AAECC
DOI 10.1007/s00200-007-0035-z

2LEV-D2P4: a package of high-performance preconditioners for scientific and engineering applications

Alfredo Buttari · Pasqua D'Ambra · Daniela di Serafino · Salvatore Filippone

Received: 22 December 2005 / Revised: 25 October 2006

Abstract We present a package of parallel preconditioners which implements one-level and two-level Domain Decomposition algorithms on the top of the PSBLAS library for sparse matrix computations. The package, named 2LEV-D2P4 (Two-LEVel Domain Decomposition Parallel Preconditioners Package based on PSBLAS), currently includes various versions of additive Schwarz preconditioners that are combined with a coarse-level correction to obtain two-level preconditioners. A pure algebraic formulation of the preconditioners is considered. 2LEV-D2P4 has been written in Fortran 95, exploiting features such as abstract data type creation, functional overloading and dynamic memory management, while providing a smooth path towards the integration in legacy application codes. The package, used with Krylov solvers implemented

A. Buttari
Innovative Computing Lab, Department of Computer Science,
University of Tennessee at Knoxville,
1122 Volunteer Blvd. Knoxville, TN 37996, USA
e-mail: buttari@cs.utk.edu

P. D'Ambra (✉)
Institute for High-Performance Computing and Networking, CNR,
Via Pietro Castellino 111, 80131 Naples, Italy
e-mail: pasqua.dambra@na.icar.cnr.it

D. di Serafino
Department of Mathematics, Second University of Naples,
Via Vivaldi 43, 81100 Caserta, Italy
e-mail: daniela.diserafino@unina2.it

S. Filippone
Department of Mechanical Engineering, University of Rome "Tor Vergata",
Viale del Politecnico, 00133 Rome, Italy
e-mail: salvatore.filippone@uniroma2.it

in PSBLAS, has been tested on large-scale linear systems arising from model problems and real applications, showing its effectiveness.

Keywords Parallel numerical software · Algebraic two-level preconditioners · Sparse linear algebra

1 Introduction

The solution of large and sparse linear systems,

$$Ax = b, \quad A \in \Re^{n \times n}, \quad x, b \in \Re^n, \tag{1}$$

is often the main computational kernel of large-scale applications in science and engineering. These systems typically arise from the discretization of partial differential equations (PDEs), but they may also come from applications in diverse areas such as circuits analysis, chemical engineering, queueing systems and economic models. Krylov iterative solvers coupled with suitable preconditioners are often the methods of choice for their solution, especially when the matrix dimension reaches the order of 10^6–10^8, as in 3D multiphysics and multiscale simulations.

Over the last few years substantial efforts have been expended in developing parallel algorithms and software implementing Krylov methods and preconditioners, in order to provide computational scientists with effective tools for building their application codes.

In this paper, we present a package of preconditioners for high-performance architectures, based on Domain Decomposition algorithms of Schwarz type. It is implemented on top of Parallel Sparse BLAS (PSBLAS) [16], a library for sparse basic linear algebra operations that provides parallel versions of the Sparse BLAS computational kernels proposed in [15] and auxiliary routines for the creation and management of distributed sparse matrices. Our package currently includes different versions of one-level and two-level Schwarz preconditioners [6,7,9,21]; therefore, it has been named *2LEV-D2P4*, that is Two-LEVel Domain Decomposition Parallel Preconditioners Package based on PSBLAS. It has been written in Fortran 95, to exploit modern features of this language, while allowing an easy integration in (Fortran) legacy application codes.

The paper is organized as follows: in Sect. 2 we review the algebraic formulation of the considered preconditioners; in Sect. 3 we outline the software architecture and the data structures of 2LEV-D2P4 and in Sect. 4 we present performance results obtained by applying to large-scale matrices different preconditioners available in the package; finally, we draw our conclusions in Sect. 5.

2 Preconditioner formulation

Let $G = (W, E)$ be the adjacency graph associated with the coefficient matrix A of the linear system (1); here W and E are the vertex set and the edge set,

respectively, and we are assuming that the sparsity pattern of A is symmetric. Two vertices are called adjacent if there is an edge connecting them. We also say that two matrix rows are adjacent if this property holds for the corresponding vertices. A δ-overlap partition of W can be defined recursively. A 0-overlap partition consists of m disjoint nonempty sets $W_i^0 \subset W$ such that $\cup_{i=1}^m W_i^0 = W$. For $\delta > 0$, the δ-overlap partition which corresponds to the 0-overlap one consists of the sets $W_i^\delta \supset W_i^{\delta-1}$ obtained by including the vertices that are adjacent to any vertex in $W_i^{\delta-1}$. Let n_i^δ be the size of W_i^δ and $R_i^\delta \in \Re^{n_i^\delta \times n}$ the restriction operator that maps a vector $v \in \Re^n$ onto the vector $v_i^\delta \in \Re^{n_i^\delta}$ containing the components of v corresponding to the vertices in W_i^δ. The transpose of R_i^δ is a prolongation operator from $\Re^{n_i^\delta}$ to $\Re^n$. The matrix $A_i^\delta = R_i^\delta A (R_i^\delta)^T \in \Re^{n_i^\delta \times n_i^\delta}$ can be regarded as a restriction of A corresponding to the set W_i^δ.

