NLAFET
Parallel Numerical Linear Algebra for Future Extreme Scale Systems

H2020–FETHPC–2014: GA 671633

D4.5

Integration

October 2018

Document information

Scheduled delivery	2018-10-31
Actual delivery	2018-10-30
Version	1.2
Responsible partner	INRIA

Dissemination level

PU — Public

Revision history

Date	**Editor**	**Status**	**Ver.**	**Changes**
2018-10-22	INRIA members	Draft	1.0	Initial version of document produced
2018-10-26	INRIA members	Draft	1.1	Revision of document produced
2018-10-29	INRIA members	Draft	1.2	Revision of document produced
2018-10-29	INRIA members	Draft	1.2	Final version of document produced

Author(s)

Simplice Donfack, Laura Grigori, Olivier Tissot, INRIA

Internal reviewers

Carl Christian Kjelgaard Mikkelsen, UMU
Florent Lopez, RAL

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

In Deliverables 4.3 and 4.4 we have introduced we have introduced a preconditioned Krylov subspace solver which aims to reduce communication when solving large sparse linear systems of equations.

In this deliverable we discuss the `preAlps` library which integrates a highly parallel and efficient implementation of enlarged CG Krylov subspace methods and the LORASC preconditioner. We also explain how to eliminate some synchronization points in ECG's iterations in order to increase its scalability, and it has been implemented in `preAlps`. We also present performance data for both methods.

2 Introduction

The *Description of Action* document states for Deliverable 4.5:

"*Report on the integration of preconditioners and iterative methods from D4.3 into the NLAFET library. Evaluation of the parallel efficiency of the new preconditioned iterative solvers.*"

This deliverable is the final deliverable related to the work performed in WP4, Task 4.2 (Iterative methods) and Task 4.3 (Preconditioners). Our work has focused on the design of communication avoiding Krylov subspace methods for large sparse linear systems of equations $Ax = b$, where A is a symmetric positive definite (SPD) matrix.

We have introduced and studied enlarged Conjugate Gradient (CG) methods [9], and LORASC, a robust and algebraic preconditioner[7]. Both algorithms are described in detail in previous deliverables as Deliverable 4.2 and Deliverable 4.4.

In this deliverable we recall and further describe the `preAlps` library, the external libraries that need to be linked with `preAlps`, and a description of the main routines of `preAlps`. Most of the routines were also described in detail in Deliverable 4.3, their input parameters remain the same, but for completeness we also include them here. However their performance was improved since that deliverable. In particular, we explain how to eliminate some synchronization points in ECG's iterations in order to increase its scalability, and it has been implemented in `preAlps`.. Hence we also present new results that show the parallel performance of `preAlps`.

3 Using the preAlps library

3.1 Overview

The `preAlps` library has been developed at INRIA as part of the NLAFET project. It is available from `https://github.com/NLAFET/preAlps`. The current version of `preAlps` library provides an efficient implementation of ECG and LORASC preconditioner. The ECG method extends the Krylov subspaces used by classical methods in a manner which dramatically reduces the need for communication during the solution of a linear system. LORASC is based on a low rank approximation of the Schur complement. These algorithms are well suited for parallelism and are described in detail in [9] and [7]. 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.

3.2 Dependencies with external librairies

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

- BLAS and LAPACK [2]: 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[13] and ParMETIS [14]: 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 [16]: 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 [1]: 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 external librairies. In order to use the ECG solver only, 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.

3.3 Installation

The complete installation of `preAlps` could be summarized as follows:

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

Table 1: Dependencies of preAlps with external librairies.

3.3.1 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`)

3.3.2 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 the 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.

3.4 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);
  ...
```

4 Major routines

We refer the reader to Deliverable 4.2, Deliverable 4.3, and Deliverable 4.4 for a more detailed description of the ECG method, and the LORASC preconditioner. We also refer to the User Guide for examples of usage of both ECG and LORASC.

4.1 Enlarged Conjugate Gradient

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)`: initialize the underlying structure,
- `preAlps_ECGIterate(ECG_t* ecg, int* rci_request)`: used within a loop to perform the iterations of the method,
- `preAlps_ECGStoppingCriterion(ECG_t* ecg, int* stop)`: compute and check the convergence of the method,
- `preAlps_ECGFinalize(ECG_t* ecg, double* solution)`: retrieve the solution and free the memory.

4.2 LORASC

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`.
- `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 *X, CPLM_Mat_Dense_t *Y)` : applies LORASC preconditioner on a dense matrix X, and returns the result in a dense matrix Y. 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.

5 ECG's new developments and experiments

5.1 Equivalence between Orthodir and Orthomin

In what follows, α_k, β_k, γ_k, and ρ_k are not necessarily real numbers, but typically denote matrices that have dimensions substantially smaller than the original matrix A. The ECG method has already been formally derived in Deliverables 4.2, 4.3 and 4.4. The general method is stated as Algorithm 1. It contains two variants: Orthomin and Orthodir. Orthomin variant corresponds to the choice

$$\beta_k = (AP_k)^\top R_k, \tag{5.1}$$
$$Z_{k+1} = R_k - P_k\beta_k, \tag{5.2}$$

whereas Orthodir corresponds to the choice

$$\gamma_k = (AP_k)^\top (AP_k), \tag{5.3}$$
$$\rho_k = (AP_{k-1})^\top (AP_k), \tag{5.4}$$
$$Z_{k+1} = AP_k - P_k\gamma_k - P_{k-1}\rho_k. \tag{5.5}$$

Algorithm 1 ECG algorithm.

