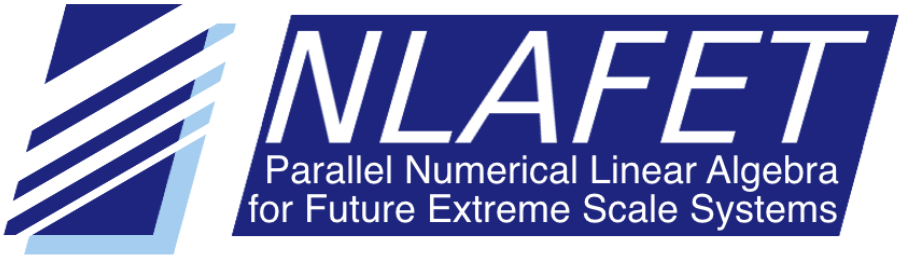


H2020–FETHPC–2014: GA 671633

D4.1

Computational Kernels for Preconditioned Iterative Methods Prototype software, phase 1

October 2016

Document information

Scheduled delivery	2016-11-01
Actual delivery	(2016-11-01)
Version	1
Responsible partner	INRIA

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2016-10-17	Inria members	Draft	0.1	Initial version of document produced
2016-10-31	Inria members	Final document	1	Final version based on comments from internal reviewers

Author(s)

Alan Ayala, Simplice Donfack, Laura Grigori, INRIA

Internal reviewers

Carl Christian Kjelgaard Mikkelsen, UMU
Florent Lopez, STFC

Contributors

Copyright

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

List of Figures

List of Tables

1 Executive Summary

One of the main challenges in high performance computing is the increased cost of communication. Several works show that there is an exponentially increasing gap between the time required to compute one floating point operation by one processor and the time required to communicate data between different levels of the memory hierarchy or between different processors [16]. Because of this gap, many algorithms do not scale to large numbers of processors. In order to address this difficulty, we have developed a new class of algorithms that we refer to as communication avoiding algorithms. These algorithms drastically reduce the communication cost with respect to classic algorithms, or even minimize it when possible.

The goal of WP4 is to address the scalability problem of existing iterative methods by focusing on two main aspects: the reformulation of Krylov subspace methods to allow a drastic reduction in the number of global communication instances with respect to classic formulations, and the design of communication avoiding preconditioners to accelerate the convergence of iterative methods.

In this deliverable, we focus on computational kernels that are requested by our communication avoiding iterative methods. We propose highly efficient kernels such as sparse matrix-matrix product and sparse low rank approximation. Our implementation leads to a library that can be used by the other work packages, or that can be incorporated in other well-known libraries.

2 Introduction

Solving large sparse linear systems of equations $Ax = b$ is essential in many scientific and engineering applications. Direct methods can be used to solve those systems and they achieve a high quality solution in a fixed number of operations. Hence these methods are known to be very robust and normally they are limited only by the available memory. If a direct method is not suitable (either it is too slow or requires too much memory), the alternative is to use an iterative method. Preconditioners are very often necessary to accelerate the convergence of iterative methods. The focus of WP4 is on preconditioned iterative methods based on Krylov subspace solvers. Each iteration of a typical Krylov subspace solver involves matrix-vector multiplications and several dot products. These dot products related to the orthogonalization of the Krylov subspace require collective communication among all processors. This collective communication does not scale to very large number of processors, and thus it is a main bottleneck in the scalability of Krylov subspace methods.

To increase the scalability of Krylov subspace solvers, our research in NLAFET focuses on enlarged Krylov subspace methods [18], 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 classic subspace. The enlarged Krylov projection subspace methods lead to faster convergence in terms of iterations and parallelizable algorithms with less communication, with respect to Krylov methods. In addition, multiplying a matrix with multiple vectors at once may lead to a better resource

utilization.

The goal of this document is to describe the computational kernels that we have developed during the first year of NLAFET. These kernels are the multiplication of a sparse matrix with a set of vectors (we refer to this operation as sparse matrix matrix product) and the computation of a low rank approximation of a sparse matrix. These two kernels are building blocks of the preconditioned enlarged Krylov subspace solvers that we develop in NLAFET. The prototype codes that are part of this deliverable will continue to be optimized in the following year of this project.

In summary, we describe a library that provides a set of linear algebra routines and optimized kernels for iterative methods, that we refer to as `PreAlps`. The library provides a routine for computing the sparse matrix matrix product and routines for computing a low rank approximation of a sparse matrix based on communication avoiding sparse QR and CUR factorization kernels. The library also provides several sequential and parallel routines that can be used to perform advanced operations on matrices stored using CSR format. Most of the parallel routines assume that the matrix is already distributed among processors by using a uni-dimensional (1D) block rows distribution. However, we provide routines for creating, converting, and distributing a CSR matrix among processors.

3 Using PreAlps library

3.1 Purpose

The `PreAlps` library is a prototype software developed as the first part of the work supported by the NLAFET project. The second part will be devoted to the development and the addition of enlarged Krylov solvers and new preconditioners that use the routines implemented so far in `PreAlps`. In this report, we present sparse matrix matrix product and sparse communication avoiding low rank approximation kernels, as well as sequential and parallel utility routines implemented in the library.

3.2 Availability of the software

The software is available on the NLAFET repository in the folder

`https://github.com/NLAFET/preAlps`.

This is a prototype software that will be improved over the following months and it is not yet publicly available. The next version of the software will be made publicly available.

However, the software can be made available upon request. Anyone interested in downloading and testing the software should send an email to `Laura.Grigori@inria.fr`.

3.3 Installation

`PreAlps` has been developed with the aim of being simple to use and flexible, and thus we have reduced as much as possible the number of external libraries used. However, our parallel functions rely on several highly optimized sequential codes. Our library uses routines from `SuiteSparse` [11] such as local sparse QR factorizations, METIS for graph

partitioning [24], and it calls BLAS routines from LAPACK [1] or MKL [30] for a local dense QR factorization. Several graph or hypergraph partitioning tools supported by the library such as ParMETIS [25] and PaToH [6] can be installed separately and linked with the library.

1. Download SuiteSparse from http://faculty.cse.tamu.edu/davis/suitesparse.html.
2. Install Suitsparse following the instructions in its Readme file, generally it is enough to type make in the main directory. To customize the installation, edit the configuration file SuiteSparse_config/SuiteSparse_config.mk where you can set a compiler, MPI version, etc.
3. At the root of the preAlps folder, edit the make.inc file setting up the appropriate path for MKL or LAPACK, and SuiteSparse.
4. Next, type make to create the library lib/libPreAlps.a and the test programs.
5. Now, functions from the lib PreAlps can be called by a program, just including their header file. See section 7 for the list of callable functions. We have created a test directory with prototype programs.

3.4 Input data formats

Most of the routines in PreAlps require a matrix stored into a compressed sparse row format (CSR). The test programs read matrices from stdin in MatrixMarket format [5] by using the routine preAlps_matrix_readmm_csr.

In this document we present blocks of code to illustrate how the routines are implemented or called in C language. The following example shows how to read a matrix from a matrix market file and how to print it.

```c
/* Including headers */
#include "preAlps_matrix.h"

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

  int m, n, nnz, *xa, *ja;
  double *a;
  char matrixName[]="cage4.mtx";

  /* Reading the matrix file */
  preAlps_matrix_readmm_csr(matrixName, &m, &n, &nnz, &xa, &ja, &a);
  preAlps_matrix_print(MATRIX_CSR, m, xa, ja, a, "CSR example matrix");
  ...
```

In next sections, we assume that the loading (and the distribution) of the matrix has been already performed.

4 Sparse Matrix-Matrix product

4.1 spMSV routine

In this section, we present the routine spMSV (sparse matrix-set of vectors product) which is used to compute the product of two sparse matrices A and B, where A is sparse and B is formed by a set of vectors.

spMSV will be used by the enlarged Krylov subspace solvers that we develop in NLAFET [18]. As such, we take into account the specific structure of the matrix B that arises in those Krylov subspace methods. Figure 1 presents an example of such a matrix. The matrix A on the left of the figure is a sparse matrix coming from the University of Florida sparse matrix collection [13], referred to as LFAT5, while the matrix B on the right is formed by a set of structured vectors.

It is important to note that spMSV can also be used to compute a parallel sparse matrix-vector multiplication or a parallel sparse matrix-dense matrix multiplication as those two operations are instances of our algorithm when the matrix B is formed by only one column or a set of dense vectors respectively.

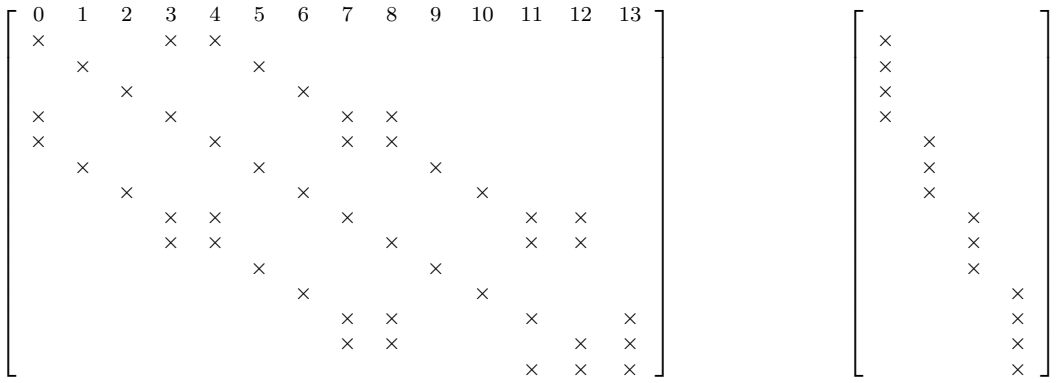


Figure 1: LFAT5 matrix from University of Florida sparse matrix collection. The matrix on the left is sparse, while the matrix on the right is sparse but structured (as arising from enlarged Krylov subspace methods).

We start by permuting the matrix A by using a graph or a hypergraph partitioning tool to obtain a matrix whose structure is displayed on the left of Figure 2. It has been shown in [31] that graph partitioning can be used to reduce the communication volume in sparse matrix-vector product while keeping good load balance between processors. In [7], the authors show that hypergraphs are more suitable to model the communication volume for non-symmetric matrices. spMSV requires the matrices A and B to be partitioned and distributed among processors before calling the routine. To do so, the final user could use his own partitioning tool to decompose the matrix A, or use our wrapper interface to partition the matrix using well-known partitioning tools such as METIS [24], ParMETIS [25], and PaToH [6]. In the remaining of this section we consider that the matrix A has been permuted to obtain the structure displayed on the left of Figure 2. The matrix B is formed by the set of vectors arising from enlarged Krylov subspace methods [18] and has the specific structure presented on the right of Figure 2.

