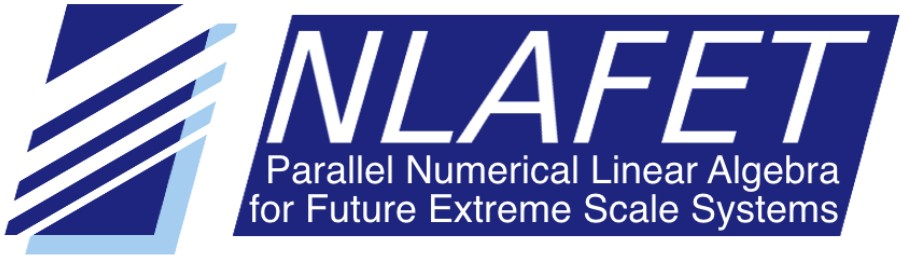

H2020–FETHPC–2014: GA 671633

D4.4

Performance evaluation

April 2018

DOCUMENT INFORMATION

Scheduled delivery	2018-04-30
Actual delivery	2018-04-27
Version	1.1
Responsible partner	INRIA

DISSEMINATION LEVEL

CO — Confidential (until June 15; after that Public)

REVISION HISTORY

Date	Editor	Status	Ver.	Changes
2018-4-13	INRIA members	Draft	1.0	Initial version of document produced
2018-4-26	INRIA members	Draft	1.1	Final version of document produced

AUTHOR(S)

Simplice Donfack, Laura Grigori, Olivier Tissot, INRIA

INTERNAL REVIEWERS

Carl Christian Kjelgaard Mikkelsen, UMU

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 deliverable 4.3, we have introduced LORASC and enlarged CG (ECG), two new approaches that aim at addressing the scalability issue of Krylov subspace solvers. LORASC is based on a domain decomposition method and a low rank approximation of the Schur-complement obtained by solving a generalized eigenvalue problem, while ECG is based on enlarging the Krylov subspace by a maximum of t vectors per iteration.

In this deliverable we discuss the communication complexity of ECG and LORASC. We also introduce a multilevel approach for LORASC that improves its scalability. We then evaluate their performance on challenging test cases. Our experiments show that these algorithms are scalable and outperform both the single-level of parallelism of LORASC and Block Jacobi preconditioner. Both implementations are available in preAlps software.

2 Introduction

The *Description of Action* document states for Deliverable 4.2:

"*Evaluation of the communication complexity and parallel performance of the prototypes from D4.3.*"

This deliverable is in the context of Task 4.2 (Iterative methods) and Task 4.3 (Preconditioners). It discusses communication avoiding Krylov subspace methods for solving large sparse linear systems of equations $Ax = b$, where A is a symmetric positive definite (SPD) matrix. In particular, it considers the iterative method enlarged CG (ECG) and the robust algebraic preconditioner LORASC discussed in Deliverable 4.3.

We first discuss two variants of ECG, based on two different recurrence formulas, Orthomin and Orthodir. We then discuss the parallel design of ECG, its communication complexity while also dynamically reducing the number of search directions added at each iteration of the Krylov method.

We then discuss a multilevel approach of the LORASC preconditioner. Although LORASC preconditioner is efficient on a small number of processors, its scalability is affected by the time required to solve a generalized eigenvalue problem which grows linearly with the number of processors. In this deliverable, we describe a multilevel approach that allows to reduce significantly the solving time of the generalized eigenvalue problem.

We then evaluate their performance on challenging test cases. Our experiments show that these algorithms are scalable and outperform both the single-level LORASC and CG preconditioned by Block Jacobi as implemented in Petsc. Both implementations are available in the preAlps library.

3 Enlarged Conjugate Gradient

In this section, we recall the derivation of the Enlarged Conjugate Gradient (ECG) method presented in the *Deliverable 4.2 - Analysis and algorithm design*.

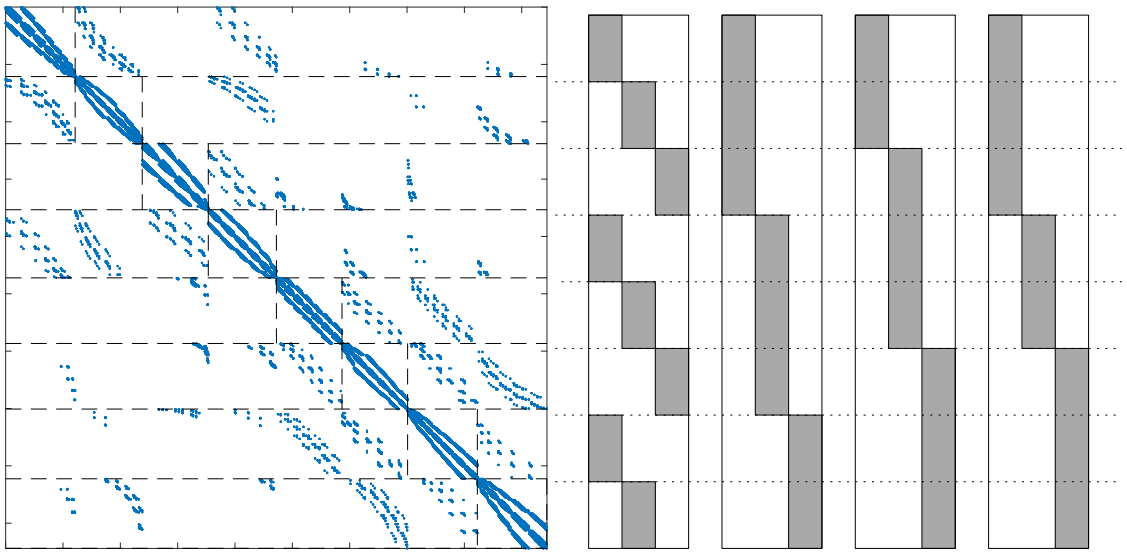

Figure 1: Illustration of the ordering of A into 8 subdomains obtained with METIS [16] and several admissible splittings of r_0 into 3 vectors.

3.1 Enlarged Krylov subspaces

In [10], the authors define so-called enlarged Krylov subspaces. First, the matrix A is reordered by partitioning its graph into $\mathcal{N}$ subdomains (using METIS [16] for example). Then, the initial residual r_0 is split into t vectors denoted $R_0^{e(i)}$, $1 \leq i \leq t$. In the original paper the authors use $t = \mathcal{N}$. It is important to notice that the case $t < \mathcal{N}$ can be dealt in many ways provided $r_0 = \sum_{i=1}^{t} R_0^{e(i)}$ (Figure 1). This is of interest in practice because typically $\mathcal{N}$ will correspond to the number of MPI processes. The parameter t is called the enlarging factor. In practice for a given t the splitting of r_0 does not have a high impact on the convergence of the method. In the numerical experiments we construct the initial enlarged residual $R_0^e = [R_0^{e(1)}, \ldots, R_0^{e(t)}]$ as the leftmost example in Figure 1.