```
Require: $A$, $R_0^e$, $k_{\max}$, $\varepsilon$
Ensure: $||R_k|| < \varepsilon$ or $k = k_{\max}$
 1: $P_0 = 0$
 2: $Z_1 = R_0^e$
 3: for $k = 1, \dots, k_{\max}$ do
 4:     $P_k = Z_k(Z_k^\top AZ_k)^{-1/2}$
 5:     $\alpha_k = P_k^\top R_{k-1}$
 6:     $X_k = X_{k-1} + P_k\alpha_k$
 7:     $R_k = R_{k-1} - AP_k\alpha_k$
 8:     if $|| \sum_{i=1}^t R_k^{(i)} ||_2 < \varepsilon$ then
 9:         stop
10:     end if
11:     construct $Z_{k+1}$ using (5.1)-(5.2) (Orthomin) or (5.3)-(5.5) (Orthodir)
12: end for
13: $x_k = \sum_{i=1}^t X_k^{(i)}$
```

In what follows, we assume exact arithmetic and we show that even if the methods are equivalent the search directions constructed by Orthodir and Orthomin are different. Indeed, by construction the approximate solutions computed by Orthodir and Orthomin are equal. Hence, the approximate residuals are also equal. But this does not imply that the search directions generated are equal even if they belong to the same space. We denote with a tilde the variables related to Orthomin and with a hat the variables related to Orthodir, e.g., $\widehat{P}_k$ are the A-normalized search directions generated during Orthodir. For the sake of brevity, we only consider the case where no breakdowns occur so that $\widehat{Z}_k$, $\widehat{P}_k$ and $\widetilde{Z}_k$, $\widetilde{P}_k$ are all well-defined.

Since Orthomin and Orthodir rely on the same projection process [8] (they both search an approximate solution in $\mathcal{K}_{t,k}$, such that the corresponding residual is orthogonal to $\mathcal{K}_{t,k}$), we know that $\widehat{X}_k = \widetilde{X}_k$. It follows that:

$$\widehat{P}_k \widehat{\alpha}_k = \widetilde{P}_k \widetilde{\alpha}_k, \tag{5.6}$$

$$\widehat{P}_k \widehat{P}_k^\top = \widetilde{P}_k \widetilde{P}_k^\top. \tag{5.7}$$

Hence, there exists $\delta_k \in \mathbb{R}^{t\times t}$ orthogonal and such that $\widetilde{P}_k = \widehat{P}_k \delta_k$. By definition we also have,

$$R_k - R_{k-1} = -A\widetilde{P}_k \widetilde{\alpha}_k. \tag{5.8}$$

A simple computation using the previous relationships gives,

$$\begin{aligned} -\widehat{Z}_{k+1}\delta_k\widetilde{\alpha}_k &= -A\widetilde{P}_k\widetilde{\alpha}_k + \widetilde{P}_k\widetilde{\gamma}_k\widetilde{\alpha}_k + \widetilde{P}_{k-1}\widetilde{\rho}_k\widetilde{\alpha}_k, & (5.9)\\ &= R_k - \widetilde{P}_k\widetilde{\beta}_k - \widetilde{Z}_k + \widetilde{P}_k\widetilde{P}_k^\top A R_{k-1}, & (5.10)\end{aligned}$$

and,

$$\begin{aligned} \widetilde{P}_k\widetilde{P}_k^\top A R_{k-1} &= \widetilde{Z}_k(\widetilde{Z}_k^\top A\widetilde{Z}_k)^{-1}\widetilde{Z}_k^\top A R_{k-1}, & (5.11)\\ &= \widetilde{Z}_k(\widetilde{Z}_k^\top A\widetilde{Z}_k)^{-1}\widetilde{Z}_k^\top A \widetilde{Z}_k, & (5.12)\\ &= \widetilde{Z}_k. & (5.13)\end{aligned}$$

Thus,

$$\widetilde{Z}_{k+1} = -\widehat{Z}_{k+1}\delta_k\widetilde{\alpha}_k. \tag{5.14}$$

This result is a generalization of a previous result presented by *Manteuffel et al.* [3] (p. 1550) for the classical CG. In fact, the authors show that $\widetilde{z}_k = \Pi_{i=0}^k(-\widetilde{\alpha}_k)\widehat{z}_k$ but they never consider explicitly the A-orthonormalized search directions. In particular, they define z_{k+1} using z_k (for Orthomin) and z_{k-1} (for Orthodir). This explains the slight difference between our generalization and their result.

When k becomes large, $\widetilde{\alpha}_k = \widetilde{P}_k^\top R_{k-1}$ and $||\widetilde{\alpha}_k||_2$ is more likely to be low because R_{k-1} is supposed to converge to 0 and $\widetilde{P}_k$ is A-orthonormalized – the same reasoning applies for $\widehat{\alpha}_k$. This result is very interesting because it shows that, since δ_k is an orthogonal matrix, when k becomes large $||\widetilde{Z}_{k+1}||_2$ can be significantly smaller than $||\widehat{Z}_{k+1}||_2$. Hence, the conditioning of $\widetilde{Z}_{k+1}^\top A\widetilde{Z}_{k+1}$ could be much worse than that of $\widehat{Z}_{k+1}^\top A\widehat{Z}_{k+1}$, possibly leading to a breakdown when computing its Cholesky factorization (line 5 in Figure 1). In practice, this phenomenon is indeed observed: there are cases where Orthomin breaks down while Orthodir does not [8]. In conclusion, Orthodir is expected to be more reliable than Orthomin. However, Orthodir requires twice as many flops and twice as much memory as Orthomin in order to construct Z_{k+1}.

5.2 Dynamic reduction of the search directions

In what follows, we briefly present an approach for reducing the block size in the Orthodir method during the iterations, proposed in [8]. The idea is to reduce the extra arithmetic and memory costs of Orthodir while maintaining its good convergence behavior.

As explained in the survey [10] the key idea to reduce the block size is to monitor the rank of R_k. Once R_k becomes rank deficient, it means that a part of the approximate solution has already converged at iteration k. More precisely for $i \geq k-1$, there exists a linear combination (independent of i) of columns of X_i that remains constant. As a consequence, the space of search directions can be reduced with no ill effects. The idea is to remove these search directions in the next iterations. As R_{k-1} is an $n \times t$ matrix with n large, it is preferable to avoid computing the rank of R_{k-1} directly. In [8], it is shown that the rank of $\alpha_k = P_k^\top R_{k-1}$ can be computed instead.

The method presented in [8] can be divided into two parts. At each iteration a Singular Value Decomposition (SVD) of α_k is computed. If the numerical rank of α_k is below a given tolerance then the search directions are reduced accordingly and some of them are kept in order to keep the A-orthogonality property. The resulting algorithm is given in Algorithm 2. Although computing the SVD of α_k at each iterations induces an extra cost compared to Orthodir, this operation does not involve any communications and it is negligible because α_k is a small matrix of size $t \times t$. Furthermore, as the search directions P_k are reduced, the dominant operation of Krylov iterations in terms of flops; the matrix product (AP_k), and the application of the preconditioner ($M^{-1}AP_k$), are cheaper.

5.2.1 Curing breakdowns in Orthomin

As explained previously Orthomin version of ECG can break down. There exist several methods to overcome this issue and in the following we recall the *Breakdown-Free block CG* method defined in [12]. Starting from the original algorithm of *O'Leary* [17], the authors propose to perform a rank-revealing QR decomposition of Z_{k+1} and then drop its null part before A-orthonormalizing it. They show that in exact arithmetic this allows to continue the algorithm with nearly no further modification. The resulting algorithm is given in Algorithm 4.

From a practical point of view the size of P_{k+1} can be reduced, but at each iteration Z_{k+1} is of size $n \times t$ because the size of R_k remains constant. Hence the matrix product AP_k is cheaper but the application of the preconditioner $M^{-1}R_k$ is not. Furthermore computing a rank-revealing QR factorization of Z_{k+1} cannot be neglected because Z_{k+1} is of size $n \times t$. In summary, as this method was not meant for efficiency, but rather for improving the stability of Orthomin, it does not allow to save as many flops as in the dynamic variant of Orthodir.

5.2.2 Cost of dynamic reduction of ECG

The implementation of the dynamic reduction of the search directions within Orthodir follows Algorithm 2. In practice, we use LAPACK routine `gesvd` and only compute the right singular vectors of α_k denoted U_k. We check the singular values obtained. If there are some smaller than $\frac{\varepsilon}{\sqrt{t}}$, which is the criterion proposed in [8], we call `geqrf` on U in order to perform the updates PU_k, APU_k and $U_k^\top \alpha_k$ in-place with `ormqr`. Since P_k and

Algorithm 2 ECG dynamic Orthodir variant.

