inter-level maps P^k and R^k we defined a new data structure, the *linear map*. This data structure was implemented in PSBLAS because the concept of a linear mapping among different vector spaces is useful in more general contexts beyond algebraic multigrid, e.g. in adaptive grid-refinement methods for PDE computations. The linear map uses the sparse matrix structure, to hold the matrix representation of the mapping among two vector spaces X and Y, as well as the communication descriptor structure, to store the information needed to properly exchange the relevant data. More precisely, the linear map type has the following definition:

```
type psb_Tlinmap_type
  integer, allocatable         :: itd_data(:), iaggr(:), naggr(:)
  type(psb_desc_type), pointer :: p_desc_X=>null(), p_desc_Y=>null()
  type(psb_desc_type)          :: desc_X, desc_Y
  type(psb_Tspmat_type)        :: map_X2Y, map_Y2X
end type psb_Tlinmap_type
```

The sparse matrices `map_X2Y` and `map_Y2X` hold the mapping from a space X to a space Y and vice versa; in the MLD2P4 case they hold the maps P^k and R^k, respectively. The two items `p_desc_X` and `p_desc_Y` are pointers to the descriptors of the index spaces associated to X and Y; in the MLD2P4 case they are the descriptors of the fine and the coarse index spaces Ω^k and Ω^{k+1}. In the multilevel algorithms the use of these pointers allows memory savings since the descriptors of the maps are the same descriptors associated to the matrices A^k and A^{k+1}; in MLD2P4 these memory savings may be quite substantial, thus justifying the implementation of this special case. On the other hand, when the linear maps are completely general, it is necessary to have independent descriptors `desc_X` and `desc_Y` for the correct application of the maps. Note also that we explicitly store both the prolongation and the restriction operators, even though in MLD2P4 they

are the transpose of each other. This choice is due to reasons of both efficiency and generality; for instance, this allows an easy path to planned extensions of MLD2P4 with Petrov-Galerkin variants of the smoothed aggregation technique, where the inter-level restriction operator is not the transpose of the prolongation [34]. Finally, `iaggr` and `naggr` hold intermediate information on the inter-level maps, i.e. information identifying the tentative prolongator P^k_{tent}, while `it_data` holds information on the internal status of `psb_Tlinmap_type`.

4.2 Preconditioner data structures

We designed two main preconditioner data structures to store all the information concerning the preconditioners implemented in MLD2P4:

- the *base preconditioner*, holding the preconditioner S^k at a single level k;
- the *multilevel preconditioner*, holding the hierarchy of preconditioners S^k, matrices A^k and maps P^k and R^k, needed to apply a general multilevel preconditioner.

We put these data structures in MLD2P4 since they are specific to the multilevel preconditioners.

In order to identify the components of the base preconditioner we took into account the needs of the preconditioners provided as smoothers in MLD2P4:

- *Diagonal scaling.* This simple preconditioner only requires the diagonal entries of the local matrix to be preconditioned.
- *Block Jacobi.* This preconditioner is a special case of AS preconditioner and requires the computation of a (usually incomplete) LU factorization of the diagonal block of the local part of matrix to be preconditioned, that is assigned to each processor; thus we need to store the L and U factors. Furthermore, we may need to store the part of the local matrix that is outside the block diagonal, to apply multiple block-Jacobi sweeps.
- *Additive Schwarz.* The variants of the AS preconditioner require the same components as the Block Jacobi preconditioner, plus a suitable use of the communication descriptor of the matrix to be preconditioned, to apply the intra-level maps.

We note that the local factorizations required by the block-Jacobi and AS preconditioner may be implemented inside MLD2P4 or provided by external packages. Thus, the base preconditioner data structure must hold the L and U factors computed by MLD2P4 as well as some hook for external packages.