The *classical Additive Schwarz* (AS) preconditioner is defined by

$$M_{AS}^{-1} = \sum_{i=1}^{m} (R_i^\delta)^T (A_i^\delta)^{-1} R_i^\delta, \tag{2}$$

while the *Restricted AS* (RAS) and *AS with Harmonic extension* (ASH) variants are defined by

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

where $\tilde{R}_i^0 \in \Re^{n_i^\delta \times n}$ is obtained by zeroing the rows of R_i^δ identified by the vertices in $W_i^\delta \setminus W_i^0$. For all the AS preconditioners A_i^δ is assumed to be nonsingular. The application of the preconditioner (2) with a Krylov solver requires the solution of a system of the form $M_{AS} z = v$, that corresponds to the following steps:

$$v_i = R_i^\delta v, \quad i = 1, \ldots, m, \tag{3}$$

$$\text{solve} \quad A_i^\delta w_i = v_i, \quad i = 1, \ldots, m, \tag{4}$$

$$z = \sum_{i=1}^{m} (R_i^\delta)^T w_i. \tag{5}$$

In RAS R_i^δ in (5) is replaced by $\tilde{R}_i^0$, while in ASH R_i^δ in (3) is replaced by $\tilde{R}_i^0$.

The AS preconditioners exhibit an intrinsic parallelism, since the computations concerning different subdomains can be performed by different processors. On the other hand, the convergence theory for the AS preconditioners shows that, when they are used in conjunction with a Krylov subspace method, the convergence rapidly improves as the overlap δ increases, while it deteriorates as the number m of subsets W_i^δ grows. Therefore, in a parallel setting, the AS preconditioners are often used in conjuction with some coarse space, where the original linear system can be approximated to provide a suitable

improvement to the solution [8,9,21]. The use of this space introduces some extra work that has a sequential nature, but is necessary for developing scalable preconditioning algorithms.

In a pure algebraic setting, a coarse-space approximation A_C of the matrix A is usually built with a Galerkin approach [21]. Given a set W_C of coarse vertices, with size n_C, and a suitable restriction operator $R_C \in \Re^{n_C \times n}$, A_C is defined as $A_C = R_C A R_C^T$ and the coarse-space correction matrix to be combined with the AS preconditioners is obtained as

$$M_C^{-1} = R_C^T A_C^{-1} R_C, \tag{6}$$

where A_C is assumed to be nonsingular. W_C and R_C may be built by using a *smoothed aggregation* technique [3,23,24]. To build W_C, the vertices of W are grouped into disjoint subsets, which are then assumed as the coarse-space vertices. The transpose of R_C is built starting from a piecewise constant interpolation operator P, from $\Re^{n_C}$ to $\Re^n$, and applying to P a smoother, in order to remove spurious oscillatory components from the range of the prolongator.

M_C can be combined with any AS preconditioner (henceforth denoted by M_{1L}), in either an additive or a multiplicative way, to obtain two-level preconditioners. The additive combination leads to

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

which corresponds to applying the coarse-level correction and the basic AS preconditioner independently and then summing up the results. An example of multiplicative combination is given by the following hybrid preconditioner, that we refer to as *2LH-post*

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

This corresponds to applying first the coarse-level correction and then, i.e. as post-smoother, the basic AS preconditioner

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

Other two-level hybrid preconditioners are obtained by applying M_{1L} as pre-smoother or pre- and post-smoother.

3 Software architecture and data structures

The main thrust of our research effort has been to develop a package of parallel Schwarz preconditioners based on standard kernels for sparse linear algebra operations. The choice of PSBLAS has been motivated by the need of having

```
type psb_dspmat_type
  integer     :: m, k
  character   :: fida(5), descra(10)
  integer     :: infoa(psb_ifa_size_)
  real(kind(1.d0)), pointer :: aspk(:)=>null()
  integer, pointer :: ia1(:)=>null(), ia2(:)=>null()
  integer, pointer :: pr(:)=>null(), pl(:)=>null()
end type psb_dspmat_type
```

Fig. 1 Sparse matrix data type

a portable, efficient and modular software infrastructure, that provides also a smooth path for integration in application codes [2,17]. 2LEV-D2P4 has been designed to fully exploit the existing functionalities of the PSBLAS library; however, its implementation has required some extensions of the existing PSBLAS kernels.

The package has a multi-layered software architecture where three main layers can be identified. The lower layer consists of the original and the new 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.

3.1 Lower layer

PSBLAS provides basic operators needed to implement iterative methods for the solution of sparse linear systems on distributed-memory parallel computers. The library is written in Fortran 95, exploiting modern features such as facilities for data encapsulation and abstract data type creation, functional overloading and dynamic memory management, which allow the development of a modular source code, while maintaining the serial efficiency of Fortran kernels. Interprocess data communications are performed using BLACS [13] and MPI [22].

PSBLAS supports a general row-block distribution of sparse matrices, with conformal distribution of vectors and dense matrices. Two main distributed data structures, implemented as Fortran 95 derived data types, are used to hold the information concerning a distributed sparse matrix: the *Sparse Matrix* and the *Communication Descriptor*.

The Sparse Matrix data type is reported in Fig. 1; the main components of the data type include the array `aspk` containing the nonzero entries in the matrix rows assigned to the local processor, while the arrays `ia1` and `ia2` contain the corresponding row and column indices, in a format determined by the labels contained in `fida` and `descra`, thus providing run-time polymorphism. Additional auxiliary information may be stored in `infoa`, depending on the storage format used.

The Communication Descriptor is an essential data structure for handling communication operations pertaining to the implementation of a solver for a specific matrix instance. It is logically associated with the sparsity pattern of the matrix as distributed on the parallel machine. We assume that there is a

```
type psb_desc_type
 integer, pointer :: matrix_data(:)=>null()
 integer, pointer :: halo_index(:)=>null()
 integer, pointer :: overlap_index(:)=>null(), overlap_elem(:)=>null()
 integer, pointer :: loc_to_glob(:)=>null(), glob_to_loc(:)=>null()
end type psb_desc_type
```

Fig. 2 Communication descriptor data type

global (row) index space $\{1, \ldots, N\}$, which is partitioned among the processors; we further assume that during computations each processor p refers to its own subset of the index space by means of a contiguous local numbering, $1, \ldots, N_p$.

