PDSLin Users Guide

Ichitaro Yamazaki[*], Xiaoye Li[†], Esmond Ng[†]

Lawrence Berkeley National Laboratory
One Cyclotron Road Berkeley, CA 94720

June 7, 2011

Abstract

This document describes a software package which implements a parallel hybrid (direct/iterative) linear solver based on the Schur complement method for solving a general sparse linear system of equations. The package is named parallel domain decomposition Schur complement based linear solver, or PDSLin in short. We give a brief description of the algorithm, installation, calling sequences, and data structures of PDSLin.[1]

[*]Email: ic.yamazaki@gmail.com

[†]Email: {xsli, 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 difficult to solve using a standard technique alone. 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 may not be readily available. This document describes a software package that implements a parallel hybrid (direct/iterative) linear solver employing techniques from both direct and iterative methods to efficiently solve these linear systems. The package is named parallel domain decomposition Schur complement based linear solver, or PDSLin in short.

PDSLin is an ANSI C library with Fortran interface and uses MPI for communication. It 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 or interface. To compute the solution of the global system, the solution on the interface is first computed by solving a so-called Schur complement system, which is obtained by eliminating the unknowns associated with the interior subdomain systems. Then, the remaining part of the solution is computed by solving the mutually-independent subdomain systems. This Schur complement method provides a framework to develop a hybrid linear solver because various options can be implemented for solving the Schur complement and subdomain systems. For instance, to avoid the potentially large amount of memory required to explicitly form the Schur complement, a preconditioned iterative method can be used to solve the Schur complement system. On the other hand, a parallel direct solver is often effective to solve interior subdomain systems in parallel.

The rest of this document is organized as follows: In Section 2, we first describe the algorithm implemented in PDSLin. Then, in Section 3, we provide the installation guide of the software, while Sections 4 and 5 respectively show typical calling sequences from C and Fortran programs. Next, in Sections 6 and 7, we discuss the inputs and outputs of PDSLin, respectively. Finally, in Section 8, we list some additional options that may be useful in some cases.

2 Algorithmic overview

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

$$AX = B, \tag{1}$$

where A is a square real or complex general matrix, B is a set of given dense right-hand-side vectors, and X is the dense solution vectors to be computed. It uses a non-overlapping domain decomposition technique called the Schur complement method. In this method, the original linear system (1) is first reordered into a system of the following block structure:

$$\left(\begin{array}{cccc|c} D_1 & & & & E_1 \\ & D_2 & & & E_2 \\ & & \ddots & & \vdots \\ & & & D_k & E_k \\ \hline F_1 & F_2 & \dots & F_k & C \end{array}\right) \left(\begin{array}{c} X_1 \\ X_2 \\ \vdots \\ X_k \\ Y \end{array}\right) = \left(\begin{array}{c} B_1 \\ B_2 \\ \vdots \\ B_k \\ Z \end{array}\right), \tag{2}$$

where D_ℓ represents the ℓ-th *interior subdomain*, C consists of *separators*, and E_ℓ and F_ℓ are the *interfaces* between D_ℓ and C. The solution of the linear system (2) is then computed by first solving the Schur complement system

$$SY = \widehat{Z}, \tag{3}$$

where S is the Schur complement defined as

$$S = C - \sum_{\ell=1}^{k} F_\ell D_\ell^{-1} E_\ell,$$

and $\widehat{Z} = Z - \sum_{\ell=1}^{k} F_\ell D_\ell^{-1} B_\ell$. Finally, the remaining part of the solution is computed by solving the mutually-independent subdomain system

$$D_\ell X_\ell = B_\ell - E_\ell Y. \tag{4}$$

See [16] and the references within for a detailed discussion on the Schur complement method.

To avoid the potentially large amount of memory required to explicitly form the Schur complement S, the default setup of PDSLin uses a preconditioned iterative solver to solve the Schur complement system (3), while a parallel direct solver is used to solve the interior subdomain system (4). With this default setup, PDSLin consists of the following two phases:

1. **Computing preconditioner.** First, the global system is partioned to extract the interior subdomains using a parallel nested graph dissection algorithm of PT-SCOTCH [3, 7] or ParMETIS [1, 11]. The interior subdomains are then factorized in parallel, using a parallel direct solver SuperLU_DIST [5, 13] on each subdomain. Finally, a preconditioner $\widetilde{S}$ to solve the Schur complement system (3) is computed as follows:

$$\begin{aligned}
S &= C - \sum_{\ell=1}^{k} (F_\ell U_\ell^{-1})(L_\ell^{-1} E_\ell), \quad \text{after the LU factorization of } D_\ell, \text{ i.e., } D_\ell = L_\ell U_\ell \\
&\approx C - \sum_{\ell=1}^{k} \widetilde{G}_\ell \widetilde{W}_\ell, \quad \text{where } \widetilde{G} \approx F_\ell U_\ell^{-1} \text{ and } \widetilde{W} \approx L_\ell^{-1} E_\ell \text{ with a drop threshold } \tau_0 \\
&\approx C - \sum_{\ell=1}^{k} \widetilde{T}_\ell, \qquad \text{where } \widetilde{T}_\ell \approx \widetilde{G}_\ell^T \widetilde{W}_\ell \text{ with a drop threshold } \tau_1 \\
&\approx \widetilde{S}, \qquad \text{where } \widetilde{S} \approx C - \textstyle\sum_{\ell=1}^{k} \widetilde{T}_\ell \text{ with a drop threshold } \tau_2.
\end{aligned} \tag{5}$$

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

2. **Computing solution.** First, the Schur complement system (3) is solved using a preconditioned Krylov subspace method of PETSc [2, 6]. Then, the subdomain systems (4) are solved in parallel, using the already-computed LU factors of the subdomains.

See [17] for a more detailed discussion on the parallel implementation of PDSLin.

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 or above) [5], which is used to solve the interior subdomain systems and possibly the Schur complement system by a parallel direct method;
- METIS [12], which is used by SuperLU_DIST for matrix ordering; and
- PETSc (version 2.3.3 or above) [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 source code of PDSLin 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              : PDSLin 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 was used on a Cray XT-4 machine at NERSC, where "-DWITH_PETSC" indicates that PDSLin is linked to PETSc.

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

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

###########################################################################
# archiver and flags used to build PDSLin
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 PDSLin library
LINKER = CC

#########################################################################
# External 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 Calling sequence 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 the 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 PDSLin. 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 PDSLin               */

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 PDSLin */
   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 PDSLin 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 PDSLin 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 PDSLin */
   dpdslin_finalize(&input, &matrix);
   MPI_Finalize();

   return 0;
}
```

PDSLin expects the input coefficient matrix A 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 A, 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 {
  /* coefficient 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 PDSLin */
  .......
} dPDSLinMatrix;
```

The local right-hand-side and solution vectors are stored in the corresponding format, i.e., in the example above, "b_loc" and "x_loc" contain 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 PDSLin subroutines used in the example above:

- **dpdslin_init**: subroutine to initialize PDSLin;
- **dpdslin_solve**: computational subroutine to compute preconditioner or solution;
- **pdslin_print_stat**: subroutine to print performance statistics; and
- **dpdslin_finalize**: subroutine to terminate PDSLin.

In the remaining of this section, we describe each of these four subroutines in more details.

Before calling any of PDSLin 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 user needs to specify the last three arguments. Specifically, 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 initialization subroutine sets up all the required MPI variables when the fifth argument argc is a non-negative value. Hence, when this 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 two arguments argc and argv can be set to be zero and NULL, respectively. 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. On return, the first three arguments "input," "matrix," and "stat" are initialized with the default values.

The default values of PDSLin parameters are as follow: The number of subdomains is the smaller of 64 and the largest power of two, which is less than or equal to the total number of processors. Hence, if the number of processor is less than or equal to 64, then each subdomain is factorized using one processor, while multiple processors are used to factorize each subdomain if more than 64 processors are available. If PDSLin is linked to PETSc, then the preconditioner $\widetilde{S}$ is computed using the drop thresholds $(\tau_0, \tau_1, \tau_2) = (10^{-6}, 0.0, 10^{-5})$, while the number of processors to compute the LU factorization of $\widetilde{S}$ is set to be the half of the number of subdomains. Finally, 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. See Section 6.3 for the brief description of the member variables.

Once PDSLin is initialized and the coefficient matrix is set, we can invoke PDSLin to solve the corresponding linear system. PDSLin allows the user to compute the preconditioner and solution, separately. Specifically, to compute the preconditioner (step 3 of the code above), PDSLin 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, both of which are not used for computing the preconditioner. The next two arguments "matrix" and "input" respectively contain the coefficient matrix and the input parameters. At the successful completion of the subroutine call, the preconditioner to solve the linear system has been computed. Furthermore, the last two arguments "stat" and "info" respectively contain performance statistics and an error code, which will be explained in Section 7.

