Enlarged Krylov Subspace Methods and Preconditioners for Avoiding Communication

Sophie Moufawad

Université Pierre et Marie Curie
École Doctorale de Sciences Mathématiques de Paris Centre 386
Laboratoire Jacques-Louis Lions / Equipe Alpines

THÈSE DE DOCTORAT
Discipline : Mathématiques Appliquées

présentée par

Sophie MOUFAWAD

Enlarged Krylov Subspace Methods and Preconditioners for Avoiding Communication

---

dirigée par Laura GRIGORI

Soutenue le ** **** 2014 devant un jury composé de :

M. Prénom NOM	Université	président
M^{lle} Prénom NOM	Institut	examinateur
M^{me} Prénom NOM	Université	rapporteur
M. Prénom NOM	Université	directeur
M. Prénom NOM	Université	rapporteur

Laboratoire Jacques Louis Lions
Boite courrier 187
4, place Jussieu
75252 Paris cedex 05

École doctorale Paris centre
Boite courrier 188
4, place Jussieu
75252 Paris cedex 05

Contents

Chapter 1

Introduction

Many scientific problems require the solution of systems of linear equations of the form $Ax = b$, where A is an $n \times n$ matrix and b is an $n \times 1$ vector. There are two broad categories for solving systems of linear equations, direct methods and iterative methods. Direct methods solve the system in a finite number of steps or operations. Examples of direct methods are matrix decompositions like LU decomposition $A = LU$, Cholesky decomposition for symmetric positive definite $A = LL^t$, and QR decomposition for full rank $A = QR$, where L is a lower triangular matrix, U and R are upper triangular matrices, and Q is an orthonormal matrix. After decomposing the matrix A, the upper triangular and lower triangular systems are solved by backward and forward substitution. The matrix A can be a dense or a sparse matrix. Several libraries that implement direct methods for solving sparse systems have been introduced, like MUMPS [1], PARADISO [69, 55], and SuperLU [57, 58]. However, when the matrix A is sparse, the factors obtained after decomposition are denser than the input matrix. Moreover, direct methods are prohibitive in terms of memory and flops when it comes to solving very large systems, and they are not easily parallelized on modern-day architectures. Thus, iterative methods that compute a sequence of approximate solutions for the system $Ax = b$ by starting from an initial guess, are a good alternative.

We are interested in solving systems of linear equations, $Ax = b$, where the matrix A is sparse. Such systems may arise from the dicretization of partial differential equations. The Krylov subspace methods are among the most practical and popular iterative methods today. They are polynomial iterative methods that aim to solve systems of linear equations ($Ax = b$) by finding a sequence of vectors $x_1, x_2, x_3, x_4, ..., x_k$ that minimizes some measure of error over the corresponding spaces

$$x_0 + \mathcal{K}_i(A, r_0), \quad i = 1, ..., k$$

where x_0 is the initial iterate, r_0 is the initial residual, and $\mathcal{K}_i(A, r_0) = span\{r_0, Ar_0, A^2r_0, ..., A^{i-1}r_0\}$ is the Krylov subspace of dimension i. Conjugate Gradient (CG) [47], Generalized Minimal Residual (GMRES) [66], bi-Conjugate Gradient [56, 30] and bi-Conjugate Gradient Stabilized [75] are some of the Krylov subspace methods. These methods compute one basis vector or one search direction vector at each iteration i, by performing a matrix-vector multiplication. Then, the i^{th}

approximate solution is defined by performing saxpy ($ax + y$) and dot products ($x^t x$), where a is scalar, and x, y are vectors.

The performance of an algorithm on any architecture is dependent on the processing unit's speed for performing floating point operations (flops) and the speed of accessing memory and disk. Moreover, the efficiency of parallel implementations is dependent on the amount of performed computations per communication, data movement. This is due to the fact that the cost of communication is much higher than arithmetic operations, and this gap is expected to continue to increase exponentially [37]. As a result, communication is often the bottleneck in numerical algorithms. In a quest to address the communication problem, recent research has focused on reformulating linear algebra operations such that the movement of data is significantly reduced, or even minimized as in the case of dense matrix factorizations like LU factorization, QR factorization, tall and skinny QR factorization [21, 38, 4]. Such algorithms are referred to as communication avoiding.

The Krylov subspace methods are governed by Blas1 and Blas2 operations like dot products and matrix vector multiplications, as discussed above. Parallelizing dot products is constrained by communication, since the performed computation is negligible. If the dot products are performed by one processor, then there is a need for a communication before (synchronization) and after the computations. In both cases, communication is a bottleneck. This problem has been tackled by different approaches. One approach is to the hide the communication's cost by overlapping it with other communications and computations, like pipelined CG [23, 43] and pipelined GMRES [34]. Another approach consists of replacing Blas1 and Blas2 operations by Blas2 and Blas3 operations, by either introducing new methods or by reformulating the algorithm itself. The first such methods to be introduced, are block methods that solve a system with multiple right-hand sides $AX = B$, like O'Leary's block CG [63]. These methods compute at each iteration a block of vectors by performing a matrix times a block of vectors. Then, the i^{th} block approximate solution is obtained by solving small systems and performing tall skinny gaxpy's, $Ax + y$, where A is an $n \times m$ matrix with $n >> m$, and x, y are vectors.

Unlike the block methods, the s-step methods solve the system $Ax = b$ by computing s basis vectors per iteration and solving small systems. Some of the s-step methods are s-step CG [19] and s-step GMRES [26]. Both methods, block and s-step, use Blas2 and Blas3 operations. Recently, communication avoiding Krylov methods, based on s-step methods, were introduced, like CA-CG, and CA-GMRES [60, 48, 12]. The communication avoiding methods aim at further avoiding communication in the Blas2 and Blas3 at the expense of performing some redundant flops. For $s = 1$, where s-step methods are equivalent to classical methods, there are many available preconditioners. One of them, block Jacobi, is a naturally parallelizable and communication avoiding preconditioner. However, except a discussion in [48], there are no available preconditioners that avoid communication and can be used with s-step methods for $s > 1$. This is a serious limitation of these methods, since for difficult problems, Krylov subspace methods without preconditioner can be very slow or even might not converge. In this thesis, we introduce a communication avoiding ILU(0) preconditioner (CA-ILU0) [40], that can be computed in parallel, and applied to s vectors

of the form $y_i = (LU)^{-1}Ay_{i-1}$ without any communication, for $i = 1, 2, .., s$. This preconditioner can be parallelized without communication due to the use of a heuristic reordering of the matrix A, that we call alternating min-max layers AMML(s). Moreover, the CA-ILU0 preconditioner can also be used with classical Krylov subspace methods, where it avoids communication.

Since communication avoiding methods are based on s-step methods which have some stability issues, we introduce a new type of Krylov subspace methods. We introduce a new approach that consists of enlarging the Krylov subspace based on domain decomposition. First, we split the initial r_0 into t vectors depending on a decomposed domain. Then, the obtained t vectors are multiplied by A at each iteration to generate the t new basis vectors of the enlarged Krylov subspace. Enlarging the Krylov subspace should lead to faster convergence and parallelizable algorithms with less communication than the classical Krylov methods, due to the use of Blas2 and Blas3 operations. In this thesis, we introduce two new versions of conjugate gradient. The first version, multiple search direction with orthogonalization CG (MSDO-CG), has the same structure as the classical conjugate gradient method, where we first define t new search directions, then find the t step lengths by solving a $t \times t$ system and update the solution and the residual. But unlike CG, the search directions are not A-orthogonal. We A-orthonormalize the search directions, to obtain a projection method that guarantees convergence at least as fast as CG. The second version, long recurrence enlarged CG (LRE-CG), is similar to GMRES in that we build an orthonormal basis for the enlarged Krylov subspace rather than finding search directions. Then, we use the whole basis to update the solution and the residual.

The thesis is organized as follows. In chapter 2 we briefly introduce some notations and kernels that are used throughout this thesis such as graphs and graph partitioning, orthonormalization schemes, A-orthonormalization schemes, the matrix powers kernel, and the set of test matrices used to test our introduced methods. In chapter 3 we discuss several variants of Krylov subspace methods, such as classical Krylov subspace methods (CG and GMRES), block Krylov methods (block CG), s-step Krylov methods (s-step CG and s-step GMRES), communication avoiding Krylov methods (CA-GMRES), and other parallelizable version (MSD-CG and coop-CG). We also discuss preconditioners, such as incomplete LU preconditioner, block Jacobi preconditioner, and restricted additive Schwarz preconditioner, which are crucial for the fast convergence of Krylov subspace methods.

In chapter 4 we introduce the enlarged Krylov subspace, the MSDO-CG method, and the LRE-CG method. We show that both methods are projection methods and hence converge at least as fast as CG in exact precision. And we compare the convergence behavior of MSDO-CG and LRE-CG methods using different A-orthonormalization and orthonormalization methods. Then we compare the most stable versions with CG and other related methods. Both methods converge faster than CG, but LRE-CG converges faster than MSDO-CG since it uses the whole basis to update the solution rather than only t search directions. We also present the parallel algorithms with their expected performance, and the preconditioned versions with their convergence behavior. This chapter is based on the article [41] which is in preparation for submission.

In chapter 5 we introduce the communication avoiding ILU(0) preconditioner (CA-ILU0) that

minimizes communication during the construction of $M = LU$ (i.e, the ILU(0) factorization), and during its application to s vectors ($z = M^{-1}y = (LU)^{-1}y$)) at each iteration of the s-step methods. In other words, it is possible to solve s upper triangular system and s lower triangular system, in addition to the s matrix vector multiplications without any communication. First, we adapt the matrix powers kernel to the case of ILU preconditioned systems. Then, we introduce the AMML(s) heuristic reordering based on nested dissection and k-way graph partitioning. Then, we show that our reordering does not affect the convergence of ILU(0) preconditioned GMRES, and we model the expected performance of our preconditioner based on the needed memory and the redundant flops introduced to reduce the communication. This chapter is based on some parts of a revised version of the technical report [40], and on the article [39], which was submitted to SIAM journal on scientific computing (SISC) and is in revision. Finally, in chapter 6 we conclude and discuss possible future work in the introduced methods.

Chapter 2

Preliminaries

We will briefly introduce some notations (section 2.1) and kernels that will be used throughout this thesis such as graphs and graph partitioning (section 2.2), orthonormalization (section 2.3), A-orthonormalization (section 2.4), and the matrix powers kernel (section 2.5) . We will also describe the test matrices (section 2.6) that will be used in Chapters 4 and 5.

2.1 Notation

We denote matrices or block of vectors by upper case letters. Whereas vectors are denoted by lower case letters. All subscripts used for matrices, vectors, graphs, and sets serve as indices, indices denoting iterations (x_k) or subparts ($A_{1,1}$). We use matlab notation for matrices and vectors. For example, given a vector y of size $n \times 1$ and a set of indices α (which correspond to vertices in the graph of A), $y(\alpha)$ is the vector formed by the subset of the entries of y whose indices belong to α. For a matrix A, $A(\alpha, :)$ is a submatrix formed by the subset of the rows of A whose indices belong to α. Similarly, $A(:, \alpha)$, is a submatrix formed by the subset of the columns of A whose indices belong to α. We have $A(\alpha, \beta) = [A(\alpha, :)](:, \beta)$, the β columns of the submatrix $A(\alpha, :)$. Note that the set of indices can be expressed explicitly like $y(1 : 20)$ which is a vector with the first 20 entries of y or $A(:, 1 : tk + i - 1)$ which is a matrix containing the first $tk + i - 1$ columns of A.

2.1.1 Communication

In this thesis, the word "processor" indicates the component performing the computations and "fetch" indicates the data movement. The definition of this "processor" and "fetch" depends on the kind of communication that we want to avoid. The two broad categories are communication in parallel computations (between processors) and communication in sequential computations (between different levels of memory hierarchy)

In the first case, communication can take the following forms, among others:

- Messages between processors, in a distributed-memory system. ("processor" = processor, "fetch" = receive message)
- Cache coherency traffic, in a shared-memory system. ("processor" = core, "fetch" = read)
- Data transfers between coprocessors linked by a bus, such as between a CPU ("Central Processing Unit" or processor) and a GPU ("Graphics Processing Unit"). ("processor" = GPU, "fetch" = copy from CPU memory to GPU memory)

In the sequential case, communication between levels of hierarchy can be between:

- cache and main memory.
- main memory and disk.
- local store (a small, fast, software-managed memory) and main memory.

In the three cases of sequential communication, the "processor" = processor and by "fetch" we mean copy the data from the slow memory to the fast one.

In general, the estimated time for computing z flops is $\gamma_c z$, where γ_c is the inverse floating-point rate, also called the floating-point throughput (seconds per floating-point operation) of the processor. In the case of distributed-memory architecture, the estimated time for sending a messages of size k is $\alpha_c + \beta_c k$, where α_c is the latency (with units of seconds) and β_c is the inverse bandwidth (seconds per word). Hence, the estimated runtime of an algorithm with a total of z computed flops and s sent messages each of size k is the sum of their corresponding estimated times $\gamma_c z + \alpha_c s + \beta_c$.

2.2 Graphs and partitioning techniques

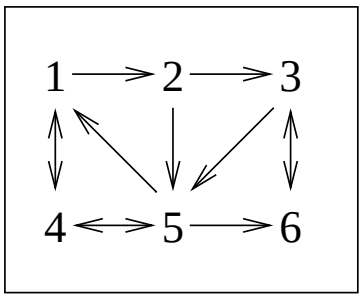
In this section we give the definitions of the notions such as graphs (section 2.2.1), nested dissection (section 2.2.2), and k-way partitioning (section 2.2.3).

2.2.1 Graphs

The structure of an unsymmetric $n \times n$ matrix A can be represented by using a directed graph $G(A) = (V, E)$, where V is a set of vertices and E is a set of edges. A vertex v_i is associated with each row i of the matrix A. An oriented edge $e_{j,i}$ from vertex j to vertex i is associated with each nonzero element $A(j, i) \neq 0$ as shown in Figure 2.1 where the vertex is represented by its index. A weight w_i and a cost $c_{j,i}$ are assigned to every vertex v_i and edge $e_{j,i}$ respectively. Let B be a subgraph of $G(A)$ ($B \subset G(A)$), then $V(B)$ is the set of vertices of B, $V(B) \subset V(G(A))$, and $E(B)$ is the set of edges of B, $E(B) \subset E(G(A))$. Let i and j be two vertices of $G(A)$. The vertex

$$\begin{pmatrix} \times & \times & 0 & \times & 0 & 0 \\ 0 & \times & \times & 0 & \times & 0 \\ 0 & 0 & \times & 0 & \times & \times \\ \times & 0 & 0 & \times & \times & 0 \\ \times & 0 & 0 & \times & \times & \times \\ 0 & 0 & \times & 0 & 0 & \times \end{pmatrix}$$

(a) Matrix A



(b) Graph of A

Figure 2.1: The figure shows the sparsity pattern of a matrix A and its corresponding graph.

j is reachable from the vertex i if and only if there exists a path of directed edges from i to j. The length of the path is equal to the number of visited vertices excluding i. Let S be any subset of vertices of $G(A)$. The set $R(G(A), S)$ denotes the set of vertices reachable from any vertex in S and includes S ($S \subset R(G(A), S)$). The set $R(G(A), S, m)$ denotes the set of vertices reachable by paths of length at most m from any vertex in S. The set $R(G(A), S, 1)$ is the set of adjacent vertices of S in the graph of A and we denote it by $Adj(G(A), S)$. The set $Adj(G(A), S) - S$ is the open set of adjacent vertices of S in the graph of A and we denote it by $opAdj(G(A), S)$. An undirected graph of a symmetric matrix is a special case of directed graphs where all the edges are bidirectional. Since there is no need to specify a direction, the edges are undirected.

Note that the structure of a sparse matrix can also be represented by a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, where $\mathcal{V}$ is a set of vertices and $\mathcal{N}$ is a set of hyperedges (nets) where each hyperedge can connect several vertices.

To exploit parallelism and reduce communication when solving a linear system $Ax = b$ using an iterative solver, the input matrix A is often reordered using graph partitioning techniques such as nested dissection [33, 52] or k-way graph partitioning [51]. These techniques assume that the matrix A is symmetric and its graph is undirected. In case A is unsymmetric, then the undirected graph of $A + A^t$ is used to define a partition for the matrix A. Graph partitioning techniques can be applied on both graphs and hypergraphs (see e.g. [13, 3]), and they rely on identifying either edge/hyperedge separators or vertex separators. In sections 2.2.2 and 2.2.3 we describe Nested Dissection and K-way briefly in the context of undirected graphs.

2.2.2 Nested dissection

Nested dissection [33] is a divide and conquer graph partitioning strategy based on vertex separators. For undirected graphs, at each step of dissection, a set of vertices that forms a separator is sought, that splits the graph into two disjoint subgraphs once the vertices of the separator are removed. We refer to the two subgraphs as $\Omega_{1,1}$ and $\Omega_{1,2}$, and to the separator as $\Sigma_{1,1}$. The vertices

of the first subgraph are numbered first, then those of the second subgraph, and finally those of the separator. The corresponding matrix has the following structure,

$$A = \begin{pmatrix} A_{11} & & A_{13} \\ & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}.$$

The algorithm then continues recursively on the two subgraphs $\Omega_{1,1}$ and $\Omega_{1,2}$. The separators subgraphs and the subdomains subgraphs introduced at level i of the nested dissection are denoted by $\Sigma_{i,j}$ and $\Omega_{i,l}$ respectively, where $j \leqslant 2^{i-1}$, $l \leqslant 2^i$, $i \leqslant t$ and $t = log(P)$ (P is the number of processors). The vertices of the separators and the final subdomains are denoted by $S_{i,j} = V(\Sigma_{i,j})$ and $D_l = V(\Omega_{t,l})$ respectively. Thus, at level i we introduce 2^{i-1} new separators and 2^i new subdomains. We illustrate nested dissection in Figure 2.2 which displays the graph of a 2D 5 point-stencil matrix A, where the vertices are represented by their indices. For clarity of the figure, the edges are not shown in the graph, but it must be noted that there are oriented edges connecting each vertex to its north, south, east and west neighbors. This corresponds to a symmetric matrix with a maximum of 5 nonzeros per row. Figure 2.2 presents the subdomains and the separators obtained by using three levels of nested dissection. All the following figures of graphs in this thesis have the same format.