The contents of this data structure, shown in Fig. 2, are divided in three groups

- `matrix_data`, that contains general information about the global matrix, such as the total number of rows and columns, the number of rows stored on the current processor, and the number of boundary rows, i.e. of local rows that are adjacent to rows owned by different processors;
- `halo_index`, `overlap_index` and `overlap_elem`, that contain processed lists of indices to be used in the communication operations among processors, such as the indices of adjacent rows owned by other processors and the indices of boundary rows that are replicated in other processors.
- `loc_to_glob` and `glob_to_loc`, that are used to handle the mapping between local and global indices;

The contents of these data structures have been described here for the sake of completeness, but the user actually needs never consider them directly. An important part of the PSBLAS library is in fact devoted to auxiliary data and environment management routines that take care of building the data structures. These routines only require from the user the choice of the global index space allocation and a list of matrix entries.

The PSBLAS library also contains the computational kernels needed to implement Krylov subspace iterations, such as sparse-matrix by vector product, dot product and vector sum. All computations are performed by means of an implementation of the serial sparse BLAS proposal of [15].

As previously mentioned, the construction of the one-level AS preconditioners has required the extension of the set of auxiliary routines, to build the matrices A_i^δ. Routines have been developed to build an extended descriptor and to enlarge A_i^0 in each processor, i.e. to gather the rows of the matrix A that correspond to the indices in $W_i^\delta \backslash W_i^0$. This functionality can be used in a more general context, e.g. to build extended stencils, that are often required by numerical simulations involving PDEs. The construction of the coarse-level matrix A_C and its application has required the extension of the set of PSBLAS sequential kernels with routines performing sparse matrix diagonal scaling, sparse matrix transpose and sparse matrix by sparse matrix multiplication. The last two operations have been implemented by integrating into PSBLAS the SMMP software [1]. We note that, although the sparse matrix multiplication is not considered in the last SBLAS standard, the possibility of its future inclusion

```
type psb_dbaseprc_type
  type(psb_dspmat_type), pointer :: av(:) => null()
  real(kind(1.d0)), pointer      :: d(:)  => null()
  type(psb_desc_type), pointer   :: desc_data => null()
  integer, pointer               :: iprcparm(:) => null()
  real(kind(1.d0)), pointer      :: dprcparm(:) => null()
  integer, pointer               :: mlia(:)  => null()
  integer, pointer               :: nlaggr(:)  => null()
  type(psb_dspmat_type), pointer :: aorig    => null()
  real(kind(1.d0)), pointer      :: dorig(:) => null()
end type psb_dbaseprc_type

type psb_dprec_type
  type(psb_dbaseprc_type), pointer :: baseprecv(:) => null()
  integer                          :: prec, base_prec
end type psb_dprec_type
```

Fig. 3 Preconditioner data types

has been foreseen by the BLAS Technical Forum [14]. Further details on the extension of PSBLAS can be found in [4,10].

3.2 Middle layer

The middle layer of 2LEV-D2P4 consists of the routines that implement the construction and the application phases of the preconditioners. A key issue in the implementation of this layer has been the definition of a *Preconditioner* data structure (see Fig. 3), that reuses the basic PSBLAS data structures to exploit the functionalities of the PSBLAS computational and auxiliary routines.

The basic idea is to have a `psb_dbaseprc_type` data type that holds a generic one-level "base" preconditioner; its contents are described by the entries in `iprcparm`, defining the type of preconditioner, how many layers of overlap are considered, which kind of factorization is employed and so on. The corresponding sparse factors L and U, are contained in the arrays `av` and `d`, and the associated descriptor is in `desc_data`.

For the two-level preconditioners, the `psb_dprec_type` data structure contains an array of two base preconditioner data types, pointed by `baseprecv`, that are associated to the fine and coarse levels. In the entry corresponding to the coarse level, `mlia` and `nlaggr` hold the mapping between the lower level indices and the upper level aggregates. When a smoothed aggregation is employed (again signaled by the contents of `iprcparm`), the aggregation operators needed to move between the levels are stored into additional entries of `av`, together with the coarse matrix and its factors. Finally, we note that `aorig` is a pointer to the lower level matrix A that allows the application of the residual operator to be used in a multiplicative multilevel framework; this avoids that the routine applying a preconditioner to a vector has to reference explicitly in its interface the system matrix A, which would be cumbersome for cases such as block-Jacobi (i.e. AS with overlap 0) in which there is no need to reference it. We note that implementing the multilevel preconditioner as a

vector of base preconditioners enables to reuse the routines for building and applying the preconditioner clearly separating the mapping between levels.

The construction of the one-level preconditioners has been implemented in two main steps: the identification of the overlap indices needed to build A_i^δ, through the algorithm described in [4], and the enlargment of the matrix A_i^0, through a new auxiliary PSBLAS routine mentioned in Sect. 3.1. For the construction of the coarse matrix A_C the smoothed aggregation technique [3,24] has been chosen. This requires the implementation of three main tasks, detailed in [10]: a parallel decoupled aggregation procedure, to build the coarse-space vertex set W_C from the original vertex set W; the application of a damped Jacobi smoother to a simple piecewise constant interpolation operator, to obtain the coarse-to-fine prolongation operator R_C; the construction of the coarse matrix $A_C = R_C A R_C^T$, where A_C can be distributed among the processors or replicated on each of them. The latter two steps have been performed by using the new sequential sparse matrix operators integrated into PSBLAS.