Taking into account the previous considerations, we came to the following definition of the base preconditioner:

```
type mld_Tbaseprec_type
  type(psb_Tspmat_type), allocatable    :: av(:)
  IntrType(kind_parameter), allocatable :: d(:)
  type(psb_desc_type)                   :: desc_data
  integer, allocatable                  :: iprcparm(:)
  real(kind_parameter), allocatable     :: rprcparm(:)
  integer, allocatable                  :: perm(:),  invperm(:)
end type mld_Tbaseprec_type
```

The sparse matrix `av` and the array `d` contain the lower and upper factors and the diagonal of the ILU factorizations computed for the block-Jacobi and AS preconditioners (note that `IntrType` denotes the real or complex data type and `kind_parameter` denotes the associated kind, according to the real/complex, single/double precision version of MLD2P4 under use). The factorizations from external packages are linked through pointers to external data structures provided by the packages; these (C) pointers are stored inside `iprcparm`. This is not a completely satisfactory solution, because it implies a dependency on the interlanguage call mechanism; a widespread availability of the `ISO_C_BINDING` interface[2] will enable a standard solution to this issue. The communication descriptor `desc_data` holds the information concerning the index space and the maps associated to the local matrix A_i^k. The arrays `iprcparm` and `rprcparm` contain integer and real parameters identifying variants of the related preconditioner, while `perm` and `invperm` include information about possible row/column permutations applied to A_i^k, e.g. to improve the performance of the factorization algorithms. We observe that the base preconditioner data structure is flexible enough to support other domain decomposition preconditioners, such as the multiplicative Schwarz ones.

In order to build the multilevel preconditioner data structure, we first packed into an intermediate data structure, `mld_Tonelev_type`, all the information needed to apply the smoothing and the coarse-space correction at a generic level k:

[2]The C binding module is already available in some compilers, including the Gnu Fortran compiler 4.3.

```
type mld_Tonelev_type
  type(mld_Tbaseprec_type)          :: prec
  integer, allocatable              :: iprcparm(:)
  real(kind_parameter), allocatable :: rprcparm(:)
  type(psb_Tspmat_type)             :: ac
  type(psb_desc_type)               :: desc_ac
  type(psb_Tlinmap_type)            :: map
  type(psb_Tspmat_type), pointer    :: base_a    => null()
  type(psb_desc_type), pointer      :: base_desc => null()
end type mld_Tonelev_type
```

Here `prec` holds the local part of the base preconditioner at the current level; `iprcparm` and `rprcparm` contain parameters defining the multilevel framework; `ac` is the local part of the matrix associated to the current level, built from the contiguous finer matrix, and `desc_ac` is its communication descriptor; `map` is the linear map type containing the inter-level restriction and prolongation operators at the current level. We note that, in the multiplicative variants of the multilevel cycle, the residual vector $r^k = v^k - A^k y^k$ has to be computed. For all levels but the finest A^k is stored in the related data structure `ac` during the build phase of the preconditioner; at the finest level A^k is the matrix A, which may be very large, thus it would be highly impractical to make a copy of it inside the preconditioner structure. Therefore, we decided to have a pointer `base_a` that is associated with A^k at all levels. This choice gives the following advantages:

- the code that computes the residual and applies the preconditioner is uniform at all levels, thanks to the additional indirection in the data structure;
- the matrix A is implicitly used without the need to pass it explicitly, which would be redundant for those preconditioners that do not use the residual;
- it is never necessary to perform a copy of A.

The same considerations apply to `base_desc`, which points to `desc_ac` except at the finest level, where it points to the descriptor associated to A.

Then we defined the *multilevel preconditioner* as an array of `mld_Tonelev_type` data types:

```
type mld_Tprec_type
  type(mld_Tonelev_type), allocatable :: precv(:)
end type mld_Tprec_type
```

4.3 Building the preconditioners

The implementation of the build phase of the multilevel preconditioners is based on the implementation of the four steps specified inside the loop in Figure 1.

We first describe the construction of the base preconditioner S^k, which is the "core" around which the multilevel preconditioner is built. This is managed by the `mld_baseprec_bld` routine, which performs three main steps:

- construction of the communication descriptors corresponding to the overlapping matrices A_i^k from the descriptors associated to the original non-overlapping row-block distribution of A^k;
- retrieval of the matrix rows needed to form the matrices A_i^k;
- factorization of these matrices.