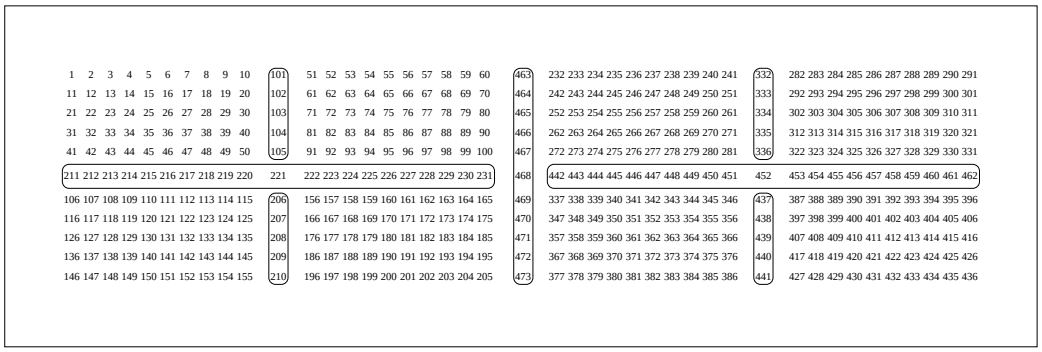


Figure 2.2: An 11 by 43 5-point stencil, partitioned into 8 subdomains using 7 separators . The bidirectional edges connecting each vertex to its north, south, east and west neighboring vertices are omitted in this figure.

2.2.3 K-way graph partitioning

K-way graph partitioning by edge separators aims at partitioning a graph $G = (V, E)$ into $k > 1$ parts $\pi = \{\Omega_1, \Omega_2, .., \Omega_{k-1}, \Omega_k\}$, where the k parts are nonempty ($\Omega_i \neq \phi$, $\Omega_i \subset V$ for $1 \leqslant i \leqslant k$, and $\cup_{i=1}^{k} \Omega_i = V$), and the partition π respects the balance criterion

$$W_i \leqslant (1 + \epsilon) W_{avg}, \tag{2.1}$$

where $W_i = \sum_{v_j \in \Omega_i} w_j$ is the weight associated to each part Ω_i, $W_{avg} = (\sum_{v_i \in V} w_i)/k$ is the perfect weight, ϵ is the maximum allowed imbalance ratio, and w_j is the weight associated to vertex v_j. In addition, k-way graph partitioning minimizes the cutsize of the partition π

$$\chi(\pi) = \sum_{e_{i,j} \in \mathcal{E}_E} c_{i,j}, \tag{2.2}$$

where $\mathcal{E}_E$ is the set of external edges of π, $\mathcal{E}_E \subset E$, and $c_{i,j}$ is the cost of the edge $e_{i,j}$ where $i < j$. In graph partitioning, the term edge-cut indicates that the two vertices that are connected by an edge belong to two different partitions. The edge which is cut is referred to as an external edge. In this thesis, we use $w_j = 1$ and $c_{i,j} = 1$.

2.3 Orthonormalization

Given an $n \times tk$ matrix Q whose tk column vectors are orthonormal and an $n \times t$ matrix P_{k+1}, we discuss in this section how to obtain $t(k+1)$ orthonormal vectors. This will be used in chapters 3 and 4 for describing CA-GMRES and LRE-CG where at each iteration k, t new vectors are computed and have to be orthonormalized against the previously computed tk orthonormal vectors. Let $D = [Q, P_{k+1}]$, then this can be done by orthonormalizing $D(:, tk+i)$ against all the previous vectors of D, $D(:, 1 : (tk+i-1))$, for $1 \leqslant i \leqslant t$ using classical Gram Schmidt or modified Gram Schmidt. However, this can not be parallelized efficiently. Thus, it is better to split the orthonormalization into two tasks. The first consists of orthonormalizing the t newly computed vectors of P_{k+1} against all the tk orthonormal vectors of Q. Then, the vectors of P_{k+1} are orthonormalized against each others.

Orthonormalizing a tall and skinny $n \times t$ matrix P_{k+1}, can be done using classical Gram Schmidt (CGS), modified Gram Schmidt (MGS) or a QR factorization like Householder factorization or based on Cholesky factorization. In [21], the authors presented a parallelizable QR version of a tall and skinny matrix based on binary reduction trees with local householder QR which they called tall and skinny QR (TSQR) factorization. As for the orthonormalization of P_{k+1} against Q, this can be done using MGS or CGS and its block version. We will briefly discuss CGS orthonormalization (section 2.3.1), MGS orthonormalization (section 2.3.2), and TSQR (section 2.3.3).

2.3.1 Classical Gram Schmidt (CGS)

We will start by introducing the orthonormalization of P_{k+1}'s vectors against Q's vectors, then against each others using CGS.

Assuming that the vectors of Q_k are orthonormal , i.e. $Q_k^t Q_k = I$ for all $j = 1, 2, .., tk$, then the orthonormalization of the vectors of P_{k+1} against the vectors of Q_k is defined by projecting $P_{k+1}(:, j)$ onto all the $Q_k(:, i)$ vectors and subtracting it from $P_{k+1}(:, j)$ as follows. For all $j =$

$1, 2, ..., t$, by letting $\widetilde{P}_{k+1}(:, j) = P_{k+1}(:, j) - \sum_{i=1}^{tk}(Q_k(:, i)^t P_{k+1}(:, j))Q_k(:, i)$, we get

$$\begin{aligned}\widetilde{P}_{k+1}(:, j)^t Q_k(:, o) &= P_{k+1}(:, j)^t Q_k(:, o) - \sum_{i=1}^{tk}(Q_k(:, i)^t P_{k+1}(:, j))Q_k(:, i)^t Q_k(:, o) \\ &= P_{k+1}(:, j)^t Q_k(:, o) - (Q_k(:, o)^t P_{k+1}(:, j))Q_k(:, o)^t Q_k(:, o) \\ &= 0\end{aligned}$$

for all $o = 1, 2, ..., tk$ since $Q_k^t Q_k = I$.

Algorithm 1 Orthonormalization against previous vectors with CGS

Input: Q_k, the tk orthonormal vectors; P_{k+1}, the t vectors to be orthonormalized against Q
Output: $\widetilde{P}_{k+1}$, the vectors orthonomalized against Q

Let $\widetilde{P}_{k+1} = P_{k+1}$
for $j = 1 : t$ **do**
  for $i = 1 : tk$ **do**
    $\widetilde{P}_{k+1}(:, j) = \widetilde{P}_{k+1}(:, j) - (Q_k(:, i)^t P_{k+1}(:, j))Q_k(:, i)$
  end for
  $\widetilde{P}_{k+1}(:, j) = \frac{\widetilde{P}_{k+1}(:,j)}{||\widetilde{P}_{k+1}(:,j)||_2}$
end for

Algorithm 2 Orthonormalization against previous vectors with BCGS

Input: Q_k, the tk orthonormal vectors; P_{k+1}, the t vectors to be orthonormalized against Q
Output: $\widetilde{P}_{k+1}$, the vectors orthonomalized against Q

$\widetilde{P}_{k+1} = P_{k+1} - Q_k(Q_k^t P_{k+1})$
for $j = 1 : t$ **do**
  $\widetilde{P}_{k+1}(:, j) = \frac{\widetilde{P}_{k+1}(:,j)}{||\widetilde{P}_{k+1}(:,j)||_2}$
end for

Algorithm 3 Orthonormalization of tall and skinny matrix using CGS

Input: P_{k+1}, the matrix to be orthonormalized
Output: P_{k+1}, the orthonomalized matrix ($P_{k+1}^t P_{k+1} = I$)

Let $\widetilde{P}_{k+1} = P_{k+1}$
for $i = 1 : t$ **do**
  for $j = 1 : (i - 1)$ **do**
    $\widetilde{P}_{k+1}(:, i) = \widetilde{P}_{k+1}(:, i) - (\widetilde{P}_{k+1}(:, j)^t P_{k+1}(:, i))\widetilde{P}_{k+1}(:, j)$
  end for
  $\widetilde{P}_{k+1}(:, i) = \frac{\widetilde{P}_{k+1}(:,i)}{||\widetilde{P}_{k+1}(:,i)||_2}$
end for

Algorithm 1 summarizes the CGS orthonormalization of P_{k+1}'s vectors against Q_k. However, a better parallelizable block version of the algorithm can be presented by eliminating the for loops as shown in Algorithm 2. As for the orthonormalization of the tall and skinny matrix P_{k+1} using CGS, it follows the same pattern as shown in Algorithm 3. In [71], the authors present a more stable version of CGS (Algorithm 3) that differs in the normalization step. The normalization in Algorithms **??** could be changed accordingly.

2.3.2 Modified Gram Schmidt (MGS)

It is known that CGS has some numerical stability problems due to the errors resulting from the projection of the initial vector repeatedly against all other vectors [36, 71] , i.e. $\widetilde{P}_{k+1}(:,j) = P_{k+1}(:,j) - \sum_{i=1}^{tk}(Q_k(:,i)^t P_{k+1}(:,j))Q_k(:,i)$. The modified Gram Schmidt partially fixes this issue by orthogonalizing the initial vector $P_{k+1}(:,j)$ against $Q_k(:,1)$ and then the obtained vector $\widetilde{P}_{k+1}(:,j)$ is orthogonalized against $Q_k(:,2)$ and so on. We will start by introducing the orthonormalization of P_{k+1}'s vectors against Q_k's vectors, then against each others using MGS. Algorithm 4 summarizes the MGS orthonormalization of P_{k+1}'s vectors against Q_k. As for the orthonormalization of the tall and skinny matrix P_{k+1} using MGS, it follows the same pattern as shown in Algorithm 5.

Algorithm 4 Orthonormalization against previous vectors with MGS

Input: Q_k, the tk orthonormal vectors
Input: P_{k+1}, the t vectors to be orthonormalized against Q
Output: $\widetilde{P}_{k+1}$, the vectors orthonomalized against Q

for $j = 1 : t$ **do**
  for $i = 1 : tk$ **do**
    $P_{k+1}(:,j) = P_{k+1}(:,j) - (Q_k(:,i)^t P_{k+1}(:,j))Q_k(:,i)$
  end for
  $P_{k+1}(:,j) = \frac{P_{k+1}(:,j)}{||P_{k+1}(:,j)||} = \frac{P_{k+1}(:,j)}{\sqrt{P_{k+1}(:,j)^t P_{k+1}(:,j)}}$
end for

Algorithm 5 Orthonormalization of tall and skinny matrix using MGS

Input: P_{k+1}, the matrix to be orthonormalized
Output: P_{k+1}, the orthonomalized matrix ($P_{k+1}^t P_{k+1} = I$)

for $i = 1 : t$ **do**
  for $j = 1 : (i-1)$ **do**
    $P_{k+1}(:,i) = P_{k+1}(:,i) - (P_{k+1}(:,j)^t P_{k+1}(:,i))P_{k+1}(:,j)$
  end for
  $P_{k+1}(:,i) = \frac{P_{k+1}(:,i)}{||P_{k+1}(:,i)||} = \frac{P_{k+1}(:,i)}{\sqrt{P_{k+1}(:,i)^t P_{k+1}(:,i)}}$
end for

2.3.3 Tall and skinny QR (TSQR)

The QR factorization of an $n \times t$ matrix P is its decomposition into an $n \times t$ orthogonal matrix Q ($Q^tQ = I$) and a $t \times t$ upper triangular matrix R. The QR factors can be obtained using Gram Schmidt orthogonalization, Givens rotations, Cholesky decomposition, or Householder reflections.

Tall and Skinny QR (TSQR) introduced in [21], refers to several algorithms that compute a QR factorization of a "tall and skinny" $n \times t$ matrix P ($n \gg t$) based on reduction trees with local Householder QR. The matrix P is partitioned row-wise $P = (P_0\ P_1\ ...\ P_{(pa-1)})^t$ where pa is the number of partitions. The sequential TSQR is based on a flat reduction tree, where pa is chosen so that the row-wise blocks of P fit into cache. Whereas, the parallel TSQR can be based on binary trees ($pa =$ number of processors) or general trees ($pa >$ number of processor). The TSQR version used in CA-GMRES [60] is a hybrid of parallel and sequential QR, where pa is chosen so that the row-wise blocks of P fit into cache. Then each processor is assigned a set of blocks that it factorizes using sequential TSQR to obtain Q and R factors. The obtained $t \times t$ R factors are stacked and factorized using LAPACK's QR.

Assuming that the number of row-wise partitions pa of P is four, then the sequential TSQR starts by performing the Householder QR factorization of the $\frac{n}{4} \times t$ matrix $P_0 = Q_0R_0$ where Q_0 is an $\frac{n}{4} \times t$ orthonormal matrix, R_0 is a $t \times t$ upper triangular matrix, and I is the $\frac{n}{4} \times \frac{n}{4}$ identity matrix.

$$P = \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix} = \begin{pmatrix} Q_0R_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix} = \left(\begin{array}{c|c|c|c} Q_0 & & & \\ \hline & I & & \\ \hline & & I & \\ \hline & & & I \end{array}\right) \begin{pmatrix} R_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix}$$

Then the obtained $t \times t$ R_0 is stacked over the $\frac{n}{4} \times t$ matrix P_1 and a Householder QR is performed, where Q_1 is an $(\frac{n}{4} + t) \times t$ orthonormal matrix and R_1 is a $t \times t$ upper triangular matrix. Similarly, the obtained R_1 is stacked over the $\frac{n}{4} \times t$ matrix P_2 and factorized using Householder QR, where Q_2 is an $(\frac{n}{4} + t) \times t$ orthonormal matrix and R_2 is a $t \times t$ upper triangular matrix . Finally, R_2 is stacked over P_3 and factorized into the $(\frac{n}{4} + t) \times t$ orthonormal matrix Q_3 and the $t \times t$ upper triangular matrix R_3.

$$\left(\begin{array}{c} R_0 \\ \hline P_1 \\ \hline P_2 \\ P_3 \end{array}\right) = \left(\begin{array}{c} Q_1R_1 \\ \hline P_2 \\ P_3 \end{array}\right), \quad \left(\begin{array}{c} R_1 \\ \hline P_2 \\ \hline P_3 \end{array}\right) = \left(\begin{array}{c} Q_2R_2 \\ \hline P_3 \end{array}\right), \quad \begin{pmatrix} R_2 \\ P_3 \end{pmatrix} = Q_3R_3$$

Then the R factor is R_3, and the Q factor is

$$Q = \left(\begin{array}{c|c|c|c} Q_0 & & & \\ \hline & I & & \\ \hline & & I & \\ \hline & & & I \end{array}\right) \left(\begin{array}{c|c|c} Q_1 & & \\ \hline & I & \\ \hline & & I \end{array}\right) \left(\begin{array}{c|c} Q_2 & \\ \hline & I \end{array}\right) Q_3$$

As for the hybrid TSQR, each of the four processors i decomposes its block P_i into the $\frac{n}{4} \times t$ orthonormal matrix Q_i and $t \times t$ upper triangular matrix R_i for $i = 0, 1, 2, 3$, using the sequential TSQR described above.

$$P = \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix} = \begin{pmatrix} Q_0 R_0 \\ Q_1 R_1 \\ Q_2 R_2 \\ Q_3 R_3 \end{pmatrix} = \left(\begin{array}{c|c|c|c} Q_0 & & & \\ \hline & Q_1 & & \\ \hline & & Q_2 & \\ \hline & & & Q_3 \end{array}\right) \begin{pmatrix} R_0 \\ R_1 \\ R_2 \\ R_3 \end{pmatrix}$$

Then the R_i factors, for $i = 0, 1, 2, 3$, are stacked and factorized using LAPACK's QR,

$$\begin{pmatrix} R_0 \\ R_1 \\ R_2 \\ R_3 \end{pmatrix} = Q_4 R_4, \text{ where } Q = \left(\begin{array}{c|c|c|c} Q_0 & & & \\ \hline & Q_1 & & \\ \hline & & Q_2 & \\ \hline & & & Q_3 \end{array}\right) Q_4, \text{ and } R = R_4.$$

Note that Q_4 is a $4t \times t$ orthonormal matrix, Q is the $n \times t$ output orthonormal matrix, and $R = R_4$ is the $t \times t$ output upper triangular matrix that satisfy $P = QR$

2.4 The A-orthonormalization

Given a set of k matrices P_i for $i = 1, 2, .., k$, of dimension $n \times t$, where the total tk column vectors are orthonormal, i.e. $P_i^t P_j = 0$ for $j \neq i$ and $P_i^t P_i = I$. Let P_{k+1} be an $n \times t$ newly computed matrix. We discuss in this section how to A-orthonormalize P_{k+1} against all the previous vectors P_i's for $i < k + 1$, and then against each others, to obtain $t(k + 1)$ orthonormal vectors. This will be used in chapters 4 for describing MSDO-CG method where at each iteration k, t new vectors are computed and have to be A-orthonormalized against the previously computed tk orthonormal vectors.