```
Require: $A$, $R_0^e$, $k_{\max}$, $\varepsilon$, $\varepsilon_{\text{def}}$
Ensure: $||R_k|| < \varepsilon$ or $k = k_{\max}$
 1: $P_0 = 0$
 2: $Z_1 = R_0^e$
 3: $H = []$
 4: for $k = 1, \dots, k_{\max}$ do
 5:     $P_k = Z_k(Z_k^\top A Z_k)^{-1/2}$
 6:     $\alpha_k = P_k^\top R_{k-1}$
 7:     $\alpha_k = U_k \Sigma_k V_k^\top$
 8:     let $s_k$ be the number of singular values of $\alpha_k$ bigger than $\varepsilon_{\text{def}}$
 9:     if $s_k < s_{k-1}$ then
10:         $\alpha_k = U_k^\top \alpha_k$
11:         $P_k = P_k U_k$
12:         $\alpha_k = \alpha_k(1 : s_k, :)$
13:         $H = [H, P(:, s_k : s_{k-1})]$
14:         $P_k = P_k(:, 1 : s_k)$
15:     end if
16:     $X_k = X_{k-1} + P_k \alpha_k$
17:     $R_k = R_{k-1} - A P_k \alpha_k$
18:     if $|| \sum_{i=1}^{t} R_k^{(i)} ||_2 < \varepsilon$ then
19:         stop
20:     end if
21:     $\gamma_k = (AP_k)^\top (AP_k)$
22:     $\rho_k = (AP_{k-1})^\top (AP_k)$
23:     $\delta_k = (AH)^\top (AP_k)$
24:     $Z_{k+1} = AP_k - P_k \gamma_k - P_{k-1} \rho_k - H \delta_k$
25: end for
26: $x_k = \sum_{i=1}^{t} X_k^{(i)}$
```

AP_k are stored in a column major fashion the selection of the columns is done at no cost. Similarly H is not explicitly defined. However the selection of the first rows of α_k implies an in-place memory rearrangement.

The implementation of Breakdown-Free Orthomin is similar to Orthomin except the computation of a rank-revealing QR decomposition of Z_{k+1}. As Z_{k+1} is distributed, it is not reasonable to use a LAPACK kernel to compute it. Instead we use a modification of Chol-QR algorithm [20] which is a cheaper but less stable alternative to TS-RRQR [6, 5]. Its implementation is very easy using the LAPACK routine `pstrf` (Cholesky with pivoting) for computing (R, π) at line 2. Following [12] we use the default tolerance of `pstrf` for detecting exact rank deficiency of Z_{k+1}.

Algorithm 3 Chol-RRQR

```
Require: $P$, $\varepsilon$
Ensure: $Q_1$ orthogonal such that
    $$P\pi = \begin{pmatrix} Q_1 & Q_2 \end{pmatrix} \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$$
    where $\pi$ is a permutation and all the diagonal elements of $R_{22}$ are larger than $\varepsilon$
 1: $\mu = P^\top P$
 2: Compute $(R, \pi)$ such that $\pi^\top \mu \pi = R^\top R$ with $R = \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}$ and all the diagonal
    elements of $R_{22}$ are larger than $\varepsilon^2$
 3: $P_1 = P\pi(:, 1 : \text{size}(R_{11}))$
 4: $Q_1 = P_1 R_{11}^{-1}$
```

Algorithm 4 BF-ECG algorithm.