The first two steps required an extension of the methods available in PSBLAS for the manipulation of descriptors and sparse matrices (essentially for gathering index lists and matrix rows); these functionalities were put in PSBLAS because they can be used to build extended stencils in a more general context, such as data communications in parallel PDE computations. A more detailed description of these steps can be found in [7] (with some differences due to the obvious evolution of MLD2P4). Various standard incomplete factorizations, such as ILU(p), MILU(p) and ILU(t, p) [32], were implemented in MLD2P4 to perform the last step; we also developed interfaces to the sparse LU factorizations provided by UMFPACK [15] and SuperLU [16].

Note that all of the previous factorizations may be also used directly as (approximate) solvers at the coarsest level; this is achieved by replicating the whole matrix A^{nlev} on all the processors. An interface to the sparse distributed LU factorization from SuperLU_DIST [17] is also provided, that may be used as coarsest-level solver when the matrix A^{nlev} is distributed.

As described in Section 2, the remaining steps of the build phase of a multilevel preconditioner are performed applying a smoothed aggregation technique. The coarse index space Ω^{k+1} is generated by the routine `mld_aggrmap_bld`, by using a decoupled aggregation, in which every processor independently aggregates the subset of indices assigned to it in the non-overlapping row-block distribution of the current-level matrix. This approach does not require any data communication, thus allowing a substantial time saving with respect to other parallel aggregation algorithms;

on the other hand, it may produce nonuniform aggregates near the boundary indices, i.e. the indices held by a processor that are halo indices for other processors, and is dependent on the number of processors and on the initial partitioning of the matrix A. Nevertheless, this approach has been shown to produce good results in practice [39]. We also note that the modularity of the MLD2P4 architecture allows for immediate extensions with different coarsening strategies.

The construction of the prolongation and restriction operators P^k and R^k and of the coarse matrix A^{k+1} is implemented in a single routine, `mld_aggrmat_asb`. Indeed, the setup of the linear map data type holding P^k and R^k requires the knowledge of the communication descriptors associated to A^k and A^{k+1}, and the descriptor of A^{k+1} is assembled when A^{k+1} is computed; thus we could not separate the construction of the coarse matrix from that of the related maps. In order to compute P^k (see (3)), $R^k = (P^k)^T$ and A^{k+1} (see (2)), we had to extend the set of basic sequential sparse matrix operators available in PSBLAS with routines performing the sparse matrix diagonal scaling, the sparse matrix transpose and the sparse matrix by sparse matrix multiplication. The last two operations were implemented by integrating into PSBLAS the SMMP software [3]. It is worth noting that, even though the sparse matrix multiplication is not considered in the SBLAS standard, the possibility of its future inclusion has been foreseen by the BLAS Technical Forum [19]. We observe also that `mld_aggrmap_bld` and `mld_aggrmat_asb` were packed into a single routine, `mld_coarse_bld`, to provide the coarsening functionality as a whole.

The construction of the multilevel preconditioners is implemented in `mld_mlprec_bld` by repeatedly calling `mld_coarse_bld` and `mld_baseprec_bld`, according to the algorithmic framework in Figure 1, and passing to these routines the data structures pertaining the level at which they are called.

4.4 Applying the preconditioners

As for the build phase, the application of the multilevel preconditioners is the result of the combination of some basic operations, i.e. smoother application, residual computation, restriction/prolongation and vector sum (see Figure 2).

The smoothers available in MLD2P4 are used through the routine `mld_baseprec_aply`. This is essentially an interface to `mld_as_aply`, which applies the AS preconditioners; additionally, it also performs the diagonal preconditioning. The original implementation of `mld_as_aply` is discussed in [7]. Since then, several local factorization algorithms have been added (see Section 4.3); fur-

thermore, the routine has been also modified to manage the solution of the coarsest-level system when this is replicated on all the processors. Shortly, the first and the third step of the application of the AS smoother S^k (see Section 2) are performed through two PSBLAS routines: `psb_halo`, which gathers, on each processor, the entries of w_i^k that correspond to the overlapping rows of the local submatrix A_i^k, and `psb_ovrl`, that sums the entries of z^k that are held by multiple processors. The computation of $z_i^k = (A_i^k)^{-1} w_i^k$ is performed by `mld_sub_aply` which applies the triangular solves involving the L and U factors obtained during the build phase. We note that `mld_baseprec_aply` applies the smoother S^k through a general operation of type

