PDSLin Users Guide

Ichitaro Yamazaki[*], Xiaoye S. Li[†], Esmond Ng[‡]
Lawrence Berkeley National Laboratory

March 21, 2011

Abstract

We describe the software package parallel domain decomposition Schur complement based linear solver, or PDSLin in short, which implements a hybrid linear solver based on the Schur complement method for solving a sparse linear system of equations. We give a brief description of the algorithm, installation, calling sequences, and data structures of the software.[1]

[*]Email: ic.yamazaki@gmail.com
[†]Email: xsli@lbl.gov
[‡]Email: egng@lbl.gov
[1]This research was supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. DOE under Contract No. DE-AC02-05CH11231.

Contents

1 Introduction

Modern numerical simulations give rise to sparse linear systems of equations that are becoming increasingly difficult to solve using standard techniques. Matrices that can be directly factorized are limited in size due to large memory and communication requirements. Preconditioned iterative solvers require less memory and communication, but often require an effective preconditioner, which is not readily available. In order to efficiently solve these challenging problems on a large number of processors, this software named parallel domain decomposition Schur complement based linear solver, or PDSLin in short, implements a parallel hybrid (direct/iterative) linear solver, which employs techniques from both sparse direct and iterative methods.

PDSLin is based on a non-overlapping domain decomposition technique called the Schur complement method. In this method, the global system is first partitioned into smaller interior subdomain systems, which are connected only through separators. To compute the solution of the global system, the unknowns associated with the interior subdomain systems are first eliminated to form the Schur complement system, which is defined only on the separators. Since most of the fill occurs in the Schur complement, to obtain the solution on the separators, the Schur complement is solved using a preconditioned iterative method. Then, the solution on the subdomains is computed by using this solution on the separators and solving another set of subdomain systems. These unknowns associated with the mutually-independent interior subdomains are eliminated in parallel using multiple processors per subdomain. PDSLin is implemented in C with Fortran interface, and uses MPI for message passing on distributed memory machine.

In Sections 2 and 3, we first describe the algorithm and the installation guide of the software, respectively. Then, in Sections 4 and 5, we show a calling sequence from C and Fortran programs, respectively. In Sections 6 and 7, we describe the input and output of the hybrid solver. Finally, in Section 8, we list some additional options that may be useful in some cases.

2 Overview of the algorithm

PDSLin is a software package for solving a linear system of equations,

$$Ax = b,$$

where A is a square real or complex general matrix, b is a given right-hand-side vector, and x is the solution vector to be computed. It uses a non-overlapping domain decomposition technique called the Schur complement method.

The original linear system is first reordered into a 2×2 block system of the following form:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}, \tag{1}$$

where A_{11} and A_{22} respectively represent *interior subdomains* and *separators*, and A_{12} and A_{21} are the *interfaces* between A_{11} and A_{22}. By eliminating the unknowns associated with the interior subdomains A_{11}, we obtain

$$\begin{pmatrix} A_{11} & A_{12} \\ 0 & S \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ \widehat{b}_2 \end{pmatrix},$$

where S is the Schur complement defined as

$$S = A_{22} - A_{21}A_{11}^{-1}A_{12},$$

and $\widehat{b}_2 = b_2 - A_{21}A_{11}^{-1}b_1$. Subsequently, the solution of the linear system (1) can be computed by first solving the Schur complement system

$$Sx_2 = \widehat{b}_2, \tag{2}$$

and then solving the interior system

$$A_{11}x_1 = b_1 - A_{12}x_2. \tag{3}$$

Since most of the fill occurs in the Schur complement S, the Schur complement system is solved using a preconditioned iterative method, while the interior subdomain system is typically solved using a direct method. Since these interior subdomains are independent from each other (i.e., D is a block diagonal matrix), unknowns associated with each interior subdomain can be eliminated in paralell. See [13] and the references within for a detailed discussion on the Schur complement method.

Our parallel implementation of the Schur complement method consists of the following two phases:

1. **Computing the preconditioner.** First, the global system is partioned to extract the interior subdomains using a parallel graph partitioning algorithm of PT-SCOTCH [3, 7] or ParMETIS [1, 10]. Each interior subdomains are then factorized in parallel, using a parallel direct solver SuperLU_DIST [5, 11] on each subdomain. Finally, a preconditioner $\widetilde{S}$ to solve the Schur complement system (2) is computed as follows:

$$\begin{aligned} S &= A_{22} - (A_{21}U_{11}^{-1})(L_{11}^{-1}A_{12}), \quad \text{with the LU factorization } A_{11} = L_{11}U_{11} \\ &\approx A_{22} - \widetilde{G}\widetilde{W}, \quad \text{where } \widetilde{G} \approx A_{21}U_{11}^{-1} \text{ and } \widetilde{W} \approx L_{11}^{-1}A_{12} \text{ with a drop threshold } \tau_0 \\ &\approx A_{22} - \widetilde{T}, \qquad \text{where } \widetilde{T} \approx \widetilde{G}^T\widetilde{W} \text{ with a drop threshold } \tau_1 \\ &\approx \widetilde{S}, \qquad\qquad \text{where } \widetilde{S} \approx A_{22} - \widetilde{T} \text{ with a drop threshold } \tau_2. \end{aligned} \tag{4}$$

At each step of computing the preconditioner $\widetilde{S}$, nonzeros with their magnitudes less than a drop threshold are discarded from the intermediate matrices. Then, the LU factorization of $\widetilde{S}$ is computed by SuperLU_DIST.

2. **Computing the solution.** First, the Schur complement system (2) is solved using a preconditioned Krylov method of PETSc [2, 6]. Then, the interior systems (3) are solved in parallel, using the already-computed LU factors of the subdomains.

Please see [14] for more details on our implementation.

3 Installation

Our package requires the following external libraries:

- PT-SCOTCH [3] or ParMETIS [1], which is used to extract interior subdomains,

- SuperLU_DIST (version 2.4) [5], which is used to solve the interior subdomain systems, and possibly the Schur complement system, and
- PETSc (version 2.3.3) [2], which is used to solve the Schur complement system by an iterative method. If the parallel direct solver SuperLU_DIST is used to solve the Schur complement system, then PETSc is not needed.