```
Require: $A$, $R_0^e$, $k_{\max}$, $\varepsilon_{\text{solver}}$, $\varepsilon_{\text{machine}}$
Ensure: $||R_k|| < \varepsilon_{\text{solver}}$ or $k = k_{\max}$
 1: $P_0 = 0$
 2: $Z_1 = R_0^e$
 3: for $k = 1, \ldots, k_{\max}$ do
 4:     $P_k = Z_k(Z_k^\top A Z_k)^{-1/2}$
 5:     $\alpha_k = P_k^\top R_{k-1}$
 6:     $X_k = X_{k-1} + P_k \alpha_k$
 7:     $R_k = R_{k-1} - AP_k \alpha_k$
 8:     if $|| \sum_{i=1}^{t} R_k^{(i)} ||_2 < \varepsilon$ then
 9:         stop
10:     end if
11:     $\beta_k = (AP_k)^\top R_k$
12:     $Z_{k+1} = R_k - P_k \beta_k$
13:     $Z_{k+1} = \text{Chol-RRQR}(Z_{k+1}, \varepsilon_{\text{machine}})$ (Algorithm 3)
14: end for
15: $x_k = \sum_{i=1}^{t} X_k^{(i)}$
```

5.3 Impact of threads on the performance of ECG

One motivation for enlarging the Krylov subspaces is to increase the arithmetic intensity of the resulting methods. This is particularly interesting to take advantage of the so-called manycore architecture as Nvidia GPUs, Intel Xeon Phi, or Sunway SW26010 used in the Sunway TaihuLight supercomputer. As the implementation relies on the MKL library which is multi-threaded [19], it is straightforward to assess its efficiency on the Xeon Phi processors.

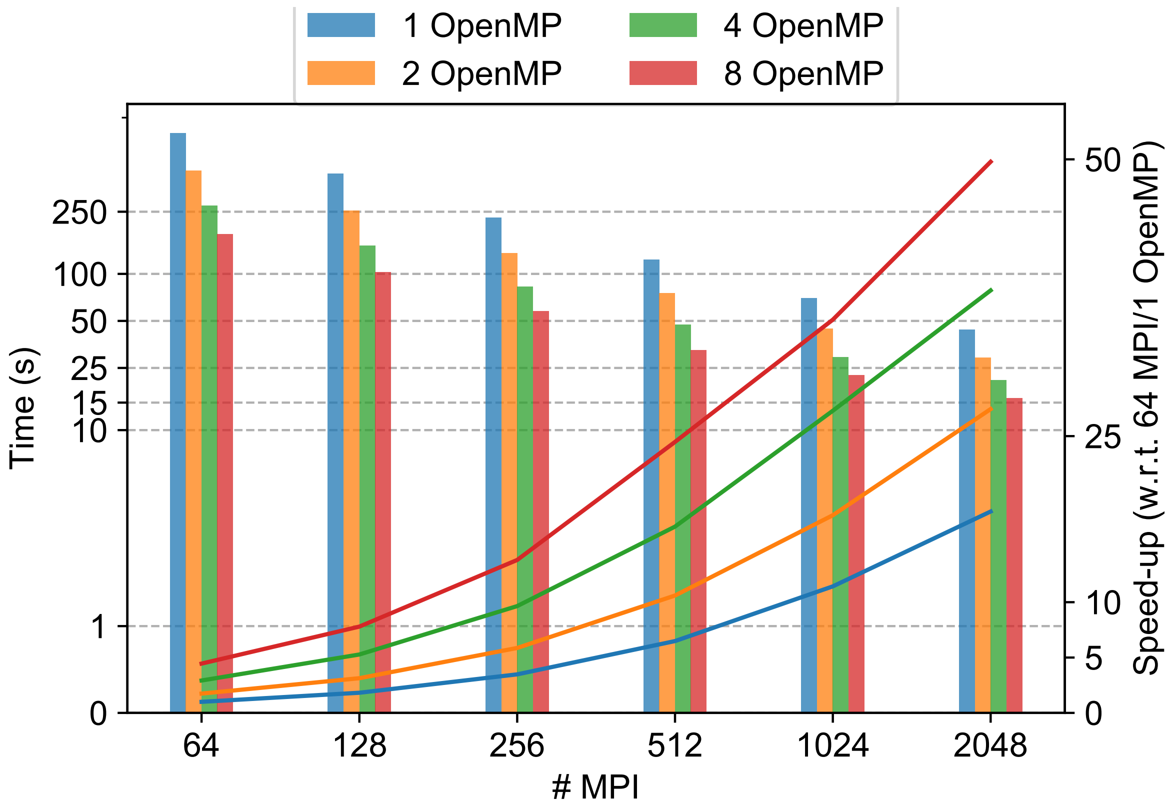
In order to do so, we perform the following experiments on NERSC's supercomputer Cori. It consists in two partitions, one with Intel Haswell processors and another one with the last generation of Intel Xeon Phi processors: Knights Landing (KNL). More precisely, the second partition consists in 9,688 single-socket Intel Xeon Phi 7250 (KNL) processors with 68 cores each. For a detailed description of the machine, we refer to the online documentation[1]. We compile the code (and its dependencies) using the default compilers and libraries installed on the machine: `version 18.0.1`, `cray-mpich` version 7.6.2, MKL version 2018.1.163 and METIS version 5.1.0. We consider Ela_30 test case and we study the impact of threads on the strong scaling of Orthodir(24). We do not use the dynamic reduction of the search directions in order to keep the cost of one iteration constant during the solve to better understand the effect of threading. We both increase the number of MPI processes from 64 to 2048 and the number of threads from 1 to 8 – this means at most $2,048 \times 8 = 16,384$ threads, each one being bound to one physical core.

The results obtained are summarized in Fig. 1a. First of all, we notice that there is a trade-off between using threads or MPI processes because the number of MPI processes dictates the preconditioner. Indeed, there are as many blocks in the block Jacobi preconditioner as the number of MPI processes, thus increasing the number of MPI processes deteriorates the quality of the preconditioner but also reduce its application cost. For instance, using 512 MPI processes takes 123s, and using 64 MPI processes with 8 threads each takes 179s: it is an increase of 50% compared to 512 MPI processes. Nevertheless, we observe that using more than 2 threads, and up to 8, has always a significant effect on the speed-up, even when the number of MPI processes is high. For instance, as shown in Table 1b, increasing the number of threads from 1 to 8 with a fixed number of $2,048$ MPI processes leads to an decrease in runtime of nearly 3. Of course, we are not close to full efficiency when using multiple threads, but we are still taking advantage of the BLAS 3 routines. This is illustrated by the Table 1b where we compare the speed-up obtained by using threads for Orthomin(1), which corresponds to the classical CG, and Orthodir(24). We observe that using more than 2 threads is not effective at all with Orthomin(1) whereas it always significantly increases the speed-up with Orthodir(24).

Finally, we are able to obtain an overall speed-up of 50 when using $16,384$ cores with respect to 64 cores. Compared to an ideal speed-up of 256, it may seem that this result is not very good (around 20% of efficiency), however it is well-known that Krylov methods may face efficiency issues at very large scale[2] — in practice, such difficulties are overcome by using preconditioning strategies well adapted to the underlying problem. Furthermore,

