PreAlps Users' Guide

V1.0, August 2018

Simplice Donfack[*] Laura Grigori[†] Olivier Tissot[‡]

[*]INRIA
[†]INRIA
[‡]INRIA

1 Introduction

We consider solving sparse linear systems of equations $Ax = b$ by using a Krylov subspace iterative method as CG[10], where A is a symmetric positive definite (SPD) matrix. Each iteration of a typical Krylov subspace solver involves matrix-vector multiplications and several dot products in order to orthogonalize the Krylov subspace. These dot products require collective communication among all processors which does not scale to a very large number of processors. This is the main bottleneck for the scalability of Krylov subspace methods. To increase the scalability of Krylov subspace solvers, we focus on enlarged Krylov subspace methods [8] and a robust and algebraic preconditioner LORASC [6].

Enlarged Krylov subspace methods [8] represent a new approach that consists of enlarging the Krylov subspace by a maximum of t vectors per iteration, based on a domain decomposition of the graph of the input matrix. The solution of the linear system is sought in the enlarged subspace, which is a superset of the classical subspace. The enlarged Krylov projection subspace methods lead to faster convergence in terms of number iterations and parallelizable algorithms with less communication, compared to classical Krylov methods. In addition, each iteration of such a method relies on multiplying a matrix with multiple vectors at once, and this typically allows for better resource utilization.

LORASC [6] is a robust algebraic preconditioner that can be built and applied in parallel. The graph of the matrix is partitioned by using nested dissection partitioning into N disjoint domains and a separator formed by the vertices connecting the N domains. The permuted matrix has an arrow head structure with $N+1$ blocks on the main diagonal. The preconditioner relies on the Cholesky factorization of the first N diagonal blocks and on approximating the Schur complement corresponding to the separator block. The approximation of the Schur complement involves the factorization of the last diagonal block and a low rank correction obtained by solving a generalized eigenvalue problem. As results, LORASC is well suitable for parallelism and converge faster with respect to the classic approaches in terms of number of iterations and solving time.

The `preAlps` library aims at solving large scale sparse linear systems on massively parallel architectures. In its current state the library contains routines for solving symmetric positive definite (SPD) linear systems with enlarged Conjugate Gradient (ECG) in parallel. It also provides routines for using LORASC as a preconditioner, which can be used in combination with ECG or with another Krylov method for SPD matrices as for example provided in PETSC[4]. We also provide preliminary results that show the parallel performance of the library.

2 Using preAlps library

2.1 Overview

`preAlps` is a parallel library that implements several preconditioners and iterative solvers based on communication avoiding principle. It focuses on two main aspects: the reformulation of Krylov-based iterative methods to allow a drastic reduction in the number of global communications with respect to classical formulations, and the design of robust preconditioners to accelerate the convergence of iterative methods. The current first version of the library provides an efficient implementation of ECG and LORASC preconditioner. ECG is based on enriching the Krylov subspace used in classical methods that allows to reduce drastically the communication cost of the iterative solver. LORASC is based on a low rank approximation of the Schur complement. These algorithms are well suited for parallelism and are described in details in [8] and [6]. In our experiments, by increasing the numbers of vectors added to the Krylov subspace by 12 instead of 1 as in the traditional conjugate gradient (CG) approach, ECG is up to 3x faster than the equivalent solver routine in PETSC, while LORASC is up to 7x faster than Block Jacobi preconditioner.

2.2 Dependencies of preAlps with external librairies

`preAlps` depends on few external librairies that need to be installed and linked with `preAlps`:

- BLAS and LAPACK [3]: BLAS is a standard library for performing basic vector and matrix operations. LAPACK is a standard software for numerical linear algebra. Although any library providing BLAS and LAPACK can be used, we recommend MKL [19].
- METIS[12] and ParMETIS [13]: the sequential and parallel graph partitioning tools. METIS is required in order to use ECG, while ParMETIS is required in order to use LORASC. We recommend to install ParMETIS as it already contains all METIS routines. `preAlps` were tested with METIS 5.1.0 and ParMETIS 4.0.3. These partitioning tools can be downloaded from `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview` and `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.
- PARPACK [14]: a parallel library used to solve eigenvalue problems. PARPACK is required in order to use LORASC. A latest version can be downloaded from `http://www.caam.rice.edu/software/ARPACK/download.html` . At the moment, only PARPACK is supported in `preAlps`, but we plan to use other eigenvalue solvers.
- MUMPS [2]: a distributed parallel sparse direct solver. MUMPS is required in order to use LORASC. `preAlps` were tested with MUMPS 5.1.2. It can be downloaded from `http://mumps.enseeiht.fr/`
- PARDISO: a sequential and multithreaded sparse direct solver. This library is optional if MKL is already provided. If MKL is not provided, PARDISO from `http://pardiso-project.org/` should be installed.

Table 1 shows the list of dependencies of the main components of preAlps with exernal librairies. In order to use ECG Solver, BLAS, METIS and MKL (with PARDISO) are required. In order to use LORASC, BLAS, ParMETIS, PARPACK and MUMPS are required. For the full installation of preAlps, BLAS, ParMETIS, PARPACK, MUMPS and MKL (with PARDISO) are required.

	BLAS	METIS	ParMETIS	PARPACK	MUMPS	PARDISO
ECG Solver	×	×				MKL
LORASC preconditioner	×		×	×	×	ANY
Full installation	×	×	×	×	×	MKL

Table 1: Dependencies of preAlps with external librairies.

2.3 Installation

`preAlps` is developed at INRIA in the context of the NLAFET project, and it can be downloaded from `https://github.com/NLAFET/preAlps`. The complete installation of `preAlps` could be summarized as follows:

2.3.1 Install the dependencies

1. Make sure to have the following libraries or install them:
   (a) MPI
   (b) MKL
   (c) METIS
   (`http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`)
2. If you want to enable LORASC preconditioner in preAlps, make sure to have the following libraries or install them:
   (a) ParMETIS
   (`http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`)
   (b) MUMPS
   (`http://mumps.enseeiht.fr/`)
   (c) PARPACK
   (`http://www.caam.rice.edu/software/ARPACK/download.html`)
3. If you want to compare ECG results with PETSc, make sure to have the following library or install it:
   (a) PETSc
   (`https://www.mcs.anl.gov/petsc/download/index.html`)

2.3.2 Get and install preAlps

1. Get the latest version of preAlps.
   ```
   $ git clone git@github.com:NLAFET/preAlps.git preAlps
   ```
2. At the top of the `preAlps` folder, edit the `make.inc` file in order to set the compiler directives and flags.

3. Copy an example of `make.lib.inc` from the directory MAKES.

   In order to use ECG Solver only, type:
   `$ cp MAKES/make.lib.inc-ecg make.lib.inc`
   For the full installation of preAlps, type:
   `$ cp MAKES/make.lib.inc make.lib.inc`

4. Edit the `make.lib.inc` file in order to set the path and compiler directives for the libraries installed in 1, 2, and 3. Make sure the LD_FLAGS of these libraries are correctly set.

5. Type `make` to compile `preAlps` and create the library `lib/libpreAlps.a` and the example programs.
   `$ make`

6. Now, the functions from the lib `preAlps` can be called by a program by including their corresponding `header` file. The folder `example` provide few stand-alone programs to test the library.