Then, the enlarged Krylov subspace of order k denoted $\mathcal{K}_{k,t}(A, r_0)$ is defined as the block Krylov subspace of order k associated to A and the enlarged residual R_0^e. More precisely, and following the notation introduced in [11],

$$\mathcal{K}_{k,t}(A, r_0) := \mathcal{K}_k^{\square}(A, R_0^e) \tag{3.1}$$

$$= \text{span}^{\square}\{R_0^e, AR_0^e, \ldots, A^{k-1}R_0^e\}. \tag{3.2}$$

Using this definition and following [9] it is possible to derive two variants (Orthomin and Orthodir) of enlarged Conjugate Gradient (ECG) algorithm (Figure 1). More precisely, the enlarged approximate solution is a matrix of size $n \times t$ denoted X_k, and the sum of its column gives the approximate solution of the original system. We denote R_k the enlarged approximate residual, and similarly we obtain the approximate residual of the original system by summing its columns. P_k is a matrix of size $n \times t$ called search directions, it corresponds to the A-orthonormalization of Z_k. We want to point out that α_k is a matrix of size $t \times t$, whereas it usually denotes a scalar in the classical CG algorithm. Depending on the method for constructing Z_{k+1}, it is possible to derive two variants of ECG: Orthomin and Orthodir.

Orthomin (Omin) corresponds to Block CG [21]:

$$\beta_k = (AP_k)^\top R_k, \tag{3.3}$$

$$Z_{k+1} = R_k - P_k\beta_k. \tag{3.4}$$

This method is very similar to the one originally proposed by *Hestenes and Stiefel* [12] because it constructs the new descent directions Z_{k+1} using R_k and P_k.

Orthodir (Odir) corresponds to the Block Lanczos algorithm but with the inner product induced by A:

$$\gamma_k = (AP_k)^\top (AP_k), \tag{3.5}$$

$$\rho_k = (AP_{k-1})^\top (AP_k), \tag{3.6}$$

$$Z_{k+1} = AP_k - P_k\gamma_k - P_{k-1}\rho_k. \tag{3.7}$$

It is the block equivalent of the homonym method defined in [2]. Unlike the previous variant, Z_{k+1} is constructed using P_k and P_{k-1}.

Both Orthodir and Orthomin produce Z_{k+1} that is A-orthogonal to P_i for $i \leq k$. Then the search directions P_{k+1} are defined as

$$P_{k+1} = Z_{k+1}(Z_{k+1}^\top AZ_{k+1})^{-1/2}. \tag{3.8}$$

Unlike CG algorithm a breakdown would occur if $Z_{k+1}^\top AZ_{k+1}$ is singular, *i.e.*, Z_{k+1} is not full rank. Although rare this situation can happen in practice and several variants have been developed in order to handle this case [6, 9, 14, 21]. Overall, both Orthomin and Orthodir generate P_{k+1} such that

$$P_{k+1}^\top AP_i = 0, \forall\ i \leq k \tag{3.9}$$

$$P_{k+1}^\top AP_{k+1} = I. \tag{3.10}$$

Consequently, the ECG method can be summarized in Algorithm 1. Another difference between ECG and the original block CG algorithm is that the search directions are A-orthonormalized at each iteration: P_k is used as search directions instead of Z_k. And it has be shown numerically that using this variant can increase the numerical stability of the method [6].

As explained in [9] it is possible to reduce the block size during the iterations of ECG. In general, the key idea is to monitor the rank of R_k [11] but 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 [9], it is shown that the rank of $\alpha_k = P_k^\top R_{k-1}$ can be computed instead. The resulting method is called D-Odir and is explained in more details in the *Deliverable 4.2* and in [9].

3.2 Parallel design

3.2.1 Data distribution

As it is usually the case in parallel implementations of Krylov methods, we assume that the unknowns are distributed among the processors. We also assume that each processor owns different unknowns. Thus all the variables whose size scales as the size of the linear system (X_k, R_k, P_k, AP_k, Z_k) are distributed row wise among the processors according to the distribution of the unknowns. And all variables whose size scales as the enlarging factor t (α_k, β_k,

$P_0 = 0$
$Z_1 = R_0^e$
$k = 1$
for $k = 1, \ldots, k_{\max}$ **do**
    $P_k = Z_k (Z_k^\top A Z_k)^{-1/2}$
    $\alpha_k = P_k^\top R_{k-1}$
    $X_k = X_{k-1} + P_k \alpha_k$
    $R_k = R_{k-1} - A P_k \alpha_k$
    if $|| \sum_{i=1}^{t} R_k^{(i)} ||_2 < \varepsilon$ **then**
        stop
    end if
    construct Z_{k+1} using (3.3)-(3.4) (Orthomin) or (3.5)-(3.7) (Orthodir)
    $k = k + 1$
end for
$x_k = \sum_{i=1}^{t} X_k^{(i)}$

Algorithm 1: ECG algorithm.

γ_k, ρ_k) are replicated on all the processors. Locally they are stored contiguously and column by column (Figure 2). There is no allocation or deallocation of memory during the iterations. In particular, when using Dynamic Orthodir or Breakdown-Free Orthomin the memory is not freed when the block size is reduced. The local memory consumption of preconditoned Orthodir and Orthomin on P processors is summarized in Table 1. For completeness, we also add the local memory consumption of the classical CG algorithm, described in [22] for instance, where only 5 vectors and 2 scalars are needed.

3.2.2 Cost analysis of ECG

Our implementation of ECG is based on Reverse Communication Interface [13] which is the standard procedure for Krylov solvers implementation. For one iteration of ECG, it requires external routines to apply the sparse matrix product and the preconditioner to a set of vectors. Indeed the implementation of these routines highly depends on the linear system to be solved. This is why we do account for these operations in our cost analysis.

Given n, t such that $t \ll n$, we denote V, W tall and skinny matrices 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 [18], 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 [18]),
- $\alpha \leftarrow V^\top W$ (`tsmtsm` in [18]),
- Cholesky factorization of α (`potrf`),
- triangular solve of α with several right-hand sides (`trsm`).

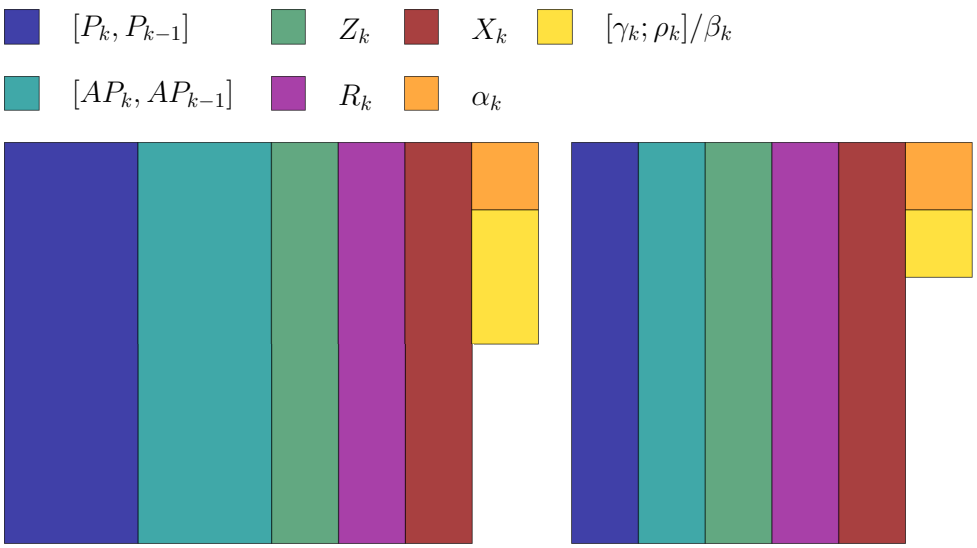

Figure 2: Local distribution of the data: Orthodir on the left and Orthomin on the left.