$$y = \beta y + \alpha\, op(S^k)x, \tag{4}$$

where x and y are vectors, and α and β are scalars and $op(S^k)$ is the preconditioner or its transpose or its conjugate transpose (depending on the real or complex version of MLD2P4). This allows to perform as a single operation the smoothing and sum steps in the second loop of Figure 2; furthermore, the transpose of S^k is used when the preconditioner is applied with Krylov methods requiring the matrix transpose.

The application of the inter-level restriction and prolongation operators R^k and P^k was put in PSBLAS, where the linear map data structure holding the operators is defined. It is implemented in `psb_map_X2Y`, which computes

$$y = \beta y + \alpha T x,$$

where $T = R^k$ or $T = P^k$, by combining PSBLAS distributed sparse matrix operators. We note that the choice of developing a routine for the application of a pair of general linear maps allows to easily manage the extension of MLD2P4 with other inter-level restriction and prolongation operators.

Finally, the computation of the residual r^k is implemented through the PSBLAS routine `psb_spmm`, which computes the sparse matrix by dense matrix (or vector) product.

The previous computational kernels are combined in `mld_mlprec_aply` to implement different multilevel frameworks, i.e. the additive one and the multiplicative variants with pre-, post- or two-side-smoothing. As the smoothing application routine, `mld_mlprec_aply` applies a multilevel preconditioner M in the general form (4) (with the obvious substitution of S^k with M). This might be useful in future extensions of MLD2P4, e.g. in the case of repeated application of a multilevel cycle to a given vector.

4.5 User interface

The MLD2P4 preconditioners are made available to the user through five interface routines, that encapsulate all the functionalities needed to build and apply any base or multilevel preconditioner implemented in the package. A detailed description of these routines can be found in [14]; we briefly describe here the main tasks performed by them, while in Section 5 we report fragments of code showing their use.

The routine `mld_precinit` allocates the `mld_prec_type` data structure that will hold the preconditioner; it also initializes it, according to a preconditioner type specified by the user, by setting `iprcparm` and `rprcparm` with default values. Four basic preconditioner types can be chosen: diagonal, block Jacobi, Additive Schwarz and multilevel (for their defaults the reader may refer to [14]). We note that, although the block Jacobi preconditioner is a special case of the AS one, it is made directly available as a separate type, to help non-expert users. The same reasoning applies to AS preconditioners, that may be also obtained by setting to 1 the number of levels of the corresponding multilevel preconditioners. A companion routine of `mld_precinit` is `mld_prec_free`, for deallocating the preconditioner data structure when the preconditioner is no longer used.

The defaults of any preconditioner type may be changed by the `mld_precset` routine, by setting a variety of parameters. These parameters have been logically divided into four groups, i.e. parameters defining the general multilevel framework, the base preconditioner, the aggregation algorithm, and the coarse-space correction at the coarsest level. Note that `mld_precset` checks also the consistency of the values of the parameters, to avoid wrong or unsupported choices.

Once the preconditioner data structure has been allocated and initialized, it may be built by `mld_precbld`, according to the user choices. This routine calls `mld_baseprec_bld` if the preconditioner selected by the user is a base preconditioner, and `mld_mlprec_bld` otherwise. Analogously, `mld_precaply` applies the selected preconditioner by calling `mld_baseprec_aply` or `mld_mlprec_aply`. We note that when MLD2P4 is used with a Krylov solver from PSBLAS, `mld_precaply` is called within the PSBLAS routine `psb_krylov`.

Finally, a utility routine, `mld_precdescr`, is also available for printing a detailed description of the preconditioner that has been set up.