In addition, the example program test_ecgsolve.c allows the end user to call CG and Block Jacobi from PETSC and compare its performance with ECG from `preAlps`. PETSC [4] provides a suite of routines for scientific applications including solvers and preconditioners.

2.4 Input data formats

Most routines in `preAlps` require matrices stored into a compressed sparse row format (CSR). PreAlps provides an internal library refers to as `CPaLAMeM` to read such matrices as *CPLM_MatCSR_t* object. The simplified *CPLM_MatCSR_t* structure is presented as follows:

- m, n : the size of the matrix.
- rowPtr: the beginning position of each rows as described in the CSR format.
- colInd: the column indexes of each non-zeros elements of the matrix.
- val: the corresponding values of each non-zeros elements of the matrix.

The following block of code presents an example for reading an external matrix.

```c
/* Including headers */
#include <preAlps_cplm_matcsr.h>

int main(int argc, char **argv){

  /* The matrix file to load */
  char matrixFileName[]="cage4.mtx";

  /* Create an empty CSR Matrix */
  CPLM_Mat_CSR_t A = CPLM_MatCSRNULL();
```

```

    /* Load the matrix file */
    CPLM_LoadMatrixMarket(matrixFileName, &A);
    ...
```

3 Enlarged Conjugate Gradient

3.1 Introduction

We briefly recall the definition of Enlarged Conjugate Gradient (ECG) [8] for solving the linear system $Ax = b$.

We use the following notations: $U^\top$ is the transpose of a matrix U, A is a symmetric ($A^\top = A$) positive definite ($x^\top Ax > 0$, $\forall x \neq 0$) real matrix of size $n \times n$, B is a real matrix of size $n \times t$ where t is the number of vectors used in Enlarged CG, $B^{(i)}$ is the i-th column of a matrix B, X_0 is an initial guess for the linear system $AX = B$, *i.e.* it is a real matrix of size $n \times t$. We denote the initial residual matrix $R_0 = B - AX_0$. We call t the initial block size. Unless otherwise stated, $||.||$ denotes the usual euclidean norm both for vectors and matrices.

Given a splitting decomposition represented by the operator T,

$$T(x) = \begin{pmatrix} * & & & & \\ \vdots & & & & \\ * & & & & \\ & * & & & \\ & \vdots & & & \\ & * & & & \\ & & \ddots & & \\ & & & * & \\ & & & \vdots & \\ & & & * & \\ & & & & * \\ & & & & \vdots \\ & & & & * \end{pmatrix}$$

where x is a vector of size n and $T(x)$ is of size $n \times t$. The first columns of $T(x)$ contains the first components of x and so on for the following columns of $T(x)$. The initial residual denoted r_0, the corresponding enlarged Krylov subspace is defined as

$$\mathcal{K}_{k,t} = \mathrm{span}^{\square}\{T(r_0), AT(r_0), \ldots, A^{k-1}T(r_0)\}. \tag{3.1}$$

Using this definition, *Grigori, Moufawad and Nataf* [8] derive a *Short Recurrence Enlarged* CG. The stopping criterion is the EUCLIDEAN norm of $r_k = \sum_i R_k^{(i)}$ and the approximate solution is $x_k = \sum_i X_k^{(i)}$. In [7], we use the framework of Block Conjugate

Gradient [17] to derive two versions of ECG: Orthodir (Algorithm 1) which corresponds to the SRE-CG in [8] and Orthomin (Algorithm 2) which corresponds to the Block Conjugate Gradient of O'Leary.

Algorithm 1 Preconditioned Orthodir Enlarged CG

```
Require: $A$, $M$, $b$, $x_0$, $k_{\max}$, $\varepsilon_{\text{solver}}$
Ensure: $||b - Ax_k|| < \varepsilon_{\text{solver}}$ or $k = k_{\max}$
 1: $R_0 = T(b - Ax_0)$
 2: $r_0 = b - Ax_0$
 3: $P_0 = 0$
 4: $P_1 = M^{-1}R_0$
 5: $k = 1$
 6: while $||r_{k-1}|| > \varepsilon_{\text{solver}}||b||$ and $k < k_{\max}$ do
 7:     $Q_k = AP_k$
 8:     A-orthonormalize($P_k$,$Q_k$)
 9:     $\alpha_k = P_k^\top R_{k-1}$
10:     $X_k = X_{k-1} + P_k\alpha_k$
11:     $R_k = R_{k-1} - Q_k\alpha_k$
12:     $Z_k = M^{-1}Q_k$
13:     $P_{k+1} = Z_k - P_kQ_k^\top Z_k - P_{k-1}Q_{k-1}^\top Z_k$
14:     $r_k = \sum_{i=1}^t R_k^{(i)}$
15:     $k = k + 1$
16: end while
17: $x_k = \sum_{i=1}^t X_k^{(i)}$
```

Algorithm 2 Preconditioned Orthomin Enlarged CG

```
Require: $A$, $M$, $b$, $x_0$, $k_{\max}$, $\varepsilon_{\text{solver}}$
Ensure: $||b - Ax_k|| < \varepsilon_{\text{solver}}$ or $k = k_{\max}$
 1: $R_0 = B - AX_0$
 2: $r_0 = b - Ax_0$
 3: $P_1 = M^{-1}R_0$
 4: $k = 1$
 5: while $||r_{k-1}|| > \varepsilon_{\text{solver}}||b||$ and $k < k_{\max}$ do
 6:     $Q_k = AP_k$
 7:     A-orthonormalize($P_k$,$Q_k$)
 8:     $\alpha_k = P_k^\top R_{k-1}$
 9:     $X_k = X_{k-1} + P_k\alpha_k$
10:     $R_k = R_{k-1} - Q_k\alpha_k$
11:     $Z_k = M^{-1}R_k$
12:     $P_{k+1} = Z_k - P_kQ_k^\top Z_k$
13:     $r_k = \sum_{i=1}^t R_k^{(i)}$
14:     $k = k + 1$
15: end while
16: $x_k = \sum_{i=1}^t X_k^{(i)}$
```

3.2 Routines

Our implementation of ECG is based on Reverse Communication Interface [11] and written in C and MPI. Following this scheme we provide 4 routines:

- `preAlps_ECGInitialize(ECG_t* ecg, double* rhs, int* rci_request)`,
- `preAlps_ECGIterate(ECG_t* ecg, int* rci_request)`,
- `preAlps_ECGStoppingCriterion(ECG_t* ecg, int* stop)`
- `preAlps_ECGFinalize(ECG_t* ecg, double* solution)`.