The A-orthonormalization is simply an orthonormalization with the A inner product ($< . , . >_A = < . , A . >$) rather than the L2 inner product ($< . , . >$). A-orthonormalizing a tall and skinny $n \times t$ matrix P_{k+1}, or alternatively computing the oblique QR factorization of P_{k+1}, has been discussed in [64] and [59] in terms of stability and ease of parallelization. The goal is to get a $\widetilde{P}_{k+1}$, such that $\widetilde{P}_{k+1}^t A \widetilde{P}_{k+1} = I$. There are two main classes for computing this oblique QR factorization of $P_{k+1} = \widetilde{P}_{k+1} R$. The first class is to factorize the matrix $A = B^t B$ using Cholesky decomposition or eigenvalue decomposition, which is expensive. Then $P_{k+1}^t A P_{k+1} = (B P_{k+1})^t (B P_{k+1})$, where the oblique QR factorization of P_{k+1} is transformed into a Euclidean QR factorization of the matrix $B P_{k+1} = Q_B R_B$ with $\widetilde{P}_{k+1} = B^{-1} Q_B$ and $R = R_B$. The second class consists of avoiding any factorization of A, like CGS, CGS2, MGS, and the Cholesky factorization of the $t \times t$ matrix $P_{k+1}^t A P_{k+1}$. For A-orthonormalizing P_{k+1} against all the previous vectors P_i with $i < k + 1$, it is possible to use CGS, CGS2, MGS and A-choleskyBGS which was discussed in Hoemmen's thesis ([48], page 115).

Thus, we start by discussing the A-orthonormalization using modified Gram Schmidt in section 2.4.1. However, this version is not easily parallelized on distributed-memory architectures, and requires a lot of communication ($(tk+1)log(t) + 2(t-1)log(t)$ messages) as compared to the classical Gram Schmidt version. Then, in section 2.4.2 we adapt the A-orthonormalization of the vectors of P_{k+1} against P_i's for $i < k+1$ using the classical Gram Schmidt (CGS) to obtain a Block Gram Schmidt (BGS) version (Algorithm 10) with A inner product that requires only $2log(t)$ messages. As for the A-orthonormalization of the P_{k+1} vectors against each others, we introduce a parallelizable version with reduced communication ($(2t-1)log(t)$ messages). Note that CGS2, section 2.4.2.3, consists of calling the algorithm CGS twice. Thus its cost is twice the cost of CGS. In section 2.4.3, we briefly discuss the A-orthonormalization of P_{k+1} using the Cholesky factorization (CholQR) of the $t \times t$ matrix $P_{k+1}^t A P_{k+1}$ which is referred to as A-CholQR and requires only $log(t)$ messages. We also present the Pre-CholQR version that was introduced in [59] and requires $3log(t)$ messages.

2.4.1 Modified Gram Schmidt A-orthonormalization

We start by introducing A-orthonormalization of the vectors of P_{k+1} against the vectors of all the P_i's for $i < k+1$, then against each others in section 2.4.1.1 . In section 2.4.1.2 , we discuss versions that save flops and reduce communication. And in section 2.4.1.3, the parallelization of both kernels is described.

2.4.1.1 The A-orthonormalization using MGS

Assuming that the vectors of P_i are A-normalized, i.e. $P_i(:,j)^t A P_i(:,j) = 1$ for all $j = 1, 2, .., t$ and $i = 1, 2, .., k$, then the A-orthonormalization of the vectors of P_{k+1} against the vectors of all the previous P_i's for $i < k+1$ is defined in Algorithm 6.

Algorithm 6 A-orthonormalization against previous vectors with MGS

Input: A, the $n \times n$ symmetric positive definite matrix
Input: P_1, P_2,.. , P_{k+1}, the $k+1$ sets of search directions
Output: P_{k+1}, the search directions A-orthonomalized against P_1, P_2,.. , P_k

1: **for** $o = 1 : t$ **do** %loop over the vectors of P_{k+1}
2: **for** $i = 1 : k$ **do** %loop over the different P_i's
3: **for** $j = 1 : t$ **do** %loop over the vectors of P_i
4: $P_{k+1}(:,o) = P_{k+1}(:,o) - (P_i(:,j)^t A P_{k+1}(:,o)) P_i(:,j)$
5: **end for**
6: **end for**
7: $P_{k+1}(:,o) = \frac{P_{k+1}(:,o)}{||P_{k+1}(:,o)||_A} = \frac{P_{k+1}(:,o)}{\sqrt{P_{k+1}(:,o)^t A P_{k+1}(:,o)}}$ %A-normalize
8: **end for**

At each inner iteration, one matrix-vector multiplication has to be computed ($AP_{k+1}(:, o)$), 1 dot product, and 1 saxpy, which costs $2\text{nnz}-n+(2n-1)+2n = 2\text{nnz}+3n-1$ flops. Then, at each outermost iteration, one matrix-vector multiplication is computed ($AP_{k+1}(:, o)$), 1 dot product, 1 square root and 1 division which costs $2\text{nnz} - n + (2n - 1) + 2 = 2\text{nnz} + n + 1$. The total cost of Algorithm 6 is $(2\text{nnz} + 3n - 1)t^2k + (2\text{nnz} + n + 1)t$, which is of the order of $\text{nnz}t^2k + nt^2k$.

As for the A-orthonormalization of the vectors of P_{k+1} against each others, it is defined in Algorithm 7. Similarly, the cost of the inner loop is $2\text{nnz} + 3n - 1$ flops and that of the outer loop is $2\text{nnz} + n + 1$, but the total cost is $(2\text{nnz} + 3n - 1)(t - 1)\frac{t}{2} + (2\text{nnz} + n + 1)t$ flops, which is of the order of $\text{nnz}t^2 + nt^2$.

Algorithm 7 A-orthonormalization against each others using MGS

Input: A, the $n \times n$ symmetric positive definite matrix
Input: P_{k+1}, the search directions to be A-orthonormalized
Output: P_{k+1}, the A-orthonomalized search directions

1: **for** $i = 1 : t$ **do** %loop over the vectors of P_{k+1}
2: **for** $j = 1 : (i - 1)$ **do** %A-orthogonalize against the vectors $P_{k+1}(:, 1 : i - 1)$
3: $P_{k+1}(:, i) = P_{k+1}(:, i) - (P_{k+1}(:, j)^t AP_{k+1}(:, i))P_{k+1}(:, j)$
4: **end for**
5: $P_{k+1}(:, i) = \frac{P_{k+1}(:,i)}{||P_{k+1}(:,i)||_A} = \frac{P_{k+1}(:,i)}{P_{k+1}(:,i)^t AP_{k+1}(:,i)}$ %A-normalize
6: **end for**

2.4.1.2 Saving flops in the A-orthonormalization using MGS

Since the A-orthonormalizations are expensive in term of flops, we present another alternative for computing the A-orthonormalizations that reduces the computed flops at the expense of storing more vectors. In Algorithm 7 and Algorithm 6, some matrix vector multiplications are repeatedly computed. For example in Algorithm 7, $AP_{k+1}(:, 1)$ is computed $t - 1$ times, $AP_{k+1}(:, 2)$ is computed $t - 2$ times, and generally, $AP_{k+1}(:, i)$ is computed $t - i$ times, which means that the matrix A is accessed $(t - 1)\frac{t}{2}$ times for every call of the algorithm. Thus, it is possible after A-orthogonalizing a vector $P_{k+1}(:, i)$ to compute and store $w_i = AP_{k+1}(:, i)$. This eliminates the redundant flops and reduces the number of accesses of A to t times, but there is a need to store t extra vectors (w_i).

Moreover, it is possible to further reduce the computations and the number of times A is accessed at the expense of storing tk vectors as shown in Algorithm 8, where the multiplication $W_{k+1} = AP_{k+1}$ is first performed by only reading the matrix A once. Then the vectors $W_{k+1}(:, i)$ are updated and stored.

The A-orthonormalization against previous vectors with flops reduction can be performed as in Algorithm 8. Then, the cost of the A-orthonormalization against previous vectors in Algorithm 8 is $(6n - 1)t^2k + (4n + 1)t$ of the order of $6nt^2k$ flops.

Algorithm 8 A-orthonormalization against previous vectors with MGS — Flops

Input: P_1, P_2,.. , P_{k+1}, the $k+1$ sets of search directions
Input: W_1, W_2,.. , W_{k+1}, the $k+1$ sets of AP_i
Output: P_{k+1}, the search directions A-orthonomalized against P_1, P_2,.. , P_k

1: **for** $o = 1 : t$ **do** %loop over the vectors of P_{k+1}
2: **for** $i = 1 : k$ **do** %loop over the different P_i's
3: **for** $j = 1 : t$ **do** %loop over the vectors of P_i
4: $P_{k+1}(:,o) = P_{k+1}(:,o) - (W_i(:,j)^t P_{k+1}(:,o))P_i(:,j)$ — $4n-1$
5: $W_{k+1}(:,o) = W_{k+1}(:,o) - (W_i(:,j)^t P_{k+1}(:,o))W_i(:,j)$ — $2n$
6: **end for**
7: **end for**
8: $pap_{k+1} = W_{k+1}(:,o)^t P_{k+1}(:,o)$ — $2n-1$
9: $P_{k+1}(:,o) = \frac{P_{k+1}(:,o)}{\sqrt{pap_{k+1}}}$ and $W_{k+1}(:,o) = \frac{W_{k+1}(:,o)}{\sqrt{pap_{k+1}}}$ — $2n+2$
10: **end for**

Algorithm 9 Flops reduction in A-orthonormalization against each others with MGS — Flops

Input: P_{k+1}, the search directions to be A-orthonormalized
Input: W_{k+1}, AP_{k+1}
Output: P_{k+1}, the A-orthonomalized search directions
Output: W_{k+1}, AP_{k+1} where P_{k+1} is the A-orthonomalized search directions

1: **for** $i = 1 : t$ **do** %loop over the vectors of P_{k+1}
2: **for** $j = 1 : (i-1)$ **do** %A-orthogonalize against the vectors $P_{k+1}(:, 1 : i-1)$
3: $P_{k+1}(:,i) = P_{k+1}(:,i) - (W_{k+1}(:,j)^t P_{k+1}(:,i))P_{k+1}(:,j)$ — $4n-1$
4: $W_{k+1}(:,i) = W_{k+1}(:,i) - (W_{k+1}(:,j)^t P_{k+1}(:,i))W_{k+1}(:,j)$ — $2n$
5: **end for**
6: $pap_{k+1} = W_{k+1}(:,i)^t P_{k+1}(:,i)$ — $2n-1$
7: $P_{k+1}(:,i) = \frac{P_{k+1}(:,i)}{\sqrt{pap_{k+1}}}$ and $W_{k+1}(:,i) = \frac{W_{k+1}(:,i)}{\sqrt{pap_{k+1}}}$ — $2n+2$
8: **end for**

As for the A-orthonormalization of P_{k+1}'s vectors against each others using MGS with flops reductions, it can be performed as in Algorithm 9. The cost of this version of A-orthonormalization (Algorithm 9) is $(6n-1)(t-1)\frac{t}{2} + (4n+1)t$, which is of the order of $3nt^2$.

2.4.1.3 Parallelization of the A-orthonormalization using MGS

In Algorithm 8, at each inner iteration we are A-orthonormalizing the updated vectors $P_{k+1}(:,o)$ against the vector $P_i(:,j)$, where the vector $P_{k+1}(:,o)$ is changed at each inner iteration. Thus it is not possible to have a block MGS by eliminating all the for loops. However, it is possible to eliminate one for loop in Algorithm 8 as shown in Algorithm 10, by A-orthonormalizing the whole block P_{k+1} against the vector $P_i(:,j)$, where $P_{k+1}(:,o) = P_{k+1}(:,o) - (P_{k+1}(:,o)^t W_i(:,j))P_i(:,j)$ for all $o = 1, 2, ..., t$. Let $[P_i(:,j)]_t$ be an $n \times t$ block containing t duplicates of the vector $P_i(:,j)$. Then, $P_{k+1} = P_{k+1} - [P_i(:,j)]_t diag(P_{k+1}^t W_i(:,j))$.

Algorithm 10 A-orthonormalization against previous vectors with MGS — Flops

Input: P_1, P_2,.. , P_{k+1}, the $k+1$ sets of search directions
Input: W_1, W_2,.. , W_{k+1}, the $k+1$ sets of AP_i
Output: P_{k+1}, the search directions A-orthonomalized against P_1, P_2,.. , P_k

1:	**for** $i = 1 : k$ **do** %loop over the different P_i's	
2:	**for** $j = 1 : t$ **do** %loop over the vectors of P_i	
3:	$P_{k+1} = P_{k+1} - [P_i(:,j)]_t diag(W_i(:,j)^t P_{k+1})$	$(4n-1)t$
4:	$W_{k+1} = W_{k+1} - [W_i(:,j)]_t diag(W_i(:,j)^t P_{k+1})$	$2nt$
5:	**end for**	
6:	**end for**	
7:	**for** $o = 1 : t$ **do**	
8:	$pap_{k+1}(o) = W_{k+1}(:,o)^t P_{k+1}(:,o)$	$2n-1$
9:	**end for**	
10:	$pap_{k+1} = (\sqrt{pap_{k+1}})$	t
11:	$P_{k+1} = P_{k+1} diag(pap_{k+1})^{-1}$ and $W_{k+1} = W_{k+1} diag(pap_{k+1})^{-1}$	$(2n+2)t$

In Algorithm 9, rather than A-orthonormalizing each vector $P_{k+1}(:,i)$ against all previous vectors $P_{k+1}(:,j)$, we can A-orthogonalize $P_{k+1}(:,i+1:t)$ against the A-normalized vector $P_{k+1}(:,i)$ as shown in Algorithm 11. Let $[P_{k+1}(:,j)]_{t-i}$ be an $n \times (t-i)$ block containing $t-i$ duplicates of the vector $P_{k+1}(:,j)$. Then $P_{k+1}(:,i+1:t) = P_{k+1}(:,i+1:t) - [P_{k+1}(:,j)]_{t-i} diag(W_{k+1}(:,i)^t P_{k+1}(:,i+1:t))$

Then the parallelization of Algorithms 10 and 11 goes as follows. We assume that we have t processors with distributed memory, and each processor pi is assigned a rowwise part of all W_j ($W_j(\delta_{pi},:)$) for $j = 1, 2, .., k+1$ and the same rowwise part of all P_j ($P_j(\delta_{pi},:)$) for $j = 1, 2, .., k+1$ where $\delta_{pi} \cap \delta_h = \phi$ for all $pi \neq h$ and $\cup_{h=1}^{t} \delta_h = \{1, 2, 3, ..., n\}$.

At each inner iteration of Algorithm 10, each processor pi has to compute $P_{k+1}(\delta_{pi},:) = P_{k+1}(\delta_{pi},:) - [P_i(\delta_{pi},j)]_t diag(W_i(:,j)^t P_{k+1})$. First, each processor pi computes a part of the matrix vector multiplication $W_i(\delta_{pi},j)^t P_{k+1}(\delta_{pi},:)$. Then, a communication of the form "all reduce" is performed to send the $1 \times t$ $W_i(:,j)^t P_{k+1}$'s value to all the processors. Finally, processor pi computes $P_{k+1}(\delta_{pi},:)$ and $W_{k+1}(\delta_{pi},:)$.

Algorithm 11 A-orthonormalization against each others with MGS

Input: P_{k+1}, the search directions to be A-orthonormalized
Input: W_{k+1}, AP_{k+1}
Output: P_{k+1}, the A-orthonomalized search directions
Output: W_{k+1}, AP_{k+1} where P_{k+1} is the A-orthonomalized search directions
1: **for** $i = 1 : (t-1)$ **do** %A-orthogonalize against the vectors $P_{k+1}(:, 1:i-1)$
2: $P_{k+1}(:, i+1:t) = P_{k+1}(:, i+1:t) - [P_{k+1}(:, j)]_{t-i} diag(W_{k+1}(:, i)^t P_{k+1}(:, i+1:t))$
3: $W_{k+1}(:, i+1:t) = W_{k+1}(:, i+1:t) - [W_{k+1}(:, j)]_{t-i} diag(W_{k+1}(:, i)^t P_{k+1}(:, i+1:t))$
4: $pap_{k+1} = W_{k+1}(:, i+1)^t P_{k+1}(:, i+1)$
5: $P_{k+1}(:, i+1) = \frac{P_{k+1}(:,i+1)}{\sqrt{pap_{k+1}}}$ and $W_{k+1}(:, i+1) = \frac{W_{k+1}(:,i+1)}{\sqrt{pap_{k+1}}}$
6: **end for**

Finally, each processor pi computes its corresponding part of the dot product $W_{k+1}(\delta_i, o)^t P_{k+1}(\delta_i, o)$ for all $o = 1, 2, .., t$ and an "all reduce" is used to send pap_{k+1}'s value to all the processors. Then, each processor A-normalizes $P_{k+1}(\delta_{pi}, o)$ and $W_{k+1}(\delta_{pi}, o)$. All the communication in Algorithm 10 is of the form "all reduce" of a $t \times 1$ vector which is equivalent to sending $log(t)$ messages and $tlog(t)$ words. So, in total $(tk+1)log(t)$ messages and $(tk+1)tlog(t)$ words are sent in Algorithm 8. Hence, by ignoring lower order terms we obtain

$$Time_{MGS1_{Aort}} \approx \gamma_c 6nt^2k + \alpha_c tklog(t) + \beta_c t^2 klog(t)$$

As for the parallelization of Algorithm 11, it is similar to that of Algorithm 10 where at each inner iteration processor pi computes a part of the matrix vector multiplication $W_{k+1}(\delta_{pi}, i)^t P_{k+1}(\delta_{pi}, i+1:t)$ and then receives the whole $1 \times t$ vector, $W_{k+1}(:, i)^t P_{k+1}(:, i+1:t)$, using an "all reduce". Then, it computes $P_{k+1}(\delta_{pi}, :)$, $W_{k+1}(\delta_{pi}, :)$ and a part of the dot product pap_{k+1}, and receives the whole dot product by an "all reduce". Finally, each processor A-normalizes its part of $P_{k+1}(:, i)$ and $W_{k+1}(:, i)$. Thus, at each iteration 2 "all reduce" communications are performed, where t words are sent in the first and one word in second. So, in total $2(t-1)log(t)$ messages are sent in Algorithm 11 where $(t-1)(t+1)log(t)$ words are sent. Hence, by ignoring lower order terms we obtain

$$Time_{MGS2_{Aort}} \approx \gamma_c 3nt^2 + \alpha_c 2tlog(t) + \beta_c t^2 log(t)$$

2.4.2 Classical Gram Schmidt A-orthonormalization

Since the MGS A-orthonormalization is costly in terms of communication, we introduce the classical Gram Schmidt (CGS) A-orthonormalization and show that it is equivalent to a QR decomposition with A inner product rather than the usual L2 inner product. Then we present the parallelization of the introduced algorithms. In section 2.4.2.1, the A-orthonormalization against previous vectors using CGS is discussed, whereas in section 2.4.2.2 we discuss the A-orthonormalization of the vectors using CGS. Then in section 2.4.2.3 we introduce the CGS A-orthonormalization with reorthogonalization.