Following ECG algorithm (Figure 1), each iteration of Orthodir and Orthomin consists of 3 `tsmm` (lines 7, 8, and 12), 4 `tsmtsm` (lines 5, 6, 9, and 12), 1 `potrf` (line 5) and 2 `trsm` (line 5). Indeed, the line 5 of the algorithm (Figure 1) can be decomposed as,

1	$AZ_k \leftarrow A * Z_k$	*sparse matrix times set of vectors*
2		
3	$C \leftarrow$ `tsmtsm`(Z_k, AZ_k)	*form* $Z_k^\top AZ_k$
4	$C \leftarrow$ `potrf`(C)	*Cholesky factorization*
5	$P_k \leftarrow$ `trsm`(Z_k, C)	
6	$AP_k \leftarrow$ `trsm`(AZ_k, C)	*update* P_k *and* AP_k

Doing so allows us to avoid calling the sparse matrix set of vectors product for computing AP_k at the price of an extra `trsm`. Hence the difference between the two algorithms is the construction of Z_{k+1} (line 12 of algorithm 1). The `tsmtsm` and `tsmm` for constructing Z_{k+1} in Orthodir (equations (3.5)-(3.7)) are twice as much costly as for Orthomin (equations (3.3)-(3.4)).

As matrices of size $t \times t$ are replicated among the processors, we notice that `tsmm`, Cholesky factorization of α and triangular solve of α are local operations without any communication. Hence we use the corresponding LAPACK routines: `gemm`, `potrf` (dense Cholesky factorization) and `trsm` (dense triangular solve with several right-hand sides). However V and W are distributed and `tsmtsm` is not a local operation. The LAPACK routine `gemm` is called to compute the local product $V_i^\top W_i$ followed by a call to `MPI_Allreduce`.

Thus, the only kernel operation that requires a communication is `tsmtsm` and 4 calls to `MPI_Allreduce` are done per iteration. It is usually assumed that during a call to `MPI_Allreduce` the number of messages sent and received on the network is equal to $\log_2(P)$ – although the exact number depends on the MPI implementation [23]. Moreover it is a blocking operation: when completed all the processors are synchronized. This is why in practice, as in the classical CG, the communication cost is dominated by 2 calls to `MPI_Allreduce`: the one after the

	# flops	# messages	# words	memory
Omin	$16\frac{nt^2}{P}+4\frac{nt}{P}+\frac{1}{3}t^3$	$4\log_2(P)$ (4)	$4t^2$	$5\frac{nt}{P}+2t^2$
Odir	$20\frac{nt^2}{P}+5\frac{nt}{P}+\frac{1}{3}t^3$	$4\log_2(P)$ (4)	$5t^2$	$7\frac{nt}{P}+3t^2$
CG	$10\frac{n}{P}$	$2\log_2(P)$ (2)	2	$5\frac{n}{P}$

Table 1: Complexity and memory consumption per processor and per iteration of Orthodir, Orthomin and CG where t is the enlarging factor, n is the number of rows of A and P is the number of processors. In parenthesis is indicated the number of calls to `MPI_Allreduce`.

sparse matrix set of vectors product (line 5) and the one after the preconditioner (line 12), because they occur after operations with a potential load imbalance between processors.

In summary, the detailed costs of one iteration of Orthodir and Orthomin in terms of flops, words, and messages are indicated in Table 1. For the sake of comparison, we recall the complexity of the CG algorithm described in [22]. 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 ECG is approximately t^2 times more costly in terms of flops than one iteration of CG. While the number of messages is of the same order, the number of words is also t^2 times larger. Indeed there is a trade-off between these extra costs and the reduction of the number of iterations due to the enlargement of the search spaces.

4 LORASC

4.1 LORASC 1-level

In this section, we recall LORASC as presented in Deliverable 4.3, we analyze its complexity, and we discuss the generalized eigenvalue problem that needs to be soved for constructing LORASC.

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

$$A=\begin{pmatrix} A_{11} & & & A_{\Gamma 1}^T \\ & \ddots & & \vdots \\ & & A_{NN} & A_{\Gamma N}^T \\ A_{\Gamma 1} & \cdots & A_{\Gamma N} & A_{\Gamma\Gamma} \end{pmatrix}. \tag{4.1}$$

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

The LORASC preconditioner of A is defined as

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

where $\tilde{D} = \text{Block}-\text{Diag}(A_{11}, A_{22}, ..., A_{NN}, \tilde{S})$ *and* L *is the block lower triangular part of matrix* A. *The matrix* $\tilde{S}$ *is defined as*

$$\tilde{S}^{-1} = A_{\Gamma\Gamma}^{-1} + E\Sigma E^T, \tag{4.2}$$

where E, Σ *are defined as*

$$E = (v_1 \quad v_2 \quad ... \quad v_i), \tag{4.3}$$

$$\Sigma = \text{Diag}(\sigma_1, \sigma_2, \ldots, \sigma_i), \quad \text{with } \sigma_k = \frac{\varepsilon - \lambda_k}{\lambda_k},\ k \in \{1, 2, ..., i\}. \tag{4.4}$$

The parallel construction of LORASC using N processors is summarized in Algorithm 2.

Require: SPD matrix A, N processors, subset of N_Γ processors, tolerance ε

1: Partition the graph of A into N disjoint domains using k-way partitioning with vertex separators, and permute the matrix into a block arrow structure as

$$A = \begin{pmatrix} A_{11} & & & A_{\Gamma 1}^T \\ & \ddots & & \vdots \\ & & A_{NN} & A_{\Gamma N}^T \\ A_{\Gamma 1} & \cdots & A_{\Gamma N} & A_{\Gamma\Gamma} \end{pmatrix}. \tag{4.5}$$

2: Distribute A on N processors such that processor j holds $A_{\Gamma j}$ and A_{jj}, distribute $A_{\Gamma\Gamma}$ on N_Γ processors by using a block row distribution

3: Processors $j = 1, \ldots, N$ compute locally the Cholesky factorization of A_{jj}

4: A subset of N_Γ processors compute the Cholesky factorization of $A_{\Gamma\Gamma}$ in parallel

5: All processors solve the generalized eigenvalue problem $Su = \lambda A_{\Gamma\Gamma} u$ and compute the eigenvalues smaller than ε and associated eigenvectors using PARPACK, where $S = A_{\Gamma\Gamma} - \sum_{j=1}^{N} A_{\Gamma j} A_{jj}^{-1} A_{\Gamma j}^T$ (the result is returned on processor 0, the shifted eigenvalues are stored in Σ and the eigenvectors in E)