[1]`http://www.nersc.gov/users/computational-systems/cori/configuration/`

[2]This is well illustrated by HCG benchmark: `http://www.hpcg-benchmark.org/custom/index.html?lid=155&slid=293`



(a) The bars represent the runtime (left) and the lines represent the corresponding speed-up with respect to 64 MPI with 1 thread each (right).

	Orthomin(1)		Orthodir(24)	
# omp	time (s)	speed-up	time (s)	speed-up
1	89	-	44	-
2	74	1.2	29	1.5
4	80	1.1	21	2.1
8	79	1.1	16	2.8

(b) Comparison between Orthomin(1) and Orthodir(24) with 2048 MPI processes. We indicate the speed-up when increasing the number of threads for each method.

Figure 1: Strong scaling study for Ela_30 matrix on Cori (# omp stands for the number of threads assigned to each MPI process).

	dynamic Orthodir(24)		PETSc's CG		
# MPI	# iter	time (s)	# iter	time (s)	speed-up
252	513	77.9	13,626	406.8	5.2
504	531	45.5	15,819	258.9	5.7
1,008	606	23.7	17,023	94.7	4.0
2,016	696	14.5	19,047	34.5	2.5

Table 2: Strong scaling study for Ela_30 presented in Deliverable 4.4. The speed-up is the ratio between PETSc runtime and ECG runtime.

it is important to notice that the matrices tested are relatively small, but they allow us to simulate extreme scale situation: with $16,384$ cores the average number of unknowns per core is around 280. We have shown that enlarged Krylov subspace methods can both increase the arithmetic intensity and reduce the total number of iterations dramatically. As a result, the processors execute a higher flop rate and the need for interprocessor communication is reduced. Thus it takes advantage of the current trend in hardware architecture for reaching exascale.

5.4 Fusing global communications

If we break down the timings obtained during the strong scaling study showed in Deliverable 4.4: Performance evaluation (Table 2), we observe that the major bottleneck of ECG at large scale is the communication involved by the calls to `MPI_Allreduce` (Figure 2). In this section, we focus on Orthodir method and we explain how to reformulate the algorithm in order fuse these calls within an iteration.

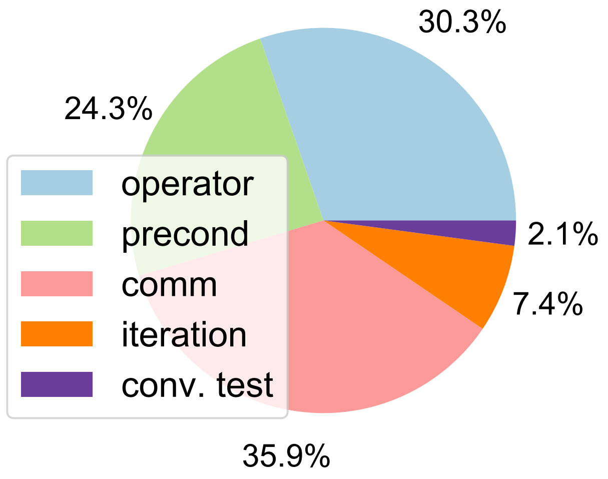


Figure 2: Comparison of the time spent in various steps of one iteration of dynamic Orthodir(24) for the Ela_30 matrix with 2016 MPI processes on Kebnekaise.

5.4.1 Derivation of the algorithm

In what follows, μ_k, α_k, β_k, and ξ_k are not necessarily real numbers, but typically denote matrices that have dimensions substantially smaller than the original matrix A. If we decompose one iteration of Orthodir there are 4 synchronizations,

$$\mu_k = Z_k^\top A Z_k, \tag{5.15}$$
$$\alpha_k = P_k^\top R_{k-1}, \tag{5.16}$$
$$\beta_k = \begin{pmatrix} AP_k & AP_{k-1} \end{pmatrix}^\top M^{-1} AP_k, \tag{5.17}$$
$$\xi_k = R_k^\top R_k. \tag{5.18}$$

Furthermore, we have $P_k = Z_k \mu_k^{-1/2}$. Thus we can reformulate α_k, and β_k,

$$\alpha_k = \mu_k^{-\top/2} Z_k^\top R_{k-1}, \tag{5.19}$$
$$\beta_k = \begin{pmatrix} AZ_k\mu_k^{-1} & AP_{k-1} \end{pmatrix}^\top M^{-1} AZ_k \mu_k^{-1/2}. \tag{5.20}$$

If we postpone the convergence test, i.e., we compute ξ_{k-1} instead of ξ_k, we can compute the following quantities at the same time,

$$\mu_k = Z_k^\top A Z_k, \tag{5.21}$$
$$Z_k^\top R_{k-1}, \tag{5.22}$$
$$\begin{pmatrix} AZ_k & AP_{k-1} \end{pmatrix}^\top M^{-1} AZ_k, \tag{5.23}$$
$$R_{k-1}^\top R_{k-1}, \tag{5.24}$$

and then it is possible to update α_k, β_k, P_k, and AP_k using the Cholesky factorization of μ_k.

The resulting Algorithm 5 is a *fused* version of the preconditioned Orthodir method. The volume of communications remains constant but the different synchronizations performed at each iterations are fused in order to have only one synchronization (`MPIAllreduce` call) per iteration. Unlike *Pipelined* methods Algorithm 5 does not overlap this global communication with computations.

Algorithm 5 Iteration of preconditioned fused Orthodir