2.4.2.1 A-orthonormalization against previous vectors using CGS

The A-orthonormalization of P_{k+1} against the vectors of all the previous P_i's for $i < k+1$ is defined as in Algorithm 12.

Algorithm 12 A-orthonormalization against previous vectors with CGS

Input: A, the $n \times n$ symmetric positive definite matrix
Input: P_1, P_2,.. , P_{k+1}, the $k+1$ sets of search directions
Output: $\widetilde{P}_{k+1}$, the search directions A-orthonomalized against P_1, P_2,.. , P_k

Let $\widetilde{P}_{k+1} = P_{k+1}$
for $o = 1 : t$ **do** %loop over the vectors of P_{k+1}
  for $i = 1 : k$ **do** %loop over the different P_i's
    for $j = 1 : t$ **do** %loop over the vectors of P_i
      $\widetilde{P}_{k+1}(:, o) = \widetilde{P}_{k+1}(:, o) - (P_i(:, j)^t A P_{k+1}(:, o)) P_i(:, j)$
    end for
  end for
  $\widetilde{P}_{k+1}(:, o) = \frac{\widetilde{P}_{k+1}(:,o)}{||\widetilde{P}_{k+1}(:,o)||_A} = \frac{\widetilde{P}_{k+1}(:,o)}{\sqrt{\widetilde{P}_{k+1}(:,o)^t A \widetilde{P}_{k+1}(:,o)}}$ %A-normalize
end for

More precisely,

$$\begin{aligned} \widetilde{P}_{k+1}(:, o) &= P_{k+1}(:, o) - \sum_{i=1}^{k} \sum_{j=1}^{t} (P_i(:, j)^t A P_{k+1}(:, o)) P_i(:, j) \\ &= P_{k+1}(:, o) - \sum_{i=1}^{k} P_i (P_i^t A P_{k+1}(:, o)) \end{aligned}$$

If we let $W_{k+1} = AP_{k+1}$, then $\widetilde{P}_{k+1}(:, o) = P_{k+1}(:, o) - \sum_{i=1}^{k} P_i (P_i^t W_{k+1}(:, o))$. Moreover, $\widetilde{P}_{k+1} = P_{k+1} - \sum_{i=1}^{k} P_i (P_i^t W_{k+1})$. Let $Q_k = [P_1, P_2, ..., P_k]$, then $\widetilde{P}_{k+1} = P_{k+1} - Q_k (Q_k^t W_{k+1})$. This represents a Block classical gram schmidt (BCGS) version of the A-orthonormalization (Algorithm 13). The total flops performed in Algorithm 13 is

$$\begin{aligned} Total\ Flops &= 2(2\text{nnz} - n)t + (2n-1)t^2 k + 2t^2 kn + 3nt \\ &= 4\text{nnz}t + nt + [4nt^2 - t^2]k \\ &\approx 4\text{nnz}t + 4nt^2 k \end{aligned}$$

As for the parallelization of Algorithm 13, it is straightforward due to the block format. Assuming that we have t processors with distributed memory, and each processor pi is assigned a rowwise part of A ($A(\delta_{pi}, :)$), a rowwise part of Q_k ($Q_k(\delta_{pi}, :)$) and a rowwise part of P_{k+1}, where $\delta_{pi} \cap \delta_h = \phi$ for all $pi \neq h$ and $\cup_{h=1}^{t} \delta_h = \{1, 2, 3, ..., n\}$.

First, processor pi computes $A(:, \delta_{pi}) P_{k+1}(\delta_{pi}, :)$ and receives the full $n \times t$ matrix W_{k+1} via an "all reduce". Then it computes $Q_k(\delta_{pi}, :)^t W_{k+1}(\delta_{pi}, :)$ and obtains the full $tk \times t$ matrix $Q_k^t W_{k+1}$ using an "all reduce". Then, processor pi computes $\widetilde{P}_{k+1}(\delta_{pi}, :) = P_{k+1}(\delta_{pi}, :) - Q_k(\delta_{pi}, :$

Algorithm 13 A-orthonormalization against previous vectors with BCGS — Flops

Input: A, the $n \times n$ symmetric positive definite matrix
Input: $Q_k = [P_1, P_2, ..., P_k]$, the tk search directions
Input: P_{k+1}, the t search directions to be A-orthonormalized
Output: $\widetilde{P}_{k+1}$, the search directions A-orthonomalized against P_1, P_2,.. , P_k

1: $W_{k+1} = AP_{k+1}$ — $(2\text{nnz} - n)t$
2: $\widetilde{P}_{k+1} = P_{k+1} - Q_k(Q_k^t W_{k+1})$ — $(2n-1)t^2k + (2tk-1)nt + nt$
3: $\widetilde{W}_{k+1} = A\widetilde{P}_{k+1}$ — $(2\text{nnz} - n)t$
4: **for** $i = 1 : t$ **do** %loop over the vectors of P_{k+1} and A-normalize
5: $\widetilde{P}_{k+1}(:,i) = \frac{\widetilde{P}_{k+1}(:,i)}{||\widetilde{P}_{k+1}(:,i)||_A} = \frac{\widetilde{P}_{k+1}(:,i)}{\sqrt{\widetilde{P}_{k+1}(:,i)^t W_{k+1}(:,i)}}$ — $3n$
6: **end for**

$)(Q_k^t W_{k+1})$. Another "all reduce" is needed so that processor pi has the full $\widetilde{P}_{k+1}$ needed to compute $\widetilde{W}_{k+1}(\delta_{pi},:) = A(\delta_{pi},:)\widetilde{P}_{k+1}$. Processor pi computes t partial dot products of the form $\widetilde{P}_{k+1}(\delta_{pi},o)^t W_{k+1}(\delta_{pi},o)$ and obtains the full dot products via an all reduce. Finally each processor A-normalizes its part of P_{k+1}, i.e $\widetilde{P}_{k+1}(\delta_{pi},i) = \frac{\widetilde{P}_{k+1}(\delta_{pi},i)}{\widetilde{P}_{k+1}(:,i)^t W_{k+1}(:,i)}$ for all $i = 1, 2, .., t$. So in total there is a need to perform 4 all reduce for parallelizing Algorithm 13.

It is possible to reduce the communication to only two by assuming that $W_{k+1} = AP_{k+1}$ has already been computed and it is an input to Algorithm 14 along with $\mathcal{W}_k = AQ_k = [W_1, W_2, ..., W_k]$. The only communication is an "all reduce" of the $tk \times t$ matrix $Q_k^t W_{k+1}$, and another "all reduce" of the vector of size t containing the norms of the columns of $\widetilde{P}_{k+1}$. We assume that it is possible to send a message of size t^2k words at once. Thus, $2log(t)$ messages are sent with $(tk+1)tlog(t)$ words where $\frac{(6n-1)t^2k+4nt}{t} = (6n-1)tk + 4n$ flops are performed in parallel. Hence, by ignoring lower order terms we obtain

$$Time_{BCGS_{Aort}} \approx \gamma_c 6ntk + \alpha_c 2log(t) + \beta_c t^2 klog(t)$$

2.4.2.2 A-orthonormalization of a set of vectors using CGS

Given a set of vectors P_{k+1} that are A-normalized, i.e the diagonal of $P_{k+1}^t AP_{k+1}$ is equal to ones, we A-orthonormalize it ($P_{k+1}^t AP_{k+1} = I$) using a classical Gram Schmidt procedure as in Algorithm 15.

The CGS A-orthonormalization can be reformulated as a QR factorization

$$P_{k+1} = \widetilde{P}_{k+1} R$$

where $\widetilde{P}_{k+1}$ is an A-orthonormal matrix, and R is a $t \times t$ upper triangular matrix defined by the

Algorithm 14 A-orthonormalization against previous vectors with BCGS — Flops

Input: $Q_k = [P_1, P_2, ..., P_k]$, the tk search directions
Input: P_{k+1}, the t search directions to be A-orthonormalized
Input: $W_{k+1} = AP_{k+1}$; $\mathcal{W}_k = AQ_k$
Output: $\widetilde{P}_{k+1}$, the search directions A-orthonomalized against Q_k; $\widetilde{W}_{k+1} = A\widetilde{P}_{k+1}$

1: $\widetilde{P}_{k+1} = P_{k+1} - Q_k(Q_k^t W_{k+1})$ — $(2n-1)t^2k + (2tk-1)nt + nt$
2: $\widetilde{W}_{k+1} = W_{k+1} - \mathcal{W}_k(Q_k^t W_{k+1})$ — $2nt^2k$
3: **for** $i = 1 : t$ **do** %loop over the vectors of P_{k+1} and A-normalize
4: Let $np = \sqrt{\widetilde{P}_{k+1}(:,i)^t \widetilde{W}_{k+1}(:,i)}$ — $2n$
5: $\widetilde{P}_{k+1}(:,i) = \frac{\widetilde{P}_{k+1}(:,i)}{np}$, and $\widetilde{W}_{k+1}(:,i) = \frac{\widetilde{W}_{k+1}(:,i)}{np}$ — $2n$
6: **end for**

Algorithm 15 A-orthonormalization against each others using CGS

Input: A, the $n \times n$ symmetric positive definite matrix
Input: P_{k+1}, the search directions to be A-orthonormalized
Output: P_{k+1}, the A-orthonomalized search directions

1: Let $\widetilde{P}_{k+1} = P_{k+1}$
2: **for** $i = 1 : t$ **do** %loop over the vectors of P_{k+1}
3: **for** $j = 1 : (i-1)$ **do** %A-orthogonalize against the vectors $P_{k+1}(:, 1 : i-1)$
4: $\widetilde{P}_{k+1}(:,i) = \widetilde{P}_{k+1}(:,i) - (\widetilde{P}_{k+1}(:,j)^t A P_{k+1}(:,i))\widetilde{P}_{k+1}(:,j)$
5: **end for**
6: $\widetilde{P}_{k+1}(:,i) = \frac{\widetilde{P}_{k+1}(:,i)}{||\widetilde{P}_{k+1}(:,i)||_A} = \frac{\widetilde{P}_{k+1}(:,i)}{\sqrt{\widetilde{P}_{k+1}(:,i)^t A \widetilde{P}_{k+1}(:,i)}}$ %A-normalize
7: **end for**

entries $r_{j,i}$ for all $j = 1, 2, .., i$ and $i = 1, 2, .., t$.

$$R = \begin{pmatrix} r_{1,1} & r_{1,2} & r_{1,3} & \cdots & r_{1,t} \\ & r_{2,2} & r_{2,3} & \cdots & r_{2,t} \\ & & r_{3,3} & \cdots & r_{3,t} \\ & & & \ddots & \vdots \\ & & & & r_{t,t} \end{pmatrix} = \begin{pmatrix} ||p_1||_A & < \tilde{p}_1, p_2 >_A & < \tilde{p}_1, p_3 >_A & \cdots & < \tilde{p}_1, p_t >_A \\ & r_{2,2} & < \tilde{p}_2, p_3 >_A & \cdots & < \tilde{p}_2, p_t >_A \\ & & r_{3,3} & \cdots & < \tilde{p}_3, p_t >_A \\ & & & \ddots & \vdots \\ & & & & r_{t,t} \end{pmatrix}$$

Although the CGS A-orthonormalization is equivalent to a QR factorization with the A inner product, we were not able to parallelize it using reduction trees with the same communication pattern as in TSQR [21]. But we can optimize the communication in Algorithm 15 by noticing that once a vector p_i is orthonormalized, we can compute the corresponding entries of the matrix R, i.e $R(i, i+1 : t)$. By taking this into consideration, algorithm 15 can be restructured, as shown in

algorithm 16.

Algorithm 16 QR factorization with A inner product using CGS — Flops

Input: A, the $n \times n$ symmetric positive definite matrix
Input: P_{k+1}, the search directions to be A-orthonormalized
Output: $\widetilde{P}_{k+1}$, the A-orthonomalized search directions $\widetilde{P}_{k+1}^t A \widetilde{P}_{k+1} = I$
Output: R, the upper triangular matrix such that $P_{k+1} = \widetilde{P}_{k+1} R$

1:	$W_{k+1} = AP_{k+1}$	$(2\text{nnz} - n)t$
2:	$R(1,1) = \sqrt{P_{k+1}(:,1)^t W(:,1)}$	$2n$
3:	$\widetilde{P}_{k+1}(:,1) = \frac{P_{k+1}(:,1)}{R(1,1)}$	n
4:	**for** $i = 2 : t$ **do**	
5:	$R(i-1, i:t) = \widetilde{P}_{k+1}(:, i-1)^t W_{k+1}(:, i:t)$	$(2n-1)(t-i+1)$
6:	$\widetilde{P}_{k+1}(:,i) = P_{k+1}(:,i) - \widetilde{P}_{k+1}(:, 1:i-1)R(1:i-1, i)$	$(2(i-1)-1)n + n$
7:	$R(i,i) = \sqrt{\widetilde{P}_{k+1}(:,i)^t A \widetilde{P}_{k+1}(:,i)}$	$(2\text{nnz} - n) + 2n$
8:	$\widetilde{P}_{k+1}(:,i) = \frac{\widetilde{P}_{k+1}(:,i)}{R(i,i)}$	n
9:	**end for**	

The total flops of Algorithm 16 is of the order of $\text{nnz}t + nt^2$.

$$\begin{aligned}
\text{Total} &= 2\text{nnz}t - nt + 3n + \textstyle\sum_{i=2}^{t}[(2n-1)(t-i+1) + 2(i-1)n + (2\text{nnz} + 2n)] \\
&= 2\text{nnz}t - nt + 3n + \textstyle\sum_{i=2}^{t}[(2n-1)(t+1) - (2n-1)i + 2ni - 2n + 2\text{nnz} + 2n] \\
&= 2\text{nnz}t - nt + 3n + \textstyle\sum_{i=2}^{t}[(2n-1)(t+1) + i + 2\text{nnz}] \\
&= 2\text{nnz}t - nt + 3n + [(2n-1)(t+1) + 2\text{nnz}](t-1) + \tfrac{t(t+1)}{2} - 1 \\
&= 4\text{nnz}t - 2\text{nnz} - nt + 3n + (2n-1)(t^2-1) + \tfrac{t^2+t}{2} - 1 \\
&= 4\text{nnz}t - 2\text{nnz} - nt + n + 2nt^2 - t^2 + \tfrac{t^2+t}{2} \\
&= 4\text{nnz}t - 2\text{nnz} - nt + n + 2nt^2 + \tfrac{-t^2+t}{2}
\end{aligned}$$

The parallelization of Algorithm 16 starts by distributing the data similarly to Algorithm 13. Processor pi computes $A(:, \delta_{pi})P_{k+1}(\delta_{pi}, :)$ and receives W_{k+1} via an "all reduce", and computes $P_{k+1}(\delta_{pi}, 1)^t W(\delta_{pi}, 1)$, and receives the full dot product $P_{k+1}(:,1)^t W(:,1)$, needed to compute $R(1,1)$, via an "all reduce". Then, it computes $\widetilde{P}_{k+1}(\delta_{pi}, 1) = \frac{P_{k+1}(\delta_{pi},1)}{R(1,1)}$.

At each iteration , processor pi computes $\widetilde{P}_{k+1}(\delta_{pi}, i-1)^t W_{k+1}(\delta_{pi}, i:t)$ and receives the full $R(i-1, i:t)$ by an all reduce. Then, it computes $\widetilde{P}_{k+1}(\delta_{pi}, i) = P_{k+1}(\delta_{pi}, i) - \widetilde{P}_{k+1}(\delta_{pi}, 1 : i-1)R(1 : i-1, i)$ and receives the $P_{k+1}(\beta_{pi}, i)$ from m_{MB} adjacent processors where $\beta_{pi} = Adjacent(G(A), \delta_{pi})$. Then it computes $\widetilde{W}_{k+1}(\delta_{pi}, i) = A(\delta_{pi}, \beta_{pi})\widetilde{P}_{k+1}(\beta_{pi}, i)$ and $\widetilde{P}_{k+1}(\delta_{pi}, i)^t \widetilde{W}_{k+1}(\delta_{pi}, i)$ and receives the full $\widetilde{P}_{k+1}(:,i)^t A \widetilde{P}_{k+1}(:,i)$, needed to compute $R(i,i)$, via an all reduce. Finally, it computes $\widetilde{P}_{k+1}(\delta_{pi}, i) = \frac{\widetilde{P}_{k+1}(\delta_{pi},i)}{R(i,i)}$. So, there is a need for a total of $2t-1$ "all reduce" and t communications with the m_{MB} neighboring processors.