The software package can be obtained by contacting the authors. Once you obtain and untar the package, the source codes are organized under the following directories:

```
 pdslin_0.0/src              : C source files
           /include          : C header files
           /examples         : C example programs
           /fortran          : Fortran interface
           /fortran/examples : Fotran example programs
           /lib              : hybrid solver library
           /make.examples    : make.inc examples
```

To install the package, you first need to modify the "make.inc" file in the top directory "pdslin_0.0." There are a number of example make.inc in "make.examples" directory, which we used on different architectures. We list below some of the parameters that may need to be modified in the make.inc file. For this example, we show the make.inc file that is used on a Cray XT-4 machine at NERSC, where "-DWITH_PETSC" indicates that the hybrid solver is linked to PETSc.

```
# directory, where the package is installed
HOME = /global/u2/y/yamazaki/franklin_Jan21.2010/pdslin_0.0

########################################################################
# make utility
MAKE = make

########################################################################
# archiver and flags used to build the library
ARCH      = ar
ARCHFLAGS = -cr

########################################################################
# C compiler and flags used to compile the code
CC    = cc
FLAGS = -fastsse -DWITH_PETSC

########################################################################
# Fortran compiler and flags
FC     = ftn
FLIB   = -lpgf90 -lpgf90_rpm1 -lpgf902 -lpgf90rtl -lpgftnrtl
FFLAGS = -fastsse

########################################################################
# linker used to link the example program with the library
LINKER = CC
```

```
############################################################################
# Libraries:
#
# BLAS/MPI libraries, and MPI include
L_BLAS =
L_MPI  =
I_MPI  =

# parallel partitioning library, and its header files
L_PPART = -L/usr/common/usg/parmetis/3.1 -lparmetis -lmetis
I_PPART = -I/usr/common/usg/parmetis/3.

# Metis library, and its header files
L_METIS = -L/global/u2/y/yamazaki/franklin_Jan21.2010/metis-5.0pre2/build/Linux-x86_64 \
          -lmetis
I_METIS = -I/global/u2/y/yamazaki/franklin_Jan21.2010/metis-5.0pre2/include

# SuperLU_DIST library, and its header files
SLUDIST = /global/u2/y/yamazaki/franklin_Jan21.2010/SuperLU_DIST_2.4
L_SLUDIST = -L$(SLUDIST)/lib/ -lsuperlu_dist_2.4
I_SLUDIST = -DDEBUGlevel=0 -DPRNTlevel=0 -DAdd_ -DUSE_VENDOR_BLAS -I$(SLUDIST)/SRC

# PETSc Library, and its header files
# double version
PETSC_ARCH = cray-xt4_g
PETSC_DIR  = /usr/common/acts/PETSc/2.3.3
I_PETSC = -I$(PETSC_DIR)/src/dm/mesh/sieve -I$(PETSC_DIR) \
          -I$(PETSC_DIR)/bmake/$(PETSC_ARCH) -I$(PETSC_DIR)/include
L_PETSC = -L$(PETSC_DIR)/lib/$(PETSC_ARCH) -lpetscksp -lpetscdm -lpetscmat \
          -lpetscvec -lpetsc

# complex version
I_ZPETSC = -I$(PETSC_DIR)/src/dm/mesh/sieve -I$(PETSC_DIR) \
           -I$(PETSC_DIR)/bmake/$(PETSC_ARCH)_complex -I$(PETSC_DIR)/include
L_ZPETSC = -L$(PETSC_DIR)/lib/$(PETSC_ARCH)_complex -lpetscksp -lpetscdm \
           -lpetscmat -lpetscvec -lpetsc
```

Once the make.inc file is properly modified, you can build the library by typing
make all
This will create the C library for solving both real and complex linear systems under the "lib" directory. If you want to create the library just for solving real or complex linear systems, then you can respectively type
make dlib
or
make zlib
Furthermore, the Fortran interface for solving both real and complex systems can be included into the library by typing
make flib

4 Example program in C

PDSLin can solve both real and complex linear systems of equations. The header files and subroutines for solving real or complex systems start with letter “d” or “z,” respectively.

There are several example programs included under the “examples” directory. The following piece of a C example program shows a typical calling-sequence to solve a real linear system using the hybrid solver. Each subroutine call will be described in more details below.

```
#include <mpi.h>
#include "dpdslin_solver.h"  /* header file for solving real systems */
#include "pdslin_solver.h"   /* header file for hybrid solver        */

int main( int argc, char* argv[] ) {
   int i, info;
   double *x_loc, *bloc;
   MPI_Comm pdslin_comm = MPI_COMM_WORLD;
   dPDSLinMatrix matrix;
   pdslin_param input;
   pdslin_stat  stat;

   /* step 0: */
   /* initialize MPI and hybrid solver */
   MPI_Init( &argc, &argv );
   dpdslin_init( &input, &matrix, &stat, pdslin_comm, argc, argv );

   /* step 1 */
   /* set up coefficient matrix (will be explained below) */
   .......

   /* step 2 */
   /* call hybrid solver to compute preconditioner */
   input.job = PDSLin_PRECO;
   dpdslin_solver( b_loc, x_loc, &matrix, &input, &stat, &info );

   /* step 3 */
   /* allocate right-hand-side and solution vectors */
   x_loc = pdslin_dmalloc( matrix.mloc );
   b_loc = pdslin_dmalloc( matrix.mloc );

   /* step 4 */
   /* setup the right-hand-side */
   for( i=0; i<matrix.mloc; i++ ) b_loc[i] = 1.0;

   /* step 5 */
   /* call hybrid solver to compute solution */
   input.job = PDSLin_SOLVE;
   dpdslin_solver( b_loc, x_loc, &matrix, &input, &stat, &info );

   /* step 6 */
   /* free right-hand-side and solution */
   pdslin_free(x_loc);
```

```
    pdslin_free(b_loc);

    /* step 7 */
    /* print out statistics */
    pdslin_print_stat( &stat, input.pdslin_comm );

    /* step 7 */
    /* terminate MPI and hybrid solver */
    dpdslin_finalize(&input, &matrix);
    MPI_Finalize();

    return 0;
}
```