Ensure: $M = (L + \tilde{D})\tilde{D}^{-1}(\tilde{D} + L^T)$ where $\tilde{D} = \texttt{Block-Diag}(A_{11}, A_{22}, ...A_{NN}, \tilde{S})$ and $\tilde{S}^{-1} = A_{\Gamma\Gamma}^{-1} + E\Sigma E^T$

Algorithm 2: Parallel construction of LORASC using N processors

4.2 Computational cost of constructing LORASC 1-level

We model the computational cost of the construction of LORASC in parallel on N processors for SPD matrices resulting from the discretization of a PDE on regular three dimensional grids. We consider here the case in which the graphs of these matrices are defined on an $n^{1/3} \times n^{1/3} \times n^{1/3}$ grid with n vertices, and each vertex is connected to its nearest 6 neighbours. We recall first results for direct methods of factorization of matrices arising from structured two dimensional (2D) and three dimensional (3D) grids [7, 20]. Within constant factors, these results apply to a more general class of sparse matrices, whose graphs have the property that their n-vertex subgraphs have separators with $O(\sqrt{n})$ vertices and $O(n^{2/3})$ vertices, respectively. A separator is a set of vertices whose removal disconnects the graph into two disjoint subgraphs of almost the same size. Computing the Cholesky factorization on N processors of an $n \times n$ matrix arising from a 2D grid of dimension $n^{1/2} \times n^{1/2}$ costs

$O(n^{3/2}/N)$ flops, and its Cholesky factor has $O(n \log n)$ nonzeros. For 3D grids of dimension $n^{1/3} \times n^{1/3} \times n^{1/3}$, the Cholesky factorization on N processors costs $O(n^2/N)$ and the Cholesky factor has $O(n^{4/3})$ nonzeros. In the case of 2D grids, if there is enough memory to store the Cholesky factor, direct methods are often faster than iterative methods. Hence in this paper we focus only on 3D grids and we consider matrices of size $n \times n$ arising from grids of dimension $n^{1/3} \times n^{1/3} \times n^{1/3}$.

The construction of LORASC requires computing the direct factorization of the first N diagonal blocks. This can be done in parallel on N processors. We assume that each processor has enough memory to complete this factorization. After this computation is performed, all processors can compute the factorization of the last block $A_{\Gamma\Gamma}$ and then the approximation of $\tilde{S}$. The approximation of $\tilde{S}$ involves solving a generalized eigenvalue problem, which can be performed by using a library as ARPACK [19] or PARPACK [19]. The cost of this computation depends on the number of eigenvalues that need to be deflated. Since this is problem dependent, we do not model further this cost. We will see in the numerical experiments in Section 5.2.2 that the number of deflated eigenvalues grows slowly when increasing the number of partitions N. For the linear elasticity problem, the number of deflated eigenvalues varies between $\sqrt{N}$ and N.

A matrix A can be permuted to a bordered block diagonal form as in equation (4.1) by using k-way partitioning with vertex separators. In our experimental results in section 5.2.2 we use the nested dissection routine from METIS [16]. Nested dissection identifies a set of vertices in the graph of A, that form a separator, which divides the graph into 2 disjoint subgraphs. The dissection process continues recursively on each disjoint subgraph until N disjoint subgraphs are obtained. The matrix A is reordered by numbering first the vertices in the N disjoint subgraphs, which form the diagonal blocks $A_{jj}, j = 1,\ldots,N$, and then the vertices in the separators, which form the block $A_{\Gamma\Gamma}$. These last vertices can be reordered in any suitable way such that the number of fill-in elements in the Cholesky factor of $A_{\Gamma\Gamma}$ is reduced.

For the ease of the analysis we estimate the cost of LORASC by using a different ordering that can be obtained when $n^{1/3} < N$. While we focus here on 3D structured grids, there are algorithms that can be used for unstructured grids [15]. The 3D regular grid is partitioned into N subgrids of dimension $n^{1/3} \times n^{1/3} \times (n^{1/3}/N - 1)$. This partitioning is obtained by using N parallel planes, each plane having dimensions $n^{1/3} \times n^{1/3}$. Each diagonal block $A_{jj}, j = 1,\ldots,N$ is of size $(n/N - n^{2/3}) \times (n/N - n^{2/3})$. To compute its Cholesky factorization, the graph corresponding to each diagonal block is reordered by using nested dissection. At the first level of nested dissection, the first separator is formed by two orthgonal planes, each of dimension $n^{1/3} \times (n^{1/3}/N - 1)$. The sub-grid corresponding to the block A_{jj} of dimension $n^{1/3} \times n^{1/3} \times (n^{1/3}/N - 1)$ is divided into 4 smaller sub-grids, each of dimension $n^{1/3}/2 \times n^{1/3}/2 \times (n^{1/3}/N - 1)$. It is known that during the Cholesky factorization of each block A_{jj}, the sub-matrix of dimension $2n^{2/3}/N \times 2n^{2/3}/N$, corresponding to the first separator, becomes dense, and computing its factorization dominates the overall cost of the Cholesky factorization of A_{jj}. Hence the Cholesky factor of each diagonal block A_{jj} has $O(n^{4/3}/N^2)$ nonzeros and its computation costs $O(n^2/N^3)$ flops.

The last diagonal block $A_{\Gamma\Gamma}$ corresponds to the N parallel planes, each of dimension $n^{1/3} \times n^{1/3}$. It is formed by N diagonal blocks, each diagonal block corresponds to one of the N parallel planes and is of size $n^{2/3} \times n^{2/3}$. Hence, computing the Cholesky factor of each

diagonal block corresponds to a 2D problem and costs $O(n)$ flops. The Cholesky factor of each diagonal block has $O(n^{2/3} \log n)$ nonzeros. Since each processor computes the factorization of one of the diagonal blocks, the overall cost of computing the Cholesky factorization of $A_{\Gamma\Gamma}$ in parallel on N processors costs $O(n)$ flops.

Ignoring the generalized eigenvalue problem, LORASC has the following costs. The total number of nonzeros in the Cholesky factors of $A_{jj}, j = 1,\ldots,N$, and of $A_{\Gamma\Gamma}$ is $O(n^{4/3}/N) + O(n^{2/3} \log n \cdot N)$. The number of nonzeros in $A_{\Gamma\Gamma}$ grows linearly with the number of processors N. Or in other words, the number of nonzeros per processor remains constant when increasing the number of processors. Computing the Cholesky factorization of the diagonal blocks $A_{jj}, j = 1,\ldots,N$, and of $A_{\Gamma\Gamma}$ in parallel on N processors costs $O(n^2/N^3) + O(n)$ flops. The construction of LORASC, ignoring the generalized eigenvalue problem, costs a factor of $O(N^2)$ less flops than the computation of the Cholesky factorization of the entire matrix A.