Algorithm 17 QR factorization with A inner product using CGS — Flops

Input: P_{k+1}, the search directions to be A-orthonormalized; W_{k+1}, $= AP_{k+1}$
Output: $\widetilde{P}_{k+1}$, the A-orthonomalized search directions; $\widetilde{W}_{k+1}$
Output: R, the upper triangular matrix such that $P_{k+1} = \widetilde{P}_{k+1}R$

1:	$R(1,1) = \sqrt{P_{k+1}(:,1)^t W(:,1)}$	$2n$
2:	$\widetilde{P}_{k+1}(:,1) = \frac{P_{k+1}(:,1)}{R(1,1)}$ and $\widetilde{W}_{k+1}(:,1) = \frac{W_{k+1}(:,1)}{R(1,1)}$	$2n$
3:	**for** $i = 2 : t$ **do**	
4:	$R(i-1, i:t) = \widetilde{P}_{k+1}(:,i-1)^t W_{k+1}(:,i:t)$	$(2n-1)(t-i+1)$
5:	$\widetilde{P}_{k+1}(:,i) = P_{k+1}(:,i) - \widetilde{P}_{k+1}(:,1:i-1)R(1:i-1,i)$	$(2(i-1)-1)n + n$
6:	$\widetilde{W}_{k+1}(:,i) = W_{k+1}(:,i) - \widetilde{W}_{k+1}(:,1:i-1)R(1:i-1,i)$	$2(i-1)n$
7:	$R(i,i) = \sqrt{\widetilde{P}_{k+1}(:,i)^t \widetilde{W}_{k+1}(:,i)}$	$2n$
8:	$\widetilde{P}_{k+1}(:,i) = \frac{\widetilde{P}_{k+1}(:,i)}{R(i,i)}$ and $\widetilde{W}_{k+1}(:,i) = \frac{\widetilde{W}_{k+1}(:,i)}{R(i,i)}$	$2n$
9:	**end for**	

It is possible to reduce the communication to only $2t-1$ "all reduce" by assuming that $W_{k+1} = AP_{k+1}$ has already been computed and it is an input to Algorithm 17. Then, at each iteration i, an "all reduce" of the vector $R(i-1, i:t)$ of size $t-i+1$ is performed and another "all reduce" of the entry $R(i,i)$ is performed. Thus, a total of $(2t-1)log(t)$ messages are sent with $(1 + \sum_{i=2}^{t} t + 2 - i)log(t) = \frac{t(t+1)}{2}log(t)$ words where $\frac{3nt^2+nt+\frac{t(1-t)}{2}}{t} = 3nt + n + \frac{(1-t)}{2}$ flops are performed in parallel. Hence, by ignoring lower order terms we obtain

$$Time_{QRCGS_{Aort}} \approx \gamma_c 3nt + \alpha_c 2tlog(t) + \beta_c t^2 log(t)$$

2.4.2.3 CGS with reorthogonalization (CGS2)

The CGS with reorthogonalization (CGS) consists of calling the CGS algorithms twice, be it for A-orthonormalizing P_{k+1} against previous vectors of Q_k (Algorithm 18), or A-orthonormalizing P_{k+1}.

Algorithm 18 A-orthonormalization of P_{k+1} against previous vectors of Q_k using CGS2

Input: Q_k, the tk search directions
Input: P_{k+1}, the t search directions to be A-orthonormalized
Input: W_{k+1}, $= AP_{k+1}$; $\mathcal{W}_k$, $= AQ_k$
Output: $\widetilde{P}_{k+1}$, the search directions A-orthonomalized against P_1, P_2,.. , P_k; $\widetilde{W}_{k+1} = A\widetilde{P}_{k+1}$

1: Call Algorithm 14 with P_{k+1} and W_{k+1} as input and with P'_{k+1} and W'_{k+1} as output
2: Call Algorithm 14 with P'_{k+1} and W'_{k+1} as input and with $\widetilde{P}_{k+1}$ and $\widetilde{W}_{k+1}$ as output

In the case of $L^{-1}AL^{-t}$-orthonormalization of P_{k+1} against previous vectors of Q_k where $L^{-t} = (L^t)^{-1}$, the CGS2 algorithm is defined in Algorithm 19. Note that we have to solve 6 systems with multiple right hand sides. If L is a lower triangular matrix, then we perform three backward substitutions and three forward substitutions.

Algorithm 19 $L^{-1}AL^{-t}$-orthonormalization against previous vectors of Q_k with CGS2

Input: A, the $n \times n$ symmetric positive definite matrix; L, $n \times n$ preconditioner
Input: Q_k, the tk search directions
Input: P_{k+1}, the t search directions to be $L^{-1}A(L^t)^{-1}$-orthonormalized
Output: $\widehat{P}_{k+1}$, the search directions $L^{-1}A(L^t)^{-1}$-orthonomalized against Q_k
Output: $\widehat{W}_{k+1} = L^{-1}AL^{-t}\widehat{P}_{k+1}$

$W_{k+1} = L^{-1}AL^{-t}P_{k+1}$
$\widetilde{P}_{k+1} = P_{k+1} - Q_k(Q_k^t W_{k+1})$
$\widetilde{W}_{k+1} = L^{-1}AL^{-t}\widetilde{P}_{k+1}$
for $i = 1 : t$ **do** %loop over the vectors of P_{k+1} and $L^{-1}AL^{-t}$-normalize
  $\widetilde{P}_{k+1}(:,i) = \frac{\widetilde{P}_{k+1}(:,i)}{||\widetilde{P}_{k+1}(:,i)||_{L^{-1}AL^{-t}}} = \frac{\widetilde{P}_{k+1}(:,i)}{\sqrt{\widetilde{P}_{k+1}(:,i)^t\widetilde{W}_{k+1}(:,i)}}$ and $\widetilde{W}_{k+1}(:,i) = \frac{\widetilde{W}_{k+1}(:,i)}{\sqrt{\widetilde{P}_{k+1}(:,i)^t\widetilde{W}_{k+1}(:,i)}}$
end for
$\widehat{P}_{k+1} = \widetilde{P}_{k+1} - Q_k(Q_k^t\widetilde{W}_{k+1})$
$\widehat{W}_{k+1} = L^{-1}AL^{-t}\widehat{P}_{k+1}$
for $i = 1 : t$ **do** %loop over the vectors of P_{k+1} and $L^{-1}AL^{-t}$-normalize
  $\widehat{P}_{k+1}(:,i) = \frac{\widehat{P}_{k+1}(:,i)}{||\widehat{P}_{k+1}(:,i)||_{L^{-1}AL^{-t}}} = \frac{\widehat{P}_{k+1}(:,i)}{\sqrt{\widehat{P}_{k+1}(:,i)^t\widehat{W}_{k+1}(:,i)}}$ and $\widehat{W}_{k+1}(:,i) = \frac{\widehat{W}_{k+1}(:,i)}{\sqrt{\widehat{P}_{k+1}(:,i)^t\widehat{W}_{k+1}(:,i)}}$
end for

2.4.3 Cholesky QR A-orthonormalization

A-orthonormalizing the $n \times t$ full rank matrix P_{k+1} is equivalent to a QR factorization $P_{k+1} = \widetilde{P}_{k+1}R$ where $\widetilde{P}_{k+1}^t A\widetilde{P}_{k+1} = I$. Thus, $P_{k+1}^t AP_{k+1} = (\widetilde{P}_{k+1}R)^t A\widetilde{P}_{k+1}R = R^tR$ and R can be obtained by performing a Cholesky factorization of the SPD matrix $P_{k+1}^t AP_{k+1}$. Then, $\widetilde{P}_{k+1} = P_{k+1}R^{-1}$ is obtained by solving the lower triangular system $R^t\widetilde{P}_{k+1}^t = P_{k+1}^t$ with multiple right-hand sides. This procedure is called A-CholQR and summarized in Algorithm 20 [64, 59]. Similarly to the other A-orthonormalization methods, we may assume that W_{k+1} is already computed, then the obtained A-CholQR is described in Algorithm 21. Note that the only difference between the A-CholQR, Algorithms 20 and 21, for A-orthonormalizing P_{k+1}, and the CholQR algorithm [72] for orthonormalizing P_{k+1} is in the definition of C. In A-CholQR $C = P_{k+1}^t AP_{k+1}$, whereas in CholQR $C = P_{k+1}^t P_{k+1}$.

The parallelization of Algorithm 21 assumes that we have t processors and each is assigned a rowwise part of P_{k+1} and W_{k+1} corresponding to the δ_i subset of indices defined previously, $P_{k+1}(\delta_i,:)$ and $W_{k+1}(\delta_i,:)$. And each processor i should compute $\widetilde{P}_{k+1}(\delta_i,:)$ and $\widetilde{W}_{k+1}(\delta_i,:)$.

Algorithm 20 A-CholQR — Flops

Input: A, the $n \times n$ symmetric positive definite matrix
Input: P_{k+1}, the search directions to be A-orthonormalized
Output: $\widetilde{P}_{k+1}$, the A-orthonomalized search directions

1: Compute $W_{k+1} = AP_{k+1}$ — $(2\text{nnz} - n)t$
2: Compute $C = W_{k+1}^t P_{k+1}$ — $(2n-1)t^2$
3: Compute the Cholesky factorization of $C = R^t R$ to obtain R — t^2
4: Solve $R^t \widetilde{P}_{k+1}^t = P_{k+1}^t$ — nt^2
5: **for** $i = 1 : t$ **do**
6: $\quad \widetilde{P}_{k+1}(:,i) = \frac{\widetilde{P}_{k+1}(:,i)}{||\widetilde{P}_{k+1}(:,i)||_A} = \frac{\widetilde{P}_{k+1}(:,i)}{\sqrt{\widetilde{P}_{k+1}(:,i)^t A \widetilde{P}_{k+1}(:,i)}}$ — $3n$
7: **end for**

Algorithm 21 A-CholQR — Flops

Input: P_{k+1}, the search directions to be A-orthonormalized, $W_{k+1} = AP_{k+1}$
Output: $\widetilde{P}_{k+1}$, the A-orthonomalized search directions; $\widetilde{W}_{k+1} = A\widetilde{P}_{k+1}$

1: Compute $C = W_{k+1}^t P_{k+1}$ — $(2n-1)t^2$
2: Compute the Cholesky factorization of $C = R^t R$ to obtain R — t^2
3: Solve $R^t \widetilde{P}_{k+1}^t = P_{k+1}^t$ — $t^2 n$
4: Solve $R^t \widetilde{W}_{k+1}^t = W_{k+1}^t$ — $t^2 n$
5: **for** $i = 1 : t$ **do**
6: $\quad$ Let $np = \sqrt{\widetilde{P}_{k+1}(:,i)^t \widetilde{W}_{k+1}(:,i)}$ — $2n$
7: $\quad \widetilde{P}_{k+1}(:,i) = \frac{\widetilde{P}_{k+1}(:,i)}{np}$, and $\widetilde{W}_{k+1}(:,i) = \frac{\widetilde{W}_{k+1}(:,i)}{np}$ — $2n$
8: **end for**

Then each processor i computes $W_{k+1}(\delta_i,:)^t P_{k+1}(\delta_i,:)$ and receives the $t \times t$ matrix C via an "all reduce" or equivalently $log(t)$ messages and $t^2 log(t)$ words. Finally, each processor i can compute the Cholesky factorization of the matrix C to obtain R which is needed to solve $R^t \widetilde{P}_{k+1}(\delta_i,:)^t = P_{k+1}(\delta_i,:)^t$ and $R^t \widetilde{W}_{k+1}(\delta_i,:)^t = W_{k+1}(\delta_i,:)^t$. Thus, it is possible to parallelize the A-CholQR A-orthonormalization, Algorithm 21, by sending $log(t)$ messages with $t^2 log(t)$ words and performing $t^2 + \frac{(4n-1)t^2}{t} \approx 4nt$ flops in parallel. Hence, by ignoring lower order terms we obtain

$$Time_{A-CholQR} \approx \gamma_c 4nt + \alpha_c log(t) + \beta_c t^2 log(t)$$

In the case of $\widehat{A}$-orthonormalization of P_{k+1}, where $\widehat{A} = L^{-1}AL^{-t}$ and $L^{-t} = (L^t)^{-1}$, the $\widehat{A}$-CholQR algorithm is defined in Algorithm 22. Note that we have to solve 2 systems with multiple right hand sides. If L is a lower triangular matrix, then we perform a backward and forward substitution.

Recently, Lowery and Langou presented a new version of A-CholQR in [59], which they call

Algorithm 22 $\widehat{A}$-CholQR

Input: A, the $n \times n$ symmetric positive definite matrix; L, $n \times n$ preconditioner
Input: P_{k+1}, the search directions to be A-orthonormalized; W_{k+1}, $= L^{-1}AL^{-t}P_{k+1}$
Output: $\widetilde{P}_{k+1}$, the A-orthonomalized search directions; $\widetilde{W}_{k+1} = L^{-1}AL^{-t}\widetilde{P}_{k+1}$

1: Compute $C = W_{k+1}^t P_{k+1}$
2: Compute the Cholesky QR factorization of C to obtain R
3: Solve $R^t \widetilde{P}_{k+1}^t = P_{k+1}^t$
4: Compute $\widetilde{W}_{k+1} = L^{-1}AL^{-t}\widetilde{P}_{k+1}$
5: **for** $i = 1 : t$ **do**
6: $\widetilde{P}_{k+1}(:,i) = \frac{\widetilde{P}_{k+1}(:,i)}{||\widetilde{P}_{k+1}(:,i)||_A} = \frac{\widetilde{P}_{k+1}(:,i)}{\sqrt{\widetilde{P}_{k+1}(:,i)^t \widetilde{W}_{k+1}(:,i)}}$ and $\widetilde{W}_{k+1}(:,i) = \frac{\widetilde{W}_{k+1}(:,i)}{\sqrt{\widetilde{P}_{k+1}(:,i)^t \widetilde{W}_{k+1}(:,i)}}$
7: **end for**

Pre-CholQR (Algorithm 23). It consists in performing a Euclidean QR factorization with L2 before calling the A-CholQR A-orthonormalization, Algorithm 20. The QR factorization of P_{k+1} can be done using the TSQR [21], which requires sending $log(t)$ messages, each of size $\frac{t^2}{2}$ words and computing $2nt + \frac{2}{3}t^3 log(t)$. Then, parallelizing Algorithm 20 requires performing two "all reduce" or $2log(t)$ messages with $(nt + t^2)log(t)$ words. In total, parallelizing Algorithm 23 requires sending $3log(t)$ messages with $(nt + 1.5t^2)log(t)$ words. Hence, by ignoring lower order terms we obtain

$$Time_{PreCholQR} \approx \gamma_c(6nt + +\frac{2}{3}t^3 log(t)) + \alpha_c 2log(t) + \beta_c(nt + \frac{3}{2}t^2 log(t))$$

Algorithm 23 Pre-CholQR

Input: A, the $n \times n$ symmetric positive definite matrix
Input: P_{k+1}, the search directions to be A-orthonormalized
Output: $\widetilde{P}_{k+1}$, the A-orthonomalized search directions

1: Compute the QR factorization of $P_{k+1} = P'_{k+1}R$
2: Call Algorithm 20 with A and P'_{k+1} as input and with $\widetilde{P}_{k+1}$ as output

2.5 Matrix powers kernel

The matrix powers kernel takes as input an $n \times n$ sparse matrix A, a dense vector $x^{(0)}$ of length n and a scalar s and computes s powers vectors $x_1 = Ax_0, x_2 = A^2x_0, ..., x_s = A^s x_0$ where $x_i = Ax_{i-1}$ for $0 < i \leqslant s$. This kernel has the advantage of performing s matrix vector computations

per fetching the matrix A once. Whereas in the classical GMRES, the matrix A is fetched at each iteration to perform one matrix-vector multiplication only. By definition, this kernel avoids communication by avoiding the fetching of A $(s-1)$ extra times. Demmel and coworkers have proposed sequential, parallel, and hybrid implementation of the matrix powers kernel that further avoid communication [22, 60, 48] by ghosting and computing redundantly on each processor the data required for computing its part of the vectors with no communication. These implementations work optimally for sparse matrices A that have a partitionable graph such that the non-overlapped partitions have a small surface-to-volume ratio, where "surface" indicates the boundary between two partitions. In other words, the ratio of the boundary vertices over the total vertices in a partition should be small, where the boundary vertices of some partition are those vertices share at least an edge with some vertex from another partition. This is true for many types of sparse matrices, including discretizations of partial differential equations.

Demmel and coworkers have presented 2 implementations PA1 and PA2 for the matrix powers kernel on distributed-memory architecture [22]. In their paper [60], they presented an implementation for shared-memory multicores architecture which is based on a simplified version of PA1. Two sequential implementations are also presented, the implicit and the explicit algorithm. The main purpose of both algorithms is to minimize the data movement between levels of memory hierarchy. As for the hybrid implementation, that nests a parallel and a sequential matrix powers kernel algorithm, it minimizes data dependency between the processors and the data movement between levels of memory hierarchy of each processor.

In this section, we present a generic algorithm for distributed-memory or shared-memory parallel implementation of the matrix powers kernel for computing the s monomial basis vectors of the Krylov subspace. First, the data and the work is split equally between the p processors where the indices set $\delta = 1 : n$ is partitioned using some graph or hypergraph partitioner into p subsets δ_i for $i = 1, 2, .., p$, where $\delta = \cup_{i=1}^{p} \delta_i$. Each processor i is assigned the δ_i part of x_0 and has to compute the same part of $x_1(\delta_i) = A(\delta_i, :)x_0$, $x_2(\delta_i) = A(\delta_i, :)x_1$, till $x^s(\delta_i) = A(\delta_i, :)x_{s-1}$ without any communication with other processors. To do so, each processor i determines (Algorithm 24) and fetches all the data needed from the neighboring processors, to compute its part δ_i of the s vectors (Algorithm 25).

Algorithm 24 s-step Dependencies for the Matrix Powers Kernel

Input: $G(A)$; s, number of steps; δ_i, subset of unknowns assigned to processor i
Output: Sets $\delta_{i,j}$ for all $j = 1$ till s
1: Let $\delta_{i,0} = \delta_i$
2: **for** $j = 1 : s$
3: Find $\delta_{i,j} = Adj(G(A), \delta i, j-1)$
4: **end for**

Due to the fact that A is a sparse matrix, $x_s(\delta_i) = A(\delta_i, \delta_{i,1})x_{s-1}(\delta_{i,1})$, where $\delta_{i,1} = Adj(A, \delta_i)$ is the adjacent of δ_i in graph of A. To compute $x_{s-1}(\delta_{i,1}) = A(\delta_{i,1}, \delta_{i,2})x_{s-2}(\delta_{i,2})$ where $\delta_{i,2} =$

Algorithm 25 Matrix Powers Kernel

Input: $A(\delta_i, :)$, $x_0(\delta_i)$, s: number of steps, δ_i: subset of unknowns assigned to processor i
Output: $y_i(\alpha_0)$, where $1 \leqslant i \leqslant s$

```
for each processor i = 1 : p do
    Processor i calls the algorithm 24
    Processor i fetches the missing parts of A(δ_{i,s−1}, δ_{i,s}) and x_0(δ_{i,s})
    for j = 1 : s do
        Processor i computes x_j(δ_{i,s−j}) = A(δ_{i,s−j}, δ_{i,s−j+1}) x_{j−1}(δ_{i,s−j+1})
        Save x_j(δ_i), which is the part that processor i has to compute
    end for
end for
```

$Adj(A, \delta_{i,1}) = R(A, \delta_i, 2)$, $x_{s-2}(\delta_{i,2})$ must be computed. And similarly, to compute $x_{s-2}(\delta_{i,2}) = A(\delta_{i,2}, \delta_{i,3})x_{s-3}(\delta_{i,3})$ where $\delta_{i,3} = Adj(A, \delta_{i,2}) = R(A, \delta_i, 3)$, $x_{s-3}(\delta_{i,3})$ must be computed.