In order to ease its usage, we encapsulate all the required information by ECG in the structure `preAlps_ECG_t`. This structure is defined as,

```
typedef struct {
  /* Input variable */
  double* b;                  /**< Right hand side */

  /* Internal symbolic variables */
  CPLM_Mat_Dense_t* X;   /**< Approximated solution */
  CPLM_Mat_Dense_t* R;   /**< Residual */
  CPLM_Mat_Dense_t* V;   /**< Descent directions ([P,P_prev] or P) */
  CPLM_Mat_Dense_t* AV;  /**< A*V */
  CPLM_Mat_Dense_t* Z;   /**< Preconditioned residual (Omin) or AP (Odir) */
  CPLM_Mat_Dense_t* alpha; /**< Descent step */
  CPLM_Mat_Dense_t* beta; /**< Step to construt search directions */

  /** User interface variables */
  CPLM_Mat_Dense_t* P;    /**< Search directions */
  CPLM_Mat_Dense_t* AP;   /**< A*P */
  double* R_p;            /**< Residual */
  double* P_p;            /**< Search directions */
  double* AP_p;           /**< A*P_p */
  double* Z_p;            /**< Preconditioned residual (Omin) or AP (Odir) */

  /** Working arrays */
  double*          work;
  int*             iwork;

  /* Single value variables */
  double           normb;    /**< norm_2(b) */
  double           res;      /**< norm_2 of the residual */
  int              iter;     /**< Iteration */
  int              bs;       /**< Block size */
  int              kbs;      /**< Krylov basis size */

  /* Options and parameters */
  int                        globPbSize; /**< Size of the global problem */
  int                        locPbSize; /**< Size of the local problem */
  int                        maxIter;   /**< Maximum number of iterations */
  int                        enlFac;    /**< Enlarging factor */
  double                     tol;       /**< Tolerance */
  preAlps_ECG_Ortho_Alg_t    ortho_alg; /**< A-orthonormalization algorithm */
```

```c
  preAlps_ECG_Block_Size_Red_t bs_red; /**< Block size reduction */
  MPI_Comm           comm;             /**< MPI communicator */
} preAlps_ECG_t;
```

where `Ortho_Alg_t` and `Block_Size_Red_t` are `enum`. The structure `CPLM_Mat_Dense_t` represents local dense matrices, it contains the pointer to the data (`val`), the number of rows (`info.m`), the number of columns (`info.n`) and an `enum` to specify the matrix storage type (`ROW_MAJOR` or `COL_MAJOR`).

First, the user has to declare a variable of type `preAlps_ECG_t` and set values for the parameters: `comm`, `globPbSize`, `locPbSize`, `maxIter`, `enlFac`, `tol`, `ortho_alg`, and `bs_red` (`ADAPT_BS` for dynamic reduction of the search directions, and `NO_BS_RED` otherwise). The variable `globPbSize` corresponds to the number of unknowns of the global solution x (the dimension of A) and `locPbSize` corresponds to the number of unknown owned locally (the number of rows of A that are stored locally).

Then, in order to allocate memory and initialize the structure, the user has to call `preAlps_ECGInitiliaze(&ecg, rhs, &rci_request)` where `rhs` is an array (`double *`) representing the right hand side and `rci_request` is an integer. After this call, he has to apply the preconditioner to `ecg.R` and put the result into `ecg.P`. And then, he has to apply the operator A to `ecg.P` and put the result into `ecg.AP`. These two operations has to be executed in parallel assuming that `ecg.R`, `ecg.P`, `ecg.AP` contains local rows of R, P and AP which are distributed as row panel over the processors.

Afterwards, the user has to call `preAlps_ECGIterate(&ecg,&rci_request)` until convergence of the method. Following RCI scheme, after this call the user has to check the value of `rci_request`. If `rci_request = 0`, then the user is requested to apply A on `ecg.P` and put the resul into `ecg.AP`. If `rci_request = 1`, then the user can check for convergence of the method. If the method did not converge, then depending on the choice of the orthogonalization algorithm (`ORTHODIR` or `ORTHOMIN`) he has to apply the preconditioner to `ecg.R` (`ORTHOMIN`) or `ecg.AP` (`ORTHODIR`) and put the result into `ecg.Z`.

When convergence is reached, it is possible to recover the solution and free the structure `ecg` by calling `preAlps_ECGFinalize(&ecg,sol)` where `sol` is an array (`double*`) already allocated.

To sum up, for solving a linear system with a block Jacobi preconditioner, the general calling sequence will be

```c
// Set parameters
ecg.comm = MPI_COMM_WORLD; /* MPI Communicator */
ecg.globPbSize = M;      /* Size of the global problem */
ecg.locPbSize = m;       /* Size of the local problem */
ecg.maxIter = maxIter;   /* Maximum number of iterations */
ecg.enlFac = 2;          /* Enlarging factor */
ecg.tol = tol;           /* Tolerance of the method */
ecg.ortho_alg = ORTHODIR; /* Orthogonalization algorithm */
ecg.bs_res = ADAPT_BS;  /* Dynamic reduction of the search directions */
// Allocate memory and initialize variables
preAlps_ECGInitialize(&ecg,rhs,&rci_request);
// Finish initialization
preAlps_BlockJacobiApply(ecg.R,ecg.P);
preAlps_BlockOperator(ecg.P,ecg.AP);
// Main loop
```

```c
while (stop != 1) {
  ierr = preAlps_ECGIterate(&ecg,&rci_request);
  if (rci_request == 0) {
    // AP = A*P
    preAlps_BlockOperator(ecg.P,ecg.AP);
  }
  else if (rci_request == 1) {
    ierr = preAlps_ECGStoppingCriterion(&ecg,&stop);
    if (stop == 1) break;
    if (ecg.ortho_alg == ORTHOMIN)
      // Z = M^-1*R
      preAlps_BlockJacobiApply(ecg.R,ecg.Z);
    else if (ecg.ortho_alg == ORTHODIR)
      // Z = M^-1*AP
      preAlps_BlockJacobiApply(ecg.AP,ecg.Z);
  }
}
// Retrieve solution and free memory
preAlps_ECGFinalize(&ecg,sol);
```