Then, to compute the solution (step 5), we can invoke PDSLin again:

```
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" and without recomputing the preconditioner.

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

```
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 PDSLin subroutine calls are completed, the user can call the following subroutine to free up all the memory allocated by PDSLin (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(ptr) is used to free up the memory allocated to ptr.

5 Calling sequence in Fortran

Just like the C subroutines described in Section 4, all the Fortran subroutines for solving real or complex linear systems start with the letter "d" or "z," respectively. Example programs that use the Fortran interface of PDSLin are included in the "fortran\examples" directory.

The following piece of the code shows how to call PDSLin 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
*
*   .. PDSLin 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 PDSLin (step 2) --
*
      call MPI_Init(ierr)
      call dpdslin_init_f( input, matrix, stat, pdslin_comm )
*
*   -- set coefficient matrix (step 3)--
*
      call dpdslin_set_matrix_f( input, matrix, frow, mloc,
     $                           n, colptr, rowind, values )
*
*   -- initialize PDSLin 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 names of the corresponding C subroutines discussed in Section 4. Here, we focus on the differences between the Fortran and C calling sequences. See Section 4 for a detailed discussion on each subroutine call.

The Fortran program first needs to include two modules that define PDSLin parameters and

interfaces (step 0 of the code above):

```
      use pdslin_param
      use pdslin_mod
```

Then, to use PDSLin from a Fortran program, appropriate handles to the C data structures, dPDSLinMatrix, pdslin_param, and pdslin_stat, need to be 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 can be set by calling the following subroutine (step 3):

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

where the distributed CSR format is used to store the matrix (see Section 4 for more details on the matrix 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." 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 PDSLin (including those of opaque objects) are deallocated:

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

6 Inputs

In this section, we describe all the input arguments to the pdslin_solver subroutine. Specifically, in Section 6.1, we describe the data structure **dPDSLinMatrix** (or **zPDSLinMatrix**) that stores the coefficient matrix A. Then, in Section 6.2, we explain how the right-hand-side vectors should be distributed among the processors. Finally, in Section 6.3, we describe the data structure **pdslin_param** that stores all the input parameters of PDSLin.

6.1 Coefficient matrix

The coefficient matrix A are stored in PDSLinMatrix structure. The only member variables of this structure, which the user can modify, are:

```
typedef struct {
  /* coefficient 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 PDSLin */
  .......
} dPDSLinMatrix;
```

These variables are used to store the matrix A in the distributed CSR format as described in Section 4.

6.2 Right-hand-side vectors

The local right-hand-side vectors should be stored in the format that corresponds to the coefficient matrix stored in PDSLinMatrix (see Section 6.1). Specifically, each processor stores the subset of the contiguous rows of the right-hand-side vectors, 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. Furthermore, the leading dimension of the vectors can be set by the member variable **ldb** of the data structure pdslin_input (see Section 6.3).

6.3 Solver input parameters

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;   /* global MPI communicator for PDSLin   */
```

```
  int        job;           /* specify job for PDSLin               */
  int        nrhs;          /* number of right-hand sides           */
  int        ldb, ldx;      /* leading dimension of b and x         */
  int        check_input;   /* to check input parameters            */
  int        gather_stat;   /* to gather statistics                 */
  t_verbose  verbose;       /* to print progress of PDSLin          */
  t_mtype    mat_type;      /* coefficient matrix type              */
  t_mtype    mat_pattern;   /* coefficient matrix pattern           */
  int        free_local;    /* to internally free coeffcient 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 PDSLin, and can be PDSLin_PRECO, PDSLin_SOLVE, PDSLin_PRESOLVE, PDSLin_CLEAN, or PDSLin_NFACT to compute preconditioner, to compute solution, to compute both preconditioner and solution, to free up all the memory allocated by PDSLin, or to compute the preconditioner with the same sparsity pattern as the previous call to pdslin_solve (see Section 8.2), respectively.
- **nrhs**: specifies the number of right-hand-side vectors. The default value is one. Information on calling PDSLin with multiple right-hand-side vectors can be found in Section 8.1.
- **ldb** and **ldx**: specify the leading dimensions of the right-hand-side and solution vectors, respectively. The default values are -1. If they are set to be a negative value, they are reset

to be equal to the dimension of the local matrix (i.e. **mloc** of PDSLinMatrix described in Section 6.1) when pdslin_solver is called to compute the solution vectors.