In general, $x_{s-j}(\delta_{i,j}) = A(\delta_{i,j}, \delta_{i,j+1})x_{s-j-1}(\delta_{i,j+1})$ where $\delta_{i,j+1} = Adj(A, \delta_{i,j}) = R(A, \delta_i, j+1)$. Thus, from the beginning, processor i fetches the missing data of $x_0(\delta_{i,s})$ and $A(\delta_{i,s-1}, \delta_{i,s})$ from its neighboring processors and store it redundantly, where $\delta_{i,s} = R(A, \delta_i, s)$. Finally, each processor i computes the set $\delta_{i,s-j} = R(G(A), \delta_i, s-j)$ of the vectors x_j for $j = 1, 2, .., s$ without any communication with the other processors as shown in Algorithm 25. Note that Algorithm 25 can be used for a sequential implementation where the number of partitions p is chosen so that the blocks would fit into cache.

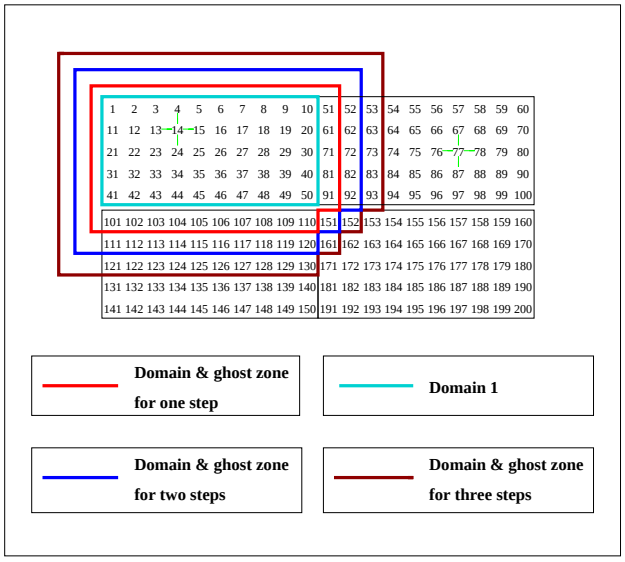


Figure 2.3: Needed data to compute three multiplications $y_i = Ay_{i-1}$ on Domain 1 using Matrix Powers kernel

Figure 2.3 shows the needed data for each step on Domain 1 with $s = 3$ where the graph of a 2D 5 point stencil matrix with n=200 is partitioned into 4 subdomains. Since $\delta_1 \subseteq \delta_{1,s} = R(G(A), \delta_1, s)$, it is obvious that the more steps are performed, the more redundant data is ghosted and flops are computed. In addition, partitioning plays an important role in reducing the size of the ghost data and balancing the load among processors. In [13] and [11], hypergraph partitioning models where introduced to reduce the volume of communication in matrix vector multiplication and matrix powers kernel.

2.6 Test matrices

In this section we describe the test matrices (Table 2.1) used in Chapters 4 and 5.

The first matrix POISSON2Dis a block tridiagonal matrix obtined from Poisson's equation (sparse) using matlab's gallery('poisson',100) function. As for the matrices, UTM3060, BCSSTK18 and WATT2, their full description can be found in [20]. The matrices refered to as NH2D, NH2D1, NH2D2, SKY2D, SKY2D1 SKY3D, SKY3D1, and ANI3D, arise from boundary value problem of the convection diffusion equations

$$\begin{aligned} \eta(x)u + div(\mathbf{a}(x)u) - div(\kappa(x)\nabla u) &= f \; in \; \Omega \\ u &= 0 \; on \; \partial\Omega_D \\ \frac{\partial u}{\partial n} &= 0 \; on \; \partial\Omega_N \end{aligned}$$

where $\Omega = [0,1]^n$, ($n = 2$, or 3) and $\partial\Omega_N = \partial\Omega \backslash \partial\Omega_D$. The function η, the vector field $\mathbf{a}$, and the tensor κ are the given coefficients of the partial differential operator. In the 2D case, we have $\partial\Omega_D = [0,1] \times \{0,1\}$, and in the 3D case, we have $\partial\Omega_D = [0,1] \times \{0,1\} \times [0,1]$. We focus on the following cases:

- NH2D, NH2D1, NH2D2: A non-homogeneous problem with large jumps in the coefficients.The coefficient η and $\mathbf{a}$ are both zero. The tensor κ is isotropic and discontinuous. It jumps from the constant value 10^3 in the ring $\frac{1}{2\sqrt{2}} \leqslant |x - c| \leqslant \frac{1}{2}$, $c = (\frac{1}{2}, \frac{1}{2})^T$, to 1 outside.

- SKY2D, SKY2D1, SKY3D, and SKY3D1 Skyscraper problems : The tensor κ is isotropic and discontinuous. The domain contains many zones of high permeability which are isolated from each other

$$\kappa(x) = \begin{cases} 10^3 * ([10 * x_2] + 1) & if \;\; [10x_i] \;\; is \;\; odd, \;\; i = 1,2 \\ 1, & otherwise. \end{cases}$$

  where we note $[x]$ as the integer value of x.

- ANI3D Anisotropic layers : the domain is made of 10 anisotropic layers with jumps of up to four orders of magnitude and an anisotropy ratio of up to 10^3 in each layer. The domain is

divided into 10 layers parallel to $z = 0$, of size 0.1, in which the coefficients are constant. We have $\kappa_y = 10\kappa_x$ and $\kappa_z = 1000\kappa_x$. The velocity field is zero.

BO1 and BO2 matrices are from a simulation of a black oil reservoir model, based on a compositional triphase Darcy flow simulator (oil, water and gas) [1], The permeability is heterogeneous, with jumps on the order of 2^8.

POISSON2D, NH2D and SKY2D1 are discretized on a 100×100 2D cartesian grid. NH2D1 and SKY2D are discretized on a 200×200 2D cartesian grid and NH2D2on 400×400 2D cartesian grid. SKY3D1 and ANI3D are discretized on a $20 \times 20 \times 20$ 3D grid and SKY3D on a $40 \times 40 \times 40$ 3D grid. BO1 and BO2 are discretized on a $15 \times 15 \times 8$ grid, and a $30 \times 30 \times 16$ grid.

The matrices CD20P1, CD50P1, CD100P1, CD200P1, CD20P2, CD50P2, and CD100P2 arise from the boundary value problem of the convection-diffusion equations $-\Delta u - 2P\frac{\partial u}{\partial x} + 2P\frac{\partial u}{\partial y} = g$ on $\Omega = (0,1) \times (0,1)$ used in [6, 25] for testing preconditioners, where $P > 0$ and the right-hand side g and the boundary conditions are determined by the solution $u(x,y) = \frac{e^{2P(1-x)}-1}{e^{2P}-1} + \frac{e^{2Py}-1}{e^{2P}-1}$. The matrices were generated by Pierre-Henri Tournier using FreeFem++ [46] with Finite Element P1 and P2 schemes with an adaptive mesh for $P = 20, 50, 100, 400, 500$.

As for the ELASTICITY3D matrix, it arises from the linear elasticity problem with Dirichlet and Neumann boundary conditions, defined as follows:

$$\begin{aligned} div(\sigma(u)) + f &= 0 && \text{on } \Omega, \\ u &= u_D && \text{on } \partial\Omega_D, \\ \sigma(u).n &= g && \text{on } \partial\Omega_N, \end{aligned}$$

where Ω is a $3D$ $30 \times 10 \times 10$ parallelepiped, Ω_D is the Dirichlet boundary, Ω_N is the Neumann boundary, u is the unknown displacement field, f is some body force, $\sigma(u)$ is the Cauchy stress tensor given by Hooke's law. The ELASTICITY3D matrix was discretized with P1 finite elements and a triangular mesh using FreeFem++ [46]. For a detailed description of the problem refer to [42].

[1] These matrices were provided to us by R. Masson, at that time at IFP Energies Nouvelles

Table 2.1: The test matrices

Matrix	Size	Nonzeros	Symetric	2D/3D	Problem
POISSON2D	10000	49600	Yes	2D	Poisson equations
NH2D	10000	49600	Yes	2D	Boundary value
NH2D1	40000	199200	Yes	2D	Boundary value
NH2D2	160000	798400	Yes	2D	Boundary value
SKY2D1	10000	49600	Yes	2D	Boundary value
SKY2D	40000	199200	Yes	2D	Skyscraper
SKY3D1	8000	53600	Yes	3D	Skyscraper
SKY3D	64000	438400	Yes	3D	Skyscraper
ANI3D	8000	53600	Yes	3D	Anisotropic Layers
BO1	1800	11670	Yes	3D	Black oil reservoir
BO2	14400	97080	Yes	3D	Black oil reservoir
UTM3060	3060	42211	No	3D	Electromagnetics
BCSSTK18	11948	149090	Yes	3D	Structural (Stiffness Matrix)
WATT2	1856	11550	No	3D	Computational fluid dynamics
CD20P1	3190	21908	No	2D	Convection diffusion P1 FE
CD50P1	3413	23439	No	2D	Convection diffusion P1 FE
CD100P1	3909	26885	No	2D	Convection diffusion P1 FE
CD200P1	5262	36224	No	2D	Convection diffusion P1 FE
CD20P2	12423	141279	No	2D	Convection diffusion P2 FE
CD50P2	13413	152589	No	2D	Convection diffusion P2 FE
CD100P2	14612	166280	No	2D	Convection diffusion P2 FE
ELASTICITY3D	11253	373647	Yes	3D	Linear Elasticity P1 FE

Chapter 3

Krylov Subspace Methods

The Krylov Subspace methods are named after the Russian applied mathematician and naval engineer Alexei Krylov. In this chapter we discuss several variants of Krylov subspace methods. In the first section, we introduce Classical Krylov subspace methods, specifically CG [47] and GMRES [66]. In the second section we briefly introduce variants of Krylov methods that are better parallelizable and require less communication like block methods, s-step methods and communication avoiding methods. Finally, we discuss preconditioners which are crucial for the fast convergence of the Krylov methods.

3.1 Classical Krylov subspace methods

In this section we define the Krylov subspaces and list its properties. Then we will define the classical Krylov Subspace methods and the Krylov projection methods like CG [47] and GMRES [66].

3.1.1 The Krylov subspaces

In linear algebra, a Krylov subspace of order-i $\mathcal{K}_i$ is generated by an $n \times n$ matrix A and an $n \times 1$ vector y where $\mathcal{K}_i(A, y) = span\{y, Ay, A^2y, ..., A^{i-1}y\}$. Thus, the Krylov Subspace is the linear subspace spanned by the images of y under the first i powers of A and it verifies the following properties:

- $\mathcal{K}_1(A, y) \subseteq \mathcal{K}_2(A, y) \subseteq \mathcal{K}_3(A, y) \subseteq \mathcal{K}_4(A, y) \subseteq ... \subseteq \mathcal{K}_n(A, y) \subseteq ...$
- $A\mathcal{K}_k(A, y) \subseteq \mathcal{K}_{k+1}(A, y)$

The proof of the first property is trivial and it is based on the definition of Krylov subspaces. As for the second poof, we let

$$\begin{aligned} x &= d_0 y + d_1 A y + d_2 A^2 y + ... + d_{k-1} A^{k-1} y \\ \implies Ax &= d_0 A y + d_1 A^2 y + d_2 A^3 y + ... + d_{k-1} A^k y \in \mathcal{K}_{k+1}(A, y) \end{aligned}$$

3.1.2 The Krylov subspace methods

The Krylov Subspace methods are polynomial iterative methods that aim to solve linear equations of the form $Ax = b$ by finding a sequence of vectors

$$x_1, x_2, x_3, x_4, ..., x_k$$

that minimizes some measure of error over the corresponding spaces

$$x_0 + \mathcal{K}_i(A, r_0), \quad \text{for } i = 1, ..., k$$

where x_0 is the initial iterate or guess, $r_0 = b - Ax_0$ is the initial residual and $\mathcal{K}_i(A, r_0)$ is the Krylov subspace of order i generated by A and r_0. Conjugate gradient (CG) [47], generalized minimum residual (GMRES) [66], bi-conjugate gradient (Bi-CG) [56, 30], and bi-conjugate gradient stabilized (Bi-CGstab) [75] are Krylov Subspace methods.

3.1.3 Krylov projection methods

The Krylov projection methods find a sequence of approximate solutions $x_k \in x_0 + \mathcal{K}_k$ $(k = 1, 2, ..)$ of the system $Ax = b$ by imposing the Petrov-Galerkin constraint on the k^{th} residual $r_k = b - Ax_k$

$$r_k \perp \mathcal{L}_k$$

where $\mathcal{L}_k \subseteq \mathbb{R}^n$ (or $\subseteq \mathbb{C}^n$) is a well-defined subspace of dimension k. The subspace $\mathcal{L}_k$ can be the same as the Krylov subspace $\mathcal{K}_k$ or different. The different choices of $\mathcal{L}_k$ give rise to different methods [65]. Thus, the different Krylov Projection methods are defined by the subspace $\mathcal{L}_k$ and the following 2 conditions:

1. Subspace condition: $x_k \in x_0 + \mathcal{K}_k$
2. Petrov-Galerkin condition: $r_k \perp \mathcal{L}_k \iff (r_k)^t y = 0 \ \ \forall \ y \in \mathcal{L}_k$

Conjugate Gradient and GMRES are Krylov projection methods where $\mathcal{L}_k = \mathcal{K}_k$, and $\mathcal{L}_k = A\mathcal{K}_k$ respectively.

3.1.4 Conjugate gradient

The conjugate gradient, which was introduced by Hestenes and Stiefel in 1952 [47], is an iterative Krylov Projection method for symmetric (Hermitian) positive definite matrices of the form

$$\begin{cases} Ax &= b \\ A &= A^t \\ x^tAx &> 0, \forall x \neq 0 \end{cases} \tag{3.1}$$

Given an initial guess or iterate x_0, at the k^{th} iteration CG finds the new approximate solution $x_k = x_{k-1} + \alpha_k p_k$ that minimizes $\phi(x) = \frac{1}{2}(x)^tAx - b^tx$ over the corresponding space $x_0 + \mathcal{K}_k(A, r_0)$, where $k > 0$, $p_k \in \mathcal{K}_k(A, r_0)$ is the k^{th} search direction and α_k is the step along the search direction.

The minimum of $\phi(x)$ is given by $\bigtriangledown\phi(x) = 0$ which is equivalent to $\bigtriangledown\phi(x) = Ax - b = 0$. Thus, by minimizing $\phi(x)$ we are solving the system (3.1). As the name of the method indicates, the gradients $\bigtriangledown\phi(x_i)$ for all i should be conjugate. And since CG is a projection Krylov method, the residual $r_k = b - Ax_k$ should respect the Petrov-Galerkin condition

$$r_k \perp \mathcal{K}_k.$$

Thus, $(r_k)^t y = 0 \ \ \forall\ y \in \mathcal{K}_k$. Hence, the residuals form an orthogonal set, $(r_k)^t r_i = 0, \ \ \forall\ i < k$. Moreover, the Petrov-Galerkin condition $r_k \perp \mathcal{K}_k(A, r_0)$ is equivalent to the conjugacy of the gradients $\bigtriangledown\phi(x_k)^t \bigtriangledown \phi(x_i) = 0\ \forall i \neq k$. Once x_k has been chosen, either x_k is the required approximate solution of $Ax = b$ or a new search direction $p_{k+1} \neq 0$ and a new approximation $x_{k+1} = x_k + \alpha_{k+1}p_{k+1}$ are computed. This procedure is repeated until convergence or until the maximum number of allowed iterations has been reached without convergence. The convergence criterion is set as

$$||r_k||_2 \leqslant \epsilon||b||_2, \ \ for\ some\ \epsilon \in \mathbb{R}$$

where $r_k = b - Ax_k \in \mathcal{K}_{k+1}(A, r_0)$ is the k^{th} residual.

Theorem 3.1.1. *The Petrov-Galerkin condition $(r_k)^t y = 0 \ \ \forall\ y \in \mathcal{K}_k$ implies the A-orthogonality of the search directions $p_i^tAp_j = 0 \ \ \forall i \neq j$.*

Proof. By definition, $p_i \in \mathcal{K}_i$ and $\mathcal{K}_i \subset \mathcal{K}_{i+1}$. Thus $p_i \in \mathcal{K}_{i+c}$ for $c \geqslant 0$. By the Petrov-Galerkin condition $r_{k-1}^t p_i = 0$ for $i \leqslant k-1$ and $r_k^t p_i = 0$. Thus, $r_k^t p_i = r_{k-1}^t p_i - \alpha p_k^t Ap_i = 0$ for $i \leqslant k-1$. This implies that $p_k^t Ap_i = 0$ for $i \leqslant k-1$ since $\alpha \neq 0$. Therefore, the A-orthogonality of the search directions. □

This theorem means that the A-orthogonality of the search directions has to be ensured or else the Petrov-Galerkin condition won't be respected. On the other hand, the search direction $p_k \in \mathcal{K}_k$ is chosen according to the following recursion relation:

$$\begin{cases} p_1 &= r_0 \\ p_k &= r_{k-1} + \beta_k p_{k-1} \end{cases} \tag{3.2}$$

where p_1 is set equal to r_0 since the initial residual is equal to negative the gradient $- \bigtriangledown \phi(x_0)$ which is the steepest descent from x_0. But p_k is not set to r_{k-1}, the steepest descent from x_{k-1} for $k > 1$, since the residuals are not A-orthogonal. It can be shown that the search directions defined in (3.2) are A-orthogonal i.e. $p_k^t A p_i = 0$ for all $i \leqslant k-1$. For $i < k-1$, we have

$$p_k^t A p_i = r_{k-1}^t A p_i + \beta_k p_{k-1}^t A p_i = \beta_k p_{k-1}^t A p_i \tag{3.3}$$

since $r_{k-1}^t A p_i = 0$ by Petrov-Galerkin condition. In addition, $r_{k-1}^t p_i = r_{k-2}^t p_i - \alpha_{k-1} p_{k-1}^t A p_i = 0$ with $r_{k-2}^t p_i = 0$ since $i \leqslant k-2$. Thus, $p_{k-1}^t A p_i = 0$. Therefore, $p_k^t A p_i = 0$ for $i < k-1$.