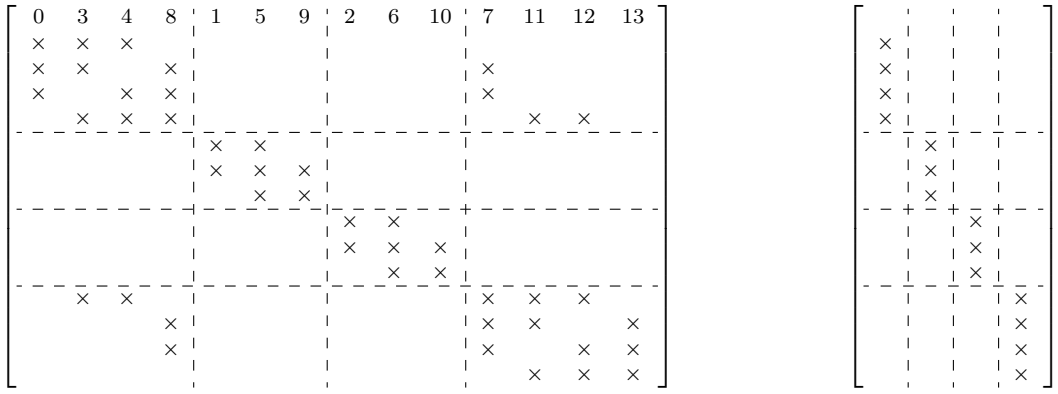


Figure 2: LFAT5 matrix partitioned in 4 parts using K-way partitioning.

Algorithm 1 presents spMSV for computing the product of two matrices A and B. We describe in the following the different steps of the algorithm.

Communication pattern (Lines 1 and 2): determines the communication that will be performed during the multiplication. This step allows each processor to determine its neighbors, that is the other processors with which it will exchange data. From the decomposition obtained in Figure 2, we build AS and BS, two dense matrices of size $P \times P$ and $P \times T$ respectively, where P is the number of processors and T is the number of block columns of the matrix B. As the matrix A is initially distributed in block rows, each processor first creates a local dense vector AS_local of size P and fills it as follows: $AS\_local[J] = 1$ if the block J of its local matrix A contains at least one non-zero element, and $AS\_local[J] = 0$ if the block J does not contain any non-zero element. Then a global collective operation is performed among all the processors to gather each AS_local in order to compute the global dense matrix AS. The same approach is used to build a global dense BS from the matrix B.

AS and BS represent the communication pattern of spMSV because they indicate which processor holds a block in a block row of A that will be multiplied by a block in a block column of B. As illustrated in Figure 3 using 4 processors, blue and green squares represent the blocks of the initial matrix A and B that contain at least one non-zero element, while gray squares represent empty blocks. The multiplication of each non-zero row block $A(I, K)$ of A with a column block $B(K, J)$ of B requires a communication when these blocks do not belong to the same processor. As illustrated in the same figure, green squares indicate blocks that cause communication. For example, the last row block of A held by processor $P0$ will be multiplied by the last column block of B held by processor $P3$, which means the former block should be sent to processor $P0$.

Asynchronous communication (Line 3 to 25): Asynchronously sends and receives blocks of the matrices needed for the matrix multiplication from the neigbors. We use asynchronous requests in MPI to send and receive data, so we overlap communication with computation. For performance reasons, all the processors first initiate asynchronous receives of the blocks they need as shown in line 11 of the algorithm, then proceed to the asynchronous sends of blocks required by the other processors as shown in line 22.

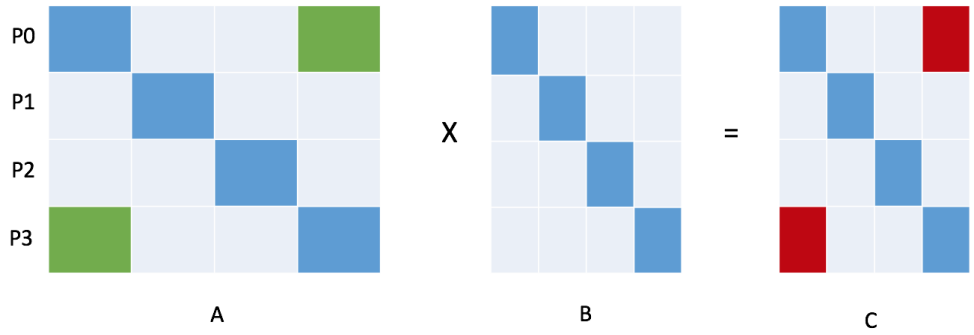


Figure 3: Communication pattern used for the sparse matrix-matrix product. Blue squares in A represent blocks that can be computed locally, while green squares represent blocks that require communication to be computed. Processor $P0$ needs to communicate with processor $P3$ in order to compute the last block column of its local submatrix. For matrix C, red squares represent the new blocks created after the multiplication.

Structure prediction (Line 3 to 34): performs a preprocessing step to predict the structure of the matrix C. This could be seen as performing a "symbolic sparse matrix multiplication". To do so, each processor needs to compute the product of the two dense matrices AS and BS. As the number of processors is usually small compared to the global matrix size, the cost of the preprocessing step is relatively small. We start by supposing that the matrix C has the same sparsity structure as the matrix B, and as new entries are added during the preprocessing step, we increase the amount the memory required for the matrix C. As illustrated in Figure 3, these new entries are represented by red squares. This allows to build the block structure CS of the resulting matrix C and to allocate the required memory using the routine *createMatrixFromStructure*.

Local matrix-matrix product(Line 35 to 50): performs a product of any non-zero blocks $A(I, K)$ and $B(K, J)$, and sums the result with the resulting block $C(I, J)$. First, each processor computes its blocks of C that do not require any communication. Then it waits for incoming blocks, as presented in line 45 of the algorithm. For each block of B received, it performs the multiplication and it sums the result with the corresponding block of C.

4.2 Implemented routines

The implementation of spMSV leads to several routines in `preAlps` that can be summarized as follows:
Matrix loading and distribution: provides a routine to load a matrix from matrix market file and distribute it to other processors using a uni-dimensional (1D) block rows distribution.
Matrix partitioning: provides a wrapper interface to partition the matrix sequentially

or in parallel, using a graph or a hypergraph to represent its structure. For the sequential case, the input matrix is partitioned on a single processor using METIS [24] or PATOH [6], then the matrix is permuted to group together the rows and column belonging to the same domain, and finally the resulting matrix is redistributed to all the processors. The matrix can also be partitioned in parallel using ParMETIS [25].
Communication pattern: provides a set of routines to determine the communication pattern of the matrices. This allows each processor to determine with which processor it exchanges data during a specific operation.
Parallel matrix matrix product: Provides a routine to perform a local sparse matrix-matrix product.

Algorithm 1 spMSV (input:Matrix A, input:Matrix B, output:Matrix C)

```
1: AS = createBlockStruct(A);
2: BS = createBlockStruct(B);
3: /*Create an array for the structure prediction of C*/
4: CS = zeros(T)
5: /* Initiate the reception of the block I need */
6: Nrecv = 0;
7: I = my_rank;
8: for J = 0 to P − 1 do
9:     for K = 0 to P − 1 do
10:        if I! = K && AS(I, K)! = 0 && BS(K, J)! = 0 then
11:            Request Block B(K,J) from processor K
12:            CS[J] = 1;
13:            Nrecv = Nrecv + 1;
14:        end if
15:    end for
16: end for
17: /* Initiate the sending of the block the other processors need */
18: K = my_rank;
19: for J = 0 to P − 1 do
20:    for I = 0 to P − 1 do
21:        if I! = K && AS(I, K)! = 0 && BS(K, J)! = 0 then
22:            Send Block B(K,J) from processor I
23:        end if
24:    end for
25: end for
26: /* Finalize the structure prediction of C */
27: I = my_rank;
28: for J = 0 to P − 1 do
29:    K = my_rank;
30:    if I! = K && AS(I, K)! = 0 && BS(K, J)! = 0 then
31:        CS[J] = 1;
32:    end if
33: end for
34: C = createMatrixFromStructure(CS);
35: /* Compute product using local blocks */
36: I = my_rank;
37: for J = 0 to P − 1 do
38:    K = my_rank;
39:    if I! = K && AS(I, K)! = 0 && BS(K, J)! = 0 then
40:        /* compute c(I,J) += A(I,K)*B(K,J) */
41:        C(I,J) += subMatrix_Product(A(K,J), B(K,J));
42:    end if
43: end for
44: /* Receive blocks and update the results */
45: while Nrecv > 0 do
46:    Wait a block from any processor
47:    Determine the coordinates J, and K from the Message TAG
48:    /* compute c(I,J) += A(I,K)*B(K,J) */
49:    C(I,J) += subMatrix_Product(A(K,J), B(K,J));
50: end while
51: Check all sending operations are completed
52: Free(AS, BS, CS);
```