```
1: $Q(:, 1 : t) = A * P(:, 1 : t)$
2: $Z = M^{-1}Q(:, 1 : t)$
3: // Beginning of synchronization
4: $\mu = P(:, 1 : t)^\top Q(:, 1 : t)$
5: $\alpha = P(:, 1 : t)^\top R$
6: $\beta = Q^\top Z$
7: $\xi = R^\top R$
8: // End of synchronization
9: $\mu = \texttt{chol}(\mu)$
10: $P(:, 1 : t) = P(:, 1 : t)\mu^{-1}$
11: $Q(:, 1 : t) = Q(:, 1 : t)\mu^{-1}$
12: // Beginning of additional triangular solves
13: $\alpha = \mu^{-\top}\alpha$
14: $\beta = \beta\mu^{-1}$
15: $\beta(1 : t, :) = \mu^{-\top}\beta(1 : t, :)$
16: $Z = Z\mu^{-1}$
17: // End of additional triangular solves
18: $X = X + P(:, 1 : t)\alpha$
19: $R = R - Q(:, 1 : t)\alpha$
20: if $\sum_{i=1}^{t} \xi(i, i) < \varepsilon$ then
21:     stop
22: end if
23: $Z = Z - P\beta$
24: $P(:, t + 1 : 2t) = P(:, 1 : t)$
25: $P(:, 1 : t) = Z$
26: $Q(:, t + 1 : 2t) = Q(:, 1 : t)$
```

It is straightforward to derive a *fused* version of dynamic Orthodir algorithm where the number of search directions is reduced dynamically during the iterations. Indeed, we have,

$$Z_{k+1} = AP_{k,1} - \begin{pmatrix} P_{k,1} & P_{k-1,1} & H \end{pmatrix} \beta_{k,1}, \tag{5.25}$$

where

$$\beta_{k,1} = \begin{pmatrix} P_{k,1}^\top AM^{-1}AP_{k,1} \\ P_{k-1,1}^\top AM^{-1}AP_{k,1} \\ H^\top AM^{-1}AP_{k,1} \end{pmatrix}.$$

Thus, $\beta_{k,1}$ is obtained by updating locally β_k,

$$\beta_{k,1} = \begin{pmatrix} U_{k,1}^\top & & \\ & I & \\ & & I \end{pmatrix} \beta_k U_{k,1}, \tag{5.26}$$

where $U_{k,1}$ denotes the left singular vectors of α_k that are kept in its low-rank approximation (see Deliverable 4.2: Analysis and algorithm design).

Remark 1. *In our implementation, we construct Z_{k+1} using β_k and then reduce its size by multiplying it with $U_{k,1}$, i.e, $Z_{k+1}U_{k,1}$. It is not as optimal as the previous discussion, but it is simpler to implement and the extra flops are expected to be small with respect to the overall iteration cost.*

5.4.2 Cost analysis

Given n, t such that $t \ll n$, we denote V, W tall and skinny matrices (tsm) of size $n \times t$ whose rows are distributed among the processors, and α is a matrix of size $t \times t$ replicated on the P processors. Following [15], it is possible to decompose the iterations of ECG (and more generally block CG) into the following kernels:

- $V \leftarrow V + W\alpha$ (`tsmm` in [15]),
- $\alpha \leftarrow V^\top W$ (`tsmtsm` in [15]),
- Cholesky factorization of α (`potrf`),
- triangular solve of α with several right-hand sides (`trsm`).

Following Algorithm 5, each iteration of the fused Orthodir variant consists of 3 `tsmm` (lines 18, 19, and 23), 1 `tsmtsm` (lines 4–7), 1 `potrf` (line 9) and 6 `trsm` (lines 10, 11, and 13–16). More precisely, the LAPACK routine `gemm` is called to compute the local products lines 4–7, and the resulting (of dimension of the order $t \times t$) matrices are put contiguously in the memory so that the reduction is done using one call to `MPI_Allreduce`. The lines 13–16 consists of the additional updates which do not involve any communication. In summary and using the previous discussion in Deliverable 4.4: Performance evaluation, we have,

$$\#flops(\text{fused Orthodir}) = 4 \times 2\frac{nt^2}{P} + 5 \times \frac{nt(2t+1)}{P} + \frac{1}{3}t^3 + 6 \times \frac{nt^2}{P}, \tag{5.27}$$

$$= 24\frac{nt^2}{P} + 5\frac{nt}{P} + \frac{1}{3}t^3. \tag{5.28}$$

As previously, matrices of size $t \times t$ are replicated among the processors, thus `tsmm`, Cholesky factorization of α and triangular solve of α are local operations without any communication. Thanks to the reformulation of the algorithm, the only communication phase occurs at lines 4–7.

In summary, the detailed costs of one iteration of Orthodir, and fused Orthodir in terms of flops, words, and messages are indicated in Table 3. For the sake of comparison, we recall the complexity of the CG algorithm described in [18]. We also report the number of `MPI_Allreduce` in parenthesis, in addition to the order of magnitude of the number of messages. In summary, one iteration of fused Orthodir involves 20% more flops — neglecting the applications of the matrix, and the preconditioner — but it reduces the number of messages by factor 4, with respect to Orthodir. fused Orthodir also reduces the number of messages by a factor 2 with respect to the usual CG method.

In Table 3, we also compare the memory requirements of the Orthodir method and the Fused Orthodir method. The fused variant requires an extra $2t^2$ words of memory

	# flops	# messages	# words	memory
Orthodir	$20\frac{nt^2}{P} + 5\frac{nt}{P} + \frac{1}{3}t^3$	$4\log_2(P)$ (4)	$5t^2$	$7\frac{nt}{P} + 3t^2$