As for $i = k-1$, $r_{k-1}^t A p_{k-1} \neq 0$ and $p_{k-1}^t A p_{k-1} \neq 0$ for $p_{k-1} \neq 0$. Thus, $\beta_k = -\frac{(r_{k-1})^t A p_{k-1}}{(p_{k-1})^t A p_{k-1}}$ is chosen so that $p_k^t A p_{k-1} = 0$

At each iteration, the step $\alpha_k = \frac{(p_k)^t r_{k-1}}{(p_k)^t A p_k} = \frac{||r_{k-1}||_2^2}{||p_k||_A^2}$ is chosen such that,

$$\phi(x_k) = min\{\phi(x_{k-1} + \alpha p_k), \forall \alpha \in \mathbb{R}\}.$$

Using the definition of α_k, $\beta_k = -\frac{(r_{k-1})^t A p_{k-1}}{(p_{k-1})^t A p_{k-1}} = \frac{||r_{k-1}||_2^2}{||r_{k-2}||_2^2}$. The CG algorithm is presented in Algorithm (26).

Algorithm 26 The CG Algorithm

Input: A, the $n \times n$ SPD matrix; b, the $n \times 1$ right-hand side
Input: ; x_0, the initial guess or iterate
Input: ϵ, the stopping tolerance; k_{max}, the maximum allowed iterations
Output: x_k, the approximate solution of the system $Ax = b$

$r_0 = b - Ax_0$, $\rho_0 = ||r_0||_2^2$, k = 1
while ($\sqrt{\rho_{k-1}} > \epsilon ||b||_2$ and $k < k_{max}$) **do**
    if (k = 1) **then** $p = r_0$
    else $\beta = \frac{\rho_{k-1}}{\rho_{k-2}}$ and $p = r + \beta p$
    end if
    $w = Ap$
    $\alpha = \frac{\rho_{k-1}}{p^t w}$
    $x = x + \alpha p$
    $r = r - \alpha w$
    $\rho_k = ||r||_2^2$
    k = k+1
end while

The CG Algorithm 26 has short recurrences, where the memory requirements and computed flops per iterations are constant. Hence, four vectors are stored along with the sparse matrix A.

After performing k_c iterations, $O(k_c(10n + 2\text{nnz}))$ flops are computed. It is shown in [65] that the speed of convergence of CG is

$$||x_* - x_k||_A \leqslant 2\left[\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right]^k ||x_* - x_0||_A$$

where x_* is the exact solution of $Ax = b$, x_k is the k^{th} approximate solution $Ax = b$, $\kappa = cond_2(A) = \frac{\lambda_{max}}{\lambda_{min}}$, λ_{max} is the maximum eigenvalue of A, and λ_{min} is the minimum eigenvalue of A.

3.1.5 Generalized minimal residual (GMRES) method

The generalized minimal residual method (GMRES), introduced by Saad and Schultz in 1986 [66], is a Krylov projection method for solving general linear systems,

$$Ax = b$$

where the residual r_k is chosen to be orthogonal to $\mathcal{L}_k = A\mathcal{K}_k$.

The GMRES method solves the system $Ax = b$ by approximating the solution at the k^{th} iteration with the vector $x_k \in x_0 + \mathcal{K}_k$ such that

$$||r_k||_2 = ||b - Ax_k||_2 = min\{||b - Ax||_2, \forall x \in x_0 + \mathcal{K}_k\}$$

The minimum of the L2 norm $||b-Ax||_2$ is zero which is equivalent to solving the system $Ax-b = 0$. It can be shown that the Petrov Galerkin condition $r_k \perp A\mathcal{K}_k$ is equivalent to minimizing $||b - Ax||_2$ for all $x_0 + \mathcal{K}_k$.

Unlike the Conjugate Gradient method, the residuals in GMRES do not form an orthonormal basis for $\mathcal{K}_k$. Thus, an orthonormal basis is built for $\mathcal{K}_k$ using the Arnoldi process, which generates basis vectors and orthonormalizes them using modified Gram Schmidt procedure. At each Arnoldi iteration a new basis vector is computed and orthonormalized against previous vectors. The Arnoldi process reduces a general, nonsymmetric $n \times n$ matrix A into an upper Hessenberg form by the similarity transform

$$A = QHQ^t \;\; or \;\; AQ = QH,$$

where H is an $i \times i$ upper Hessenberg matrix and Q is an $n \times i$ matrix with $Q^tQ = I$. Note that $i \leqslant n$ is the largest index such that $q_i \neq 0$. Since Q satisfies $Q^tQ = I$, then the columns of Q, $\{q_1, q_2, ..., q_i\}$ form an orthonormal basis for $\mathcal{K}_i$. For $1 < k < i$ we have the following relation:

$$AQ_{k-1} = Q_kH_{k-1}$$

where Q_k is an $n \times k$ orthogonal matrix and H_{k-1} is a $k \times (k-1)$ upper Hessenberg matrix. Note that at the k^{th} iteration of the Arnoldi process the k^{th} basis vector, $q_k = Aq_{k-1}$, is computed and

orthonormalized against $\{q_1, q_2, ..., q_{k-1}\}$, thus producing the last column of H_{k-1}, where $Q_k = [Q_{k-1}, q_k]$ and

$$H_{k-1} = \left(\begin{array}{ccc|c} & & & h_{1,1} \\ & H_{k-2} & & \vdots \\ & & & \vdots \\ \hline 0 & ... & 0 & h_{k,k-1} \end{array}\right)$$

After building up the Arnoldi orthonormal basis $\{q_1, q_2, ..., q_k\}$ for the subspace $\mathcal{K}_k$, the least squares problem $\min_{x \in x_0 + \mathcal{K}_k} ||b - Ax||_2$ is transformed into $\min_{y \in \mathbb{R}^k} ||\rho_0 e1_{k+1} - H_k y||_2$ where $x_k = x_0 + Q_k y$, Q_k is an $n \times k$ orthogonal matrix, H_k is a $(k+1) \times k$ upper Hessenberg matrix, $\rho_0 = ||r_0||_2$ and $e1_{k+1} = [1\ 0\ 0\ ...\ 0]^t$ is a $k+1 \times 1$ vector.

The final step is finding the optimal $y_k \in \mathbb{R}^k$ that minimizes $||\rho_0 e1_{k+1} - H_k y||_2$. This is equivalent to solving the system

$$H_k y = \rho_0 e_1. \tag{3.4}$$

By exploiting the special structure of H_k, the system (3.4) is transformed into an upper triangular system using Givens rotations. Then the obtained upper triangular system is solved using backward substitution. Note that it is not necessary to solve for y_k, and compute x_k at each iteration. This can be done once convergence is attained. Thus, the norm of the k^{th} residual $\rho_k = ||p_k - H_k y_k||_2$ has to be evaluated without finding y_k, where p_k is a $(k+1) \times 1$ vector corresponding to the application of Givens rotations on $\rho_0 e1_{k+1}$. Since H_k is a $(k+1) \times k$ upper triangular matrix, where the last row has zero entries, and $H_k(1:k, 1:k)y_k = p_k(1:k)$, we conclude that $\rho_k = ||p_k - H_k y_k||_2 = |p_k(k+1)|$.

Algorithm 27 The GMRES algorithm

Input: A, the $n \times n$ matrix; b, the $n \times 1$ right-hand side; x_0, the initial guess or iterate
Input: ϵ, the stopping tolerance; k_{max}, the maximum allowed iterations
Output: x_k, the approximate solution of the system $Ax = b$
1: Compute $r_0 = b - Ax_0$, $\rho_0 = ||r_0||_2$, $q_1 = \frac{r_0}{\rho}$, $e_1 = [1, 0]$, $p_1 = \rho_0 e_1^t$, k = 1
2: **while** ($\rho_0 \geqslant \epsilon ||b||_2$ and $k < k_{max}$) **do**
3: Generate the $(k+1)^{th}$ vector of the Arnoldi basis Q_{k+1} and the k^{th} column of the Upper Hessenberg matrix H_k
4: Let $H_{k+1} = H_k$ and apply Givens rotations on H_k and p_k
5: $\rho_k = |p_k(k+1)|$, $e_1 = [e_1, 0]$, $p_{k+1} = \rho_0 e_1^t$, $k = k + 1$
6: **end while**
7: Let $k = k - 1$, and perform backward substitution on $H_k(1:k, 1:k)y = p_k(1:k)$
8: $x_k = x_0 + Q_k y_k$

The GMRES Algorithm 27 has long recurrences, where at each iteration a new basis vector q_{k+1} is computed and orthonormalized against all the previous vectors $q_1, q_2, ..., q_k$. Thus the memory

requirements and computed flops increase with every iteration. In case of limited memory, it is possible to restart GMRES with $x_0 = x_k$. However, there is a possibility that it stagnates.

After performing k_c iterations, $k_c + 1$ vectors of length n and a $(k_c + 1) \times k_c$ upper Hessenberg matrix have to be stored along with the $n \times n$ sparse matrix A. And $O(k_c n^2 + k_c^2 n)$ flops are computed.

As for the convergence, GMRES method is known for its superlinear convergence behavior, where the rate of convergence seems to improve as the iterations proceed [76]. Assuming that $||I - A||_2 \leqslant \rho < 1$ Kelley [50] proves the following relation between the k^{th} error and the initial one,

$$||x_k - x^*||_2 \leqslant \rho^k ||x_0 - x^*||_2 \tag{3.5}$$

where x^* is the exact solution, x_0 is the initial guess, and x_k is the k^{th} approximate solutions.

3.2 Parallelizable variants of the Krylov subspace methods

The classical Krylov subspace methods, discussed in the previous section, are governed by Blas1 and Blas2 computations like dot products and matrix vector multiplication. Parallelizing dot products is not efficient due to the negligible amount of performed flops with respect to the cost of the data movement. The solution multiply a matrix by a set of vectors and solve small systems instead of matrix vector multiplications and dot products. For example, in block methods the idea is to solve a system with multiple right hand sides. Whereas in s-step methods the idea is to merge the computations of s iterations of classical Krylov methods in order to compute s matrix-vector multiplications at a time. Communication avoiding methods are based on s-step methods with algorithmic and implementation level improvements for avoiding communication communication. In sections 3.2.1, 3.2.2, and 3.2.3 we discuss block methods, s-step methods and communication avoiding methods respectively. Finally in section 3.2.4 we discuss two CG parallelizable variants, cooperative CG (coop-CG) and multiple search direction CG (MSD-CG). In coop-CG, the idea is having multiple agents or threads that cooperate to solve the system $Ax = b$, t times in parallel starting with t initial distinct guesses. In MSD-CG, a domain decomposition method, the idea is to have multiple search directions at each iteration. These search directions are defined on different subdomains.

3.2.1 Block Krylov methods

The block Krylov methods solve a system with multiple right-hand sides

$$AX = B$$

where A is an $n \times n$ matrix, X is an $n \times t$ block of vectors, B is an $n \times t$ block of vectors, and t is the number of right-hand sides. The block Krylov methods are iterative methods that approximate

the solution of $AX = B$ at the k^{th} iteration by $X_k \in X_0 + \mathcal{K}_k(A, R_0)$, where $R_0 = B - AX_0$ is the initial block residual, and $\mathcal{K}_k(A, R_0) = block - span\{R_0, AR_0, A^2R_0, ..., A^kR_0\}$ is the block Krylov subspace. Every $n \times t$ block $Z \in \mathcal{K}_{k+1}(A, R_0)$ is defined as $Z = \sum_{i=1}^{k} A^i R_0 \zeta_i$ where ζ_i is a $t \times t$ matrix.

The first block method, block CG [63], was introduced in 1980 by O'Leary. As for block GMRES which was introduced in Vital's PhD thesis [78], it is based on the block Arnoldi method (refer to [45]). Block CG is only described in section 3.2.1.1. For a brief description of block GMRES refer to [45].

3.2.1.1 Block conjugate gradient (B-CG) method

In 1980 O'Leary introduced a block CG version [63] that solves an SPD system with multiple right-hand sides

$$\left\{ \begin{array}{rcl} AX & = & B, \\ A & = & A^t, \\ x^tAx & > & 0, \forall x \neq 0 \end{array} \right. \tag{3.6}$$

where A is an $n \times n$ matrix, $X \in \mathbb{R}^{n \times t}$ is a block vector, and B is a block vector of size $n \times t$ containing the multiple right hand sides.

Starting with an initial guess $X_0 \in \mathbb{R}^{n \times t}$, initial residual $R_0 = B - AX_0$, $P_1 = R_0\gamma_1$ with γ_1 a $t \times t$ full rank freely chosen matrix, the B-CG searches for an approximate solution $X_{k+1} \in X_0 + \mathcal{K}_{k+1}(A, R_0)$ where $\mathcal{K}_{k+1}(A, R_0) = block - span\{R_0, AR_0, A^2R_0, ..., A^kR_0\}$ is the block Krylov subspace. By the Petrov-Galerkin condition we have that $R_{k+1} \perp \mathcal{K}_{k+1}(A, R_0)$. Then, $R_{k+1}^t Y = 0$ for all $Y \in \mathcal{K}_{k+1}(A, R_0)$, which implies that $R_{k+1}^t R_i = 0$ and $R_{k+1}^t AP_i = 0$ for all $i < k + 1$.

Then, for $k \geqslant 0$ the iterates are defined similarly to CG:

$$\begin{array}{rcll} X_{k+1} & = & X_k + P_{k+1}\alpha_{k+1} & \in \mathcal{K}_{k+1}(A, R_0) \\ R_{k+1} & = & R_k - AP_{k+1}\alpha_{k+1} & \in \mathcal{K}_{k+2}(A, R_0) \\ P_{k+2} & = & (R_{k+1} + P_{k+1}\beta_{k+2})\gamma_{k+2} & \in \mathcal{K}_{k+2}(A, R_0) \end{array}$$

where

$$\begin{array}{rcl} \alpha_{k+1} & = & (P_{k+1}^t AP_{k+1})^{-1}\gamma_k^t(R_k^t R_k) \\ \beta_{k+2} & = & \gamma_{k+1}^{-1}(R_k^t R_k)^{-1}(R_{k+1}R_{k+1}) \end{array}$$

Note that α_{k+1} is chosen such that $\phi(X_{k+1}) = min\{\phi(X_k + P_{k+1}\alpha), \text{for all} \alpha \in \mathbb{R}^{t,t}\}$. As for β_{k+1}, it is chosen to ensure the A-orthogonality of the P_{k+1} and P_k ($(P_{k+1})^t AP_k = 0$). Whereas γ_k is a $t \times t$ full rank matrix that can be chosen freely to decrease roundoff errors in the implementation.

Moreover, the search direction $P_{k+1} \in \mathcal{K}_{k+1}(A, R_0)$ of the block conjugate gradient method is A-conjugate, $(P_{k+1})^t AY = 0$, for all $Y \in \mathcal{K}_k(A, R_0)$. This leads to the A-orthogonality of the search direction $\Longrightarrow (P_{k+1})^t AP_i = 0$, for all $i < k + 1$. The Block-CG algorithm is presented in Algorithm 28.

Algorithm 28 The Block CG Algorithm

Input: A, the $n \times n$ symmetric positive definite matrix
Input: B, the $n \times t$ block of t right-hand sides; X_0, the block of t initial guesses or iterates
Input: ϵ, the stopping tolerance; k_{max}, the maximum allowed iterations
Output: X_k, the block of t approximate solutions of the multiple right-hand side system $AX = B$

$R_0 = B - AX_0$, k = 1
while (Not converged and $k < k_{max}$) **do**
  if (k = 1) **then** Let $P = R_0$
  else Let $P = (R + P\beta)$
    Orthogonalize the vectors of P against each others and define γ
  end if
  $\alpha = (P^t AP)^{-1}\gamma^t(R^t R)$
  $X = X + P\alpha$
  $R_{k-1} = R$
  $R = R - AP\alpha$
  $\beta = \gamma^t(R_{k-1}^t R_{k-1})^{-1}(R^t R)$
  k = k+1
end while

3.2.2 The s-step Krylov methods

The s-step Krylov methods are parallelizable version of classical Krylov methods where s iterations of classical Krylov methods are merged and computed simultaneously. The first introduced s-step method was Van Rosendale's s-step CG [77]. However, Chronopoulos and Gear where the first to call their method "s-step" CG [19]. On the other hand, in 1985 Walker introduced s-step GMRES [79] for numberical stability purposes. Since then many improved s-step CG and GMRES versions were introduced. For a brief overview, refer to [48] on page 34. In this section, Chronopoulos and Gear's s-step CG (section 3.2.2.1) and Walker's s-step GMRES (section 3.2.2.2) are briefly described.