3.3 Example program

The following listing is an abbreviated version of usage of Enlarged Conjugate Gradient routines for solving a parallel linear system. In this example, we provide routines for calling block Jacobi as a preconditioner. However the user can combine ECG with any preconditioner. The routines for initializing and applying a preconditioner and associated data structures can evolve in our library. The user can also use his own routine for computing the matrix set of vectors product instead of using the `preAlps_BlockOperator` routine.

```c
  /* STD */
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <math.h>
/* MPI */
#include <mpi.h>
/* MKL */
#include <mkl.h>

/* CPaLAMeM */
//#include <cpalamem_macro.h>
//#include <cpalamem_instrumentation.h>

/* preAlps */
#include "operator.h"
#include "block_jacobi.h"
#include "ecg.h"

/* Command line parser */
#include <ctype.h>
#include <getopt.h>
```

```c
/******************************************************************************/

/******************************************************************************/
/*                                 CODE                                      */
/******************************************************************************/
int main(int argc, char** argv) {
  /*================= Initialize ================*/
  MPI_Init(&argc, &argv);
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /*======== Construct the operator using a CSR matrix ========*/
  CPLM_Mat_CSR_t A = CPLM_MatCSRNULL();
  int M, m;
  int* rowPos = NULL;
  int* colPos = NULL;
  int sizeRowPos, sizeColPos;
  // Read and partition the matrix
  preAlps_OperatorBuild(matrixFilename,MPI_COMM_WORLD);
  // Get the CSR structure of A
  preAlps_OperatorGetA(&A);
  // Get the sizes of A
  preAlps_OperatorGetSizes(&M,&m);
  // Get row partitioning of A
  preAlps_OperatorGetRowPosPtr(&rowPos,&sizeRowPos);
  // Get col partitioning induced by this row partitioning
  preAlps_OperatorGetColPosPtr(&colPos,&sizeColPos);

  /*======== Construct the preconditioner ========*/
  preAlps_BlockJacobiCreate(&A,rowPos,sizeRowPos,colPos,sizeColPos);

  /*============= Construct a normalized random rhs =============*/
  double* rhs = (double*) malloc(m*sizeof(double));
  // Set the seed of the random generator
  srand(0);
  double normb = 0.0;
  for (int i = 0; i < m; ++i) {
    rhs[i] = ((double) rand() / (double) RAND_MAX);
    normb += pow(rhs[i],2);
  }
  // Compute the norm of rhs and scale it accordingly
  MPI_Allreduce(MPI_IN_PLACE,&normb,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
  normb = sqrt(normb);
  for (int i = 1; i < m; ++i)
    rhs[i] /= normb;

  /*================ ECG solve ================*/
  preAlps_ECG_t ecg;
  // Set parameters
  ecg.comm = MPI_COMM_WORLD;
  ecg.globPbSize = M;
  ecg.locPbSize = m;
  ecg.maxIter = maxIter;
  ecg.enlFac = enlFac;
  ecg.tol = tol;
  ecg.ortho_alg = (ortho_alg == 0 ? ORTHODIR : ORTHOMIN);
  ecg.bs_red = (bs_red == 0 ? NO_BS_RED : ADAPT_BS);
  int rci_request = 0;
  int stop = 0;
  double* sol = NULL;
  sol = (double*) malloc(m*sizeof(double));
  // Allocate memory and initialize variables
  preAlps_ECGInitialize(&ecg,rhs,&rci_request);
  // Finish initialization
  preAlps_BlockJacobiApply(ecg.R,ecg.P);
  preAlps_BlockOperator(ecg.P,ecg.AP);
  // Main loop
  while (stop != 1) {
    preAlps_ECGIterate(&ecg,&rci_request);
    if (rci_request == 0) {
      preAlps_BlockOperator(ecg.P,ecg.AP);
    }
    else if (rci_request == 1) {
      preAlps_ECGStoppingCriterion(&ecg,&stop);
      if (stop == 1) break;
      if (ecg.ortho_alg == ORTHOMIN)
        preAlps_BlockJacobiApply(ecg.R,ecg.Z);
      else if (ecg.ortho_alg == ORTHODIR)
        preAlps_BlockJacobiApply(ecg.AP,ecg.Z);
    }
  }
  // Retrieve solution and free memory
  preAlps_ECGFinalize(&ecg,&sol);

  if (rank == 0) {
    preAlps_ECGPrint(&ecg,0);
  }

  /*================= Finalize =================*/

  // Free arrays
  if (rhs != NULL) free(rhs);
  if (sol != NULL) free(sol);
  preAlps_OperatorFree();

  CPLM_printTimer(NULL);
  MPI_Finalize();
  return 0;
}
```

4 LORASC

4.1 LORASC preconditioner

We briefly recall LORASC, a robust algebraic preconditioner of the form $M = (L + \tilde{D})\tilde{D}^{-1}(\tilde{D} + L^T)$ as presented in the [6]. The graph of the input matrix is first partitioned by using k-way partitioning with vertex separators into N disjoint domains and a separator Γ formed by the interface vertices connecting the N domains. Such a partitioning can be obtained by using existing software as METIS [12]. The permuted matrix has a block arrow structure, as presented in equation (4.3), in which the first N diagonal blocks correspond to the disjoint domains, while the last diagonal block $A_{\Gamma\Gamma}$ corresponds to the separator.

Consider a symmetric positive definite matrix A of size $n \times n$, which has a bordered block diagonal structure as in Equation (4.3). We refer in the following to LORASC preconditioner as M. The preconditioner M is defined by the following approximate factorization

$$\begin{aligned} M &= (L+\tilde{D})\tilde{D}^{-1}(\tilde{D}+L^T) \qquad (4.1) \\ &= \begin{pmatrix} A_{11} & & & \\ & \ddots & & \\ & & A_{NN} & \\ A_{\Gamma 1} & \cdots & A_{\Gamma N} & \tilde{S} \end{pmatrix} \begin{pmatrix} A_{11}^{-1} & & & \\ & \ddots & & \\ & & A_{NN}^{-1} & \\ & & & \tilde{S}^{-1} \end{pmatrix} \begin{pmatrix} A_{11} & & & A_{\Gamma T} \\ & \ddots & & \vdots \\ & & A_{NN} & A_{\Gamma N}^T \\ & & & \tilde{S} \end{pmatrix}, \end{aligned}$$

where $\tilde{D} = \text{Block-Diag}(A_{11}, A_{22}, ..., A_{NN}, \tilde{S})$, L is defined as

$$L = \begin{pmatrix} 0 & & & \\ & \ddots & & \\ & & 0 & \\ A_{\Gamma 1} & \cdots & A_{\Gamma N} & 0 \end{pmatrix}, \qquad (4.2)$$

and $\tilde{S}$ is an approximation of the Schur complement $S = A_{\Gamma\Gamma} - \sum_{j=1}^{N} A_{\Gamma j} A_{jj}^{-1}$.

Definition 4.1 (LORASC preconditioner)**.** *Let A be an $n \times n$ symmetric positive definite matrix with a bordered block diagonal structure,*

$$A = \begin{pmatrix} A_{11} & & & A_{\Gamma 1}^T \\ & \ddots & & \vdots \\ & & A_{NN} & A_{\Gamma N}^T \\ A_{\Gamma 1} & \cdots & A_{\Gamma N} & A_{\Gamma\Gamma} \end{pmatrix}. \qquad (4.3)$$

Let $S = A_{\Gamma\Gamma} - \sum_{j=1}^{N} A_{\Gamma j} A_{jj}^{-1} A_{\Gamma j}^T$. Given a tolerance τ, a lower bound $\varepsilon = \frac{1}{\tau}$ for the generalized eigenvalue problem $Su = \lambda A_{\Gamma\Gamma} u$, let $\lambda_1, \lambda_2, ..., \lambda_i$ be the generalized eigenvalues less than ε, i.e. for all $k \in \{1, ..., i\}$ then $\lambda_k < \varepsilon$, and let $v_1, v_2, ..., v_i$ be the corresponding $A_{\Gamma\Gamma}$-orthonormal generalized eigenvectors.

The LORASC preconditioner of A is defined as

$$M_{\text{LORASC}} := (L+\tilde{D})\tilde{D}^{-1}(\tilde{D}+L^T).$$

where $\tilde{D} =$ Block−Diag$(A_{11}, A_{22}, ..., A_{NN}, \tilde{S})$ and L is given by (4.2). The matrix $\tilde{S}$ is defined as

$$\tilde{S}^{-1} = A_{\Gamma\Gamma}^{-1} + E\Sigma E^T, \qquad (4.4)$$

where E, Σ are defined as

$$\begin{aligned} E &= (v_1 \quad v_2 \quad \dots \quad v_i), \qquad (4.5) \\ \Sigma &= \text{Diag}(\sigma_1, \sigma_2, \dots, \sigma_i), \quad \textit{with } \sigma_k = \frac{\varepsilon - \lambda_k}{\lambda_k}, \ k \in \{1, 2, ..., i\}. \qquad (4.6) \end{aligned}$$

4.2 Routines

LORASC preconditioner can be built separately and used in any sparse iterative solvers. The implemented routines are described as follows:

- `preAlps_LorascAlloc (preAlps_Lorasc_t **lorasc)` : creates an object of the type `preAlps_Lorasc_t`. The resulting object can be used by the end-user to replace the default parameters such as the `deflation_tolerance` (ε in definition 4.1).