fused Orthodir	$24\frac{nt^2}{P} + 5\frac{nt}{P} + \frac{1}{3}t^3$	$\ln(P)$ (1)	$5t^2$	$7\frac{nt}{P} + 5t^2$
CG	$10\frac{n}{P}$	$2\log_2(P)$ (2)	2	$5\frac{n}{P}$

Table 3: Complexity of Orthodir, fused Orthodir and CG where t is the enlarging factor.

because it has to hold the results of the four reductions given in (5.21)–(5.24). However, this memory overhead is not significant with respect to the overall memory consumption of both methods ($\mathcal{O}(\frac{nt}{P})$).

5.5 Numerical experiments

In order to evaluate the gain of this reformulation, we perform several experiments both on Kebnekaise and Cori.

5.5.1 Kebnekaise

The following experiments are performed on Kebnekaise. We use the same parameters and test cases as in Deliverable 4.4. In particular, the tolerance is set to 10^{-5}, and we use a block Jacobi preconditioner where each MPI process factorizes its corresponding diagonal block of the matrix A, and then performs a forward and backward substitution at each iteration. We do not remake the whole parameter study (see Deliverable 4.4) but rather compare the performance of the fused algorithms against the non-fused ones, and with PETSc's CG, when the number of MPI processes is relatively high.

		fused dynamic Orthodir		dynamic Orthodir		PETSc's CG
Case	# MPI	comm	total	comm	total	total
Hook_1498	1,008	1.1	1.6	1.4	2.1	0.9
	2,016	0.9	1.2	1.2	1.5	0.9
	3,024	0.7	0.9	2.0	2.3	1.0
Flan_1565	1,008	1.9	3.8	2.6	4.1	4.9
	2,016	1.3	2.2	2.6	3.6	2.6
	3,024	1.4	1.9	2.2	2.9	1.6
Ela_30	1,008	11.1	21.7	12.2	23.7	94.7
	2,016	6.9	13.0	8.2	14.5	34.5
	3,024	8.0	11.9	7.7	11.6	19.5

Table 4: Timings in seconds of the fused dynamic Orthodir variant with dynamic Orthodir and PETSc's CG. The enlarging factor t is set to 8 for Hook, 12 for Flan, and 24 for Ela_30, and "comm" stands for the time spend in the global communication during the iterations of ECG.

Figure 3: Numerical comparison between dynamic Orthodir and the fused dynamic Orthodir variant on Flan_1565. In both cases, the horizontal axe represent the iteration count, and the title correspond to the vertical axe (right: normlized residual, left: number of search directions). The number of MPI processes is 1008 and the enlarging factor t is set to 12. As expected, the two plots coincide perfectly.

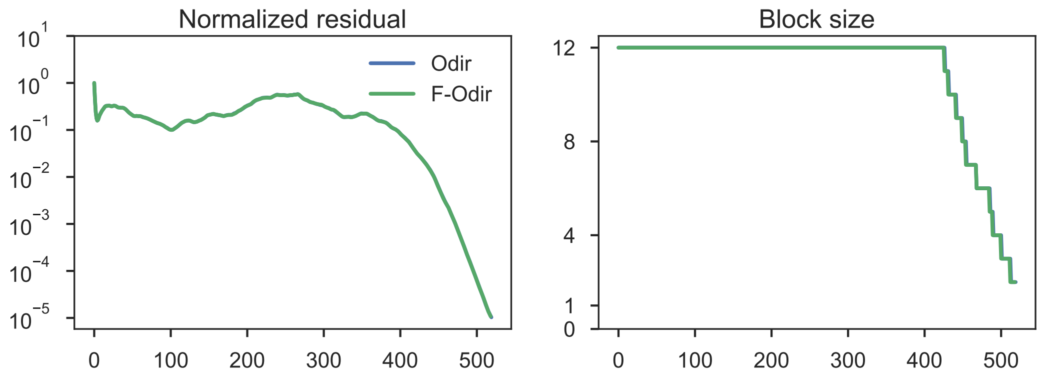
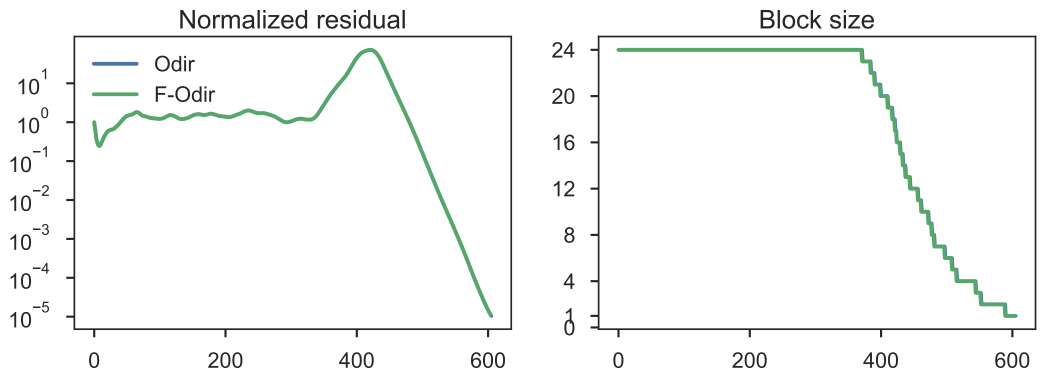


Figure 4: Numerical comparison between dynamic Orthodir and the fused dynamic Orthodir variant on Ela30. In both cases, the horizontal axe represent the iteration count, and the title correspond to the vertical axe (right: normlized residual, left: number of search directions). The number of MPI processes is 1008 and the enlarging factor t is set to 24. As expected, the two plots coincide perfectly.



In Table 4, we summarize the results obtained when we compare the fused dynamic Orthodir method with dynamic Orthodir, and PETSc's CG. More precisely, we compare these 3 methods on Hook_1498, Flan_1565, and Ela_30 respectively using the same parameters as in the strong scaling study, and $t = 8$ for Hook, 12 for Flan, and 24 for Ela_30. Thus in any case the fused version is faster than the non–fused one. Indeed, it does not suffer from numerical instabilities and the number of iterations remains exactly the same (see Figures 3 and 4) but the time spent in the communication is reduced. For example, with the largest number of MPI processes, the fused variant is more than two times faster than the non–fused variant for the Hook test case, and around 1.5 times faster than the non–fused variant for the Flan test case. Depending on the test case, fusing the global communications even allows us to be slightly faster than PETSc's CG at large scale; this is the case for Hook, but not for Flan.

In summary, fusing the global communications almost always leads to a decrease in the runtime with respect to the non–fused variant. This decrease is significant at large scale because the communication phase usually dominates the overall runtime. Furthermore, the fused variant can also be slightly better than PETSc's CG when the MPI processes count is the highest (3024).

5.5.2 Cori

We now make the exact same experiments on Cori using KNL processors: both the matrices and the parameters are the same as in section 5.5.1. In all the experiments we use 2 threads per MPI processes, each one being bound to one physical core. It means that the total number of processors used is # MPI $\times 2$. Once again, we are interested in comparing the performance of the fused algorithms against the non-fused ones, and with PETSc's CG, when the number of MPI processes is relatively high: starting to $2,048$, and up to $16,385$ — meaning up to $32,768$ cores at the largest scale.