The application of the preconditioners has been implemented by exploiting different PSBLAS kernels. Sparse matrix management routines are used to perform the steps (3) and (5) of the basic AS preconditioners, while the parallel sparse matrix by vector multiplication and dense vector sum routines are used to apply the coarse-level correction matrix (6) and to combine it with the one-level preconditioners. The solution of the system (4), involving the matrices A_i^δ, is accomplished by either the ILU or the LU factorization. The sequential ILU routine available in PSBLAS is used in the former case; an interface to UMFPACK [12], version 4.4, has been developed to deal with the latter case. The same options are available for the system involving A_C, when this matrix is replicated among the processors. On the other hand, to solve the coarse-level systems when A_C is distributed, a block-Jacobi routine has been developed, that uses the ILU or the LU factorization on the coarse matrix diagonal blocks held by the processors.

3.3 Upper layer

At the upper layer of 2LEV-D2P4, two black-box routines encapsulate all the functionalities for the construction and the application of any of the one-level and two-level preconditioners. The values of the parameters that define a specific preconditioner may be set by means of a third routine.

To build a preconditioner, a user of 2LEV-D2P4 first sets the preconditioner parameters, through the routine `psb_precset`, and then creates and defines the Preconditioner data structure, through `psb_precbld`. The APIs of these routines are shown in Fig. 4. In the `psb_precset` API, the main input parameters are the Preconditioner data structure, named `prec`, and the `ptype` string variable, defining the choice of the preconditioner made by the user. Currently, the following choices are available: no preconditioner, Jacobi, block-Jacobi (special case of one-level Schwarz), one-level Schwarz and two-level Schwarz preconditioners. The different items of the `iv` optional array are used to specify

```
subroutine psb_precset(prec,ptype,info,iv,rs,rv)
  type(psb_dprec_type), intent(inout)    :: prec
  character(len=*), intent(in)           :: ptype
  integer, intent(out)                   :: info
  integer, optional, intent(in)          :: iv(:)
  real(kind(1.d0)), optional, intent(in) :: rs
  real(kind(1.d0)), optional, intent(in) :: rv(:)
end subroutine psb_precset

subroutine psb_precbld(a,prec,desc_a,info,upd)
  integer, intent(out)                :: info
  type(psb_dspmat_type), intent(in), target  :: a
  type(psb_dprec_type),intent(inout)         :: prec
  type(psb_desc_type), intent(in)            :: desc_a
  character, intent(in), optional            :: upd
end subroutine psb_precbld
```

Fig. 4 APIs of the routines for the preconditioner setup

```
subroutine psb_precaply(prec,x,y,desc_data,info,trans,work)
  type(psb_desc_type), intent(in)   :: desc_data
  type(psb_dprec_type), intent(in)  :: prec
  real(kind(0.d0)), intent(inout)   :: x(:), y(:)
  integer, intent(out)              :: info
  character(len=1), optional        :: trans
  real(kind(0.d0)), intent(inout), optional, target :: work(:)
end subroutine psb_dprecaply
```

Fig. 5 API of the routine for the preconditioner application

the parameters of the Schwarz preconditioners. For the one-level preconditioners, the entries specify the number of overlap layers, the types of restriction (R_i^δ or $\tilde{R}_i^0$) and prolongation, that define the AS variants, and the type of factorization to be employed. For the two-level preconditioners, the user first sets the parameters of the fine level, and afterwards the parameters concerning the coarse-level correction. The user may specify the type of two-level framework (additive or multiplicative), details of the aggregation algorithm, details on the application of the one-level preconditioner (as pre-smoother, post-smoother or both), the coarse matrix storage (distributed or replicated), the type of solver to be employed at the coarse level and related details. Default values are provided for all the optional parameters.

Once the preconditioner has been built, it may be applied at each iteration of a Krylov solver by calling the routine `psb_prcaply`, which has the API shown in Fig. 5. This routine computes $y = op(M^{-1})\,x$, where M is the previously built preconditioner, stored in the `prec` data structure, and op denotes the matrix itself or its transpose, according to the value of `trans`.

An example of use of 2LH-post, using RAS with overlap 1 as basic preconditioner, a distributed coarse matrix and four block-Jacobi sweeps with the UMFPACK LU factorization on the blocks, is sketched in Fig. 6. In this example, the preconditioner is coupled with the BiCGSTAB solver implemented in PSBLAS. More details on the use of the routines for the setup and the application of the preconditioners can be found in [5].

```
! 2LEV-D2P4 example program
  use psb_sparse_mod
! sparse matrices
  type(psb_dspmat_type) :: a
  type(psb_desc_type)   :: desc_a
  type(psb_dprec_type)  :: pre
!
  ...
! read and assemble matrix A and right-hand
! side vector b
  ...
! set preconditioner options
  novr = 1
  nsweep = 4
  call psb_precset(pre,'asm',iv=(/novr,halo_,none_/))
  call psb_precset(pre,'ml',&
    & iv=(/mult_ml_prec_,post_smooth_,loc_aggr_,mat_distr_,&
    &      f_umf_,nsweep/))
!
! build preconditioner
  call psb_precbld(a,pre,desc_a,info)
!
! set solver parameters
  ...
! solve Ax=b with preconditioned BiCGSTAB
  call psb_bicgstab(a,pre,b,x,tol,desc_a,info)
  ...
```

Fig. 6 Example of use of 2LEV-D2P4

Table 1 Dimension and number of nonzeros of the test matrices

Matrix	Dimension	Nonzeros
shipsec5	179,860	4,598,604
thm-rod	3,180,184	15,818,394
thm-plate	600,000	2,996,800

4 Performance results

The package 2LEV-D2P4 has been tested on different large-scale matrices, arising either from real applications or from model problems. For the sake of space, we show here only the results concerning selected matrices and preconditioners.