- **verbose** (of type t_verbose): specifies the level of messages to be printed by PDSLin as it runs, and can be PDSLin_VALL, PDSLin_VWARN, or PDSLin_VNONE to print all the status information, only warning messages, or nothing, respectively. The default is PDSLin_VNONE.

- **mat_type** and **mat_pattern** (of type t_mtype): can be either SYMMETRIC or UNSYMMETRIC, and specify the matrix type and its sparsity pattern. The default is UNSYMMETRIC for both.

- **schur_pattern** (of type t_mtype): can be either SYMMETRIC or UNSYMMETRIC, and specifies if the Schur complement should be kept symmetric or not, respectively. 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 of the local matrix (i.e., matrix.lrowptr, matrix.lcolind, and matrix.lnzval described in Section 6.1) are not changed.

- **num_doms** (of type integer): specifies the number of interior subdomains. In the current version, the interior subdomains are extracted using a nested graph dissection 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 the number of subdomains (i.e., 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 a power of two and greater than or equal to num_doms.

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

- **equil_dom** (of type integer): specifies the equilbration and row permutation used on the subdomains by SuperLU. 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 used. A negative integer specifies that only the equilbration is applied. See [9] for more information on the equilbration. The default is -1 to apply just the equilbration.

- **equl_schur** (of type integer): 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. If it is set to be an integer value between 1 and 5, then it is used to call an external subroutine MC64 [10]. The default value is 5, which will compute a row permutation to move large elements to diagonal and equilbration to scale the matrix so that the diagonals are all of modulus one, and off diagonals have modulus less than or equal to one. See [17] for more information on this parallel matrix preprocessing techniques.

- **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. The description of the iterative methods can be found, for example, in [15].

- **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 compute the LU factorization of $\widetilde{S}$. The default is set to be half of the number of subdomains, but at least one. It is dynamically adjusted so that each processor has at least $1,000$ rows of $\widetilde{S}$.

- **check_input** (of type integer): specifies if PDSLin 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 is PDSLin_YES. When pdslin_solver is called to compute preconditioner or solution, only the relevant parameters are checked. Specifically, the preconditioner cannot be changed by changing a parameter to compute the preconditioners (e.g., num_doms or drop_tau0) when calling pdslin_solver to compute solution. See Section 8.2 for more details to recompute a preconditioner.

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

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 vectors are distributed among the processors. Then, in Section 7.2, we list the error codes that may be returned by pdslin_solver. Finally, in Section 7.3, we describe the data structure **pdslin_stat** that stores the performance statistics of PDSLin.

7.1 Solution vectors

The local solution vectors (i.e., the second argument in pdslin_solver) is returned in the same format as the local right-hand-side vectors (see Section 6.2). Specifically, each processor stores the subset of the contiguous rows of the global solution vectors, where the index of the first row and the number of rows in the local vectors are specified by the member variables **frow** and **mloc** of PDSLinMatrix (see Section 6.1), respectively. The leading dimension of the local vectors can be set by the member variable **ldx** of the data structure pdslin_param (see Section 6.3), which must be greater than mloc.

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: PDSLin successfully completed the requested task.

-x Invalid parameters: The x-th parameter 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 has successfully completed 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 occurence of a zero pivot. The memory requirement may be reduced by increasing the drop thresholds or the number of subdomains, while the zero pivot may be avoided by using row permutation on each subdomain. If the row permutation did not help, the global matrix may need to be preprocessed 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 pivot. You may be able to reduce the memory requirement by increasing the drop thresholds or the number of subdomains. The zero pivot may be avoided by using row permutation on the approximate Schur complement, but 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 or the number of subdomains. Alternatively, you can increase the relative residual norm used for the 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 version of PDSLin, each processor may return a different error code. Furthermore, if one processor fails, the rest of the processors may hung. To print out the error or warning messages, set the verbose level of PDSLin to be PDSLin_VERROR or PDSLin_VWARN, respectively (see Section 6.3).