4.3 Example program

The following listing is an abbreviated version of the sparse matrix-matrix multiplication routine. It illustrates how spMSV can be used in an example program.

```c
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include "preAlps_matrix.h"
#include "preAlps_matrix_mp.h"
#include "spMSV.h"

void main(int argc, char *argv[]) {
  MPI_Comm comm;
  char matrix_filename[150]="";
  int m, n, nnz, matrixDim[3];
  int mloc, nzloc, nloc, rloc, nparts, b_nparts;
  int *xa = NULL, *ja = NULL, *xb = NULL, *jb = NULL, *xc = NULL, *jc = NULL;
  double *a = NULL, *b = NULL, *c = NULL, *A, *v;
  int A_size, *ncounts=NULL, *part, *b_ncolcounts = NULL;
  int i, ds, nrhs = -1, nbprocs, my_rank, root = 0;
  int spmsv_options[3] = {0, 0, 0};

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

  /* 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],"-nrhs") == 0) nrhs = atoi(argv[i+1]);
  }

  /* Read the matrix file on processor 0 in csr format */
  if(my_rank==0){
    preAlps_matrix_readmm_csr(matrix_filename, &m, &n, &nnz, &xa, &ja, &a);
    /* Prepare the matrix dimensions for the broadcast */
    matrixDim[0] = m; matrixDim[1] = n; matrixDim[2] = nnz;
  }

  /* Broadcast the global matrix dimension among all processors */
  MPI_Bcast(&matrixDim, 3, MPI_INT, root, comm);
  m   = matrixDim[0]; n = matrixDim[1]; nnz = matrixDim[2];

  /* Initialize the vector v */
  v   = (double *) malloc(m*sizeof(double));
  if(my_rank==0) for(i=0;i<m;i++) v[i] = i*1.0;

  /* Set the number of partitions to create. */
  nparts = nbprocs;
  /* Partition the matrix */
  part = (int *) malloc((m*sizeof(int)));
  if(my_rank==root){
    preAlps_matrix_partition_sequential(m, xa, ja, nparts, part);
  }
  /* Broadcast the partitioning array to all processors */
  MPI_Bcast(part, m, MPI_INT, root, comm);
```

```
  /* Redistribute the matrix according to the global partitioning array 'part' */
  preAlps_matrix_kpart_redistribute(comm, m, n, &xa, &ja, &a, &mloc, part);

  /* Compute the number of columns in each block column */
  ncounts  = (int *) malloc(nbprocs*sizeof(int));
  for(i=0;i<nparts;i++) ncounts[i] = 0;
  for(i=0;i<m;i++) ncounts[part[i]]++;
  /* Determine the dimension of the local matrix */
  mloc = ncounts[my_rank]; nloc = n; nzloc = xa[mloc]; rloc = nrhs;
  /* Number of block columns to create (assume one column per part) */
  b_nparts = rloc;

  /* Allocate memory for the matrix B */
  xb   = (int *) malloc((mloc+1)*sizeof(int));
  jb   = (int *) malloc(nzloc*sizeof(int));
  b    = (double *) malloc(nzloc*sizeof(double));

  /* Distribute v as Block Diagonal CSR matrix (one column per partition) */
  preAlps_matrix_vshift_distribute(comm, mloc, ncounts, b_nparts, v, xb, &jb, &b);
  /* Set the number of columns for each block column of B*/
  b_ncolcounts  = (int *) malloc(b_nparts*sizeof(int));
  for(i=0;i<b_nparts;i++) b_ncolcounts[i] = 1;

  /* Create a sparse block struct of A(nbprocs x nbprocs) */
  A_size = nbprocs*nbprocs;
  A    = (int *) malloc(A_size*sizeof(int));
  preAlps_matrix_createBlockStruct(comm, MATRIX_CSR, mloc, nloc, xa, ja, nbprocs,
      ncounts, A);
  /* Locally convert B from CSR to CSC */
  preAlps_matrix_convert_csr_to_csc(mloc, rloc, &xb, jb, b);

  /* Compute the matrix-matrix product */
  ds = preAlps_spMSV(comm, mloc, nloc, rloc, xa, ja, a, xb, jb, b, nparts, ncounts,
            b_nparts, b_ncolcounts, A, &xc, &jc, &c, spmsv_options);

  /* Free memory */
  free(part); free(ncounts); free(A);
  free(xa);free(ja);free(a);
  free(xb);free(jb);free(b);
  free(xc);free(jc);free(c);
  free(v); free(b_ncolcounts);
  MPI_Finalize();
}
```

5 Sparse tournament pivoting factorization

In this section we present a collection of algorithms for computing a low-rank approximation of a large sparse matrix. We use sparse QR based factorizations which are effective in revealing the singular values in terms of both accuracy and speed. These factorizations use tournament pivoting which is communication optimal [14].

Experiments on matrices that arise from scientific computing, data analysis, statistics and engineering show that our functions provide a good low rank approximation for sparse matrices and are less expensive than the rank revealing QR factorization in terms of computational and memory usage costs, while also minimizing the communication cost.

We present two functions to factorize a large sparse matrix. A local (inside a processor) and global (among all processors) tournament pivoting scheme is used to perform a sparse QR factorization, from which a sparse CUR factorization is derived, preserving sparsity and interpretability.

In addition, when tournament pivoting is performed on a matrix, it computes a good and fast approximation of its singular values, being scalable (tested for square matrices of size up to 3.5 million and up to 512 processors) and communication avoiding.

5.1 Column tournament pivoting

Given a matrix $A \in \mathbb{R}^{m \times n}$, the classical QR factorization with column pivoting (QRCP) selects a column of maximum norm at each step of the factorization. This requires a reduction operation among processors which costs $\mathcal{O}(\log P)$ messages. QRCP exchanges $\mathcal{O}(k \log P)$ messages for computing a rank-k approximation, and if the factorization proceeds until the end, it exchanges $\mathcal{O}(n \log P)$ messages.

When A is dense and the memory per processor is $\mathcal{O}(n^2/P)$, a lower bound on the number of messages for computing a QR factorization is $\Omega(\sqrt{P})$, where P is the number of processors [2]. Hence QRCP is not optimal in terms of the number of messages exchanged.

A communication avoiding rank revealing factorization, referred to as CARRQR, was introduced in [14]. This factorization is communication optimal modulo polylogarithmic factors, and uses tournament pivoting to select k linear independent columns at each step of the algorithm. CARRQR uses a reduction tree to perform tournament pivoting, in each tree node two sets of k columns are combined and a standard QR factorization with column pivoting is performed to obtain a permutation of these $2k$ columns. From the permuted set, the first k columns are the ones selected to be evaluated in the next node of the tree.

After the distribution of the global matrix among all processors, each processor has a local matrix A_{local} and it calls the function `preAlps_tournamentPivoting` which selects k linear independent columns using tournament pivoting.

- Inputs: A_{local} (local matrix, read into the vectors `xa,ja` and `a`, as presented in Section 3.4), matrix dimensions (m, n, nnz), rank of approximation k.
- Flags: printSVal (to print the singular values) and ordering (to activate `METIS`).
- Outputs: J_c (vector of selected columns indexes), S_{val} (vector of singular values).

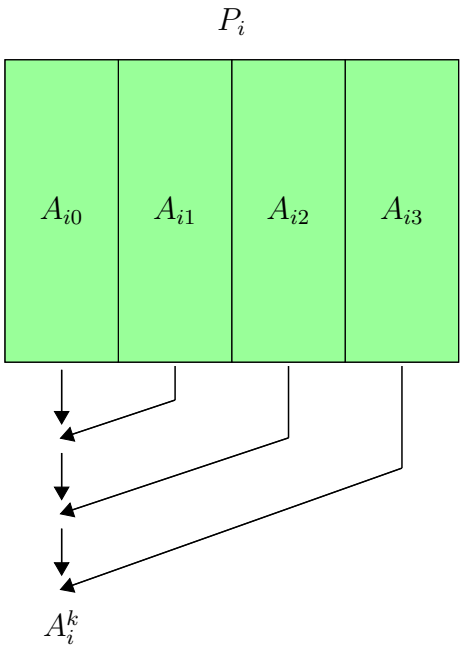


Figure 4: Flat tree inside a local processor

```
...
/* Tournament Pivoting*/
int info;
MPI_Comm comm;
int m, n, nnz, *xa, *ia;
double *a;
int printSVal = 1;
int ordering = 1; // Metis ordering activated
long *Jc,*Sval;
Jc = malloc(sizeof(long)*k); /* Indexes of selected columns */
Sval = malloc(sizeof(long)*k); /* k first singular values */

info = preAlps_tournamentPivoting(comm,xa,ia,a,m,n,nnz,col_offset,k,Jc,
                                  &Sval,printSVal,ordering);
...
```

To illustrate how this function works, let us consider $P = 4$ processors, we perform a local and a global reduction operation using a flat and binary tree respectively.

On each node of the reduction tree, a sparse QR factorization is performed on a set of $2k$ columns using the package `Suitesparse` from Tim Davis [4], next a dense QR factorization with column pivoting is performed on the small upper triangular matrix R using the function `dgeqp3` from either `LAPACK` or `MKL` to obtain a permutation of the set that gives the final k columns.

For $i \in \{1, 2, 3\}$, let $A_i = [A_{i0}\ A_{i1}\ A_{i2}\ A_{i3}]$ be the local matrix on processor P_i, assumed to be formed by 4 panels of k columns each. The local reduction selects a set of k columns A_i^k, as it is presented in Figure 4.

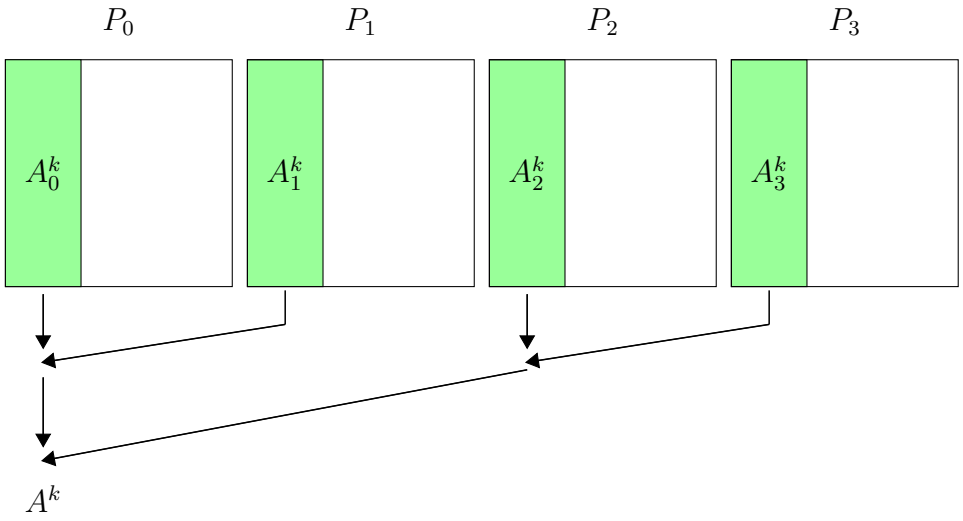


Figure 5: Binary tree among all processors

The global reduction is performed among all processors using a binary tree to select the final set of k columns A^k, the process is showed in Figure 5.

The function `preAlps_tournamentPivoting` gives as output a size n permutation vector J_c such that

$$A^k = A(:, J_c)$$

(we use `MATLAB` notation in this report).