Three matrices are considered, named shipsec5, thm-rod and thm-plate. The first one comes from a ship modelling application (DNV-PARASOL ship section 5-1999-01-15) and is available from the University of Florida Sparse Matrix Collection [11]. The matrix thm-rod arises from an industrial application, modelling the steady-state thermal diffusion in a copper rod; the problem is discretized by using a finite volume cell-centered scheme on an unstructured tetrahedral mesh. Finally, the matrix thm-plate arises from a central finite-difference discretization, on a regular mesh, of the equation describing the steady-state thermal diffusion in a homogeneous plate. The dimension and the number of nonzeros of the three matrices are reported in Table 1.

We show the results concerning RAS and 2LH-post with RAS at the fine level, that turn out to be the most effective among the one-level and the two-level preconditioners, respectively. Three variants of 2LH-post are considered,

differing in the coarse-space correction. In the ones named 2LH-DI and 2LH-DU the coarse matrix is distributed among the processors and four block-Jacobi sweeps are applied to the corresponding system; the number of sweeps has been determined experimentally, as the one that generally minimizes the execution time. At each block-Jacobi sweep, the ILU factorization from PSBLAS and the LU factorization from UMFPACK are used by 2LH-DI and 2LH-DU, respectively, to deal with the diagonal blocks of the coarse matrix held by each processor. In the remaining variant, named 2LH-RU, the coarse-space matrix is replicated on all the processors and the corresponding system is solved by using the UMFPACK LU factorization. In all the preconditioners, the RAS subdomain systems are solved by ILU. The results concerning the overlaps 0 and 1 are reported.

The preconditioners have been applied as right preconditioners with the BiCGSTAB solver available in PSBLAS, choosing the null vector as starting guess and the vector of all 1's as right-hand side. The iterations have been stopped when the ratio between the 2-norms of the residual and of the right-hand-side is less than 10^{-6}; a maximum number of 5,000 iterations has been also set, but it has been never reached.

Two data distributions have been considered for each matrix: a row-block one, where each processor holds equal-sized disjoint blocks of consecutive rows according to the well-known BLACS one-dimensional pure-block mapping, and a distribution obtained by using the multilevel recursive bisection graph partitioning algorithm implemented in Metis [19]. For each problem we show the results for the most efficient data distribution, i.e. the Metis distribution for shipsec5 and thm-rod and the block one for thm-plate. Conformal distributions have been applied to the right-hand side and solution vectors. Note that the data distributions implicitly define domain decompositions such that the number of subdomains is equal to the number of processors.

We performed a comparison between 2LEV-D2P4 and the ML package, included in Trilinos [18], version 6.0 that implements algebraic multilevel preconditioners based on the smoothed aggregation. Trilinos provides a C++ object-oriented framework where ML is interfaced with various solvers and preconditioners, that can be used as smoothers, coarse-level solvers and outer iterative solvers. Two ML versions of 2LH-post have been run, that use the RAS implementation provided by Trilinos-AztecOO, a parallel decoupled smoothed aggregation algorithm and different coarse-level solvers. The ILU factorization has been applied within RAS; the coarse matrix has been either gathered in a single processor and factorized by the UMFPACK LU implementation, or distributed among the processors and factorized by the LU implementation provided by SuperLU-DIST [20]. The RAS version provided by AztecOO has been also compared with the corresponding one available in 2LEV-D2P4. In the following the above three Trilinos preconditioners are referred to as Tril-2LH-U, Tril-2LH-S and Tril-RAS. The AztecOO BiCGSTAB implementation has been used as Krylov solver. The comparison has been performed on thm-plate, which is a reference test case for multilevel Schwarz preconditioners.

Table 2 Number of iterations on shipsec5

shipsec5						
np	RAS		2LH-DI		2LH-DU	
	ov 0	ov 1	ov 0	ov 1	ov 0	ov 1
1	402	402	396	396	367	367
2	479	436	464	386	422	367
4	461	384	498	384	464	442
8	644	457	709	436	776	430
16	818	483	717	508	630	619
32	839	488	553	412	581	513
64	963	499	561	535	732	557

Table 3 Number of iterations on thm-rod

thm-rod						
np	RAS		2LH-DI		2LH-DU	
	ov 0	ov 1	ov 0	ov 1	ov 0	ov 1
1	120	120	36	36	–	–
2	143	144	44	38	–	–
4	162	142	41	39	–	–
8	164	140	43	41	–	–
16	173	149	42	38	19	19
32	166	147	42	37	22	21
64	162	151	39	41	23	22

All the tests have been carried out on a Beowulf-class cluster installed at the Innovative Computing Laboratory of the University of Tennessee at Knoxville. Each node of the cluster has an AMD Opteron dual-processor (model 240, 1.4 GHz), running the Debian Linux 3.1 operating system with kernel 2.6.13, and 2 GBytes of memory; the nodes are connected with Myrinet network interfaces. We used a development snapshot of the GNU Compiler Collection version 4.2, including C, C++ and Fortran 95 compilers, and the specific MPI implementation for the Myrinet interface.

Tables 2, 3, 4 show, for all the test problems, the number of BiCGSTAB iterations with the different preconditioners on np = 1, 2, 4, 8, 16, 32, 64 processors (note that in Table 4 there is only one column for Tril-2LH-U and Tril-2LH-S, since they have the same iteration counts). The corresponding execution times, in seconds, are plotted in Figs. 7, 8, 9. The data concerning 2LH-RU on shipsec and thm-rod and 2LH-DU on thm-rod with np = 1, 2, 4, 8 are missing, since these cases could not be run due to excessive memory requirements for the UMFPACK LU factorization.