4.3 Multilevel formulation

In this section, we present a multilevel version of LORASC using N processors. Without loss of generality, we consider two levels of parallelism. First, we partition the input matrix into N_1 subdomains using k-way partitioning with vertex separators, then we assign N_2 processors to each subdomain such as $N_1 \times N_2 \leq N$. Since $N_1 \leq N$ and the size of the interface between N domains grows linearly with the number of domains N, it is straightforward that the size of the interface between N_1 domains is smaller than the size of the interface between N domains. Indeed, N_1 is chosen such as to keep the size of the interface relatively small compared to the global number of processor N. This decomposition leads to N_1 groups of processors of size N_2 each.

At the first level of the parallelism, N_1 processors are used to partition the matrix as presented on the left of Figure 3. We denote by M_1, the group formed by all the N_1 prcoessors participating in this operation. This initial partitioning creates the blocks A_{ii}, A_{ig}, A_{gi} and $A_{\Gamma\Gamma}$. At the second level of parallelism, each block A_{ii} is assigned to N_2 processors in order to perform a new partitioning of these blocks as presented on the right of Figure 3. We denote by W_{1i}, the group formed by all the N_2 processors assigned to each block A_{ii}. We consider that the master of each group W_{1i} is the processors from each group W_{1i} which belong also to the group M_1;

After the initial partitioning, the factorisation of each subdomain can be performed at the second level of parallelism by a parallel direct method or a new call of LORASC using N_2 processors. This recursive formulation is also illustrated in Figure 3. The figure at the left shows the initial partitioning of the matrix into 4 subdomains and permuted into a block arrow structure, then each subdomain A_{ii} is partitioned again into 2 subdomains as presented at the right of the figure.

The parallel construction of LORASC multilevel is summarized in algorithm 3. In line 3 of the algorithm, the processors from the group M_1 partition the graph of the input matrix into N_1 disjoint domains using k-way partitioning with vertex separators, and permute the matrix into a block arrow structure as in equation 4.5. In line 5, the master processor from each group W_i distributes the matrices A_{ii}, A_{ig}, A_{gi} and $A_{\Gamma\Gamma}$ to group of processors using a block row distribution. In line 8, the processors of each group W_i receive a block row part of each matrices A_{ii}, A_{ig}, A_{gi} and $A_{\Gamma\Gamma}$ from the master processor of the group. In line 11,

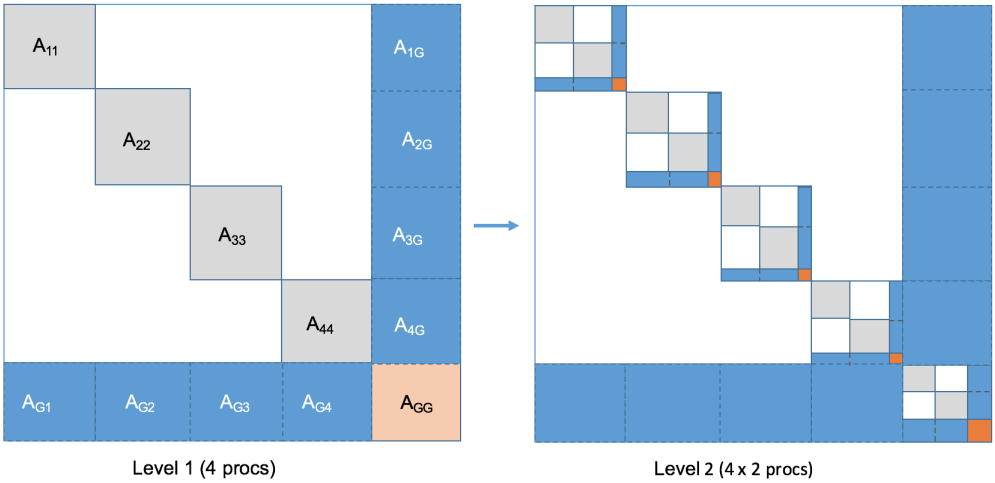

Figure 3: Multilevel LORASC

a symbolic factorization of each subdomain is performed in order to predict the maximum memory required if a direct method will be used. Such routine is available in parallel solvers such as MUMPS [1]. We note that if the size of the subdomain is large, its direct factorization might fail due to fill-in. This analysis step aims at determining if we can use a direct method.

From line 13 to line 17 of algorithm 3, we determine which approach can be used to factorize each subdomain. If there is enough memory, a parallel direct method can be used, otherwise a new instance of LORASC should be used. This step constructs a parallel solver Aii_{solver} which will be used each time a computation with A_{ii}^{-1} is required. We note that with this approach each subdomain has its own optimal parameters independently of the other subdomains. In line 18, the Schur complement matrix is factorized using a parallel direct method. Finally, in line 19, we solve the eigenvalue problem. We note that solving the eigenvalue problem is similar to *Deliverable 4.3*, with the only difference that some computations are performed in parallel. For example, when the product $z_i = A_{gi} \cdot A_{ii}^{-1} \cdot A_{gi} \cdot v$ is required, first we use a parallel matrix-vector product routine to compute $x_i = A_{gi} \cdot v$ using all the processors in the group W_{1i}, then we use a parallel solver (direct method or LORASC) to compute $y_i = A_{ii}^{-1}$, and finally we use again parallel matrix-vector product routine to compute $z_i = A_{gi} \cdot y_i$.

Require: Matrix A_i, current level of parallelism i, N_i processors at level i, myid is the number of the processor running this routine

if myid belongs to level i
  /* Permute the matrix into a block arrow structure as in (4.5) */
  BlockArrowCreate(comm_level[i], A_i, A_{ii}, A_{ig}, A_{gi}, $A_{\Gamma\Gamma}$)
  /* Distribute blocks on N_{i+1} processors by using a block row distribution */
  MatricesBlockRowDistribute(comm_level[i+1], A_{ii}, A_{ig}, A_{gi})
else if myid belongs to level $i+1$
  /* Receive a block row of each matrices */
  MatricesBlockRowReceive(root_level[i+1], A_{ii}, A_{ig}, A_{gi})
endif
if myid belongs to level $i+1$
  mem_required = symbolicFactorization(comm_level[i+1], A_{ii})
endif
if mem_required < MAX_MEMORY
  Aii_{solver} = NumericFactorize(comm_level[i+1], A_{ii})
else
  Aii_{solver} = LorascBuild(comm_level[i+1], A_{ii}, $i+1$)
endif
$A_{\Gamma\Gamma solver}$ = Factorize(comm_level[i+1], $A_{\Gamma\Gamma}$)
SolveEigenValues(comm_level, A_{ii}, A_{ig}, A_{gi}, $A_{\Gamma\Gamma}$, Aii_{solver}, $A_{\Gamma\Gamma solver}$)

Algorithm 3: LorascBuild: Parallel construction of LORASC multilevel using N_i processors.

5 Experiments

This section is dedicated to the numerical experiments we performed in order to assess the parallel design of ECG and LORASC we presented in Sections 3 & 4

5.1 Enlarged Conjugate Gradient

In this section, we present the numerical results we obtained with the ECG implementation detailed in Section 3.