5.2 QR factorization

Let us consider an $m \times n$ matrix A and set $r = \min(m, n)$. The column pivoting QR factorization (QRCP) of A takes the form

$$\underset{(m\times n)}{A} \quad \underset{(n\times k)}{P} = \underset{(m\times k)}{Q} \quad \underset{(m\times k)}{S}, \tag{5.1}$$

where P is a permutation matrix, Q is an orthonormal matrix, and S is upper triangular matrix (commonly the upper triangular matrix is written as R but we use S to avoid confusion with the factors in the CUR factorization).

QRCP computes the factorization via Householder transformations [15] using rank-1 updates to the matrix. QRCP can be halted at step k to produce a rank-k approximation A_{approx} to the matrix A. To illustrate this, after k steps of the factorization the matrices can be split as

$$Q = \begin{array}{c} \\ m \end{array} \begin{array}{c} \begin{array}{cc} k & r-k \end{array} \\ \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \end{array} \quad \text{and} \quad S = \begin{array}{c} \begin{array}{cc} k & n-k \end{array} \\ \begin{array}{c} k \\ r-k \end{array} \begin{bmatrix} S_{11} & S_{12} \\ 0 & S_{22} \end{bmatrix} \end{array}.$$

Equation (5.1) can be written as

$$A = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} S_{11} & S_{12} \\ 0 & S_{22} \end{bmatrix} P^T = \underbrace{Q_1 \begin{bmatrix} S_{11} & S_{12} \end{bmatrix} P^T}_{=:A_{\text{approx}}} + \underbrace{Q_2 \begin{bmatrix} 0 & S_{22} \end{bmatrix} P^T}_{\text{"remainder"}}. \tag{5.2}$$

A BLAS-3 version of QRCP algorithm can be obtained using the routine `dgeqp3` [1], or its parallel version given in [4]. However, a rank k pivoted sparse factorization as (5.2), is not provided as a built in function in existing packages.

In contrast to the scheme of QRCP that obtains the factorization (5.2) in k steps, we obtain it directly in 1 step using the tournament pivoting scheme introduced previously. The function `preAlps_tournamentPivotingQR` performs this factorization, we present the details below.

Using notation from Subsection (5.1), we can identify the factors in (5.2):

- The global permutation vector J can be split into the selected columns and the remaining columns,
$$\underset{(1\times n)}{J} = [\underset{1\times k}{J_c} \quad \underset{1\times(n-k)}{J_{\text{rem}}}],$$
so that $A^k = A(:, J_c)$, $A^{\text{rem}} = A(:, J_{\text{rem}})$ and $P = I(:, J)$, where I is the $n \times n$ identity matrix.

- Q_1 and S_{11} are the factors of the QR factorization of A^k,
$$A^k = Q_1 S_{11}.$$

- And
$$\begin{bmatrix} S_{12} \\ S_{22} \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix}^T A^{\text{rem}}.$$

From (5.2), we compute the approximation error

$$\|A - A_{approx}\| = \|Q_2[0 \quad S_{22}]P^T\| = \|[0 \quad S_{22}]\| = \|S_{22}\|. \tag{5.3}$$

Computing A_{approx} is typically much faster than computing the truncated SVD. The results obtained in [14] and [19] shows that this factorization produces a very good approximation of the singular values, and the error decreases while incrementing the rank k of the approximation. However for certain extremely rare matrices, substantial sub-optimality can result [23].

The function `preAlps_tournamentPivotingQR` performs the previous factorization in parallel trying to maintain a good load balancing among processors with a small communication cost. Next, we present a call of `preAlps_tournamentPivotingQR` from a single processor.

– Inputs: A_{local} (local matrix, read into the vectors `xa,ja` and `a`, as presented in Section 3.4), matrix dimensions (m, n, nnz), rank of approximation k.

– Flags: printSVal (to print the singular values), checkFact (to print the factorization error), printFact (to print the matrices Q and R) and ordering (to activate `METIS`).

– Outputs: J_c (vector of selected columns indexes), S_{val} (vector of singular values).

– As this is a reduction based algorithm, the processor 0 (Master processor) has the complete $Q(m \times k)$ and $R(k \times n)$ matrices, and we can find the approximation error

$$\|A_{\text{global}} - A_{\text{aaprox}}\| = \|A_{\text{global}} - Q * R\|.$$

```
...
/* Tournament Pivoting QR */
int info;
MPI_Comm comm;
int m, n, nnz, *xa, *ia;
double *a;
int printSVal = 1;
int checkFact = 1, printFact = 1;
int ordering = 1; // Metis ordering activated
long *Jc,*Sval;
Jc = malloc(sizeof(long)*k); /* Indexes of selected columns */
Sval = malloc(sizeof(long)*k); /* k first singular values */

info = preAlps_tournamentPivotingQR(comm,xa,ia,a,m,n,nnz,col_offset,k,Jc,&Sval,
                                    printSVal,checkFact,printFact,ordering);
...
```

In the directory `preAlps/test/` we provide a prototype code that shows how to perform a sparse QR factorization using tournament pivoting.

5.3 CUR factorization

The library `preAlps` includes a sparse CUR factorization, which is based on a QR truncated factorization, it is performed by the function `preAlps_tournamentPivotingCUR`. The motivation for developing this function is to provide a structure preserving factorization which is useful in important modern applications arising nowadays, such as genetic, medical imaging, and internet data.

A sparse CUR factorization helps to solve classical problems that arise when using QR, SVD or Eigen-Analysis, for instance:

- sparsity is destroyed by orthogonalization,
- storage requirements are small for CUR factorizations,
- interpretability is very important in diverse applications, for instance in genetics is more valuable to have a list of representative set of genes than a linear combination of them (which does not have a practical meaning).

The implemented CUR factorization is a continuation of the factorization (5.2), in which we have

$$A_{\text{approx}} = Q_1 \begin{bmatrix} S_{11} & S_{12} \end{bmatrix} P^T. \tag{5.4}$$

As we already know the factors of the S matrix , we rewrite the previous factorization as

$$A_{\text{approx}} = Q_1 S_{11} \begin{bmatrix} I_k & T \end{bmatrix} P^T, \tag{5.5}$$

where T is the $k \times (n-k)$ matrix resulting from solving the linear system

$$S_{11} T = S_{12}. \tag{5.6}$$

We denote

$$V = P \begin{bmatrix} I_k \\ T \end{bmatrix}. \tag{5.7}$$

1. `preAlps_tournamentPivotingCUR` returns two length k vectors J_c and J_r, which are the indexes of the selected columns and rows respectively.

2. J_c is obtained from tournament pivoting of the matrix A, it is the same J_c vector from the previous Subsection, so that we obtain the $m \times k$ matrix $C := A^k = A(:, J_c)$.

3. J_r is obtained with the same idea as J_c, performing a QR factorization of C^T, which returns a permutation vector P_r of size $1 \times m$, take $J_r = P_r(1:k)$. Next, we obtain the $k \times n$ matrix $R := A(J_r, :)$.

4. Replacing (5.7) in (5.5) we get

$$A_{\text{approx}} = C V^T. \tag{5.8}$$

   Factorization (5.8) is know as the one-sided interpolative decomposition (ID), it is called interpolative since the columns of A_{approx} are obtained from an interpolation of the columns of C being V^T the matrix of interpolation weights.

5. Finally, the matrix U comes from solving the linear system

$$UR = V. \tag{5.9}$$

6. In order to solve the sparse linear systems (5.9) and (5.6) we use the solver function `SuiteSparseQR_C_backslash_sparse`.

We use the previous QR-based CUR factorization, since it is in general more accurate than randomized algorithms [29], and its implementation in parallel is efficient maintaining a good load balancing between processors.

The function `preAlps_tournamentPivotingCUR` performs the previous factorization efficiently. Next, we present a call of `preAlps_tournamentPivotingQR` from a single processor.

– Inputs: A_{local} (local matrix, read into the vectors `xa,ja` and `a`, as presented in Section 3.4), matrix dimensions (m, n, nnz), rank of approximation k.

– Flags: printSVal (to print the singular values), checkFact (to print the factorization error), printFact (to print the vectors Jc and Jr and the matrix U) and ordering (to activate `METIS`).

– Outputs: J_c (vector of selected columns indexes) , J_r (vector of selected rows indexes), S_{val} (vector of singular values).

– At the end of computations, the master processor has the matrices $C = A_{\text{global}}(:, J_c)$, $U(k \times k)$ and $R = A_{\text{global}}(J_r, :)$ and we compute the approximation error

$$\|A_{\text{global}} - A_{\text{aaprox}}\| = \|A_{\text{global}} - CUR\|.$$

```
...
/* Tournament Pivoting CUR */
int info;
MPI_Comm comm;
int m, n, nnz, *xa, *ia;
double *a;
int printSVal = 1;
int checkFact = 1, printFact = 1;
int ordering = 1; // Metis ordering activated
long *Jc,*Jr,*Sval;
Jc = malloc(sizeof(long)*k); /* Indexes of selected columns */
Jr = malloc(sizeof(long)*k); /* Indexes of selected rows */
Sval = malloc(sizeof(long)*k); /* k first singular values */

info = preAlps_tournamentPivotingCUR(comm,xa,ia,a,m,n,nnz,col_offset,k,Jr,Jc,&Sval,
                                     printSVal,checkFact,printFact,ordering);
...
```