With shipsec5, the use of a coarse-level correction does not result in an improvement in terms of iterations, except in a few cases; instead, the number of iterations of 2LH-DI and 2LH-DU is often greater than that of RAS. On the

Table 4 Number of iterations on thm-plate

thm-plate												
	RAS		Tril-RAS		2LH-DI		2LH-DU		2LH-RU		Tril-2LH-U/S	
np	ov 0	ov 1	ov 0	ov 1	ov 0	ov 1	ov 0	ov 1	ov 0	ov 1	ov 0	ov 1
1	740	740	769	769	183	183	5	5	5	5	5	5
2	802	732	690	693	205	182	19	18	6	5	6	5
4	726	870	798	762	201	177	29	26	6	5	6	5
8	729	758	813	736	200	193	44	35	6	5	6	5
16	692	783	782	832	182	191	61	56	6	5	6	5
32	765	705	827	729	196	191	81	78	7	5	7	5
64	840	844	797	737	180	171	113	103	7	5	6	5

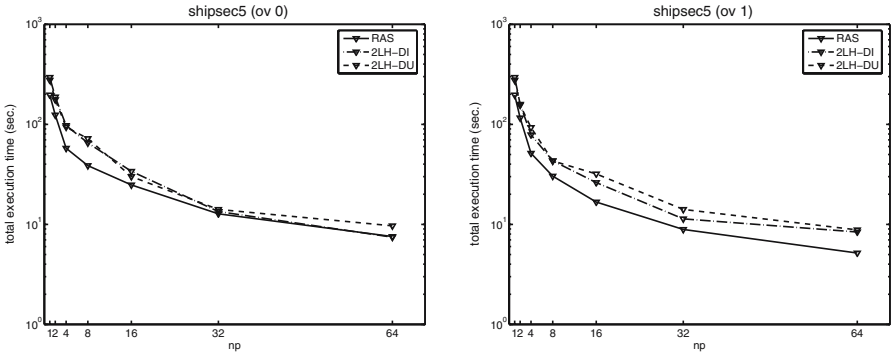

Fig. 7 Execution times on shipsec5 (*left* overlap 0; *right* overlap 1)

other hand, a significant reduction of the iterations can be observed when RAS goes from overlap 0 to overlap 1; on more than 8 processors, RAS generally shows a number of iterations smaller than the 2LH-post versions. Accordingly, the smallest execution times are generally obtained by RAS with overlap 1, followed by RAS with overlap 0. Even when 2LH-DI and 2LH-DU have an iteration count smaller than RAS, this is not enough to balance the large execution times needed for building and applying the coarse-space corrections.

The preconditioners behave differently on thm-rod and thm-plate, that come from pure elliptic problems. With all the versions of 2LH-post, the number of iterations is substantially reduced with respect to RAS; as expected, the more accurate the solution of the coarse-level system, the stronger the reduction in the number of iterations. In particular, with the thm-plate test case, the two-level version that uses the LU factorization on the overall coarse matrix performs a number of iterations that is less than 1% of the RAS one; the same holds for Trilinos. With both test cases, the execution times of all the preconditioners do not vary significantly with the overlap. On thm-rod, the smallest execution times are obtained with 2LH-DI, where the cost of the coarse grid correction is offset by the time gain resulting from the reduction of the iteration count. This is not the case for 2LH-DU, where the cost of the LU factorization inside

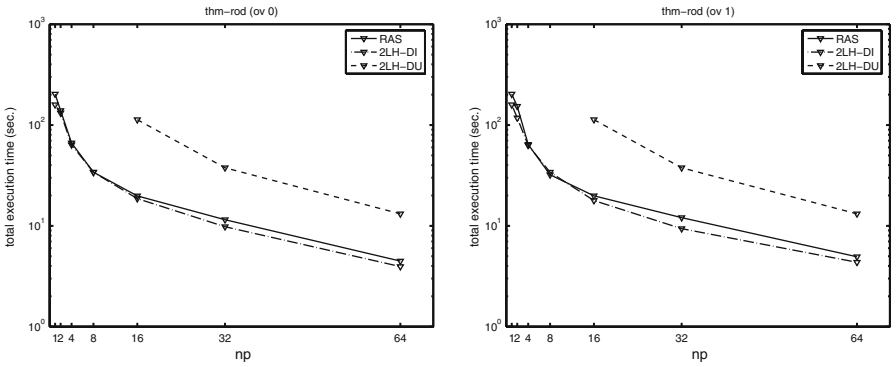

Fig. 8 Execution times on thm-rod (*left* overlap 0; *right* overlap 1)

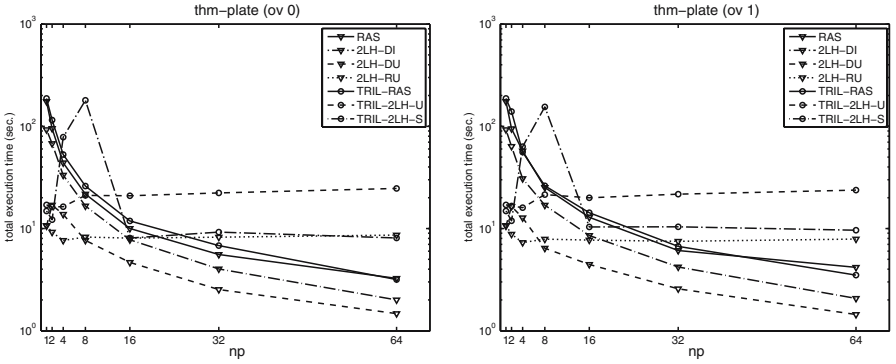

Fig. 9 Execution times on thm-plate (*left* overlap 0; *right* overlap 1)

the block-Jacobi procedure increases the execution time above that of RAS. On thm-plate the smallest execution times are obtained with 2LH-RU when up to 4 processors are used, and with 2LH-DU when more processors are used; with large numbers of processors 2LH-DI also outperforms 2LH-RU. These results show that the performance of the two-level preconditioners depends on a tradeoff between the accuracy and the cost of the coarse-level correction. RAS is slower than all the two-level preconditioners, except 2LH-RU on 32 and 64 processors.