- `preAlps_LorascBuild (preAlps_Lorasc_t *lorasc, CPLM_Mat_CSR_t *A, CPLM_Mat_CSR_t *locAP, MPI_Comm comm)` : constructs LORASC preconditioner from an input matrix A and stores all the required internal workspace in the object `lorasc`. First, it partitions and permutes the matrix A into a block arrow structure, then distributes it to each processor. After this distribution, each processor stores in the output matrix $locAP$ its block from a 1-D block row distribution of the permuted matrix A. Finally it constructs the preconditioner itself.

- `preAlps_LorascApply (preAlps_Lorasc_t *lorasc, double *x, double *y)` : applies LORASC preconditioner on a vector x and return the result in the vector y.

- `preAlps_LorascApplyMat (preAlps_Lorasc_t *lorasc, CPLM_Mat_Dense_t *A, CPLM_Mat_Dense_t *B)` : applies LORASC preconditioner on a dense matrix A, and returns the result in a dense matrix B. This routine does the same computation as `preAlps_LorascApply` routine with the difference that it applies the preconditioner on a dense matrix.

- `preAlps_LorascDestroy (preAlps_Lorasc_t **lorasc)` : frees the internal memory allocated by LORASC preconditioner and destroys `lorasc` object.

4.3 Example program

The following listing illustrates how LORASC can be used as a preconditioner in an example program for solving a sparse linear system. In this example, we use ECG as iterative solver with an enlarging factor of 1, which is similar to using the classical CG algorithm.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include <preAlps_cplm_matcsr.h>
#include "preAlps_utils.h"
#include "preAlps_preconditioner.h"
#include "preAlps_lorasc.h"
#include "preAlps_ecg.h"

int main(int argc, char** argv){

  int i, bsize, ierr, nbprocs, my_rank;
  char matrix_filename[150]="", rhs_filename[150]="";
  CPLM_Mat_CSR_t A = CPLM_MatCSRNULL(), locAP = CPLM_MatCSRNULL();
```

```c
double *x = NULL, *b = NULL;

/* Generic preconditioner object */
preAlps_preconditioner_t *precond = NULL;

/* Lorasc preconditioner */
preAlps_Lorasc_t *lorascA = NULL;

/* Start MPI*/
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nbprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Load the matrix on processor 0 */
if(my_rank==0){
  /* Get user parameters */
  for(i=1;i<argc-1;i+=2){
    if (strcmp(argv[i],"-mat") == 0) strcpy(matrix_filename,argv[i+1]);
    if (strcmp(argv[i],"-rhs") == 0) strcpy(rhs_filename,argv[i+1]);
  }

  /* Read the matrix in CSR format */
  CPLM_LoadMatrixMarket(matrix_filename, &A);

  /* Read the rhs */
  CPLM_DVectorLoad(rhs_filename, &b, &bsize, 0);
}

/* Memory allocation for LORASC preconditioner */
ierr = preAlps_LorascAlloc(&lorascA); preAlps_checkError(ierr);

/* Set LORASC parameters */
lorascA->deflation_tolerance = 1e-2; //eigenvalues deflation tolerance

/* Build the preconditioner and distribute the matrix according to the Block arrow
    structure */
ierr = preAlps_LorascBuild(lorascA, &A, &locAP, MPI_COMM_WORLD);
    preAlps_checkError(ierr);

/* Create a generic preconditioner object compatible with EcgSolver*/
preAlps_PreconditionerCreate(&precond, PREALPS_LORASC, (void *) lorascA);

/* Solve the system using ECGSolve*/
ECG_t ecg;                    /* Enlarge CG solver */
/* Set the solver parameters */
ecg.comm = MPI_COMM_WORLD;    /* MPI Communicator */
ecg.globPbSize = A.info.m;    /* Size of the global problem */
ecg.locPbSize = locAP.info.m; /* Size of the local problem */
ecg.maxIter = 10000;          /* Maximum number of iterations */
ecg.enlFac = 1;               /* Enlarging factor */
ecg.tol = 1e-8;               /* Tolerance of the method */
ecg.ortho_alg = ORTHOMIN;     /* Orthogonalization algorithm */


/* Call Ecg Solve */
```

```c
  /* ... */
  /* See listing in ECG section */
  /* ... */

  if (my_rank == 0)
    printf("=== ECG ===\n\titerations: %d\n\tnorm(res): %e\n",ecg.iter,ecg.res);

  /* Destroy Lorasc preconditioner */
  ierr = preAlps_LorascDestroy(&lorascA); preAlps_checkError(ierr);

  /* Destroy the generic preconditioner object*/
  preAlps_PreconditionerDestroy(&precond);

  /* Free memory*/
  if(b!=NULL) free(b);
  CPLM_MatCSRFree(&locAP);

  if(my_rank==0){
    CPLM_MatCSRFree(&A);
  }

  MPI_Finalize();
  return EXIT_SUCCESS;
}
```

5 Experiments

This section presents the performance that can be expected by using ECG and LORASC.

5.1 Enlarged Conjugate Gradient

5.1.1 Environment

Concerning ECG, the experiments were performed on a machine located at Umeå University as part of High Performance Computing Center North (HPC2N), called Kebnekaise. It is an heterogeneous machine formed by a mix of Intel Xeon E5-2690v4 with 2x14 cores (and E7-8860v4 for large memory computations), Nvidia K80 GPU and Intel Xeon Phi 7250 (Knight's Landing) with 68 cores. In our experiments, we use the so-called compute nodes, which means formed by Intel Xeon E5-2690v4. For a detailed description of the machine, we refer to the online documentation (https://www.hpc2n.umu.se/resources/hardware/kebnekaise).

5.1.2 Description of test matrices

We compare the performance of ECG and PETSc PCG on a set of matrices that are also used in [6, 1, 16] where they are described in more details. In this document, we present the results for one matrix in this set which is detailed in Table 2 where we present its size and its number of nonzeros.

The Ela matrices arise from the linear elasticity problem with Dirichlet and Neumann boundary conditions defined as follows

$$\mathrm{div}(\sigma(u)) + f = 0 \quad \text{on } \Omega \tag{5.1}$$

$$u = 0 \quad \text{on } \partial\Omega_D \tag{5.2}$$

$$\sigma(u) \cdot n = 0 \quad \text{on } \partial\Omega_N \tag{5.3}$$

where Ω is a unit square cube. The matrices Ela$N$ correspond to this equation discretized using a triangular mesh with $N \times 10 \times 10$ points on the corresponding vertices and P1 finite elements scheme. $\partial\Omega_D$ is the Dirichlet boundary, $\partial\Omega_N$ is the Neumann boundary, f is some body force, u is the unknown displacement field. $\sigma(.)$ is the Cauchy stress tensor given by Hooke's law: it can be expressed in terms of Young's Modulus E and Poisson's ratio ν. For a more detailed description of the problem see [15] and [6]. We consider discontinuous E and ν, $(E_1, \nu_1) = (2 \times 10^{11}, 0.25)$ and $(E_2, \nu_2) = (10^7, 0.45)$.

Table 2: Test matrix we use in our tests, its size, the number of nonzeros and the type of problem it is coming from.

	Size	Nonzeros	Problem
Ela400	145 563	4 907 997	3D Linear Elasticity P1 FE

In the experiments we use a block Jacobi preconditioner with as many blocks as processors. Before calling ECG, each processor factorizes the diagonal block of A corresponding to the local row panel that it owns. At each iteration of ECG, each processor performs a backward and forward solve locally in order to apply the preconditioner. Hence, the application does not need any communication.

5.1.3 Performance of ECG

We compile `preAlps` (and its dependencies) using Intel toolchain installed on the machine: `mpiicc` (based on `icc` version 17.0.1 20161005) and `MKL` version 2017.1.132.

We are using `PETSc` in order to compare our ECG implementation to `PETSc` PCG implementation. `PETSc` 3.7.6 is compiled using the following configuration,