5.4 Example program

The following example presents how to use our functions to perform a CUR factorization of a sparse matrix read by `stdin`.

```
/*
============================================================================
Description : Performs a CUR factorization using tournament pivoting on a sparse
matrix.
============================================================================
*/
#include "preAlps_matrix.h"
#include "tournamentPivoting.h"
#include "spTP_utils.h"

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

/* Global variables */
double *Sval; // vector of singular values.
int k; // rank of approximation.
int rank,size;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm comm;
MPI_Comm_dup(MPI_COMM_WORLD, &comm);
```

```
22
23  /* Initialize variables (can be donde by stdin too) */
24  int ordering = 0; // if set to 1 -> Uses Metis ordering.
25  int printSVal = 0; // if set to 1 -> Prints the singular values into a file.
26  int checkFact = 0; // if set to 1 -> Returns factorization error (infinty norm).
27  int printFact = 0; // if set to 1 -> Prints the matrix U and the vectors Jr, Jc.
28  char *matrixName = NULL;
29
30  /* Reading parameters from stdin */
31  preAlps_TP_parameters_display(comm,&matrixName,&k,ordering,&printSVal,
32  &checkFact,&printFact,argc,argv);
33
34  /* Reading matrix in Master processor */
35  int *xa = NULL, *ia = NULL;
36  double *a = NULL;
37  int m=0, n=0, nnz=0;
38
39  if(rank ==0){
40    preAlps_matrix_readmm_csc(matrixName, &m, &n, &nnz, &xa, &ia, &a);
41  }
42
43  free(matrixName);
44
45  /* Distribute the matrix among all processors */
46  long col_offset=0;
47  preAlps_spTP_distribution(comm,&m, &n, &nnz, &xa, &ia, &a, &col_offset, checkFact);
48
49  /* Alocate memory for the vectors Jc, Jr and the singular values */
50  ASSERT(k>0);
51  long *Jc,*Jr;
52  Jr  = malloc(sizeof(long)*k);
53  if(rank == 0) {
54    Jc  = malloc(sizeof(long)*k);
55    if(printSVal) Sval = malloc(sizeof(long)*k);
56  }
57
58  /* Call tournamentPivotingCUR, gets Jc, Jr and the singular values (if required). */
59  double t_begin, t_tp;
60  t_begin=MPI_Wtime();
61  preAlps_tournamentPivotingCUR(comm,xa,ia,a,m,n,nnz,col_offset,k,Jr,Jc,&Sval,
62  printSVal,checkFact,printFact,ordering);
63  t_tp = MPI_Wtime()-t_begin;
64
65
66  /* Print the results */
67  if(rank==0) {
68    printf("Time for tournamentPivotingCUR = %f \n", t_tp );
69    free(Jc);
70    if(printSVal) free(sval);
71  }
72  free(Jr); free(xa); free(ia); free(a);
73
74  MPI_Finalize();
75  return 0;
76  }
```

6 Experiments

6.1 Environment

In this section we evaluate the performance of the routines spMSV, tournamentPivotingQR, and tournamentPivotingCUR on two machines running on Linux. The first machine has 32 nodes, each node is equipped with a socket having 10 cores based on "Intel Xeon IvyBridge E5-4650 v2" processor, and each core has a frequency of 2.40GHz. The second machine has 28 nodes, each node is equipped with a socket having 24 cores based on "Intel Xeon E5-2670", and each core has a frequency of 2.6GHz. Since the computing machine is shared with other users, we were able to run some of our experiments up to 200 cores on the first machine and 512 cores on the second machine. We assign one MPI task per core. We determine the speedup of the sparse matrix-matrix product with respect to MKL 12.0 vendor library [30]. As MKL supports only shared memory, we tested two routines of MKL on a single node. The first routine, `mkl_dcsrcsr`, performs the product of two sparse matrices stored using CSR format. The second routine, `mkl_dcsrmm`, performs the product of two matrices where the first is sparse and stored using CSR format and the second is dense. In the second case, the matrices formed by a set of vectors in our test set are converted to dense matrices before calling the `mkl_dcsrmm` routine. We consider both routines as the best single-node sparse matrix-matrix product. For simplicity reasons, we refer to `mkl_dcsrcsr` and `mkl_dcsrmm` as `MKL(sparse x sparse)` and `MKL(sparse x dense)` respectively.

6.2 Sparse matrix matrix product

6.2.1 Description of test matrices

In order to evaluate the performance of spMSV, we have used a set of matrices from the University of Florida sparse matrix collection [13] arising from various real applications. We describe these matrices in Table 1. For each test, we use A as the sparse matrix, and B as the matrix formed by a set of vectors.

6.2.2 Performance of spMSV

Figure 6 shows the speedup of spMSV with respect to MKL for 32 and 128 processors when the number of block columns of B is 32 (T = 32). For 32 processors, spMSV is on average 37 times faster than `MKL(sparse x dense)` with a maximum speedup of 102, and on average 18 times faster than `MKL(sparse x sparse)` with a maximum speedup of 27. For 128 processors, spMSV is on average 28 times faster than `MKL(sparse x dense)` with a maximum speedup of 90, and on average 12 times faster than `MKL(sparse x sparse)` with a maximum speedup of 25. We observe that maximum speedup is always obtained for the matrix crankseg_1, which is much denser than the other test matrices. We see a decrease in the speedup for 128 processors versus 32 processors due to the fact that for large number of processors, the communication cost tends to dominate the computation. When the problem size is fixed and the number of processors increases, the number of nonzero elements per processor decreases considerably, and finally the computation represents less than 10% of the total execution time. This can be better illustrated in Figure 7 which

Name	Matrix size	nnz	Description
bcsstk37	25,503	1,140,977	Stiffness Matrix, Track Ball
msc23052	23,052	1,142,686	Symmetric test matrix from MSC/Nastran
raefsky4	19,779	1,316,789	Buckling problem for container model
turon_m	189,924	1,690,876	System modelling the underground
cfd1	70,656	1,825,580	CFD, symmetric pressure matrix
bcsstk39	46,772	2,060,662	Stiffness Matrix, Shuttle Solid Rocket Booster
ct20stif	52,329	2,600,295	CT20 Engine Block
shipsec8	114,919	3,303,553	Ship section/detail from production run
Dubcova3	146,689	3,636,643	Univ. Texas at El Paso, from a PDE solver
cant	62,451	4,007,383	FEM/Cantilever
s3dkq4m2	90,449	4,427,725	FEM, cylindrical shell
Si34H36	97,569	5,156,379	Real-space pseudopotential method
consph	83,334	6,010,480	FEM/Spheres: FEM concentric spheres
thermal2	1,228,045	8,580,313	Unstructured FEM, steady state thermal problem
crankseg_1	52,804	10,614,210	OUTPUT4-Matrix
kkt_power	2,063,494	12,771,361	Optimal power flow, nonlinear optimization
Queen_4147	4,147,110	316,548,962	3D structural problem
nlpkkt240	27,993,600	760,648,352	3D PDE-constrained optimization problem

Table 1: Description of matrices sorted by number of nonzeros

shows the time for each operation of spMSV for an arbitrary matrix. We observe that for 2 processors, the cost of the computation represents about 70% of the total time, while for 64 processors, it represents only 15% of total time. Similarly, the communication cost for 2 processors represents only 12% of the computations, while for 64 processors, it represents 60% of the total time.

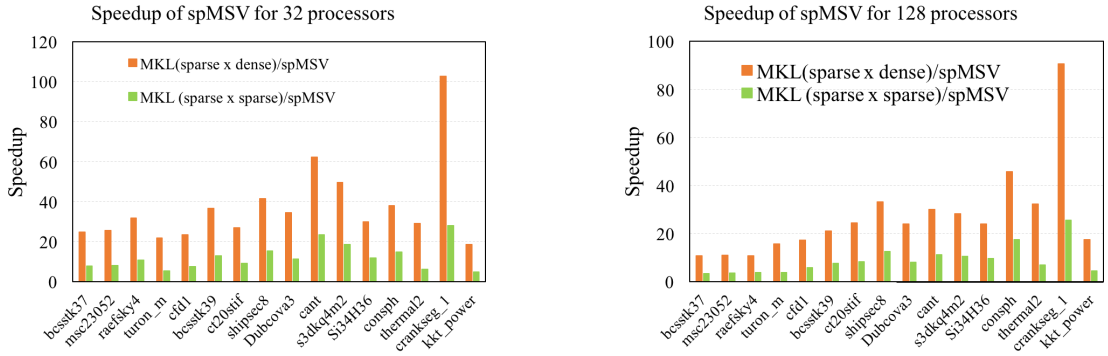


Figure 6: Speedup of SPMSV for 32 and 128 processors with respect to `MKL(sparse x dense)` and `MKL(sparse x sparse)`.

Figure 8 shows the scalability of spMSV when increasing the number of processors to 128. For the matrices with less than 12 millions non-zeros as represented at the left of the figure, spMSV is scalable up to 64 processors. While for larger matrices with over

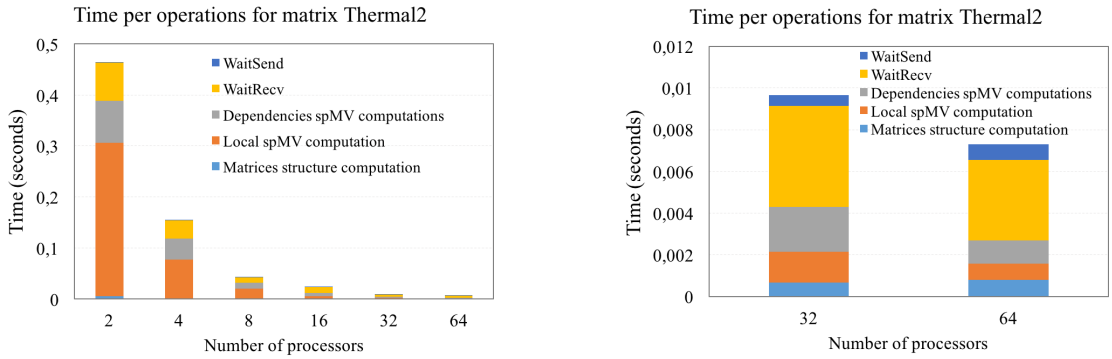


Figure 7: Time of each operation of spMSV for the matrix Thermal2 when the number of processor varies.

300 millions non-zeros as represented at the right of the figure, it is scalable up to 128 processors.

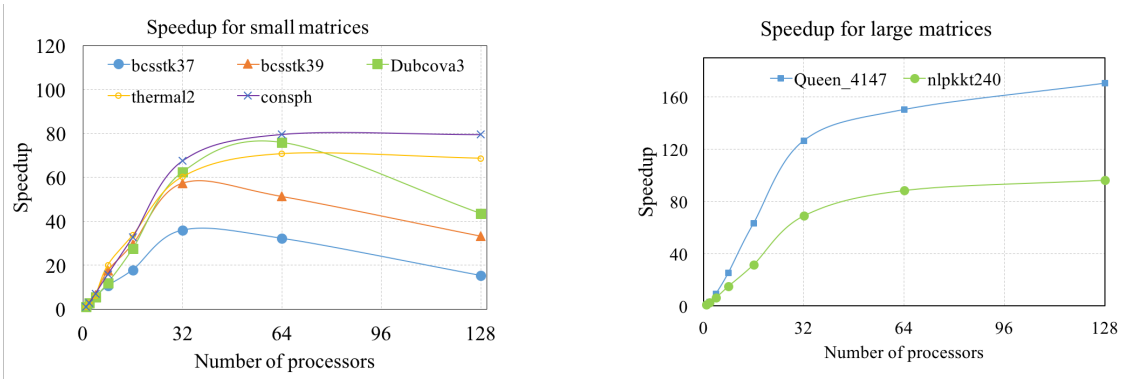


Figure 8: Speedup of spMSV for T=32. At the left, the test matrices have less than 12 millions of non-zeros. At the right, the test matrices have more than 300 millions of non-zeros.