Our hybrid solver expects the input matrix in the distributed Compressed Sparse Row (CSR) format (step 1 of the code above); namely, each processor owns a set of contiguous rows of the global matirx, where only the numerical values and column indices of the nonzeros in these rows are stored in the row-wise order and in the ascending order of their row indices. This coefficient matrix is set by modifying the corresponding member variables of "matrix" in the example above, which is a variable of type dPDSLinMatrix:

```
typedef struct {
  /* input matrix in distributed CSR format */
  int_t n;          /* global matrix dimension                             */
  int_t nnz;        /* total number of nonzeros in the global matrix       */
  int_t frow;       /* the index of the first row of the local matrix      */
  int_t mloc;       /* the number of rows in the local matrix              */
  double *lnzval;   /* pointer to array of nonzero values, packed by row   */
  int_t *lcolind    /* pointer to array of columns indices of the nonzeros */
  int_t * lrowptr   /* pointer to array of beginning of rows in lnzval[] and */
                    /* lcolind[]                                           */

  /* the rest of member variables used internally by the hybrid solver */
  .......
} dPDSLinMatrix;
```

The local right-hand-side and solution vector are stored in the corresponding format, i.e., in the example above, "b_loc" and "x_loc" contains the frow-th through the (frow+mloc-1)-th elements of the global right-hand-side and solution vectors, respectively. Note that the row and array indices are zero-based.

Below, we list the four key subroutines used in the example above, each of which will be describe in more details in the remaining of this section:

- **dpdslin_init**: subroutine to initialize the solver.
- **dpdslin_solve**: computational subroutine to compute preconditioner or solution.
- **pdslin_print_stat**: subroutine to print statistics.

- **dpdslin_finalize**: subroutine to finalize the solver.

Before calling any of the hybrid solver subroutines, the following initialization subroutine needs to be called (step 0 of the code above):

```
dpdslin_init( &input, &matrix, &stat, pdslin_comm, argc, argv ),
```

where the first three arguments "input," "matrix," and "stat" are initialized with the default values on return, the third argument "pdslin_comm" specifies the global MPI communicator used to solve the linear system, and the last two arguments are the C command line arguments, which can be used to initialize PETSc. This subroutine also sets up required MPI variables when the fifth argument argc is a non-negative value. Hence, when this initialization subroutine is called for the first time, argc needs to be non-negative. If there are no command-line arguments to initialize PETSc, then the last argument argv can be set to be NULL. Finally, if this subroutine needs to be called more than once, then argc and argv should be negative and NULL, respectively, for the second call and on.

The default parameters are set as follow: The number of subdomains is the smaller of the total number of processors and 64.[2] Hence, if the number of processor is less than or equal to 64, each subdomain is factorized using one processor, while multiple processors are used to factorize each subdomain if more than 64 processors are available. The number of processors to compute the preconditioner for solving the Schur complement system is set to be the half of the number of subdomains. If the hybrid solver is linked to PETSc, then the preconditioner is computed using the drop thresholds $(\tau_0, \tau_1, \tau_2) = (10^{-6}, 0.0, 10^{-5})$, and the Schur complement system is solved using GMRES, where the iteration is restarted after 100 iterations, and terminated when the relative residual norm of 10^{-12} is obtained, or $1,000$ iterations are performed. These parameters can be changed by modifying the corresponding member variables of the variable "input" of type pdslin_param. Please see Section 6.3 for the brief description of the member variables.

Once the hybrid solver is initialized and the coefficient matrix is set, we can invoke the hybrid solver to solve the corresponding linear system. The hybrid solver allows the user to compute the preconditioner and solution, separately. Specifically, to compute the preconditioner (step 3 of the code above), the hybrid solver is invoked as

```
input.job = PDSLin_PRECO;
dpdslin_solver( b_loc, x_loc, &matrix, &input, &stat, &info ),
```

where the first two arguments "b_loc" and "x_loc" are the local right-hand-side and solution vectors, respectively, but for the computation of the preconditioner, they are not used. The last two arguments "stat" and "info" respectively contain the statistics and error code on return, which will be explained in Section 7.

To solve to the linear system, we can invoke the hybrid solver again to compute the solution vector (step 5):

```
input.job = PDSLin_SOLVE;
dpdslin_solver( b_loc, x_loc, &matrix, &input, &stat, &info ).
```

The user can compute the solution vectors for the same coefficient matrix and for different right-hand-side vectors by repeatedly calling this subroutine with different "b_loc."

The follwing subroutine can be used to print out the performance statistics of pdslin_solver:

[2]Currently, the number of subdomains needs to be a power of two. When the number of subdomains is not a power of two, it is set to be the largest number that is a power of two and less than or equal to the number of processors.

```
   pdslin_print_stat( stat, input.pdslin_comm ),
```

where the second argument is the global MPI communicator used to solve the system. To use this subroutine, the performance profiling needs to be turned on before calling pdslin_solver (see Section 6.3).

When all the subroutine calls to the hybrid solver are completed, the user can call the following subroutine to free up all the memory allocated by the hybrid solver (step 7 of the code above):

```
   dpdslin_finalize(&input, &matrix);
```

In the example above, the subroutine pdslin_dmalloc(mloc) allocates memory for b_loc and x_loc to store "mloc" double variables. In addition, the subroutine pdslin_free is used to free up the memory allocated.

5 Fortran interface

Just like the C subroutines, all the Fortran subroutines for solving real or complex linear systems start with letter "d" or "z," respectively. Example programs that use the Fortran interface of our hybrid solver are included in the "fortran\examples" directory.