A graphical representation of the software architecture described so far is provided in Figure 3, where the different layers resulting from the modu-

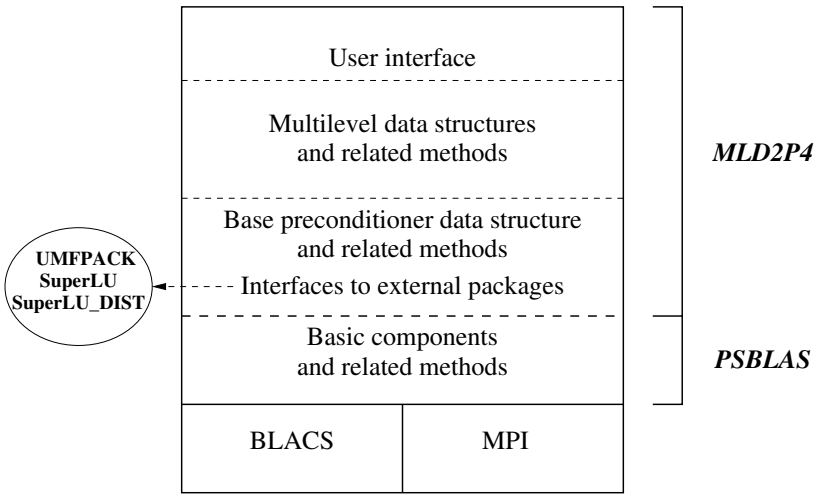

Figure 3: Software architecture of MLD2P4.

lar design approach and the splitting between MLD2P4 and PSBLAS are outlined.

5 Using MLD2P4

MLD2P4 allows the user to build and apply, with the Krylov solvers included in PSBLAS, any multilevel preconditioner resulting from a combination of the following elements:

- the base preconditioner, i.e. the classical AS preconditioner or variants of it, with arbitrary overlap and a local LU or ILU solver chosen among the available ones (see Section 4.3);
- the multilevel framework, i.e. additive or multiplicative with pre-, post- or two-side smoothing;
- the coarsening strategy, i.e. the decoupled smoothed aggregation algorithm, with a selected aggregation threshold to identify strongly connected indices, and a selected damping parameter for the smoothed prolongator;
- the layout of the coarsest-level matrix, i.e. distributed or replicated;

- the coarsest-level solver, i.e. any of the sequential or distributed LU or ILU solvers mentioned in Section 4.3, or the block-Jacobi solver with LU or ILU on the blocks; the SuperLU_DIST and block-Jabobi solvers can be applied if the coarsest-level system is distributed, while the other solvers can be applied if the system is replicated.

The setup and the application of a preconditioner is obtained by performing a few simple steps, through the user interface described in Section 4.5:

1. declare the preconditioner data structure `mld_Tprec_type`;
2. select a preconditioner type (diagonal, block Jacobi, additive Schwarz, Multilevel), and allocate and initialize accordingly the preconditioner data structure, by using the routine `mld_precinit`, which also sets defaults for the selected type;
3. modify the selected preconditioner type by properly setting preconditioner parameters, through `mld_precset`;
4. build the preconditioner for a given matrix, via `mld_precbld`;
5. apply the preconditioner at each iteration of a Krylov solver, through `mld_precaply` (this step is completely transparent to the user, since `mld_precaply` is called by the Krylov solvers implemented in PSBLAS);
6. free the preconditioner data structure, through `mld_precfree`.

The code fragment reported below shows how to set and apply the default multilevel preconditioner (real double precision version), which is a two-level hybrid preconditioner with post-smoothing. The smoother is RAS with overlap 1 and the ILU(0) factorization of the local blocks; the coarsest-level system is distributed among the processors and is approximately solved by using four block-Jacobi sweeps, with the LU factorization of the blocks via UMFPACK.

```
  use psb_base_mod
  use psb_krylov_mod
  use mld_prec_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(psb_dpk_), allocatable :: b(:), x(:)
!
! Build  matrix A and right-hand side b
... ...
!
! initialize the default multi-level preconditioner
  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)
```

We see that the 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. We also note that the modules `psb_base_mod`, `mld_prec_mod` and `psb_krylov_mod`, containing definitions of data types and interfaces to MLD2P4 or PSBLAS routines, must be used by the calling program.