6.3 Low rank approximation

6.3.1 Approximation results

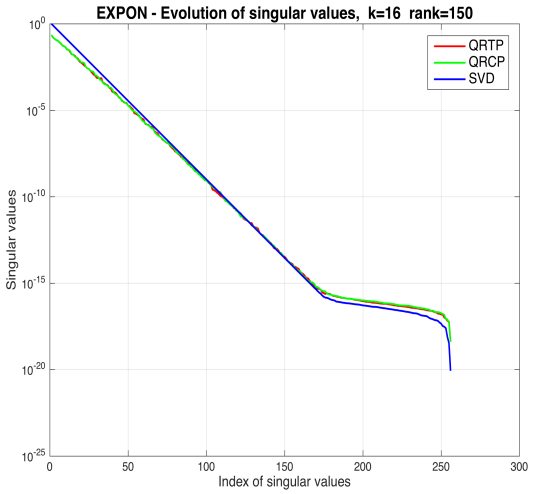
Singular value approximation: We present the approximation of the whole spectrum of 2 challenging matrices of size $n \times n$, which are often among the matrices used for testing rank revealing algorithms, see e.g. [14] and [17].

Table 2: Test matrices generated in MATLAB

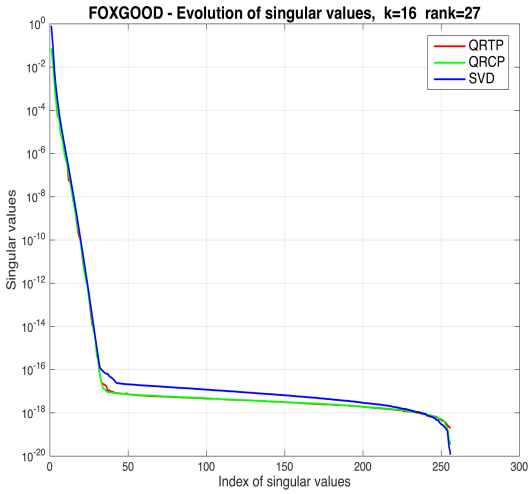
No.	Matrix	Description
1	`EXPONENTIAL`	Exponential Distribution, $\sigma_1 = 1$, $\sigma_i = \alpha^{i-1} \quad (i = 2, \ldots, n)$ [3]
2	`FOXGOOD`	Severely ill-posed test problem of the 1st kind Fredholm integral equation used by Fox and Goodwin [21]

We construct these matrices setting the size $n = 256$. Setting $k = 16$, we apply recursively the sequential `MATLAB` implementation of our function `preAlps_tournamentPivotingQR` to obtain all the singular values, in the context of a rank revealing QR factorization.

Figure 9 shows approximations of the singular values using two rank revealing factorizations with pivoting, QRTP uses tournament pivoting, while QRCP uses the classic column pivoting. The singular values obtained by the `svd` routine from `MATLAB` are also displayed. The results from Figure 9 show that the singular values are well approximated.



Singular values of Exponential matrix



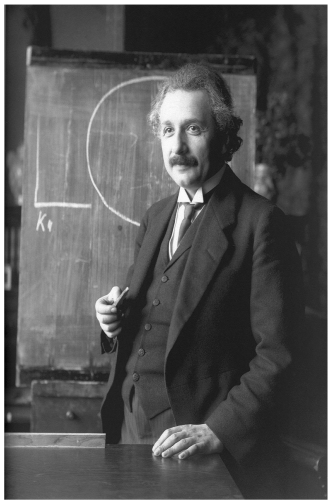
Singular values of Foxgood matrix

Figure 9: Singular values approximation using QR with column pivoting and QR with tournament pivoting.

Image approximation and compression: Using a `MATLAB` implementation of the algorithms presented in section 5, we analyze the approximation and compression of each

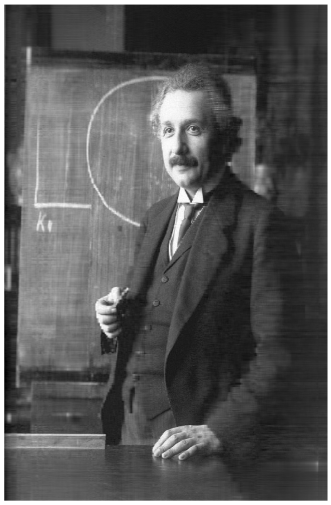
of them. For that, we define the compression ratio C_r as the amount of memory space required by A_{approx} with respect to the memory space required by A. For instance, for the CUR and QR factorizations of dense matrices, the compression ratios are

$$C_r^{CUR} = \frac{k \times (m+n+k)}{m \times n}, \qquad C_r^{QR} = \frac{k \times (m+n)}{m \times n}.$$

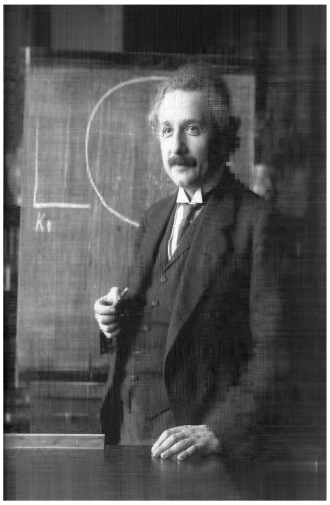


Original image

QRNP, $C_r = 0.17$, $k = 100$



QRTP, $C_r = 0.17$, $k = 100$



CUR, $C_r = 0.18$, $k = 100$

Figure 10: Approximation of image of size 1345×1024 using different algorithms, image source: `https://en.wikipedia.org/wiki/Albert_Einstein#/media`

Figure 10 shows graphically the approximation of a low rank approximation using different methods (QRNP: QR without pivoting, QRTP: QR with tournament pivoting). We can see that using 17% (for QRTP) and 18% (for CUR) of the memory required by the full matrix, we obtain a good approximation of the image.

The result is due to the fact that the singular values of the matrix representing the image decrease rapidly. Indeed, the first singular value is larger than 400 and the singular values beyond the hundredth are smaller than 2. Also, note that the images obtained from the approximations based on QR and CUR show similar results. But the error from CUR factorization is typically larger than the error from QR factorization, since the computation of the matrix U incurs an additional error.

6.3.2 Scalability results

We test the scalability of our algorithms using a a set of square sparse matrices of size $n \times n$ coming from the University of Florida sparse matrix collection [13]. We divide the tests between those for small matrices (with $n \leq 100,000$) and those for larger matrices (with $n \geq 500,000$) . The results show that the algorithms are scalable up to 512 processors for the set of matrices used in our tests.

Table 3: Scalability for small matrices, time in seconds to select $k = 16$ columns, with LAPACK and without METIS.

Matrix	**Dimensions**			**Number of MPI processes**				
	nrows	*ncols*	*nnz*	4	8	16	32	64
photogrametry	1,388	390	11,816	0.2	0.1	0.1		
dictionary28	52,652	52,652	178,076	118.1	57.0	26.4	13.0	6.7
water_tank	60,740	60,740	203,5281	21.7	11.6	6.2	3.2	1.7
delaunay_n16	65,536	65,536	393,150	21.6	11.8	6.6	3.4	1.8
gas_sensor	66,917	66,917	170,3365	28.4	15.2	8.2	4.3	2.2
oilpan	73,752	73,752	359,7188	118.9	65.9	40.1	27.7	11.2
shallow_water2	81,920	81,920	327,680	33.5	18.7	10.8	5.6	2.9
onera_dual	85,567	85,567	419,201	36.1	20.4	12.1	6.2	3.3

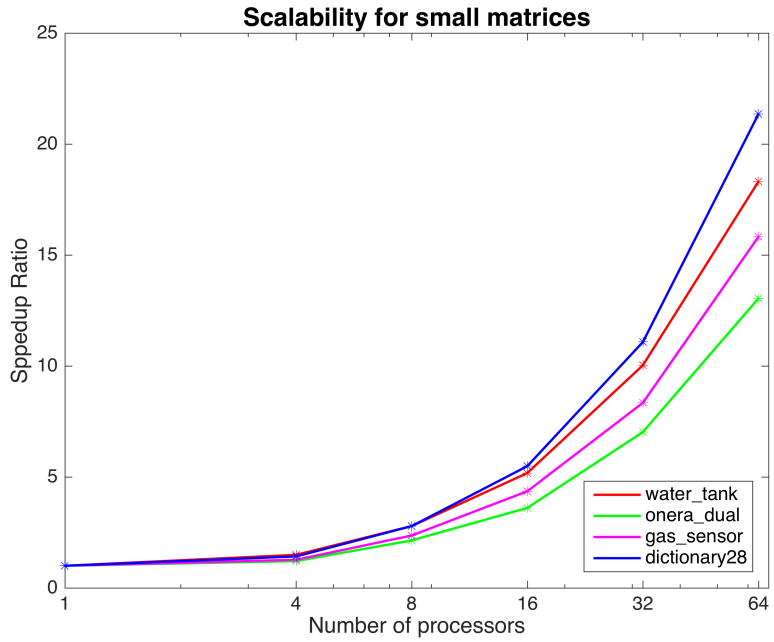


Figure 11: Scalability for small matrices.