7.3 Performance statisitics

The features described in this section are mainly for experts users who like to study the performance of PDSLin 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 local 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 nonzeros in LU of approximate 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) */
                   /* default values are PDSLin_NO                               */
} pdslin_error;
```

The detailed error codes are organized in a hierarchical 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 additional options of PDSLin, which may be useful in some cases.

8.1 Solving with multiple right-hand-side vectors

The structure pdslin_param (see Section 6.3) has a member variable called "nrhs," which stands for the number of right-hand-side vectors, and specifies the number of columns in the right-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.

The member variable "ldb" and "ldx" of pdslin_param specify the leading dimensions of the

local right-hand-side and solution vectors, respectively. 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 $((k-1)\times$ldb$)$-th through $((k-1)\times$ldb$+$mloc$-1)$-th locations.

There are several advantages of 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 coefficient matrix

When the coefficient matrix of the linear system changes, the preconditioner to solve the Schur complement system may need to be recomputed. In this case, the old preconditioner needs to be first freed by calling pdslin_solver with PDSLin_CLEAN:

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

Then, a new preconditioner can be computed by setting the new coefficient matrix as described in Section 4, and calling pdslin_solver with PDSLin_PRECO:

```
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 old 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 symbolic factorization) are skipped.

8.3 Using serial SuperLU to solve a subdomain system

Instead of using the parallel direct solver SuperLU_DIST, the user can use a serial direct solver SuperLU [4, 8] to solve the subdomain system. This restricts the user to use one processor per subdomain. However, SuperLU version 4.0 or greater supports an incomplete LU factorization of the subdomain, which may reduce the potential bottleneck of the direct solution of the subdomain systems.

In order to link the serial SuperLU to PDSLin, the user must modify the make.inc file before compiling the source code; specifically, -DWITH_SLU needs to be included in the compiler flag and the location of SuperLU on the target machine must be specified:

```
############################################################################
# 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 factors of subdomains. 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. 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 );
```

The details on the default setup of the ILU subroutine can be found in the SuperLU user guide [9].

When an ILU factorization of the subdomain is used, the inner-iteration of PDSLin solves an approximate Schur complement system,

$$\widehat{S}\widehat{Y} = \widehat{Z}, \tag{6}$$

where

$$\widehat{S} = C - \sum_{\ell=1}^{k} (F_\ell \widehat{U}_\ell^{-1})(\widehat{L}_\ell^{-1} E_\ell),$$

$\widehat{L}_\ell$ and $\widehat{U}_\ell$ are the ILU factors of the ℓ-th subdomain D_ℓ, and $\widehat{Z} = Z - \sum_{\ell=1}^{k} F_\ell (\widehat{L}_\ell \widehat{U}_\ell)^{-1} B_\ell$. Hence, in order to obtain the solution to the global system, an outer-iteration is invoked. Since the solution of (6) 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 [14].

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 "inner_restart." The default values for outer_max and outer_tol are 100 and 10^{-12}, respectively.[2] Since the inner-iteration solves an approximate Schur complement system, only a crude approximate solution is typically needed (e.g., the relative residual norm of 10^{-2}).

In some cases, we have observed that the computational and memory requirements of PDSLin can be reduced using the ILU factorization of the subdomains, but its effects depend on the properties of the linear system.

8.4 Assigning a different number of processors to each subdomain

When multiple processors are used to solve each subdomain system, 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 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 effects depend on the structure of the coefficient matrix. The default is PDSLin_NO, which distributes the processors evenly among the subdomains.

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 (5) requires only $\widetilde{W}_\ell$, i.e., $\widetilde{G}_\ell = \widetilde{W}_\ell^T$. Hence, $\widetilde{G}$ is not computed 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 $\widetilde{W}_\ell$ and $\widetilde{G}_\ell$ are stored by rows. This storage scheme allows an efficient computation of $\widetilde{T}_\ell$. Hence, even when $\widetilde{G}_\ell = \widetilde{W}_\ell^T$, both $\widetilde{W}_\ell$ and $\widetilde{W}_\ell^T$ are stored by rows. This can be avoided by setting the member variable "column_w" to be PDSLin_YES. However, the computation of $\widetilde{T}_\ell$ is typically slower using this option. Hence, this option is recommended only when the computation of $\widetilde{S}$ requires large memory.

[2] 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.

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] J. Demmel, S. Eisenstat, J. Gilbert, X. Li, and J. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20:720–755, 1999.

[9] J. Demmel, J. Gilbert, and X. Li. SuperLU Users' Guide. Technical Report LBNL-44289, Lawrence Berkeley National Laboratory, September 1999. `http://crd.lbl.gov/~xiaoye/SuperLU/`. Last update: September 2007.

[10] 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.

[11] 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.

[12] Karypis Lab, Digital Technology Center, Department of Computer Science and Engineering, University of Minesota. METIS - Serial Graph Partitioning and Fill-reducing Matrix Ordering. http://glaros.dtc.umn.edu/gkhome/metis/metis.

[13] 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, 2003.

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

[15] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2nd edition, 2003.

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

[17] 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.