```
./configure -with-cc=mpiicc --with-cxx=0 --with-fc=0 COPTFLAGS="-O3 -march=native
    -mtune=native" --with-debugging=0 -with-blas-lapack-dir=${MKLROOT}
    --with-mkl_pardiso-dir=${MKLROOT} --download-metis
```

In particular, `PETSc` is using `MKL-PARDISO` as exact solver for sparse matrices in the block Jacobi preconditioner. CPaLAMeM has been linked with `PETSc`, and the `METIS` library downloaded and installed by `PEtSc`.

When increasing the enlarging factor t the number of iterations decreases and the number of flop per iteration is increasing. In other words, if t is small the communication cost should dominate and if t is large the computational cost should dominate. There is no simple way to choose a priori the optimal enlarging factor t. In Figure 1 we plot the

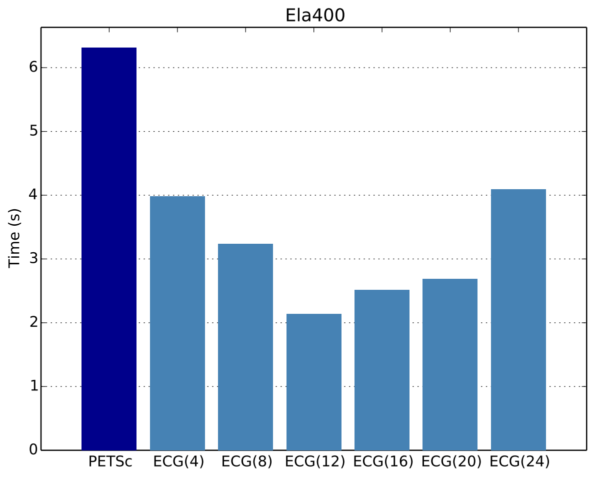

Figure 1: Runtimes for Ela400 with a tolerance of 10^{-5} on 48 processors.

runtime as a function of t when solving Ela400 on 48 processors. Overall, ECG performs better than PETSc PCG, both with the same preconditioner. Indeed, ECG runtime is 2 to 3 times faster than PCG runtime. On the other hand, experimental results confirm that there exists a good choice of t. Indeed, the runtime is decreasing until $t = 12$ but then it increases. It is important not to choose an enlarging factor too small, for example, ECG(4) is two times slower than ECG(12). On the other side, increasing t is not useful up to a certain number of right-hand sides, for example, ECG(12) is nearly two times faster than ECG(24).

In Figure 2 we summarize the results obtained when performing a strong scalability study on Ela400 with PETSc PCG and ECG(12). We start with 24 processors and double the number of processors until 192. First, ECG(12) scaling is sub-linear and PETSc PCG scales a little more than linearly. However, this test case is relatively small and thus the runtime with 192 processors is very small (around 1 second). When using 8 times more processors (from 24 to 192) PETSc PCG becomes around 8.9 times faster and ECG(12) around 6.5 times faster. However, ECG(12) remains around 2 times faster than PETSc PCG. With 192 processors ECG(12) is 86% faster and with 48 processors it is 3 times faster.

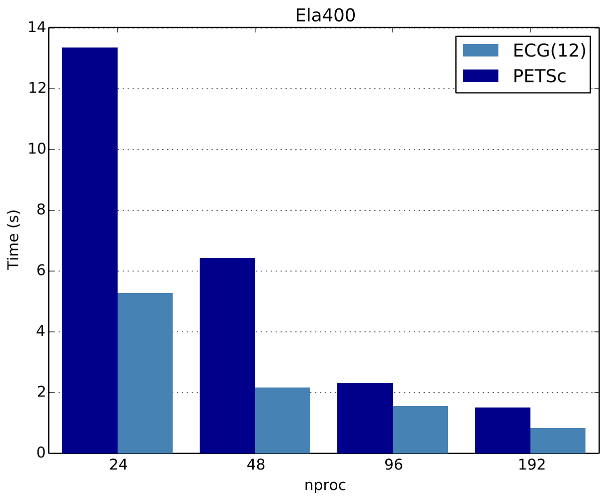

Figure 2: Strong scaling study for Ela400 with a tolerance of 10^{-5} and an enlarging factor of 12.

Table 3: Runtimes comparison between ECG(12) and PETSc PCG for Ela400 with a tolerance of 10^{-5} when increasing the number of processors.

	nprocs	runtimes (s)
ECG(12)	24	5.28
	48	2.16
	96	1.55
	192	0.83
PETSc PCG	24	13.35
	48	6.43
	96	2.31
	192	1.51

5.2 LORASC

5.2.1 Environment

The experiments of LORASC were performed on a parallel machine formed by 28 compute nodes. Each node is equipped with a 24 cores socket based on Intel Xeon E5-2670 (Sandy Bridge), each core has a frequency of 2.6 GHz. We assign one MPI task per core. As described in [6], the construction of LORASC is performed in parallel and requires computing the Cholesky factorization of the diagonal blocks of A and solving a generalized eigenvalue problem. When N processors are used, first the input matrix A is permuted by using METIS into a matrix with a block arrow structure as in equation (4.3). The Cholesky factorization of each domain is computed by one of the N processors using PARDISO from MKL's scientific library. The Cholesky decomposition of $A_{\Gamma\Gamma}$ is computed in parallel by a subset of the N processors using MUMPS 5.0.2 [2]. In our experiments, the number of processors used for $A_{\Gamma\Gamma}$ is determined by the minimum of N, and $size(A_{\Gamma\Gamma})/2000$. Hence each processor will have at least 2000 rows of $A_{\Gamma\Gamma}$. We observed that increasing the number of processors used for the factorization of $A_{\Gamma\Gamma}$ over this threshold leads to more communication between processors, and no performance gain. Once the factorization of the diagonal blocks is computed, the generalized eigenvalue problem is solved by using the parallel version of ARPACK, PARPACK 2.1 [14]. We provide to PARPACK a parallel routine that applies S to a vector and $A_{\Gamma\Gamma}^{-1}$ to a vector.

Once LORASC is built, the linear system is solved using ECG solver with an enlarging factor of 1 which corresponds to the classical CG.

5.2.2 Performance of LORASC

We compare the performance of LORASC with block Jacobi from PETSc. On N processors, block Jacobi preconditioner uses the Cholesky factorization of N diagonal blocks of the matrix A permuted as following. The vertices corresponding to the separator Γ are partitioned into N sets. The rows and columns of the input matrix A are permuted such that the j-th set of vertices of the separator Γ are ordered after the vertices of the j-th domain. Processor j owns the rows corresponding to these vertices. The residual tolerance of the solver is fixed to 10^{-6} for all the preconditioners. In the following tables, *No prec* refers to the version of CG without preconditioning.