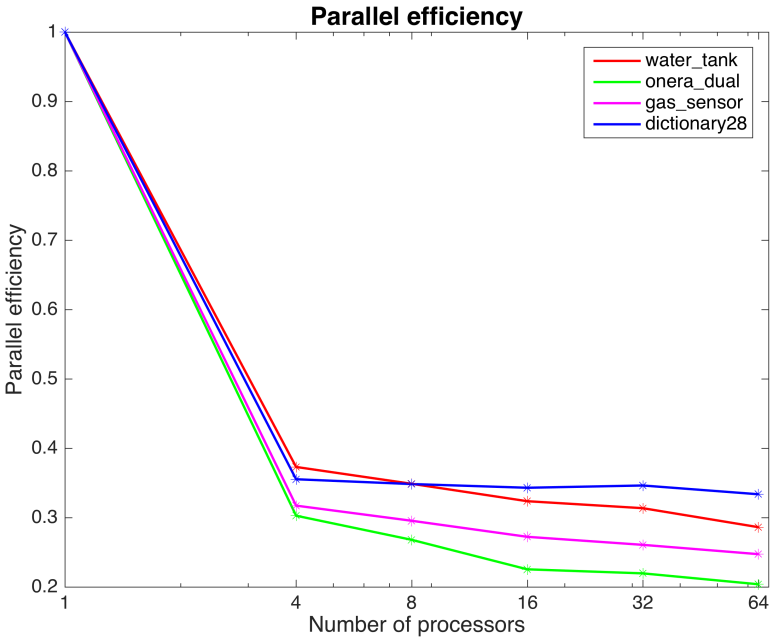


Figure 12: Parallel efficiency for small matrices.

For the larger matrices, we have done many tests using different combinations of the parameters given to our functions. We swap `LAPACK` for `MKL` during the calls for computing a QR factorization of a small dense matrix in our reduction tree. We also swap between the choice of a natural ordering or `METIS` ordering. Below we present the results using

Table 4: Scalability up to 200 processors, time in seconds to select $k = 256$ columns, with MKL and METIS.

Matrix	Dimensions			Number of MPI processes			
	nrows	*ncols*	*nnz*	32	64	128	200
parabolic_fem	525,825	525,825	3,674,625	49.3	40.0	31.5	25.1
mac_econ_fwd500	206,500	206,500	1,273,389	90.3	39.4	22.6	19.0
atmosmodd	1,270,432	1,270,432	8,814,880	374.1	210.9	128.8	99.7
circuit5M_dc	3,523,317	3,523,317	19,194,193	874.2	441.3	239.6	160.2

the combination that has produced the best results for each of the two machines.

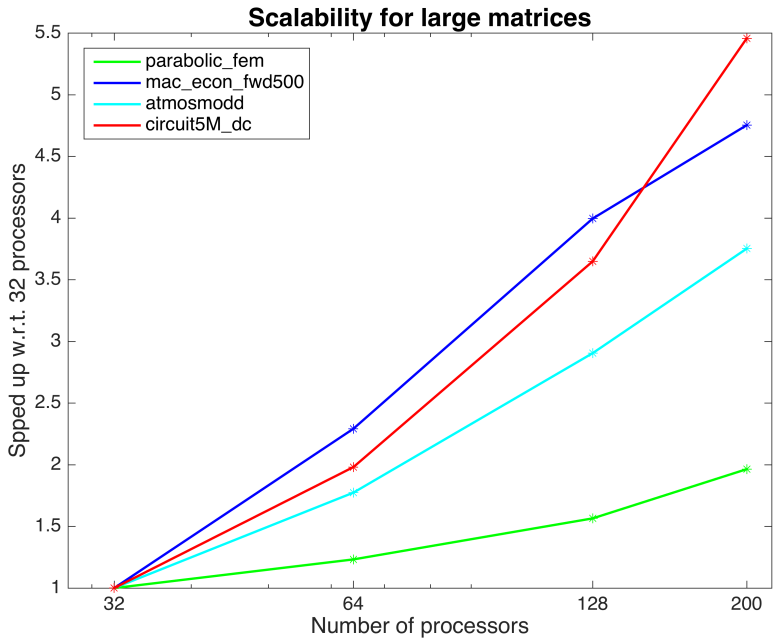


Figure 13: Scalability for large matrices up to 200 cores.

Table 5: Scalability up to 512 processors, time in seconds to select $k = 256$ columns, with LAPACK and METIS.

Matrix	Dimensions			Number of MPI processes				
	nrows	*ncols*	*nnz*	32	64	128	256	512
parabolic_fem	525,825	525,825	3,674,625	57.66	44.0	25.7	12.4	6.9
mac_econ_fwd500	206,500	206,500	1,273,389	94.0	55.1	28.2	13.1	7.2
atmosmodd	1,270,432	1,270,432	8,814,880	370.3	203.3	150.1	86.0	44.0
circuit5M_dc	3,523,317	3,523,317	19,194,193	916.0	465.9	245.4	143.1	80.7

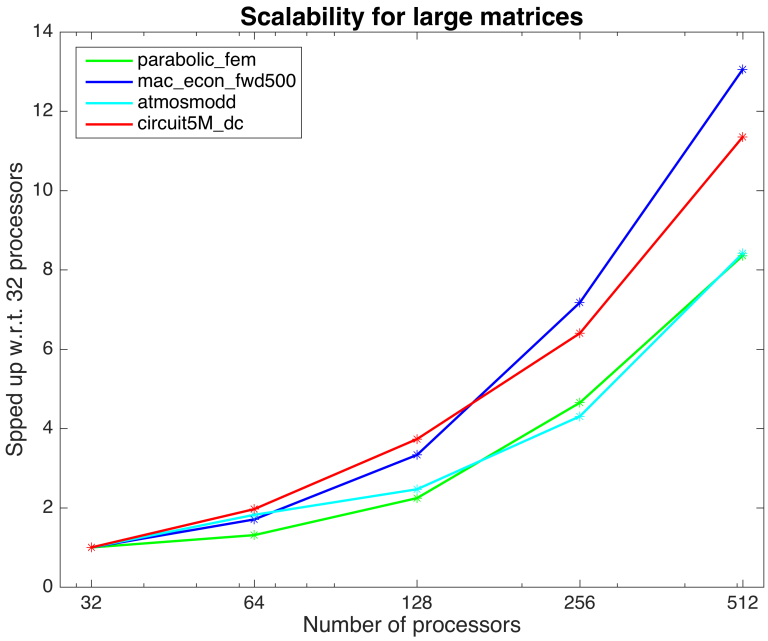


Figure 14: Scalability for large matrices up to 512 cores.

7 Routines

In this section, we present the utility routines implemented in `PreAlps` library.

7.1 Sequential Utility routines

```
/* Convert a matrix from csc to csr*/
int preAlps_matrix_convert_csc_to_csr(int m, int n, int **xa, int *asub, double *a);

/* Convert a matrix from csr to csc*/
int preAlps_matrix_convert_csr_to_csc(int m, int n,
                                      int **xa, int *asub, double *a);

/* Convert a matrix from csr to dense */
int preAlps_matrix_convert_csr_to_dense(int m, int n,
```

```
                                    int *xa, int *asub, double *a,
                                    preAlps_matrix_layout_t mlayout, double *a1, int lda1);

/* Copy a matrix A into A1 */
void preAlps_matrix_copy(int m,
                   int *xa, int *asub, double *a,
                   int *xa1, int *asub1, double *a1);

/* Print a dense matrix*/
void preAlps_matrix_dense_print(preAlps_matrix_layout_t mlayout, int m, int n,
                          double *a, int lda, char *s);

/*
 * Partition a matrix using an hypergraph partitioning tools
 * part_loc:
 *     output: part_loc[i]=k means row i belongs to subdomain k
 */

int preAlps_matrix_hpartition_sequential(int m, int n, int *xa, int *asub,
                                    int nparts, int *part);


/*
 * Partition a matrix using Metis
 * part_loc:
 *  output: part_loc[i]=k means row i belongs to subdomain k
 */
int preAlps_matrix_partition_sequential(int m, int *xa, int *asub,
                                    int nparts, int *part);


/*
 * Compute A1 = P'AQ where P and Q are permutations of 0..m-1 and 0..n-1.
 * if pinv or q is NULL it is considered as the identity
 */
void preAlps_matrix_permute (int n, int *xa, int *asub, double *a,
                        int *pinv, int *q,
                        int *xa1, int *asub1,double *a1);

/* Print a CSR matrix */
void preAlps_matrix_print(preAlps_matrix_storage_t mtype, int m,
                     int *xa, int *asub, double *a, char *s);

/* Read a matrix market data file and stores the matrix using CSR format */
int preAlps_matrix_readmm_to_csr(char *filename, int *m, int *n, int *nnz,
                           int **xa, int **asub, double **a);

/*
 * Perform C = A(i_begin:i_end, :) * B, where the matrix submatrix
```

```
 * A(i_begin:i_end,:) is sparse,
 * and the matrix B is dense.
 * The result is
 * ptrRowBegin:
 * input: ptrRowBegin[i] = j means the first non zero element of row i is in column j
 * ptrRowEnd:
 * input: ptrRowEnd[i] = j means the last non zero element of row i is in column j
 */
int preAlps_matrix_subMatrix_CSRDense_Product(int m,
                                              int *ptrRowBegin, int *ptrRowEnd,
                                              int a_colOffset,
                                              int *asub, double *a,
                                              double *b, int ldb, int b_nbcol,
                                              double *c, int ldc);


/* Create a full symmetric matrix from a lower triangular matrix */
int preAlps_matrix_symmetrize(int m, int n, int nnz,
                              int *xa, int *asub, double *a, int *nnz2,
                              int **xa2, int **asub2, double **a2);
```

7.2 Parallel Utility routines

```
/*
 * Assemble a previous distributed matrix into one CSR matrix,
 * All the processors calling this routines send their part
 * of the matrix to the master processor.
 */
int preAlps_matrix_assemble(MPI_Comm comm, int mloc,
                            int *xa, int *asub, double *a, int *ncounts,
                            int **xa1, int **asub1, double **a1);

/*
 * Create a sparse block structure of a CSR matrix.
 * The matrix is initially 1D row block distributed.
 */
int preAlps_matrix_createBlockStruct(MPI_Comm comm, preAlps_matrix_storage_t mtype,
                                     int mloc, int nloc,
                                     int *xa, int *asub, int nparts, int *ncounts,
                                     int *ABlockStruct);

/*
 * Distribute a CSR matrix to all the processors in the communicator
 * using 1D block rows distribution.
 */
int preAlps_matrix_distribute(MPI_Comm comm, int m,
                              int **xa, int **asub, double **a, int *mloc);
```

```
/*
 * 1D block row redistribution of the matrix to each proc guided by an array 'part'
 * which indicates to which subdomain each row belongs to.
 * part_loc: part_loc[i]=k means row i belongs to subdomain k.
*/
int preAlps_matrix_kpart_redistribute(MPI_Comm comm, int m, int n,
                                int **xa, int **asub, double **a,
                                int *mloc,
                                int *part);

/*
 * Partition a matrix in parallel using parMetis
 * part_loc:
 *  output: part_loc[i]=k means rows i belongs to subdomain k
 */
int preAlps_matrix_partition(MPI_Comm comm, int *vtdist,
                       int *xa, int *asub, int nbparts, int *part_loc);


/* Every processor prints its local matrix*/
void preAlps_matrix_print_global(MPI_Comm comm, preAlps_matrix_storage_t mtype, int m,
                           int *xa, int *asub, double *a, char *s);


/*
 * 1D block row distribution of the matrix to all the processors in the communicator
 * using ncounts to indicate the number of rows per processor.
 */
int preAlps_matrix_scatterv(MPI_Comm comm, int mloc,
                      int **xa, int **asub, double **a, int *ncounts);
```