5.1.1 Description of the parallel environment

In the experiments we use a block Jacobi preconditioner, associating at each block a MPI process. Before calling ECG, each MPI process factorizes the diagonal block of A corresponding to the local row panel that it owns. At each iteration of ECG, each MPI process performs a backward and forward solve locally in order to apply the preconditioner. Hence the application of the preconditioner does not need any communication. It is likely that there exists better preconditoners than block Jacobi for our test cases, however we are interested in the iterative method rather than in the preconditioner. In particular, we do not want to target specific applications and aim at being as generic as possible. Although in theory it is possible to apply any preconditioner within this implementation, in practice it is essential that

Name	Size	Nonzeros	Problem
Flan_1565	1,564,794	117,406,044	Structural problem
Bump_2911	2,911,419	130,378,257	Reservoir simulation

Table 2: Test matrices.

applying this preconditioner to several vectors at the same time is not too costly, *e.g*, a sublinear complexity with respect to the number of vectors.

The following experiments are performed on a machine located at Umeå University as part of High Performance Computing Center North (HPC2N), called Kebnekaise. In our experiments, we use the so-called compute nodes, which are formed by Intel Xeon E5-2690v4 (Broadwell) with 2x14 cores. For a detailed description of the machine, we refer to the online documentation[1].

We compile the code (and its dependencies) using Intel toolchain installed on the machine: `mpiicc` (based on `icc` version 18.0.1 20171018) and MKL [24] version 2018.1.163. We use PETSc [3] in order to compare ECG implementation to PETSc PCG implementation. In particular, PETSc is configured to use MKL-PARDISO as exact solver for sparse matrices in the block Jacobi preconditioner. For partitioning the matrix we are using the METIS library downloaded and installed by PETSc.

5.1.2 Test cases

As previously pointed out, ECG is an algebraic method that does not rely on any particular assumption on the matrix, except that it is symmetric positive definite. As an illustration, we test the implementation on two SPD matrices coming from the Sparse Matrix Collection of Tim Davis [4]: Flan_1564 and Bump_2911. Numerical properties of the test matrices are summarized in Table 2.

In all the experiments the tolerance is set as the default tolerance of PETSc, *i.e.*, 10^{-5} and the maximum number of iterations is set to 5000. The right-hand side is chosen uniformly random and then normalized. The initial guess is set to 0. We do not use any kind of threading and use 28 MPI processes per node, *i.e.*, each MPI rank is assigned a physical core.

5.1.3 Impact of the enlarging factor

First we study the impact of the enlarging factor t on the methods. We fix the number of processors and we vary the value of t for the 4 methods: Orthodir (Odir), Orthodir with dynamic reduction of the search directions (D-Odir) and Orthomin (Omin). The results obtained are summarized in Table 3

For Flan_1565 the number of MPI processes is fixed to 56. We remark that the runtime is decreasing until $t = 12$ and then it increases slightly. When t is relatively small the 4 methods are comparable. However as t increases the effect of dynamic reduction becomes more visible. With $t = 28$, D-Odir is almost 10% faster than Odir. Overall, for this matrix, the best method is D-Odir with $t = 12$.

[1] https://www.hpc2n.umu.se/resources/hardware/kebnekaise

	t	Odir	D-Odir	Omin		t	Odir	D-Odir	Omin
Flan_1565	1	56.9	62.8	56.7	Bump_2911	1	54.4	53.3	53.4
	4	36.3	36.4	35.5		4	76.9	72.4	75.4
	8	30.0	29.6	29.0		8	93.6	85.4	91.5
	12	30.2	29.1	29.8		12	123.1	104.1	122.1
	16	31.3	29.3	30.2		16	151.2	123.6	147.1
	20	33.1	30.7	32.0		20	179.7	143.3	174.0
	24	37.9	33.7	36.2		24	198.3	158.3	195.5
	28	39.2	34.9	37.6		28	223.6	171.8	219.0

Table 3: Runtime results (in seconds) for Flan_1565 ($P = 56$) and Bump_2911 ($P = 112$)..

For Bump_2911 we fix the number of MPI processes to 112. For this matrix the reduction of the number of iteration is not balancing the increase in flops. However, we also notice that using the dynamic Orthodir variant (D-Odir) allows to reduce significantly the runtime when t is large: D-Odir is around 20% faster than Odir.

In conclusion, D-Odir is the best method over the different variants of ECG that we tested: it is a good compromise between the stability of Odir and the efficiency of the classical CG. Nevertheless, there exists matrices such as Bump_2911 for which the reduction of the number of iterations does not compensate the extra cost of ECG compared to the classical CG, even when using the dynamic reduction of the search directions. These results support the theoretical convergence study that has been done but that is not presented in this document. ECG(t) is acting as if the t smallest eigenvalues of the matrix were deflated. Finally, we notice that values of t between 8 to 24 are good default parameters. Indeed, such values allow to effectively reduce the number of iterations while maintaining an affordable cost per iteration.

5.1.4 Strong scaling study

Following the parameter study, we perform a strong scaling study on Flan_1565. As Bump_2911 does not seem particularly well suited for the method we do not perform the strong scaling study on this matrix.

Hence, for Flan_1565 we compare PETSc PCG and D-Odir with $t = 12$, the best choice over the parameters we tested. The resulting runtimes are plot in Figure 4. When the number of MPI processes is relatively low ECG scales as well as PETSc, *i.e.*, almost linearly. As the number of iterations is significantly reduced with D-Odir(24), there is about 20% speed-up compared to PETSc at such scales. Nevertheless, we notice that for 2,016 MPI processes PETSc is significantly faster than ECG. This is likely because the number of iterations with PETSc is reduced with respect to 1,008 MPI processes. This behavior is not expected because it is known that block Jacobi preconditoners are not scalable (see [5] for instance). Indeed, we observe that the number of iterations is effectively increasing both for ECG and CG when the number of MPI processes increases.

	D-Odir(12)		CG	
# MPI	# iter	res	# iter	res
252	332	1.3E-4	1,709	1.3E-4
504	405	1.8E-4	2,430	1.9E-4
1,008	519	2.6E-4	3,179	2.6E-4
2,016	637	3.7E-4	2,687	3.7E-4

Table 4: Iteration count and residual for Flan_1565.