Figure 3 presents the performance of LORASC on matrices arising from the discretization of the 3D linear elasticity problem. The dimension of the matrices increases from $n = 4719$ to $n = 72963$. In the results presented in Figure 3, the number of processors, and hence the number of domains, is fixed to 16. For all the results, we include both the time to construct the preconditioner and the time to solve the system. The runtime of LORASC increases from 0.2 to 3 when the matrix size increases from 4719 to 72963. LORASC is significantly faster than Block Jacobi, up to a factor of 6 for the matrix of dimension $n = 72963$. In these results, all the eigenvalues smaller than $\varepsilon = 10^{-2}$ are deflated in LORASC. For example, for $n = 9438$, 8 eigenvalues are deflated, and for $n = 72963$, 23 eigenvalues are deflated.

Figure 4 presents the performance of LORASC and Block Jacobi for a medium size matrix with $n = 145563$ and 4907997 nonzero elements. The number of processors increases from 16 to 256. The number of domains used in LORASC and the number of diagonal

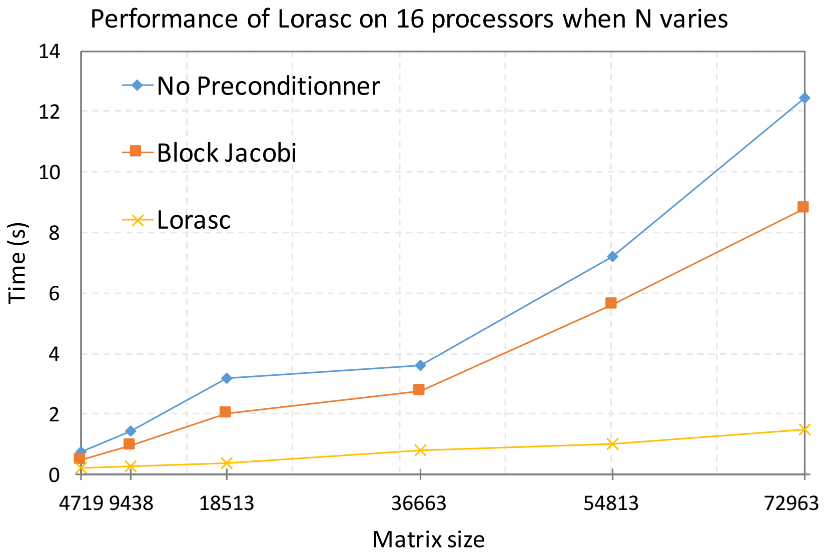

Figure 3: Performance of LORASC on 16 processors on 3D linear elasticity problems when n varies from 4719 to 72963.

blocks used in Block Jacobi are equal to the number of processors. In addition, table 4 displays for both preconditioners the number of iterations (`Iter`), the time to construct the preconditioner (`Prec`), the time to solve (`Solve`), and the total time of the iterative process (`Total` $=$ `Prec` $+$ `Solve`). Similarly in Figure 4 the runtime is broken down into the time to construct the preconditioner (`Preconditioner`), and the solve time (`Solve`). One of the parameters that we need to pass to PARPACK is the number of eigenvalues and eigenvectors that need to be computed. In this experiment the number of eigenvalues computed by PARPACK is fixed to 150, and the eigenvalues smaller than the threshold $\varepsilon = 10^{-2}$ are deflated. In the future, we will work on estimating the number of eigenvalues smaller than the threshold ϵ, and thus have an estimate of the number of eigenvalues that needs to be computed by PARPACK and deflated by LORASC.

The results in Figure 4 show that LORASC is faster than Block Jacobi when up to 128 processors are used. It is 7 times faster than Block Jacobi on 16 processors. However, the time required to compute LORASC increases when the number of domains increases. More than 80% of the time is spent in solving the generalized eigenvalue problem with PARPACK. Since the size of the interface Γ becomes larger when the number of domains increases, the construction of LORASC requires solving a larger generalized eigenvalue problem, and this becomes more expensive, even if we use a larger number of processors to solve it. The time to build Block Jacobi is very small, but since the number of iterations to converge grows when the number of processors increases, the solve time is important. On 256 processors, even if Block Jacobi requires a factor of 64 more iterations to converge than LORASC, the time to apply Block Jacobi is almost equal with the time to construct LORASC.

These results show that for medium size problems and for a small to medium number of processors, LORASC is an efficient preconditioner. When the number of processors

N	Block Jacobi				LORASC			
	`Iter`	`Prec`	Solve	Total	`Iter`	`Prec`	`Solve`	`Total`
16	4058	3.5E-4	132.1	132.1	102	15.4	2.7	18.1
32	6020	3.5E-4	91.8	91.8	131	12.6	1.4	14.1
64	9096	3.7E-4	64.3	64.3	184	13.3	1.0	14.3
128	13321	3.7E-4	41.3	41.3	227	22.4	1.2	23.6
256	15573	3.7E-4	37.3	37.3	243	36.3	1.8	38.1

Table 4: Parallel runtime of Block Jacobi and LORASC on N processors for a matrix arising from the discretization of a 3D linear elasticity problem. The matrix has dimension $n = 145563$ and 4907997 nonzeros. `Iter` denotes the number of iterations until convergence of preconditioned CG, `Prec` denotes the time to construct the preconditioner and `Solve` denotes the solve time.

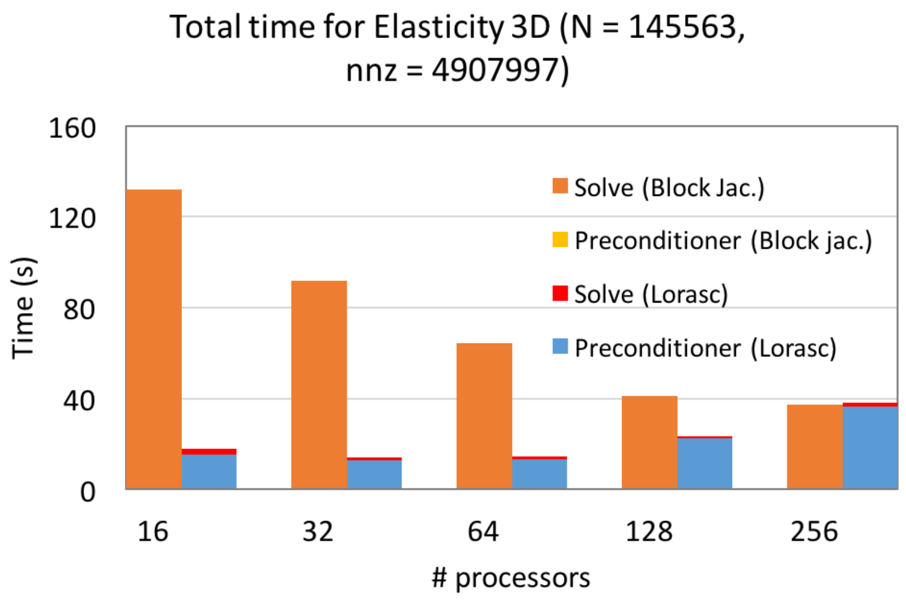

Figure 4: Strong scaling performance of LORASC on elasticity 3D problem of size $n = 145563$ with 4907997 nonzeros.

increases and the interface Γ becomes too large, the resolution of the generalized eigenvalue problem becomes expensive. However, if multiple linear systems with the same matrix A need to be solved, and the right hand sides are not available simultaneously, then LORASC remains an interesting option since the solve phase is very fast.

6 Routines