7.3 spMSV routine

```
int preAlps_spMSV(MPI_Comm comm,
          int mloc, int nloc, int rloc,
          int *xa, int *asub, double *a,
          int *xb, int *bsub, double *b,
          int a_nparts, int *a_nrowparts,
          int b_nparts, int *b_ncolparts,
          int *ABlockStruct,
          int **xc, int **csub, double **c,
          int *options);

/*
 * Purpose
 * =======
 *
```

```
* Perform a matrix matrix product C = A*B,
* where A is a CSR matrix, B is a CSR matrix formed by a set of vectors,
* and C is a CSR matrix.
* The matrix is initially 1D row block distributed.
*
* Arguments
* =========
*
* mloc:
*   input: local number of rows of A owned by the processor calling this routine
*
* nloc:
*   input: local number of columns of A
*
* rloc:
*   input: local number of columns of B
*
* xa,asub,a:
*   input: matrix A of size (mloc x nloc) stored using CSR.
*
* xb,bsub,b:
*   input: matrix B of size (nloc x rloc) stored using CSC.
*          If B is stored using CSR , then see options below;
*
* a_nparts:
*   input: the number of block rows of matrix A.
*          The global block structure of A has size (a_nparts x a_nparts).
*
* a_nrowparts:
*   input: Array of size a_nparts to indicate the number of rows in
*          each block column of A.
*
* b_nparts:
*   input: number of block columns for matrix B
*          The global sparse block struct size of B is (a_nparts x b_nparts).
*
* b_ncolparts:
*   input: Array of size b_nparts to indicate the number of columns
*          in Block column of B.
*
* ABlockStruct
*   input: array of size (a_nparts x a_nparts) to indicate
*          the sparse block structure of A.
*          ABlockStruct(i,j) = 0 if the block does not contains any element.
*          This array must be precomputed by the user before calling this routine.
*
* xc,csub,c:
*   output: matrix C stored as CSC.
```

```
 *           If C is NULL, it will be created.
 *           If the size of C is smaller than the expected size of A*B,
 *           it will be reallocated with the enlarged size.
 *           Otherwise C will be reused.
 *           The sparse block struct size of C is (a_nparts x b_nparts)
 *
 * options: array of size 3 to control this routine. (Coming soon)
 *   input:
 *          options[0] = 1 if the input matrix B is stored as CSR instead of CSC.
 *                           An internal buffer will be required.
 *          options[1] = 1 if the result matrix must be stored as CSR instead of CSC.
 *          options[2] = 1 switch off the possibility to switch to dense matrix.
 *
 * Return
 * ======
 * 0: the resulting matrix C is sparse, use (xc, asubc, c) to manipulate it.
 * 1: the resulting matrix C is converted to dense, use only (c) to manipulate it.
 * < 0: an error occured during the execution.
 */
```

7.4 Tournament pivoting routines

```
int preAlps_tournamentPivoting(MPI_Comm comm, int *xa, int *ia, double *a,
                               int m,  int n,  int nnz, long col_offset,
                               int k, long *Jc, double **Sval,
                               int printSVal, int ordering);


  /*
   * Purpose
   * =======
   * Performs tournament pivoting to choose k pivot columns of a sparse
   * matrix.
   * Arguments
   * =========
   * Inputs:
   *  xa,ia,a: vectors that define the CSC matrix (column pointers, row
   *           indexes and matrix values respectively),
   *  m,n,nnz: dimensions of the matrix,
   *  k: rank of the approximation,
   *  col_offset: offset of local column indexes with respect to global
   *              indexes,
   *  Flags: printSVal (to print the singular values) and ordering
   *         (to activate METIS).
   * Outputs:
   *  Jc: vector of indexes of selected columns,
   *  Sval: vector containing the approximated k first singular values.
```

```
   */


int preAlps_tournamentPivotingQR(MPI_Comm comm, int *xa, int *ia, double *a,
                                 int m, int n, int nnz, long col_offset,
                                 int k, long *Jc, double **Sval,
                                 int printSVal, int checkFact,
                                 int printFact, int ordering);

 /*
  * Purpose
  * =======
  * Performs a sparse QR factorization using tournament pivoting.
  *
  * Arguments
  * =========
  * Inputs:
  *  xa,ia,a: vectors that define the CSC matrix (column pointers, row
  *           indexes and matrix values respectively),
  *  m,n,nnz: dimensions of the matrix,
  *  col_offset: offset of local column indexes with respect to global
  *              indexes,
  *  k: rank of the approximation,
  *  Flags: printSVal (to print the singular values), checkFact (to print
  *         the factorization error), printFact (to print the matrices Q
  *         and R) and ordering (to activate METIS).
  * Outputs:
  *  Jc: vector of indexes of selected columns,
  *  Sval: vector containing the approximated k first singular values.
  */


int preAlps_tournamentPivotingCUR(MPI_Comm comm, int *xa, int *ia, double *a,
                                  int m,  int n,  int nnz, long col_offset,
                                  int k, long *Jr, long *Jc, double **Sval,
                                  int printSVal, int checkFact,
                                  int printFact, int ordering);

 /*
  * Purpose
  * =======
  * Performs a sparse CUR factorization using tournament pivoting.
  *
  * Arguments
```

```
* =========
* Inputs:
*  xa,ia,a: vectors that define the CSC matrix (column pointers, row
*           indexes and matrix values respectively),
*  m,n,nnz: dimensions of the matrix,
*  col_offset: offset of local column indexes with respect to global
*              indexes,
*  k: rank of the approximation,
*  Flags: printSVal (to print the singular values), checkFact (to print
*         the factorization error), printFact (to print the vectors Jc
*         and Jr and the matrix U) and ordering (to activate METIS).
* Outputs:
*  Jr: vector of indexes of selected rows,
*  Jc: vector of indexes of selected columns,
*  Sval: vector containing the approximated k first singular values.
*/
```

8 Conclusion

We have designed and tested `PreAlps` library which implements efficient matrix-matrix product and sparse communication avoiding low rank approximation routines widely used in iterative solvers. Our experiments show that the results are encouraging for the development of our preconditioner and enlarged Krylov subspace solvers which will be integrated gradually into the library as part of this work. The application of our singular value approximation using tournament pivoting in image approximation and compression also show good results compared with the classic singular value decomposition.

9 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] 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.

[2] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[3] Christian H. Bischof. A parallel QR factorization algorithm with controlled local pivoting. *SIAM Journal on Scientific and Statistical Computing*, 12(1):36–57, 1991.

[4] 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.

[5] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix Market: a web resource for test matrix collections. In *Quality of Numerical Software*, pages 125–137. Springer, 1997.

[6] Ümit Çatalyürek and Cevdet Aykanat. PaToH (partitioning tool for hypergraphs). In *Encyclopedia of Parallel Computing*, pages 1479–1487. Springer, 2011.

[7] Umit V. Catalyurek and Cevdet Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[8] Yanqing Chen, Timothy A. Davis, William W. Hager, and Sivasankaran Rajamanickam. Algorithm 887: CHOLMOD, Supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)*, 35(3):22, 2008.

[9] Anthony T. Chronopoulos and Charles William Gear. s-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25(2):153–168, 1989.

[10] Anthony T. Chronopoulos and Charles D. Swanson. Parallel iterative s-step methods for unsymmetric linear systems. *Parallel Computing*, 22(5):623–641, 1996.

[11] Timothy A. Davis. Algorithm 8xx: SuiteSparseQR, a multifrontal multithreaded sparse QR factorization package. *ACM Trans. Math. Software*, 2008.

[12] Timothy A. Davis. Multifrontral multithreaded rank-revealing sparse QR factorization. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[13] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[14] James W. Demmel, Laura Grigori, Ming Gu, and Hua Xiang. Communication avoiding rank revealing QR factorization with column pivoting. *SIAM Journal on Matrix Analysis and Applications*, 36(1):55–89, 2015.

[15] Gene H. Golub and Charles F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[16] Susan L. Graham, Marc Snir, Cynthia A. Patterson, et al. *Getting up to speed: The future of supercomputing*. National Academies Press, Washington, D.C., USA, 2005.

[17] Laura Grigori, Sebastien Cayrols, and James W. Demmel. Low rank approximation of a sparse matrix based on LU factorization with column and row tournament pivoting. *[Research Report] RR-8910, INRIA. 2016, pp.35. <hal-01313856>.*

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

[19] Ming Gu and Stanley C. Eisenstat. Efficient algorithms for computing a strong rank-revealing QR factorization. *SIAM Journal on Scientific Computing*, 17(4):848–869, 1996.

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

[21] Per Christian Hansen. Regularization tools version 4.0 for matlab 7.3. *Numerical algorithms*, 46(2):189–194, 2007.

[22] Mark Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, University of California, Berkeley, 2010.

[23] William Kahan. Numerical linear algebra. *Canadian Math. Bull*, 9(6):757–801, 1966.

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

[25] 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.

[26] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 36. ACM, 2009.

[27] Yousef Saad. *Iterative methods for sparse linear systems*. Society for Industrial Mathematics, 2003.

[28] Henk A. Van der Vorst. *Iterative Krylov methods for large linear systems*, volume 13. Cambridge University Press, 2003.

[29] Sergey Voronin and Per-Gunnar Martinsson. RSVDPACK: An implementation of randomized algorithms for computing the singular value, interpolative, and CUR decompositions of matrices on multi-core and GPU architectures. *arXiv preprint arXiv:1502.05366*, 2015.

[30] 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.

[31] Michael M. Wolf, Erik G. Boman, and Bruce Hendrickson. Optimizing parallel sparse matrix-vector multiplication by corner partitioning. *PARA08, Trondheim, Norway*, 2008.