The following piece of the code shows how to call our hybrid solver for solving a real linear system from a Fortran program. Each of the subroutine calls will be explained in more details below.

```
      program dtest_pdslin
*
*   .. hybrid modules (step 0) ..
      use pdslin_param
      use pdslin_mod
      implicit none
      include 'mpif.h'
*
*   .. local variables (step 1)..
      integer(pdslin_ptr) matrix
      integer(pdslin_ptr) input
      integer(pdslin_ptr) stat
      integer ierr, nproc, info, pdslin_comm
      parameter( pdslin_comm = MPI_COMM_WORLD )
*
*   -- initialize MPI and hybrid solver (step 2) --
*
      call MPI_Init(ierr)
      call dpdslin_init_f( input, matrix, stat, pdslin_comm )
*
*   -- set input matrix (step 3)--
*
      call dpdslin_set_matrix_f( input, matrix, frow, mloc,
     $                           n, colptr, rowind, values )
*
*   -- initialize hybrid solver and its parameters (step 4)--
*
```

```
      call MPI_Comm_size( pdslin_comm, nproc, ierr )
      call set_pdslin_params_f( input, verbose       = PDSLin_YES  )
      call set_pdslin_params_f( input, mat_type      = UNSYMMETRIC )
      call set_pdslin_params_f( input, num_doms      = nproc       )
      call set_pdslin_params_f( input, drop_tau0     = 0d0         )
      call set_pdslin_params_f( input, drop_tau1     = 1d-5        )
      call set_pdslin_params_f( input, drop_tau2     = 0d0         )
      call set_pdslin_params_f( input, inner_solver  = PDSLin_GMRES)
      call set_pdslin_params_f( input, inner_max     = 100         )
      call set_pdslin_params_f( input, inner_restart = 10          )
      call set_pdslin_params_f( input, inner_tol     = 1d-9        )
*
*   -- compute the preconditioner (step 5)--
*
      call set_pdslin_params_f( input, job=PDSLin_PRECO )
      call dpdslin_solver_f( b, x, matrix, input, stat, info )
*
*   -- compute the solution (step 6) --
*
      call set_pdslin_params_f( input, job=PDSLin_SOLVE )
      call dpdslin_solver_f( b, x, matrix, input, stat, info )
*
*   -- clean up (step 7)--
*
      call dpdslin_finalize_f( input, matrix, stat )
      call MPI_Finalize(ierr)
*
*   .. end of program ..
*
      end
```

The Fortran subroutine names are appended by "_f" to the corresponding C subroutine names, which are discussed in Section 4. Here, we focus on the differences between the Fortran and C calling sequences. Please see Section 4 form more details on each subroutine call.

The Fortran program first needs to include two hybrid modules that define the hybrid parameters and interfaces (step 0 of the code above):

```
use pdslin_param
use pdslin_mod
```

To use the hybrid solver from a Fortran program, appropriate handles to the C data structures, dPDSLinMatrix and pdslin_param, need to be first declared (step 1 of the code above). These handles are of type integer, and their sizes are given by "pdslin_ptr" defined in the pdslin_param module. These objects then need to be allocated and initialized by calling the following initialization subroutine (step 2 of the code above):

```
call dpdslin_init_f( input, matrix, stat, pdslin_comm ),
```

where the last argument specifies the global MPI communicator used to solve the linear system. This subroutine also initializes all the input parameters to their default values.

Note that these C data structures are *opaque* (i.e., their sizes and structures are not visible) from the Fortran user, and can be modified only through appropriate Fortran subroutine calls. For instance, the coefficient matrix in dPDSLinMatrix can be set by the following subroutine call (step 3):

```
      call dpdslin_set_matrix_f( input, matrix, frow, mloc,
     $                           n, colptr, rowind, values ),
```

where the local matrix is stored in the distributed CSR format (see Section 4 for more details of the format). The column and array indices are one-based on input, but they are shifted to zero-based on return.

Furthermore, we provide an interface to modify the individual parameters of pdslin_param (step 4). For example, the iterative solver used on the Schur complement system can be changed to FGMRES by the following subroutine call:

```
      call set_pdslin_params_f( input, inner_solver = PDSLin_FGMRES ).
```

All the available parameters are defined in the module "pdslin_mod." Please see Section 6.3 for the descriptions of the input parameters.

Finally, the preconditioner and solution can be computed by

```
*
*   -- compute the preconditioner --
      call set_pdslin_params_f( input, job=PDSLin_PRECO )
      call dpdslin_solver_f( b, x, matrix, input, stat, info )
*
*   -- compute the solution --
      call set_pdslin_params_f( input, job=PDSLin_SOLVE )
      call dpdslin_solver_f( b, x, matrix, input, stat, info )
```

When the following finalizing subroutine is called, all the memory allocated by the hybrid solver (including those of opaque objects) are deallocated:

```
      call dpdslin_finalize_f( input, matrix, stat )
```

6 Inputs

In this section, we describe the required inputs of the pdslin_solver subroutine. Specifically, in Section 6.1, we describe the data structure **dPDSLinMatrix** (or **zPDSLinMatrix**) that stores the input matrix. Then, in Section 6.2, we explain how the right-hand-side vector needs to be distributed. Finally, in Section 6.3, we describe the data structure **pdslin_param** that stores all the input parameters.

6.1 PDSLinMatrix structure

The only member variables of this structure, which the user can modify, are:

```
typedef struct {
  /* input matrix in distributed CSR format */
  int_t n;          /* global matrix dimension                             */
  int_t nnz;        /* total number of nonzeros in the global matrix       */
  int_t frow;       /* the index of the first row of the local matrix      */
  int_t mloc;       /* the number of rows in the local matrix              */
  double *lnzval;   /* pointer to array of nonzero values, packed by row   */
  int_t *lcolind    /* pointer to array of columns indices of the nonzeros */
  int_t * lrowptr   /* pointer to array of beginning of rows in lnzval[] and */
                    /* lcolind[]                                           */

  /* the rest of member variables used internally by the hybrid solver */
  .......
} dPDSLinMatrix;
```

These parameters are used to set the local coefficient matrix in the distributed CSR format as discussed in Section 4.

6.2 Right-hand-side vector

The local right-hand-side vector should be stored in the format that corresponds to the input matrix stored in PDSLinMatrix (see Section 6.1). Specifically, each processor stores the subset of the contiguous rows of the right-hand-side vector, where the index of the first row and the number of rows in the local vector are specified by the member variables **frow** and **mloc** of PDSLinMatrix, respectively.