		fused dynamic Orthodir		dynamic Orthodir		PETSc's CG
Case	# MPI	comm	total	comm	total	total
Hook_1498	2,048	1.6	3.0	2.3	3.8	3.4
	4,096	1.8	2.8	2.7	3.8	3.4
	8,192	0.8	1.4	1.4	2.2	1.6
Flan_1565	2,048	2.0	5.5	3.0	6.4	5.5
	4,096	1.6	3.7	2.6	4.9	3.7
	8,192	1.4	2.9	2.4	4.1	3.2
Ela_30	2,048	18.2	32.4	24.6	39.0	151.7
	4,096	6.5	14.3	8.3	16.2	53.7
	8,190	5.1	11.2	7.0	13.5	34.6
	16,384	4.0	8.0	6.3	10.1	31.7

Table 5: Timings in seconds of the fused dynamic Orthodir variant with dynamic Orthodir and PETSc's CG. The enlarging factor t is set to 8 for Hook, 12 for Flan, and 24 for Ela_30, and "comm" stands for the time spend in the global communication during the iterations of ECG.

We summarize the results we obtained in Table 5. As before, we compare the fused dynamic Orthodir method with dynamic Orthodir and PETSc's CG on Hook_1498, Flan_1565, and Ela_30. For Cori is a much larger machine than Kebnekaise, we use a larger number of MPI processes in these experiments. For the Hook test case, we observe that the fused variant is always faster than PETSc, unlike the non–fused. Indeed, the communication time is drastically reduced: up to a factor almost 2 for the largest count of MPI processes we considered. For the Flan matrix, the same behavior is observed. The fused variant allows to obtain a speed–up for the largest count of MPI processes considered with respect to PETSc's CG. Thus it scales slightly better than PETSc's CG. Eventually for the Ela_30 test case, we observe a significant improvement of the scalability on the largest count of MPI processes. The fused dynamic Orthodir variant is indeed 33% faster using $16,384$ MPI processes than when using $8,192$. On the other hand, PETSc's CG is around 8% times faster using $16,384$ MPI processes than when using $8,192$. As a consequence, the fused dynamic Orthodir method is almost 4 times faster than PETSc's CG when using $16,384 \times 2 = 32,768$ physical processors. In summary, we observe that the fused variant is always faster that the non–fused one. However, on this machine the fused variant is at least as fast (for the Flan test case) as PETSc's CG when the number of MPI processes is relatively low ($2,048$ and $4,096$), but slightly faster than PETSc when using more than $8,192$ MPI processes. Thus the fused, dynamic Orthodir method scales better than PETSc's CG up to $16,384$ cores on this machine

6 Acknowledgments

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

References

[1] 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.

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

[3] S. F. Ashby, T. A. Manteuffel, and P. E. Saylor. A taxonomy for conjugate gradient methods. *SIAM Journal on Numerical Analysis*, 27(6):1542–1568, 1990.

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

[5] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing*, 34(1):A206–A239, 2012.

[6] J. W. Demmel, L. Grigori, M. Gu, and H. Xiang. Communication avoiding rank revealing QR factorization with column pivoting. *SIAM Journal on Matrix Analysis and Applications*, 36(1):55–89, 2015.

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

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

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

[10] M. H. Gutknecht. Block Krylov space methods for linear systems with multiple right-hand sides: an introduction. *in: Modern Mathematical Models, Methods and Algorithms for Real World Systems (A.H. Siddiqi, I.S. Duff, and O. Christensen, eds.)*, pages 420–447, 2007.

[11] Dongarra J., Eijkhout V., and Kalhan A. Reverse communication interface for linear algebra templates for iterative methods. *UT, CS-95-291, May*, 1995.

[12] H. Ji and Y. Li. A breakdown-free block conjugate gradient method. *BIT Numerical Mathematics*, 57(2):379–403, Jun 2017.

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

[14] 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.

[15] M. Kreutzer, J. Thies, M. Röhrig-Zöllner, A. Pieper, F. Shahzad, M. Galgon, A. Basermann, H. Fehske, G. Hager, and G. Wellein. GHOST: Building blocks for high performance sparse linear algebra on heterogeneous systems. *International Journal of Parallel Programming*, 45(5):1046–1072, Oct 2017.

[16] 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.

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

[18] Y. Saad. *Iterative methods for sparse linear systems*, volume 82. siam, 2003.

[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.

[20] I. Yamazaki, S. Tomov, and J. Dongarra. Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multiple GPUs. *SIAM Journal on Scientific Computing*, 37(3):C307–C330, 2015.