By comparing RAS and Tril-RAS on thm-plate, we see that RAS generally outperforms Tril-RAS with overlap 0, except on 64 processors, where they have about the same execution time. The two preconditioners lead to very similar execution times with overlap 1, except on 64 processors, where TRIL-RAS is slightly faster. We also note that the number of iterations is significantly different for the two solvers. Indeed, a detailed analysis has shown that for this test case the number of iterations is quite sensitive to the parallel algorithm used for the computation of the dot products, which is different in PSBLAS and AztecOO. The situation is very different when the two-level preconditioners are considered. The time plots of 2LH-RU and Tril-2LH-U have similar behaviours, but 2LH-RU is much faster than the Trilinos counterpart for both overlap

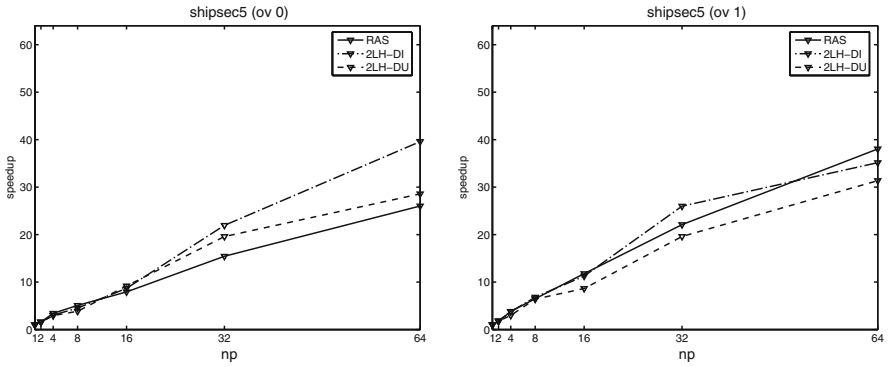

Fig. 10 Speedup on shipsec5 (*left* overlap 0; *right* overlap 1)

0 and 1. Tril-2LH-S and 2LH-RU have close execution times, except for 4 and 8 processors, on which the time of Tril-2LH-S is much geater than expected. In general, none of these two-level preconditioners is able to get a time decrease as the number of processors grows: we note also that the 2LH-RU, Tril-2LH-U and Tril-2LH-S preconditioners give essentially the same iteration counts, since the coarse-level correction removes the dot product effects mentioned above.

In Figs. 10, 11, 12 the speedup values corresponding to the previous execution times are plotted. For thm-rod the speedup of 2LH-DU is computed with respect to the execution time on 16 processors. On shipsec5 all the preconditioners show a satisfactory speedup; in particular, RAS with overlap 1, which leads to the smallest execution time, has a speedup of 22 on 32 processors and of 38 on 64 processors. On thm-rod, RAS and 2LH-DI achieve good speedups; for example, with 64 processors and overlap 0, we obtain speedups of 45 and 40, respectively. For the thm-plate test case, high speedup values are obtained by RAS, 2LH-DI and Tril-RAS; with overlap 0, their values on 64 processors are 54, 46 and 59, respectively. On the other hand, the fastest preconditioner, i.e. 2LH-DU, achieves smaller speedup values (e.g. around 9 on 64 processors, with overlap 0 and 1); this is due to the large increase in the number of iterations, and hence in the execution time, with the number of processors. No gain in terms of execution time is obtained with 2LH-RU and Tril-2LH-U on more than 4 processors and with Tril-2LH-S on more than 2 processors. This confirms that the use of distributed inexact coarse-level corrections pays off in a parallel setting.

5 Conclusions and future work

In this paper, we have described 2LEV-D2P4, a package of one-level and two-level algebraic Schwarz preconditioners for high-performance computers, based on the PSBLAS library. We have provided a general overview of the package, focusing on its software architecture and data structures, functionalities and user interface. We have also presented performance results on large-scale problems, showing the effectiveness of 2LEV-D2P4. We note that the choice of a

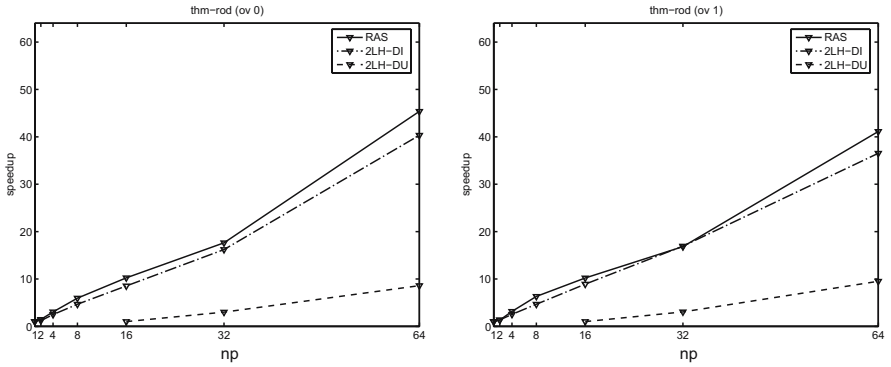

Fig. 11 Speedup on thm-rod (*left* overlap 0; *right* overlap 1)

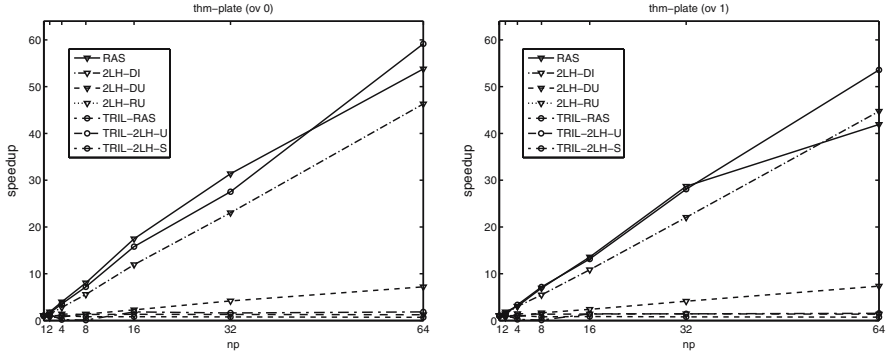

Fig. 12 Speedup on thm-plate (*left* overlap 0; *right* overlap 1)

portable, efficient and modular infrastructure such as PSBLAS has allowed the development of a layered software package, that can be easily maintained and extended.