6.3 pdslin_param structure

All the input parameters are stored in a variable of type "pdslin_param." In this section, we describe a subset of these parameters, which are most useful to the users.

```
typedef struct {
  MPI_Comm   pdslin_comm;   /* MPI communicator for hybrid solver */

  int        job;           /* specify job for hybrid solver      */
  int        check_input;   /* to check input parameters          */
  int        gather_stat;   /* to gather statistics               */
  t_verbose  verbose;       /* to print progress of hybrid solver */
  t_mtype    mat_type;      /* input matrix type                  */
  t_mtype    mat_pattern;   /* input matrix type                  */
  int        free_local;    /* to internally free input matrix    */

  /* inputs for partitioning */
  int        num_doms;      /* number of subdomains to be extracted       */
  int        nproc_dcomp;   /* number of processors to extract subdomains */
  /* input for preconditioner */
                            /* dropping thresholds */
  double     drop_tau0, drop_tau1, drop_tau2, drop_tau3;
  int        equil_dom;     /* equilbration/row permutation for subdomains       */
  int        equil_schur;   /* equilbration/row permutation for schur complement */

  /* input for iterative solver */
  t_itsolver inner_solver;  /* iterative solver for schur complement     */
  int        inner_max;     /* maximum number of matrix operations       */
  int        inner_restart; /* number of operations at restart           */
  double     inner_tol;     /* stopping criteria                         */
  int        nproc_schur;   /* number of processors for schur complement */
  .....
} pdslin_param;
```

Below, we provide some brief description of these parameters:

- **pdslin_comm** (of type MPI_Comm): specifies the global MPI communicator used to solve the linear system.
- **job** (of type integer): specifies the job to be performed by the hybrid solver, and can be PDSLin_PRECO, PDSLin_SOLVE, PDSLin_PRESOLVE, PDSLin_CLEAN, or PDSLin_NFACT to compute preconditioner, compute solution, compute both preconditioner and solution, free up all the memory allocated by the hybrid solver, or perform numerical computation of the preconditioner with the same sparsity pattern, respectively.
- **verbose** (of type t_verbose): specifies to print the progress of the hybrid solver as it runs, and can be PDSLin_VALL, PDSLin_VWARN, or PDSLin_VNONE to print all the status information, only warning status, or nothing. The default is PDSLin_VNONE.
- **mat_type** and **mat_pattern** (of type t_mtype): can be either SYMMETRIC or UNSYMMETRIC, and specifies the matrix type and its sparsity pattern. The default is UNSYMMETRIC.
- **schur_pattern** (of type t_mtype): can be either SYMMETRIC or UNSYMMETRIC, and specifies if the Schur complement should be kept symmetric or not. The default is UNSYMMETRIC.
- **free_local**: can be either PDSLin_YES or PDSLin_NO, and specifies if the original matrix can be internally freed. The default is PDSLin_NO. When this is set to be PDSLin_NO, then the entries in matrix.lrowptr, matrix.lcolind, and matrix.lnzval are not changed.
- **num_doms** (of type integer): specifies the number of interior subdomains. Note that the interior subdomains are extracted using a graph partitioning algorithm of ParMETIS or PT-SCOTCH, which requires the number of subdomains to be a power of two. The default is the smaller of 64 and the largest power of two, which is less than or equal to the total number of processors in pdslin_comm.

- **nproc_dcomp** (of type integer): specifies the number of processors used to extract the interior subdomains. The default is set to be equal to num_doms. A larger number may reduce the time required to extract the subdomains, but may degrade the quality of the partition. This needs to be greater than or equal to num_doms.

- **drop_tau0, drop_tau1, drop_tau2** (of type double): specifies the dropping thresholds used to compute the preconditioner (i.e., the thresholds for enforcing sparsity of $\widetilde{G}$ and $\widetilde{W}$, $\widetilde{T}$, and $\widetilde{S}$, respectively). The default values are 10^{-6}, 10^{-5}, and 0.0. By increasing the drop threshold, the user can reduce the time and memory required to compute the preconditioner, but the number of iterations may increase. When they are all set to be zero, the direct solver is used to solve the Schur complement system.

- **equil_dom** (of type integer): specifies the equilbration and row permutation on the subdomains. If it is set to be zero, neither equilbration nor row permutation is applied. If it is set to be a positive integer, then both equilbration and row permutation are applied to move large elements to diagonal, while a negative integer specifies that only the equilbration is applied. The default is -1 to apply just the equilbration.

- **equl_schur**: specifies the equilbration and row permutation used on the approximate Schur complement. If it is set to be zero, then neither equilbration nor row permutation is applied. If it is negative, then only the equilbration is applied. Otherwise, if it is set to a value between 1 and 5, then it is used to call an external subroutine MC64 [9]. The default value is 5, which will compute a row permutation to move large elements to diagonal and scaling so that diagonal elements are ones.[3]

- **inner_solver** (of type t_itsolver): specifies the iterative solver used to solve the Schur complement system, and can be PDSLin_GMRES, PDSLin_FGMRES, PDSLin_BICGSTAB, or PDSLin_TFQMR. The default is PDSLin_GMRES.

- **inner_max** (of type integer): specifies the maximum total number of iterations for solving the Schur complement system. The default is $1,000$.

- **inner_restart** (of type integer): specifies the maximum number of GMRES or FGMRES iterations before restart for solving the Schur complement system. The default is 100.

- **inner_tol** (of type double): specifies the relative residual norms for stopping the iteration for solving the Schur complement system. The default is 10^{-12}.

- **nproc_schur** (of type integer): specifies the number of processors used to solve the Schur complement system. The default is set to be half of the number of subdomains, but at least one. The larger value may or may not reduce the time to compute and solve the Schur complement system. It is dynamically adjusted so that each processor has at least $1,000$ rows of the Schur complement.

- **check_input** (of type integer): specifies if the hybrid solver checks for the validity of the input parameters when the subroutine pdslin_solver is called. Invalid parameters are automatically reset to their default values. It can be either PDSLin_YES or PDSLin_NO, and the default

[3]This is done in parallel. See [14] for more information.

is PDSLin_YES. When pdslin_solver is called to compute preconditioner or solution, only the relevant parameters are checked. For instance, you cannot change the preconditioner by changing a parameter to compute the preconditioners (e.g., num_doms or drop_tau0) when calling pdslin_solver to compute solution. In order to change the preconditioner, you need to first call pdslin_solver with PDSLin_CLEAN to free the previous preconditioner (free_local should have been set to be PDSLin_NO for the previous call to pdslin_solver so that the original matrix is not freed). Then, you need to call pdslin_solver again with PDSLin_PRECO to recompute the preconditioner with new parameters. Please see Section 8.2 for more details to recompute the preconditioner.

- **gather_stat** (of type integer): specifies if the hybrid solver gather statistics, and can be either PDSLin_YES or PDSLin_NO. The default is PDSLin_YES.