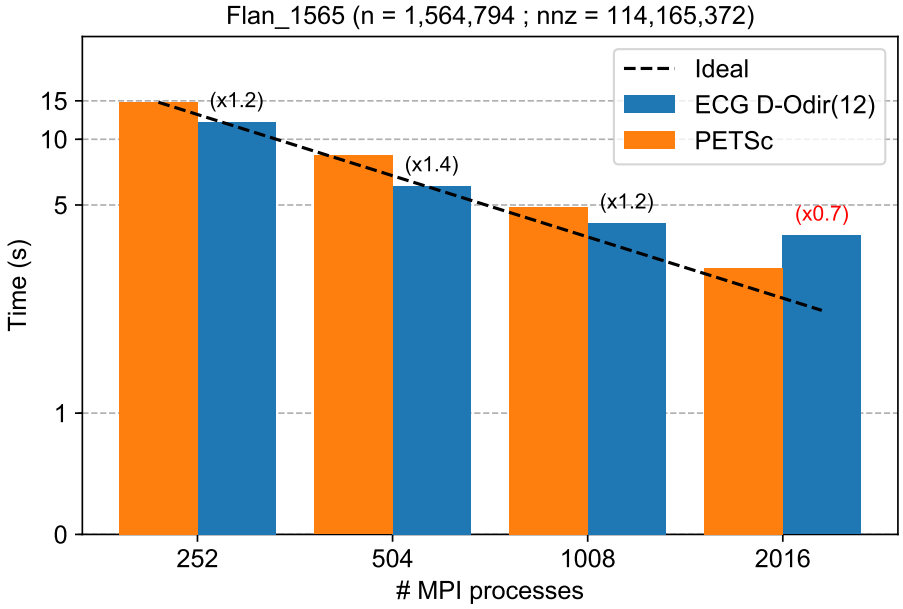

Figure 4: Runtime comparison between ECG and PETSc PCG (in parenthesis the ratio between PETSc runtime and ECG runtime) for Flan_1565.

5.2 LORASC

In this section, we present the results of LORASC for one and two levels of parallelism. We denote by LORASC 1-level, our previous implementation of LORASC, and LORASC 2-level the multilevel version of LORASC using 2 levels of parallelism.

5.2.1 Environment

In this section we discuss the performance of LORASC on a parallel machine formed by 28 compute nodes. This follows the description provided in [8]. Each node is equipped with a 24 cores socket based on Intel Xeon E5-2670 (Sandy Bridge), each core has a frequency of 2.6 GHz. We use ParMetis 4.0.3 [17] to partition the graph of the input matrix in parallel. The symbolic and Cholesky factorizations of each domain are computed using MUMPS 5.0.2 [1]. The generalized eigenvalue problem is solved by using the parallel version of ARPACK, PARPACK 2.1 [19]. ECG solver with an enlarging factor of 1 is used to solve the system,

which corresponds to the classical CG. Both LORASC and Block Jacobi preconditioners are implemented in the reverse communication interface provided by ECG solver.

5.2.2 LORASC 1-level

Figure 5 shows the performance of Block Jacobi and LORASC 1-level for an elasticity 3D problem of size N =145 563. The orange and blue dashed lines represent the global time to solve the system using Block Jacobi and LORASC 1-level respectively. Each line corresponds to the sum of the time to construct the preconditioner and solve the preconditioned system. The blue bars represent the size of the Schur-complement after the initial partitioning of the matrix. In accordance with the results from Deliverable 4.3, we observe that LORASC 1-level outperforms Block Jacobi up to 64 processors, but Block Jacobi becomes better from 128 processors. As presented in Deliverable 4.3, the main reason is that the size of the Schur-complement increases linearly with the number of processors, hence solving the generalized eigenvalue problem becomes the most expensive part on large number of processors. For example, using 512 processors, the size of the Schur complement is 62000, which corresponds to 40% of the initial problem size.

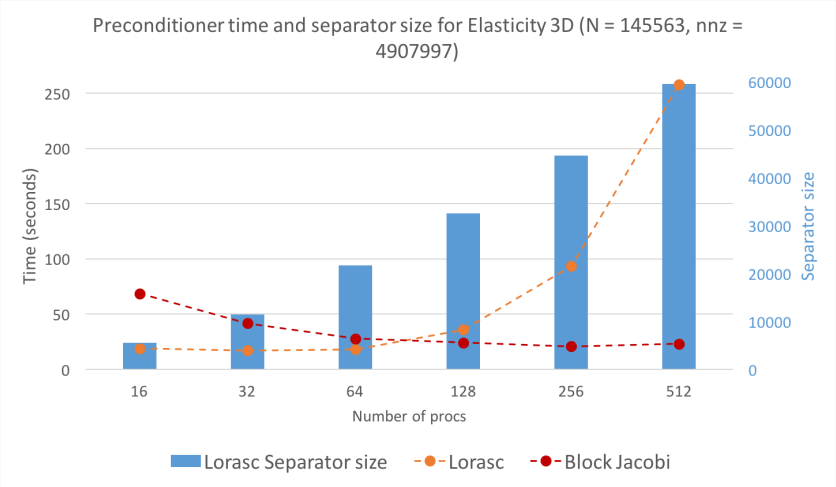

Figure 5: Performance of Block Jacobi, LORASC and the separator sizes.

The first approach used to address the scalability of LORASC is based on tuning the parameters of the generalized eigenvalue problem. The number of eigenvalues to compute is fixed independently of the size of the Schur-complement, so the selected number is smaller than the number of eigenvalues recommended by our analysis. The main advantage of this choice is that we compute fewer eigenvalues and therefore the time to solve the generalized eigenvalue problem is reduced. The disadvantage of this approach is that all eigenvalues lower than the tolerance ε might not be computed, which will lead to a slower convergence and increase the solve time. Figure 6 shows the performance of Block Jacobi and LORASC for the same matrix as in figure 5 when the number of eigenvalues to compute is fixed to $nev = 150$. The yellow and orange bars represent the time to construct Block Jacobi preconditioner and solve the preconditioned system respectively, while blue and red bars represent the same

times but using LORASC 1-level. The time to construct Block Jacobi preconditioner is negligible and it does not appear in the figure. For LORASC, we observe that fixing the number of eigenvalues to 150 has a very small impact on the solving time while reducing considerably the time to construct the preconditioner. However, the construction of the preconditioner is still the most expensive part. This version of LORASC is much faster than Block Jacobi up to 128 processors, while for 256 processors Block Jacobi and LORASC have almost the same performance. Beyond 512 processors, the performance improvements of LORASC are not significant.

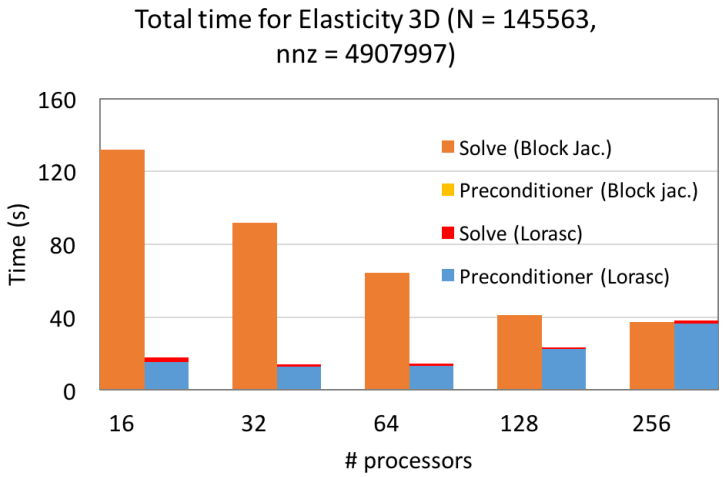

Figure 6: Strong scaling performance of LORASC on elasticity 3D problem of size n =145 563 with 4 907 997 nonzeros.

5.2.3 LORASC 2-level

As presented in section 4.3, the goal of the multilevel approach is to reduce the size of the Schur-complement at each level of the partitioning, while increasing the parallelism within each domain.