Other versions of multi-level preconditioners can be obtained by changing the default values of the preconditioner parameters. The code fragment reported below shows how to set up a three-level hybrid Schwarz preconditioner, which uses block Jacobi with ILU(0) on the local blocks as post-smoother, and solves the coarsest-level system, which is replicated on the processors, with the LU factorization from UMFPACK.

```
... ...
  call mld_precinit(P,'ML',info,nlev=3)
  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)
... ...
```

The number of levels is specified by using `mld_precinit`; the other preconditioner parameters are set by calling `mld_precset`. Note that some features of the preconditioners are not specified through `mld_precset` (e.g. the type of multilevel framework or the ILU(0) factorization used by the block Jacobi smoother), since they are set by default when `mld_precinit` is called.

6 Related work

The large interest toward using algebraic multilevel preconditioners in the solution of large-scale linear systems has led to their implementation in various parallel software packages.

Algebraic multilevel preconditioners can be built and applied within PETSc [2], which provides a suite of data structures and of routines (Krylov solvers, preconditioners, nonlinear solvers and support routines) for the parallel solution of scientific applications modeled by partial differential equations. It is written in C, with an object-oriented programming style, and uses MPI for message passing communications. PETSc has been developed for C/C++ users, but provides also a Fortran 77/90 interface that allows to access most of its functionalities. However, it includes limited support for direct use of Fortran 90 pointers. One-level AS preconditioners are directly available to users; conversely, multilevel preconditioners can be built by using PETSc facilities, but this require the user to provide at least the restriction and prolongation operators, unless the problem comes from a PDE discretization on a structured grid. On the other hand, PETSc supplies interfaces to the Trilinos/ML and Hypre packages, that implement algebraic multilevel preconditioners (see below).

Classical algebraic multigrid preconditioners are implemented in *BoomerAMG* [28], which is included in *Hypre* [23], a library of solvers and preconditioners for the solution of large and sparse linear systems on massively parallel computers. Different parallel "classical" coarsening schemes are available in BoomerAMG, together with various interpolation and relaxation techniques, including Schwarz smoothers. Hypre is written C (with some exceptions, that are in C++), but provides interfaces for the C, C++, Fortran, and Python languages; it uses MPI for inter-process communications.

ML [27] is a package implementing various types of algebraic multilevel preconditioners, with emphasis on preconditioners based on smoothed aggregation. It also implements other multilevel approaches, such as edge-element eddy current algebraic multigrid for Maxwell's equations, finite-element based two-level schemes, refinement-based multigrid and classical algebraic multigrid. A variety of parallel smoothers is available, including the parallel Schwarz preconditioners from *AztecOO*, an object-oriented version of the Aztec library [38]. All the computational kernels are written in C, with some C++ interfaces provided; MPI is used for message passing. ML can be used as a stand-alone package; however, it is designed to interoperate with other packages of the *Trilinos* [29] framework, which provides an object-oriented C++ software infrastructure, making available common core data structures and abstract solver APIs for different packages, and modern tools for developing and implementing new algorithms for scientific and engineering applications.

Recursive ILU-based multilevel preconditioners are implemented in *pARMS* [33], which is a library of parallel solvers and preconditioners for distributed sparse linear systems. It is written in C and Fortran 77 and uses MPI for inter-process communications. Basic AS and Schur-complement preconditioning techniques are available through the recursive ILU approach.

7 Conclusions and future work

We presented the design objectives, the software architecture and an example of use of MLD2P4, a package of parallel algebraic multilevel preconditioners based on AS methods and on smoothed aggregation. The object-based design coupled with the choice of Fortran 95 was a key issue in achieving goals such as flexibility, extensibility, portability and ease of use, while preserving efficiency.

We plan to extend MLD2P4 with further functionalities, including

- other aggregation algorithms, in particular smoothed aggregation variants for highly nonsymmetric matrices;
- other multilevel cycles, such as general V-cycles, W-cycles and full multigrid ones;
- other base preconditioners.