Future work includes the extension of the package to multilevel Schwarz preconditioners and the integration and testing of the preconditioners inside engineering applications; we will also address the inclusion of other factorizations, such as ILU(n) and ILU with threshold, and more sophisticated aggregation algorithms.

Acknowledgments We thank Jack Dongarra for the usage of the machine on which the results of Sect. 4 have been obtained. We thank also Stefano Toninel and David Schmidt for providing the test case concerning the thermal diffusion in a copper rod. Finally, we thank the referees for their advice, that helped us to improve the quality of the paper.

References

1. Bank, R.E., Douglas, C.C.: SMMP: sparse matrix multiplication package. Adv. Comput. Math. **1**, 127–137 (1993)
2. Bella, G., Filippone, S., De Maio, A., Testa, M.: A simulation model for forest fires. In: Dongarra, K.M.J., Wasniewski, J. (eds.) Proceedings of PARA04 Workshop on State of the Art in Scientific Computing, pp. 546–553. Springer, Heidelberg (2005)

3. Brezina, M., Vaněk, P.: A black-box iterative solver based on a two-level Schwarz method. Computing **63**, 233–263 (1999)
4. Buttari, A., D'Ambra, P., di Serafino, D., Filippone, S.: Extending PSBLAS to build parallel schwarz preconditioners. In: Dongarra, K.M.J., Wasniewski, J. (eds.) Proceedings of PARA04 Workshop on State of the Art in Scientific Computing, pp. 593–602. Springer, Heidelberg (2005)
5. Buttari, A., D'Ambra, P., di Serafino, D., Filippone, S.: 2LEV-D2P4 User's Guide (in preparation)
6. Cai, X.C., Saad, Y.: Overlapping domain decomposition algorithms for general sparse matrices. Num. Linear Algebra Appl. **3**(3), 221–237 (1996)
7. Cai, X.C., Sarkis, M.: A restricted additive Schwarz preconditioner for general sparse linear systems. SIAM J. Sci. Comput. **21**(2), 792–797 (1999)
8. Cai, X.C., Widlund, O.B.: Domain decomposition algorithms for indefinite elliptic problems. SIAM J. Sci. Stat. Comput. **13**(1), 243–258 (1992)
9. Chan, T., Mathew, T.: Domain decomposition algorithms. In: Iserles, A. (ed.) Acta Numerica, pp. 61–143. Cambridge University Press, Cambridge (1994)
10. D'Ambra, P., di Serafino, D., Filippone, S.: On the development of PSBLAS-based parallel two-level Schwarz preconditioners. Appl. Num. Math. (to appear) (2007)
11. Davis, T.: University of Florida sparse matrix collection home page. http://www.cise.ufl.edu/research/sparse/matrices
12. Davis, T.A.: Algorithm 832: UMFPACK—an unsymmetric-pattern multifrontal method with a column pre-ordering strategy. ACM Trans. Math. Softw. **30**, 196–199 (2004)
13. Dongarra, J., Whaley, R.: A user's guide to the BLACS v1.0. LAPACK working note #94 CS-95-281, University of Tennessee (1995) http://www.netlib.org/lapack/lawns
14. Duff, I., Heroux, M., Pozo, R.: 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)
15. Duff, I., Marrone, M., Radicati, G., Vittoli, C.: Level 3 basic linear algebra subprograms for sparse matrices: a user level interface. ACM Trans. Math. Softw. **23**(3), 379–401 (1997)
16. Filippone, S., Colajanni, M.: PSBLAS: A library for parallel linear algebra computation on sparse matrices. ACM Trans. Math. Softw. **26**(4), 527–550 (2000)
17. Filippone, S., D'Ambra, P., Colajanni, M.: Using a parallel library of sparse linear algebra in a fluid dynamics applications code on linux clusters. In: Joubert, G., Murli, A., Peters, F., Vanneschi, M. (eds.) Proceedings of Parallel Computing—Advances and Current Issues, pp. 441–448. Imperial College Press, London (2002)
18. Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., Kolda, T.G., Lehoucq, R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T., Salinger, A.G., Thornquist, H.K., Tuminaro, R.S., Willenbring, J.M., Williams, A., Stanley, K.S.: An overview of the Trilinos project. ACM Trans. Math. Softw. **31**(3), 397–423 (2005)
19. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1999)
20. Li, X.S., Demmel, J.W.: SuperLU-DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. ACM Trans. Math. Softw. **29**(2), 110–140 (2003)
21. Smith, B., Bjorstad, P., Gropp, W.: Domain decomposition: parallel multilevel Methods for Elliptic Partial Differential Equations. Cambridge University Press, Cambridge (1996)
22. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J.: MPI: The Complete Reference, The MPI Core, vol. 1, 2nd edn. MIT, Cambridge (1998)
23. Tuminaro, R.S., Tong, C.: Parallel smoothed aggregation multigrid: aggregation strategies on massively parallel machines. In: Donnelley, J. (ed.) Proceedings of SuperComputing 2000. Dallas, (2000)
24. Vaněk, P., Mandel, J., Brezina, M.: Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. Computing **56**, 179–196 (1996)