```
/*
 * Read a sparse matrix from file (mtx or petsc binary) then partition it
 * using METIS K-Way algorithm into the number of processor in the communicator
 * comm and distribute it among those processors.
 * input: matrixFilename: MatrixMarket or PETSc binary file
 *        comm          : MPI communicator
 */
int  preAlps_OperatorBuild(const char* matrixFilename, MPI_Comm comm);

/*
 * Release memory allocated during preAlp_OperatorBuild.
 */
void preAlps_OperatorFree();

/*
 * Print informations about the operator.
 */
void preAlps_OperatorPrint(int rank);

/*
 * Apply the operator to a group of vector X and put the result into AX.
 * input : X : CPLM_Mat_Dense_t
 * output: AX: CPLM_Mat_Dense_t
 */
int  preAlps_OperatorGetSizes(int* M, int* m);

/*
 * Return the local part of the operator as a row panel CSR matrix.
 */
int  preAlps_BlockOperator(CPLM_Mat_Dense_t* X, CPLM_Mat_Dense_t* AX);

/*
 * Return the global size and local size of the operator A.
 */
int  preAlps_OperatorGetA(CPLM_Mat_CSR_t* A);

/*
 * Return a vector containing the global row index corresponding to the beginning
 * of each row panel.
 */
int  preAlps_OperatorGetRowPosPtr(int** rowPos, int* sizeRowPos);

/*
 * Return a vector containing the global col index corresponding to the beginning
 * of each local column block (column block i corresponds to values at the interface
 * between processor i and my_rank)
 */
int  preAlps_OperatorGetColPosPtr(int** colPos, int* sizeColPos);

/*
 * Return a vector containing the indexes of the neighbor processors.
 */
int  preAlps_OperatorGetDepPtr(int** dep, int* sizeDep);

/*
 * Factorize the diagonal block associated to the local row panel owned locally
 * input: A         : a CSR matrix CPLM_Mat_CSR_t
 *        rowPos    : the vector returned by preAlps_OperatorGetRowPosPtr
 *        sizeRowPos: the size of rowPos
 *        colPos    : the vector returned by preAlps_OperatorGetColPosPtr
 *        sizeColPos: the size of colPos
 */
int preAlps_BlockJacobiCreate(CPLM_Mat_CSR_t* A,
                              int* rowPos,
                              int sizeRowPos,
                              int* colPos,
                              int sizeColPos);

/*
 * Solve Mx = rhs and put the result into rhs.
 * Internal usage only.
 */
int preAlps_BlockJacobiInitialize(CPLM_DVector_t* rhs);

/*
 * Solve M B_out = A_in with M a block Jacobi preconditioner.
 */
int preAlps_BlockJacobiApply(CPLM_Mat_Dense_t* A_in, CPLM_Mat_Dense_t* B_out);

/*
 * Free the memory allocated during the construction of the preconditioner.
 */
void preAlps_BlockJacobiFree();
```

7 Conclusion

In this document we described a first version of preAlps library devoted to solving large sparse linear systems of equations by using preconditioned iterative methods. In its current version, the library implements ECG iterative solver and LORASC preconditioner.

In the context of enlarged Krylov methods, our future work will focus on implementing the dynamic reduction of the number of vectors added during each iteration of ECG such that the computational cost per iteration is reduced while the same convergence rate is maintained. This technique is described in [7]. In the context of LORASC, our performance results show that its application during the iterative process is fast and scalable, but on larger number of processors, its construction becomes expensive since it requires solving a generalized eigenvalue problem of a larger size. Our future work will focus on approaches that allow to speed up the resolution of the generalized eigenvalue problem, which for the moment increases when the number of processors increases.

8 Acknowledgments

This project is funded from the European Union's Horizon 2020 research and innovation programme under the NLAFET grant agreement No 671633.

References

[1] Y. Achdou and F. Nataf. Low frequency tangential filtering decomposition. *Numerical Linear Algebra with Applications*, 14:129–147, 2007.

[2] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM J. Matrix Anal. Appl.*, 23(1):15–41, 2001.

[3] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' guide*, volume 9. SIAM, 1999.

[4] Satish Balay, Shrirang Abhyankar, M Adams, Peter Brune, Kris Buschelman, L Dalcin, W Gropp, Barry Smith, D Karpeyev, Dinesh Kaushik, et al. Petsc users manual revision 3.7. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2016.

[5] L. Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. *ScaLAPACK users' guide*, volume 4. SIAM, 1997.

[6] L. Grigori, F. Nataf, and S. Youssef. Robust algebraic schur complement based on low rank correction. Technical report, ALPINES-INRIA, Paris-Rocquencourt, 6 2014.

[7] L. Grigori and O. Tissot. Reducing the communication and computational costs of enlarged krylov subspaces conjugate gradient. Research Report RR-9023, Feb 2017.

[8] Laura Grigori, Sophie Moufawad, and Frederic Nataf. Enlarged Krylov subspace conjugate gradient methods for reducing communication. *SIAM J. Matrix Anal. Appl.*, 2016.

[9] Martin H. Gutknecht. Block Krylov space methods for linear systems with multiple right-hand sides: an introduction. 2006.

[10] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–436, December 1952.

[11] A.Kalhan J. Dongarra, V.Eijkhout. Reverse communication interface for linear algebra templates for iterative methods. Technical report, May 1995.

[12] George Karypis and Vipin Kumar. METIS –unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[13] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. *Version 1.0, Dept. of Computer Science, University of Minnesota*, 1997.

[14] R. B. Lehoucq, D. C. Sorensen, and C. Yang. Arpack User's Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods, 1997.

[15] F. Nataf, F. Hecht, P. Jolivet, and C. PrudâĂŹHomme. Scalable domain decomposition preconditioners for heterogeneous elliptic problems. *SC13, (Denver, Colorado, United States)*, 2013.

[16] Q. Niu, L. Grigori, P. Kumar, and F. Nataf. Modified tangential frequency filtering decomposition and its fourier analysis. *Numerische Mathematik*, 116:123–148, 2010.

[17] D. P. OâĂŹLeary. The block conjugate gradient algorithm and related methods. *Linear Algebra and Its Applications*, 29:293–322, 1980.

[18] A. Sluis and H.A. Vorst. The rate of convergence of conjugate gradients. *Numerische Mathematik*, 48(5):543–560, 1986.

[19] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer, 2014.