When the number of domains at the first level is very small, the factorization of each subdomain using a direct method requires more memory because of the fill-in, or when the maximum amount of memory is limited, a new call of LORASC is required in order to factorize the subdomain. In our experiments, we use 32 domains at the first level of parallelism, which corresponds to the best trade-off between the required amount of memory and the time to construct the preconditioner. By using 32 processors at the first level of parallelism, the size of the Schur-complement corresponds exactly to the same as using 32 processors in Figure 5, with the only difference that the number of processors within each block is increased. Hence, using 64 processors for LORASC 2-level corresponds to creating 32 domains at the first level of parallelism and using 2 MPI processors within each subdomain. Using 128 processors corresponds to the same partitioning but using 4 processors within each block and so on. The number of processors used by PARPACK corresponds to 32 processors, but all the processors participate in the computation of the matrix-vector product in the reverse

communication interface. By using a reduced number of processors in PARPACK we also reduce the number of communication involved in the dot-product of PARPACK iterative method.

Figure 7 shows the performance of Block Jacobi, LORASC 1-level and LORASC 2-level for a large elasticity 3D problem of size N =19 613 997. We observe that for this problem size, Block Jacobi is not scalable, LORASC 1-level is scalable up to 64 processors, while LORASC 2-level is scalable up to 1024 processors. Using an increasing number of processors within each of the 32 subdomains helps to further decrease the total time of LORASC 2-level. As a result, the time of LORASC 2-level does not increase with respect to LORASC 1-level.

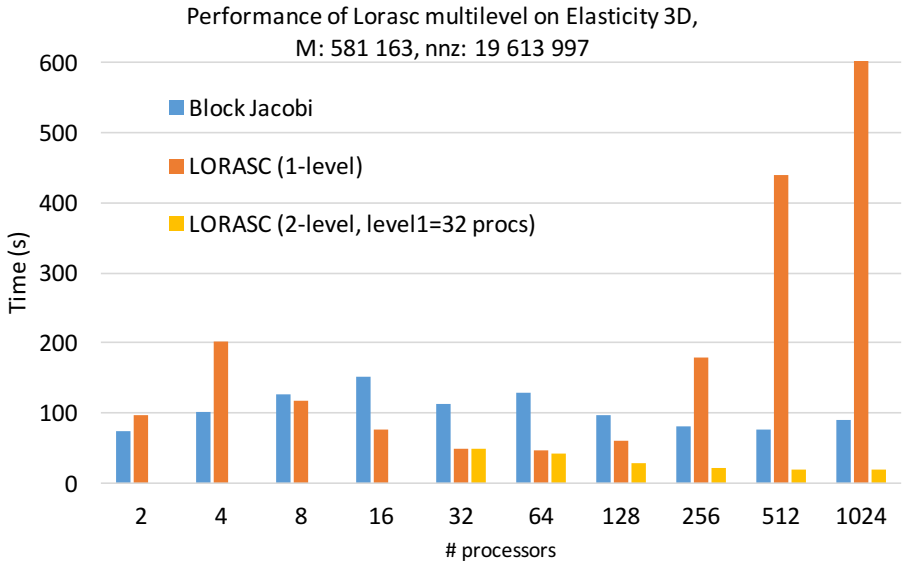

Figure 7: Strong scaling performance of LORASC 2-level on elasticity 3D problem.

Figure 8 shows the weak scaling performance of Block Jacobi, LORASC 1-level and LORASC 2-level. The size of the problem on each processor is almost constant when the number of processors and the matrix size increase. The figure on the left shows the time for small matrices, while the figure on the right shows the time for larger matrices. The number of processors doubles with the increasing size of the problem. For example, the problem of size $N = 4719$ is solved using 2 processors, $N = 9438$ is solved using 4 processors, $N = 18513$ is solved using 8 processors and so on. We observe that the time of Block Jacobi and LORASC 1-level increase considerably with the size of the problem, while the time of LORASC 2-level is slightly impacted. This result shows that LORASC 2-level is suited for a larger problem using a relatively higher number of processors.

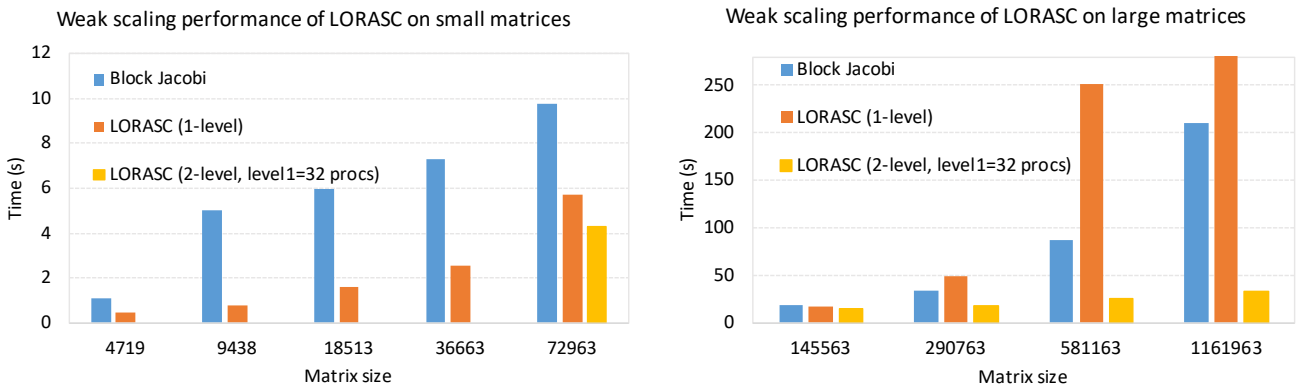

Figure 8: Weak scaling performance of LORASC on elasticity 3D problem.

6 Acknowledgments

This project is funded from the European Union's Horizon 2020 research and innovation program under the NLAFET grant agreement No 671633. We thank the High Performance Computing Center North (HPC2N) at Umeå University for providing computational resources and valuable support.

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] 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.

[3] 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.

[4] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[5] V. Dolean, P. Jolivet, and F. Nataf. *An introduction to domain decomposition methods*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2015. Algorithms, theory, and parallel implementation.

[6] A.A. Dubrulle. Retooling the method of block conjugate gradients. *ETNA. Electronic Transactions on Numerical Analysis [electronic only]*, 12:216–233, 2001.

[7] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

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

[9] L. Grigori and O. Tissot. Reducing the communication and computational costs of enlarged Krylov subspaces conjugate gradient. NLAFET Working Note WN 13. Also as Research Report RR-9023, Feb 2017.

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

[11] 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.

[12] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards.*, 49:409–436, 1952.

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

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

[15] G. Atenekeng Kahou, L. Grigori, and M. Sosonkina. A partitioning algorithm for block diagonal matrices with overlap. *Parallel Computing*, 34(6):332–344, 2008.

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

[17] 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.

[18] 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.

[19] 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.

[20] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM J. Numer. Anal.*, 16:346–358, 1979.

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

[22] Y. Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[23] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[24] 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.