Please see the header file "pdslin_solver.h" for the information on the remaining parameters.

7 Outputs

In this section, we describe the outputs from the pdslin_solver subroutine. Specifically, in Section 7.1, we explain how the computed solution vector is distributed. Then, in Section 7.2, we list the error code that may be returned by the hybrid solver. Finally, in Section 7.3, we describe the data structure **pdslin_stat** that stores the performance statistics of the hybrid solver.

7.1 Solution vector

The local slution vector (i.e., the second argument in pdslin_solver) is returned in the same format as the local right-hand-side vector (see Section 6.2). Specifically, each processor stores the subset of the contiguous rows of the global right-hand-side vector, where the index of the first row and the number of rows in the local vector are specified by the member variables **frow** and **mloc** of PDSLinMatrix (see Section 6.1), respectively.

7.2 Error codes

An integer error code is returned in the fourth argument of pdslin_solver subroutine. We list below the possible error codes, and our suggestions on how to resolve the error by adjusting the input parameters. See Section 6.3 for more details on the input parameters.

0 Success: The hybrid solver successfuly completed the requested task.

-x Invalid parameters: The x-th paremeter of pdslin_param had an illegal value (see the header file include/pdslin_util.h for the mapping between x and the actual parameter). If you have asked pdslin_solver to check for invalid parameters, then the invalid parameter has been reset to its default value, and pdslin_solver have successfully complted the task.

1 Out of memory: The total memory requirement may be reduced by increasing the drop tolerances or number of subdomains. Alternatively, you may be able to reduce the memory required by each processor by increasing the total number of processors.

2 Failure to factorize subdomain: This is likely to be due to either out of memory or zero pivot. The memory requirement may be reduced by increasing the number of subdomains, while the zero pivod may be avoided by using row permutation on each subdomain. If the row permutation did not help, a global preprocessing of the whole matrix may be needed before calling pdslin_solver (e.g., to move large elements to diagonals).

3 Failure to factorize approximate Schur complement: Similar to the error code 2, the factorization failed due to either out of memory or zero pivots. You may be able to reduce the memory requirement by reducing the drop thresholds or the number of subdomains. The zero pivot may be avoided by using row permutation on the approximate Schur complement, but a preprocessing may be needed on the global matrix.

4 Failure to converge: The iterations to solve the Schur complement system did not converge within the specified maximum number of iterations. You may be able to avoid this by increasing the maximum number of iterations, or by decreasing the drop tolerances, number of subdomain, or stopping criteria.

5 Other failures. More detailed information on the error can be obtained by looking at the member variable **error** of the data structure **pdslin_stat** (see Section 7.3).

In the current implementation, each processor may return a different error code. Furthermore, if one of the processor fails, the rest of the processors may hung. To print out the error or warning messages, please set the verbose level to be PDSLin_VERROR or PDSLin_VWARN, respectively.

7.3 pdslin_stat structure

The features described in this section are mainly for experts users who like to study the performance of the hybrid solver in detail.

On return from the pdslin_solver subroutine, some performance statistics are returned in the fifth argument, which is of type **pdslin_stat**:

```
typedef struct {
  /* partition info */
  int num_doms;   /* number of subdomains                            */
  int id;         /* index of subdomain this processor is assigned  */
  int n1;         /* total dimension of interior subdomains         */
  int n2;         /* dimension of interface/separator               */
  int nnz_subdom; /* number of nonzeros in local interior subdomain */
  int nnz_interf; /* number of nonzeros in local interface          */
  int nnz_seprat; /* number of nonzeros in local separators         */

  /* workspace info */
  int nnz_w; /* number of nonzeros in intermediate local matrix W        */
  int nnz_g; /* number of nonzeros in intermediate local matrix G        */
  int nnz_s; /* number of nonzeros in locla approximate schur complement */

  /* preconditioner info */
  int_t nnz_L, nnz_U;    /* number of nonzeros in LU of "id"-th subdomain  */
  int_t nnz_PL, nnz_PU;  /* number of nonozeros in LU of apporixmate Schur */

  /* processor info */
  int proc_id;  /* processor id */
  int *sepid;   /* list of processor ids to compute preconditioner */

  /* timing results in seconds */
  double time_dd,     /* time for matrix partitioning               */
         time_dst,    /* time for matrix redistribution             */
         time_lu,     /* time for subdomain factorization           */
         time_schur,  /* time for approximate schur computation     */
         time_prep,   /* time for preprocessing approximate schur   */
         time_prec,   /* time for factorizing approximate schur     */
         time_fact,   /* total time for preconditioner computation */
         time_solve;  /* total time for solution computation        */

  /* iteration results */
  int inner_itrs;     /* total number of inner-iterations */

  /* error code */
  pdslin_error error; /* error code */

  ......
} pdslin_stat;
```

The last member variable **error** of pdslin_stat contains detailed information of error generated by the pdslin_solver subroutine:

```
typedef struct {
  int  code;       /* detailed error code */
  int *info;       /* pointer to error code (see Section 7.2) */

  int error_print; /* specify to print out error location (PDSLin_YES/PDSLin_NO) */
  int error_tau;   /* specify to abort program with error (PDSLin_YES/PDSLin_NO) */
                   /* defaul values are PDSLin_NO                                */
} pdslin_error;
```

The detailed error codes are organized in a hierachical fashion based on the location where the error is generated. For instance, if the error is generated during the computation of the preconditioner or solution, then the last digit of the error code is "1" or "2," respectively. The error code is further subdivided into several phases (e.g., Initialization, and Partitioning of the matrix for the computation of the preconditioner) which is indicated by the second digit of the error code from the last. Then, the third digit from the last indicates the subroutine, which generated the error, and finally, the remaining numbers specify the type of the error. We list below the error codes:

1. Preconditioner computation	xxx1
1.1. Initialization	xx11
1.1.1. Top-level subroutine	x111
- memory allocation error	1111
- invalid parameter error	2111
1.2 Partitioning of matrix	xx21
1.2.1. Top-level subroutine	x121
- memory allocation error	1121
1.3 Redistribution of matrix	xx31
1.3.1. Top-level subroutine	x131
- memory allocation error	1131
1.4. Subdomain factorization	xx41
1.4.1. Top-level subroutine	x141
- memory allocation error	1141
1.4.2. Preprocessing subroutine (e.g., MC64)	x241
where "x" is replaced by the error code from the subroutine	
1.4.3. Factorization subroutine (e.g., SuperLU_DIST)	x641
where "x" is replaced by the error code from the subroutine	
1.5. Approximate Schur computation	xx51
1.5.1 Top-level subroutine	x151
- memory allocation error	1151
1.5.2. Triangular solve subroutine (e.g.,SuperLU_DIST)	x651
where "x" is replaced by the error code from the subroutine	
1.6. Preconditioner computation	xx61
1.6.1. Top-level subroutine	x161
- memory allocation error	1161
1.6.2. Preprocessing subroutine (e.g., MC64)	x261
where "x" is replaced by the error code from the subroutine	
1.6.3. Factorization subroutine (e.g., SuperLU_DIST)	x661
where "x" is replaced by the error code from the subroutine	
2.0. Solution computation	xxx2
2.1. Initialization	xx12
2.1.1. Top-level subroutine	x112
- memory allocation error	1112
- invalid parameter	2112
2.2. Forward-substitution	xx22
2.2.1. Top-level subroutine	x122
- memory allocation error	1122
2.2.2. Solver subroutine (e.g., SuperLU_DIST)	x622
2.3. Schur-complement solution	xx32
2.3.1. Top-level subroutine	x132
- memory allocation error	1132
2.3.2. Solver subroutine (e.g., SuperLU_DIST/PETSc)	x632
2.4. Backward-substitution	xx42
2.4.1. Top-level subroutine	x142

- memory allocation error	1142
2.4.2. Solver subroutine (e.g., SuperLU_DIST)	x642

8 Additional options

In this section, we describe some of the additional options included in the solver, which may be useful in some cases. The details and numerical results of these options can be found in [].

8.1 Solving with multiple right-hand-side vectors

The structure pdslin_param has a member variable called "nrhs," which stands for the number of right-hand-sides, and specifies the number of columns in the righ-hand-side vectors. The default value of this variable is one. When this is set to be a value greater than one, a single call to pdslin_solver with PDSLin_SOLVE computes the solution of a linear systems with multiple right-hand-side vectors. In the current version (i.e., version 0.0), the direct solution of the subdomain systems with multiple right-hand-side vectors is computed at the same time, while the iterative solution of the Schur complement system is computed one vector at a time.

Another member variable "ldb" of pdslin_param specifies the leading dimension of the local right-hand-side and solution vectors. This variable is set to be the number of rows in the local coefficient matrix by default (i.e., matrix.mloc), but can be any value greater than or equal to matrix.mloc. The storage for the right-hand-side and solution vectors need to be large enough to store nrhs×ldb elements, where the k-th vector is stored at $((k-1)\times\text{ldb})$-th through $((k-1)\times\text{ldb}+\text{mloc}-1)$-th locations.

There are several advantages with computing the solution of the multiple right-hand-side vectors at a time; 1) symbolic computation needs to be computed once, 2) fewer messages needs to be sent to compute the solution, and 3) the data locality may be improved. These advantages may lead to the reduction in the solution time.

8.2 Recomputing preconditioner for a different matrix

When the coefficient matrix of the linear system changes, the preconditioner to solve the Schur complement system may need to be recomputed. This can be done by first calling pdslin_solver with PDSLin_CLEAN:

```
input.job = PDSLin_CLEAN;
dpdslin_solver( b_loc, x_loc, &matrix, &input, &stat, &info ),
```

then by setting the new coefficient matrix as described in Section 4, and finally calling pdslin_solver with PDSLin_PRECO to compute a new preconditioner:

```
input.job = PDSLin_PRECO;
dpdslin_solver( b_loc, x_loc, &matrix, &input, &stat, &info ).
```

The first call to pdslin_solver with PDSLin_CLEAN frees all the memory allocated to compute the preconditioner, and the second call to pdslin_solver with PDSLin_PRECO recompute the preconditioner for the new matrix.

Several optimizations can be applied, when the second matrix has the same sparsity pattern as that of the first matrix. In order to take advantage of the same sparsity pattern, when calling the pdslin_solver to compute the preconditioner for the first matrix, the user needs to set the member variable "save_pattern" of pdslin_param to be PDSLin_YES:

```
/* compute preconditioner for the first matrix */
input.save_pattern = PDSLin_YES;
input.job = PDSLin_PRECO;
dpdslin_solver( b_loc, x_loc, &matrix, &input, &stat, &info );
```

With this option, all the necessary information for the optimizations will be saved, e.g., the symbolic computation, communication pattern, and the row pointer and column indexes of the local matrix (i.e., matrix.lrowptr and matrix.lcolind do not have to be reset).

Then, when calling pdslin_solver to compute the preconditioner for the second matrix, the user needs to set the member variable "job" of pdslin_param to be PDSLin_NFACT:

```
/* free memory */
input.job = PDSLin_CLEAN;
dpdslin_solver( b_loc, x_loc, &matrix, &input, &stat, &info );

/* compute preconditioner for the second matrix */
input.job = PDSLin_NFACT;
dpdslin_solver( b_loc, x_loc, &matrix, &input, &stat, &info );

/* compute solution */
input.job = PDSLin_SOLVE;
dpdslin_solver( b_loc, x_loc, &matrix, &input, &stat, &info );
```

During the numerical computation of the preconditioner, all the redundant computation for the same sparsity pattern (e.g., the communication setup and symblic factorization) are skipped.

8.3 Using serial SuperLU for solving subdomain problems

Instead of using the parallel direct solver SuperLU_DIST, the user can use a serial SuperLU [4] to solve the subdomain problem. This will restric the user to use one processor per subdomain. However, SuperLU version 4.0 or greater supports an incomplete LU factorization of the matrix, which may reduce the potential bottleneck of the direct solution of the subdomain problems.