We also plan to exploit new object-oriented feautures of Fortran 2003, as soon as a wide coverage of such standard is ensured by the compiler vendors.

The source code distribution of MLD2P4, along with its User's and Reference Guide, is available from `http://www.mld2p4.it`.

References

[1] Andrea Aprovitola, Pasqua D'Ambra, Filippo Denaro, Daniela di Serafino, and Salvatore Filippone. Scalable algebraic multilevel preconditioners with application to CFD. In *Proceedings of the International Conference on Parallel Computational Fluid Dynamics*, Lecture Notes in Computational Science and Engineering, Berlin/Heidelberg, Germany, 2009. Springer-Verlag. To appear.

[2] Satishand Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 2.3.3, Argonne National Laboratory, Argonne, IL, USA, 2007. See also http://www.mcs.anl.gov/petsc/petsc-as/.

[3] Randolph E. Bank and Craig C. Douglas. Sparse matrix multiplication package (SMMP). *Adv. Comput. Math.*, 1(1):127–137, 1993.

[4] A. Brandt. Multi-level adaptive solutions to boundary value problems. *Math. Comp.*, 31:333–390, 1977.

[5] A. Brandt. Multiscale scientific computation: review 2001. In T. Chan T.J. Barth and R. Haimes, editors, *Multiscale and Multiresolution Methods: Theory and Applications*, Lecture Notes in Computational Science and Engineering, pages 1–96. Springer-Verlag, Heidelberg, Germany, 2002.

[6] Marian Brezina and Petr Vaněk. A black-box iterative solver based on a two-level Schwarz method. *Computing*, 63(3):233–263, 1999.

[7] Alfredo Buttari, Pasqua D'Ambra, Daniela di Serafino, and Salvatore Filippone. Extending PSBLAS to build parallel Schwarz preconditioners. In J. Dongarra, K. Madsen, and J. Wasniewski, editors, *Applied Parallel Computing*, volume 3732 of *Lecture Notes in Computer Science*, pages 593–602, Berlin/Heidelberg, Germany, 2006. Springer-Verlag.

[8] Alfredo Buttari, Pasqua D'Ambra, Daniela di Serafino, and Salvatore Filippone. 2LEV-D2P4: a package of high-performance preconditioners

for scientific and engineering applications. *Appl. Algebra Engrg. Comm. Comput.*, 18(3):223–239, 2007.

[9] Alfredo Buttari, Victor Eijkhout, Julien Langou, and Salvatore Filippone. Performance optimization and modeling of blocked sparse kernels. *Internat. J. High Perf. Comp. Appl.*, 21(4):467–484, 2007.

[10] Xiao-Chuan Cai and Yousef Saad. Overlapping domain decomposition algorithms for general sparse matrices. *Numer. Linear Algebra Appl.*, 3(3):221–237, 1996.

[11] Xiao-Chuan Cai and Marcus Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM J. Sci. Comput.*, 21(2):792–797, 1999.

[12] Tony F. Chan and Tarek P. Mathew. Domain decomposition algorithms. In *Acta numerica, 1994*, Acta Numer., pages 61–143. Cambridge Univ. Press, Cambridge, UK, 1994.

[13] Pasqua D'Ambra, Daniela di Serafino, and Salvatore Filippone. On the development of PSBLAS-based parallel two-level Schwarz preconditioners. *Appl. Numer. Math.*, 57(11-12):1181–1196, 2007.

[14] Pasqua D'Ambra, Daniela di Serafino, and Salvatore Filippone. *MLD2P4 User's and Reference Guide*, September 2008. Available from http://www.mld2p4.it.

[15] Timothy A. Davis. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Software*, 30(2):196–199, 2004.

[16] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl.*, 20(3):720–755, 1999.

[17] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM J. Matrix Anal. Appl.*, 20(4):915–952, 1999.

[18] Jack J. Dongarra and R. Clint Whaley. A user's guide to the blacs v1.1. Technical report, LAPACK Working Note 94, 1997. Available from http://www.netlib.org/blacs/lawn94.ps.

[19] Iain S. Duff, Michael A. Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Trans. Math. Softw.*, 28(2):239–267, 2002.