In order to link the serial SuperLU to the source code, the user must modify the make.inc file before compiling the source code, and include WITH_SLU in the compiler flag and specify the location of SuperLU on the target machine:

```
############################################################################
# C compiler flags
FLAGS = -fastsse -DWITH_PETSC -DWITH_SLU

############################################################################
# serial SuperLU for interior subdomain solves
I_SLU = -I$(TOP)/SuperLU_4.0/SRC
L_SLU = $(TOP)/SuperLU_4.0/lib/libsuperlu_4.0.a
```

Then, when calling pdslin_solver to compute the preconditioner, the user needs to set the member variable "dom_solver" of pdslin_param to be SLU, which is set to be SLU_DIST by default:

```
/* compute preconditioner for the first matrix */
input.dom_solver = SLU;
input.job = PDSLin_PRECO;
dpdslin_solver( b_loc, x_loc, &matrix, &input, &stat, &info );
```

Another member variable “tau_sub” of pdslin_param specifies the drop threshold to compute the ILU preconditioner. The default value of the variable is zero, which indicates the LU factorization subroutine of SuperLU is used to compute the preconditioner. When this variable is set to be a positive value, it is used as the drop threshold, while a negative value indicates that the default setup of the ILU subroutine is used. See the SuperLU user guide for the details on the default setup [8]. Hence, to use the default setup of the ILU subroutine, pdslin_solver needs to be called as follows:

```
/* compute preconditioner for the first matrix */
input.dom_solver = SLU;
input.tau_sub = -1.0;
input.job = PDSLin_PRECO;
dpdslin_solver( b_loc, x_loc, &matrix, &input, &stat, &info );
```

When an ILU factorization of the subdomain is used, the hybrid solver solves an approximate Schur complement system, i.e.,

$$\widehat{S}x_2 = \widehat{b}_2, \tag{5}$$

where

$$\widehat{S} = A_{22} - (\widetilde{U}_{11}^{-T} A_{21}^T)^T (\widetilde{L}_{11}^{-1} A_{12}),$$

$\widetilde{L}_{11}$ and $\widetilde{U}_{11}$ are the ILU factors of A_{11}, and $\widehat{b}_2 = b_2 - A_{21}(\widetilde{L}_{11}\widetilde{U}_{11})^{-1}b_1$. Hence, in order to obtain the solution to the global system, an outer-iteration is invoked. Since the solution of (5) is computed using a Krylov subspace method with the stopping criteria specified by the residual norm tolerance, the flexible version of GMRES (FGMRES) is used as the outer-iteration [12].

There are several parameters to control the performance of FGMRES. The maximum number and stopping criteria of the iterations are set by the member variables “outer_max” and “outer_tol” of pdslin_param, respectively, while the maximum number of iterations before the restart is set by the member variable “restart” (i.e., the inner and outer iterations are restarted after the same number of iterations). The default values for outer_max and outer_tol are 100 and 10^{-12}, respectively.[4] We also note that since the inner-iteration solves an approximate Schur complement system, only a crude approximate solution are typically required (e.g., the relative residual norm of 10^{-2}).

In some cases, we have observed the computational and memory costs can be reduced by using ILU factorization of the subdomains, but its effect depends on the properties of the linear system.

8.4 Assigning different number of processors to each subdomain

When multiple processors are used to solve each subdomain problem, we have the flexibility of assigning different number of processors to each subdomain. An option available for the user is to set the number of processors assigned to a subdomain to be proportional either to the size of the subdomain or to the number of nonzeros in the subdomain. This can be done by setting the

[4]It is possible to solve the exact Schur complement system with the ILU preconditioner by setting the member variable “exact_schur” to be PDSLin_YES. This requires the computation of both LU and ILU factorization of the subdomains.

member variable "pmap" of pdslin_param to PDSLin_PmapN or PDSLin_PmapNNZ, respectively. This may improve the load balance among the processors assigned to different subdomains, but its effect depends on the structure of the coefficient matrix.

8.5 Reducing memory requirement by taking advantage of symmetry

When the coefficient matrix A is symmetric, the computation of the preconditioner $\widetilde{S}$ of (4) requires only $\widetilde{W}$, i.e., $\widetilde{G} = \widetilde{W}^T$. This option is used when the user sets the member variables "mat_type" and "mat_pattern" of pdslin_param to be SYMMETRIC (see Section 6.3). However, there is an additional option to further reduce the memory required to compute the preconditioner.

With the default setup, both intermediate matrices $\widetilde{W}$ and $\widetilde{G}$ are stored by rows. This storage scheme allows an efficient computation of $\widetilde{T}$. Hence, even when $\widetilde{G} = \widetilde{W}^T$, both $\widetilde{W}$ and $\widetilde{W}^T$ are stored by rows. This can be avoided by setting the member variable "column_w" to be PDSLin_YES. However, the computation of $\widetilde{T}$ is typically slower using this option. Hence, this option is recommended only when the computation of $\widetilde{S}$ requires large memory.

References

[1] ParMETIS - Parallel Graph Partitioning and Fill-reducing Matrix Ordering. http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview.

[2] PETSc - the portable, extensible, toolkit for scientific computation. www.mcs.anl.gov/petsc.

[3] PT-SCOTCH - Software package and libraries for graph, mesh and hypergraph partitioning, static mapping, and sparse matrix block ordering. `http://www.labri.fr/perso/pelegrin/scotch/`.

[4] SuperLU - a general purpose library for the serial direct solution of large, sparse, nonsymmetric systems of linear equations. `http://crd.lbl.gov/~xiaoye/SuperLU/#superlu`.

[5] SuperLU_DIST - a general purpose library for the parallel direct solution of large, sparse, nonsymmetric systems of linear equations. `http://crd.lbl.gov/~xiaoye/SuperLU/#superlu_dist`.

[6] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[7] C. Chevalier and F. Pellegrini. PT-SCOTCH. *Parallel Computing*, 34(6–8):318–331, 2008.

[8] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. SuperLU Users' Guide. Technical Report LBNL-44289, Lawrence Berkeley National Laboratory, September 1999. `http://crd.lbl.gov/~xiaoye/SuperLU/`. Last update: September 2007.

[9] I. Duff and J. Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):889–901, 1999.

[10] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48:71 – 85, 1998.

[11] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

[12] Y. Saad. A flexible inner-outer preconditioned gmres algorithm. *SIAM J. Sci. Comput.*, 14:461–469, 1993.

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

[14] I. Yamazaki and X. Li. On techniques to improve the robustness and scalability of the schur complement method. In *the proceedings of the nineth international meeting on high performance computing for computational science (VECPAR)*, 2010.