[20] Iain S. Duff, Michele Marrone, Giuseppe Radicati, and Carlo Vittoli. Level 3 basic linear algebra subprograms for sparse matrices: A user-level interface. *ACM Trans. Math. Software*, 23(3):379–401, 1997.

[21] Evridiki Efstathiou and Martin J. Gander. Why restricted additive Schwarz converges faster than additive Schwarz. *BIT*, 43(suppl.):945–959, 2003.

[22] Robert D. Falgout. An introduction to algebraic multigrid. *Computing in Science and Engineering*, 8(3):24–33, 2006.

[23] Robert D. Falgout, Jim E. Jones, and Ulrike Meier Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In *Numerical solution of partial differential equations on parallel computers*, volume 51 of *Lect. Notes Comput. Sci. Eng.*, pages 267–294. Springer-Verlag, Berlin, Germany, 2006. See also https://computation.llnl.gov/casc/linear_solvers/sls_hypre.html.

[24] Salvatore Filippone and Alfredo Buttari. *PSBLAS: User's and Reference Guide*, 2008. Available from http://www.ce.uniroma2.it/psblas/.

[25] Salvatore Filippone and Michele Colajanni. PSBLAS: A library for parallel linear algebra computation on sparse matrices. *ACM Trans. Math. Software*, 26(4):527–550, 2000. See also http://www.ce.uniroma2.it/psblas/.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[27] Michael W. Gee, Christofer M. Siefert, Jonathan J. Hu, Ray S. Tuminaro, and Marzio G. Sala. ML 5.0 smoothed aggregation user's guide. Technical Report SAND2006-2649, Sandia National Laboratories, Albuquerque, NM, and Livermore, CA, USA, 2006.

[28] Van Emden Henson and Ulrike Meier Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Appl. Numer. Math.*, 41:155–177, 2000.

[29] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos Project. *ACM Trans. Math. Soft.*, 31(3):397–423, 2005.

[30] D. Keyes. Domain decomposition methods in the mainstream of computational science. In *Proceedings of the 14th International Conference on Domain Decomposition Methods*, pages 79–93, Mexico City, Mexico, 2003. UNAM Press.

[31] J. Mandel, M. Brezina, and P. Vaněk. Energy optimization of algebraic multigrid bases. *Computing*, 62(3):205–228, 1999.

[32] Yousef Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2003.

[33] Yousef Saad and Masha Sosonkina. pARMS: A package for the parallel iterative solution of general large sparse linear systems user's guide. Technical Report UMSI2004-8, Minnesota Supercomputing Institute, Minneapolis, MN, USA, 2004.

[34] Marzio Sala and Raymond S. Tuminaro. A new Petrov-Galerkin smoothed aggregation precondiioner for nonsymmetric linear systems. *SIAM J. Sci. Comput.*, 31(1):143–166, 2008.

[35] Barry F. Smith, Petter E. Bjørstad, and William D. Gropp. *Domain decomposition. Parallel multilevel methods for elliptic partial differential equations*. Cambridge University Press, Cambridge, UK, 1996.

[36] Marc Snir, Steve Otto, Steven Huss-Lederman, David W. Walker, and Jack J. Dongarra. *MPI: The Complete Reference. Vol. 1 – The MPI Core*. Scientific and Engineering Computation. The MIT Press, Cambridge, MA, USA, second edition, 1998.

[37] Klaus Stüben. Algebraic multigrid (AMG): An introduction with applications. Technical Report 70, GMD, Schloss Birlinghoven, Sankt Augustin, Germany, 1999.

[38] Ray S. Tuminaro, Michael A. Heroux, Scott A. Hutchinson, and John N. Shadid. Official Aztec user's guide - version 2.1, 1999. See also http://www.cs.sandia.gov/CRF/aztec1.html.

[39] Ray S. Tuminaro and Charles Tong. Parallel smoothed aggregation multigrid: Aggregation strategies on massively parallel machines. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Dallas, TX, USA, 2000. CDROM.

[40] P. Vaněk, J. Mandel, and M. Brezina. Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, 56(3):179–196, 1996.