3.2.2.1 The s-step conjugate gradient

Chronopoulos and Gear's s-step CG [19] starts by defining the first s search directions as the basis vectors, $p_{1,j} = A^{j-1}r_0$ where $1 \leqslant j \leqslant s$. Let $P_1 = [p_{1,1}, p_{1,2}, ..., p_{1,s}]$, then at the k^{th}

iteration $x_k = x_{k-1} + P_k\alpha_k$ where α_k is the $s \times 1$ step lengths vector. The α_k is chosen so that $\phi(x_k) = min\{\phi(x| \quad for\ all\ x\ \in \mathcal{K}_{sk})\} = min\{x_{k-1} + P_k\alpha | for\ all\ \alpha \in \mathbb{R}^s\}$ where $\phi(x) = \frac{1}{2}x^tAx - b^tx$.

Let $F(\alpha) = \phi(X_{k-1} + P_k\alpha)$. Then,

$$\begin{aligned} F(\alpha) &= \frac{1}{2}(x_{k-1} + P_k\alpha)^tA(x_{k-1} + P_k\alpha) - (x_{k-1} + P_k\alpha)^tb \\ &= \phi(x_{k-1}) + \frac{1}{2}[(x_{k-1})^tAP_k\alpha + \alpha^t(P_k)^tAx_{k-1} + \alpha^t(P_k)^tAP_k\alpha] - \alpha^t(P_k)^tb \\ &= \phi(x_{k-1}) + \frac{1}{2}[(x_{k-1})^tAP_k\alpha - \alpha^t(P_k)^tAx_{k-1}] + \frac{1}{2}\alpha^t(P_k)^tAP_k\alpha - \alpha^t(P_k)^tr_{k-1} \\ &= \phi(x_{k-1}) + \frac{1}{2}\alpha^t(P_k)^tAP_k\alpha - \alpha^t(P_k)^tr_{k-1}, \quad since\ \ A\ is\ spd. \end{aligned}$$

The minimum of $F(\alpha)$ is given by $F'(\alpha) = (P_k^tAP_k)\alpha - P_k^tr_{k-1} = 0$. Thus, $\alpha_k = (P_k^tAP_k)^{-1}(P_k^tr_{k-1})$, and $r_k = b - Ax_k = r_{k-1} - AP_k\alpha_k$.

As for the new s search directions, $P_{k+1} = B_k + P_k\beta_k$ where $B_k = [r_k, Ar_k, A^2r_k, ..., A^{s-1}r_k]$, and β_k is an $s \times s$ matrix chosen to ensure the A-orthogonality of P_{k+1} against P_k, $P_k^tAP_{k+1} = 0$. Thus $\beta_k = (P_k^tAP_k)^{-1}(P_k^tAB_k)$. The s-step CG algorithm is summarized in Algorithm 29.

Algorithm 29 The s-step CG Algorithm

Input: A, the $n \times n$ symmetric positive definite matrix ; s, the number of steps per iteration
Input: b, the $n \times 1$ right-hand side; x_0, the $n \times 1$ initial guess or iterate
Input: ϵ, the stopping tolerance; k_{max}, the maximum allowed iterations
Output: x_k, the approximate solution of the system $Ax = b$

$r = b - Ax_0$,$P = [r, Ar, A^2r, .., A^{s-1}r]$, $\rho = r^tr$, k = 1
while ($\sqrt{\rho} > \epsilon||b||_2$ and $k < k_{max}$) **do**
  $\alpha = (P^tAP)^{-1}(P^tr)$
  $x = x + P\alpha$
  $r = r - AP\alpha$
  $B = [r, Ar, A^2r, .., A^{s-1}r]$
  $\beta = (P^tAP)^{-1}(P^tAB)$
  $P = (B + P\beta)$
  $\rho = r^tr$
  k = k+1
end while

3.2.2.2 The s-step GMRES

s-step GMRES ([79, 26] and references therein) is based on replacing the Arnoldi iteration by the Arnoldi(s) process where s basis vectors $Ar_0, A^2r_0, ..., A^sr_0$ are computed and then $V_{s+1} =$

$[r_0, Ar_0, A^2r_0, ..., A^sr_0]$ is orthonormalized using MGS, CholQR, Householder QR, or any other QR factorization. In Walker's s-step GMRES Houselder QR is used for its numerical stability where $V_{s+1} = Q_{s+1}R_{s+1}$. After orthonormalizing the s basis vectors, the upper Hessenberg matrix is reconstructed. Assuming that the orthonormal vectors obtained form Arnoldi(s) with Householder QR are the same as those obtained from Arnoldi process with MGS , i.e. the diagonal entries of R are real positive numbers, then the upper Hessenberg is reconstructed as follows.

By definition of the construction of s basis vectors, $AV_s = V_{s+1}E_s$ where $E_s = [e_2, e_3, ..e_{s+1}]$ is an $(s+1)\times s$ matrix, and e_i is the i^{th} canonical vector with one at the i^{th} entry and zero elsewhere. By Householder QR we have that $V_s = Q_sR_s$ and $V_{s+1} = Q_{s+1}R_{s+1}$ where $Q_s = Q_{s+1}(:, 1 : s)$ and $R_s = R_{s+1}(1 : s, 1 : s)$. Then

$$\begin{aligned} AV_s &= V_{s+1}E_s \\ AQ_sR_s &= Q_{s+1}R_{s+1}E_s \\ AQ_s &= Q_{s+1}R_{s+1}E_sR_s^{-1} \\ AQ_s &= Q_{s+1}H_s, \end{aligned}$$

where $H_s = R_{s+1}E_sR_s^{-1}$ is an upper Hessenberg matrix. Solving the least square $||\rho e_1 - H_sy||$ is equivalent to solving the system $H_sy = \rho e_1$ by reducing it into an upper triangular system using Givens rotations. Then, y is obtained by backward substitution of the upper triangular system. If $x_s == x_0 + Q_sy$ is not the desired solution, then s-step GMRES Algorithm 30 is restarted.

Algorithm 30 s-step GMRES

Input: A, the $n \times n$ matrix; b, the $n \times 1$ right-hand side; x_0, the initial guess or iterate
Input: ϵ, the stopping tolerance; s, the maximum allowed iterations before restart
Input: k_{max}, number of restarts
Output: x_s, the approximate solution of the system $Ax = b$

Compute $r_0 = b - Ax_0$, $\rho_0 = ||r_0||_2$, $v_1 = \frac{r_0}{\rho}$, $k = 1$
Let $E_{s+1} = I_s$, $e_1 = E_{s+1}(:, 1)$, $p_1 = \rho_0 e_1$, $E_s = E_{s+1}(:, 2 : s + 1)$
Perform Arnoldi(s) process
    Compute $v_2 = Av_1$, $v_3 = Av_2$,.., $v_{s+1} = Av_s$ and let $V_{s+1} = [v_1, v_2, .., v_{s+1}]$
    Factorize $V_{s+1} = Q_{s+1}R_{s+1}$ using Householder QR and let $R_s = R_{s+1}(1 : s, 1 : s)$
    Reconstruct the upper Hessenberg matrix $H_s = R_{s+1}E_sR_s^{-1}$
Solve Least Square $||p_1 - H_sy||$
    Apply Givens rotations on H_s and p_1
    Solve the upper triangular system $H_sy = p_1$ by backward substitution to obtain y_s
    $\rho_s = ||p_1 - H_sy_s||$, $k = k + 1$
$x_s = x_0 + Q_sy_s$
if ($\rho_s \geqslant \epsilon||b||_2$ and $k < k_{max}$) **then**
    Let $x_0 = x_s$ and restart by calling s-step GMRES with $k_{max} = k_{max} - 1$
else x_s is the approximate solution
end if

Although the memory requirement in s-step GMRES are fixed for a given s and are much less than that of GMRES, however the method has to restart after computing s basis vectors, where $s << n$.

3.2.3 Communication avoiding methods

The communication avoiding methods are based on s-step methods with the goal of further reducing communication. There are several CA methods like CA-GMRES [60] , CA-CG [48], CA-BiCG, CA-CGS, and CA-BiCGStab [12]. In this section we will briefly discuss CA-GMRES which is based on s-step GMRES with several improvements like differentiating between the restart length and the s-step basis length .

3.2.3.1 Communication avoiding GMRES (CA-GMRES)

CA-GMRES [60, 48] is based on s-step GMRES (section 3.2.2.2) where the Arnoldi(s) process is replaced by Arnoldi(s,t). The Arnoldi(s) process restarts after computing s basis vectors in one iteration, where the restart length is equal to the s-step basis length. Whereas, the Arnoldi(s,t) restarts after t iterations and after computing st basis vectors, where the choice of t is independent from s.

At the i^{th} iteration of the Arnoldi(s,t) process where $1 \leqslant i \leqslant t$, s basis vectors are computed using the matrix powers kernel [22] without any communication. Then they are orthonormalized against the previous $s(i-1)$ vectors using BCGS (Algorithm 2) and finally the s basis vectors are orthonormalized using TSQR (section 2.4.2.2). A total of $3log(p)$ messages are sent at the i^{th} iteration of the Arnoldi(s,t) process , where p is the number of processors. Whereas in classical GMRES, $\frac{s^2+s}{2}log(p)$ messages are sent after performing s iterations of the Arnoldi process. Thus by replacing the MGS in Arnoldi process by the BCGS+TSQR in the Arnoldi(s,t), the communication is reduced.

Then similarly to s-step GMRES, the upper Hessenberg is reconstructed (refer to [48]) and the least square problem $y_{si} = min||\rho_0 e_1 - H_{si}y||_2$ is solved where e_1 is an $si+1$ vector and H_{si} is an $(si+1) \times si$ upper Hessenberg matrix.

At the end of the t iterations of the Arnoldi(s,t) process, $st+1$ computed orthonormal basis vectors, $\widetilde{Q}_{st+1}$, satisfy the following relation, $A\widetilde{Q}_{st} = \widetilde{Q}_{st+1}\widetilde{H}_{st}$, where $\widetilde{Q}_{st} = \widetilde{Q}_{st+1}(:, 1:st)$ is an $n \times st$ column orthogonal matrix, and $\widetilde{H}_{st}$ is an $st+1 \times st$ upper Hessenberg matrix. The $st+1$ basis vectors, $\widetilde{Q}_{st+1}$, obtained from Arnoldi(s,t) process may differ by a unitary scaling θ_{s+1} from the $st+1$ basis vectors Q_{st+1} obtained from Arnoldi process with MGS. $Q_{st+1} = \widetilde{Q}_{st+1}\theta_{s+1}$ where the absolute value of θ_{s+1} is the $(s+1) \times (s+1)$ identity matrix. It is shown in [48] how obtain the approximate solution x_{st} of CA-GMRES equal to that of GMRES.

Algorithm 31 summarizes the main steps in CA-GMRES. For a detailed algorithm refer to [48], where the author shows that it is possible to delay the reconstruction of the $(si+1) \times si$ upper Hessenberg matrix H_{si} and the solution of the least square problem until convergence is attained

Algorithm 31 CA-GMRES

Input: A, the $n \times n$ matrix; b, the $n \times 1$ right-hand side; x_0, the initial guess or iterate
Input: ϵ, the stopping tolerance; s, the s-step basis length; t, the number of Arnoldi(s,t) iterations
Input: k_{max}, the maximum number of restarts
Output: $x_s t$, the approximate solution of the system $Ax = b$

Compute $r_0 = b - Ax_0$, $\rho_0 = ||r_0||_2$, $v_1 = \frac{r_0}{\rho}$, $i = 1$, $k = 1$
Let $E_{s+1} = I_s$, $e_1 = E_{s+1}(:, 1)$, $p_1 = \rho_0 e_1$
while ($\rho_s \geqslant \epsilon||b||_2$ and $i < t$) **do**
Perform Arnoldi(s,t) iteration
    Compute $v_{s(i-1)+2} = Av_{s(i-1)+1}$, $v_{s(i-1)+3} = Av_{s(i-1)+2}$,.., $v_{si+1} = Av_{si}$ using the Matrix Powers kernel (Algorithm 25)
    if $i = 1$ **then**
        Let $V_{s+1} = [v_1, v_2, .., v_{s+1}]$ be an $n \times (s + 1)$ matrix
        Orthonormalize V_{s+1} using TSQR algorithm (section 2.4.2.2) and let $Q_{s+1} = [V_{s+1}]$
    else
        Let $V_{si+1} = [v_{s(i-1)+2}, v_{s(i-1)+3}, .., v_{si+1}]$ be an $n \times s$ matrix
        Orthonormalize V_{si+1} against V_{s+1},.., $V_{s(i-1)+1}$ using BCGS (Algorithm 2)
        Orthonormalize V_{si+1} using TSQR algorithm (section 2.4.2.2)
        Let $Q_{si+1} = [Q_{s(i-1)+1}, V_{si+1}]$ and $Q_{si} = Q_{si+1}(:, 1 : si)$
    end if
    Reconstruct the $si + 1 \times si$ upper Hessenberg matrix $H_s i$ where $AQ_{si} = Q_{si+1}H_{si}$
Solve Least Square $y_{si} = min_y||p_1 - H_{si}y||$
        Apply Givens rotations on H_{si} and p_1
        Solve the upper triangular system $H_{si}y = p_1$ by backward substitution to obtain y_{si}
        $\rho_{si} = ||p_1 - H_{si}y_{si}||$, $i = i + 1$, $k = k + 1$, $p_1 = [p_1, 0_s]$
end while
if ($\rho_{si} \geqslant \epsilon||b||_2$) and $k \leqslant k_{max}$ **then**
    Let $i = i - 1$, and $x_{si} = x_0 + Q_{si}y_{si}$
    Let $x_0 = x_{si}$ and restart by calling CA-GMRES with $k_{max} = k_{max} - 1$
else $x_{si}$is the approximate solution
end if

or a restart is needed. In total, as explained in [48], CA-GMRES communicates a factor of $\Theta(s)$ fewer messages in parallel than GMRES. In a sequential machine with two levels of fast and slow memory, it reads the sparse matrix and vectors from slow memory to fast memory a factor of $\Theta(s)$ fewer times.

3.2.4 Other CG methods

Apart from s-step and communication avoiding methods that merge s iterations of the classical Krylov methods to reduce communication in parallel implementations, other ideas were introduced. In this section we discuss two CG variants that are related to our introduced enlarged Krylov subspace CG variants. The first method is called cooperative-CG (coop-CG) [8] which was recently introduced, solves the system $Ax = b$ by starting with t distinct initial guesses. This is equivalent to solving the system $AX = b * \mathbb{1}_t$ (algorithmically very similar to Block CG) where $\mathbb{1}$ is a vector of ones of size t. The authors also present a parallel implementation that needs 2 to 3 synchronizations per iteration (section 3.2.4.1). As for the multiple search directions CG (MSD-CG) [44], it solves $Ax = b$ by partitioning A's domain into t subdomains and defining a search direction on each of the t subdomains. Then $x_k = x_{k-1} + P_k\alpha_k$, where P_k is a matrix containing all the t search directions and α_k is a vector of size t (section 3.2.4.2). Unlike CG, block CG and coop CG, MSD-CG does not have the A-orthogonality condition of the search directions, i.e. $P_k^t A P_i$ is not equal to zero for all i not equal to k. Hence it is not a projection method. This causes MSD-CG to have slower convergence than CG as we will see later in section 4.4.

3.2.4.1 Coop-CG

Recently, in 2012, Bhaya et al. presented a new version of conjugate gradient which is similar in structure to the Block conjugate gradient method. The coop-CG [8] solves the system $Ax = b$ by starting with t different initial guesses and solving the same system t times in parallel, where t threads/agents cooperate to find the solution. This is equivalent to solving the system $AX = b * ones(1, t)$ where X_0 is a block-vector containing the t initial guesses, $R_0 = AX_0 - b * \mathbb{1}_t$ is the block residual, $P_1 = R_0$ is the initial block search direction. Then the derivations and the algorithm of the coop-CG (Algorithm 32) are the same as the Block-CG with $\gamma_k = I$, where $R_{k+1} \perp \mathcal{K}_{k+1}(A, R_0)$.

For $k \geqslant 0$ the iterates are defined similarly to B-CG:

$$\begin{aligned} X_{k+1} &= X_k + P_{k+1}\alpha_{k+1} && \in \mathcal{K}_{k+1}(A, R_0) \\ R_{k+1} &= R_k - AP_{k+1}\alpha_{k+1} && \in \mathcal{K}_{k+2}(A, R_0) \\ P_{k+2} &= R_{k+1} + P_{k+1}\beta_{k+2} && \in \mathcal{K}_{k+2}(A, R_0) \end{aligned}$$

where

$$\begin{aligned} \alpha_{k+1} &= (P_{k+1}^t A P_{k+1})^{-1}(R_k^t P_k) = (P_{k+1}^t A P_{k+1})^{-1}(R_k^t R_k) \\ \beta_{k+2} &= (P_{k+1}^t A P_{k+1})^{-1}(P_{k+1}^t A R_{k+1}) = (R_k^t R_k)^{-1}(R_{k+1} R_{k+1}) \end{aligned}$$

Algorithm 32 The Coop-CG Algorithm

Input: A, the $n \times n$ symmetric positive definite matrix
Input: b, the $n \times 1$ right-hand side; X_0, the $n \times t$ initial guesses or iterates
Input: ϵ, the stopping tolerance; k_{max}, the maximum allowed iterations
Output: x_k, the approximate solution of the system $Ax = b$

$R_0 = b * ones(1, t) - AX_0$,$P = R_0$, k = 1
$\rho = min(||R_0(:,1)||_2^2, ||R_0(:,2)||_2^2, ..., ||R_0(:,t-1)||_2^2, ||R_0(:,t)||_2^2)$
while ($\sqrt{\rho} > \epsilon||b||_2$ and $k < k_{max}$) **do**
    $\alpha = (P^tAP)^{-1}(R^tP)$
    $X = X + P\alpha$
    $R = R - AP\alpha$
    $\beta = (P^tAP)^{-1}(P^tAR)$
    $P = (R + P\beta)$
    $\rho = min(||R(:,1)||_2^2, ||R(:,2)||_2^2, ..., ||R(:,t-1)||_2^2, ||R(:,t)||_2^2)$
    k = k+1
end while

A parallel implementation has also been introduced in [8]. First, $W = AP$ is computed where each agent i performs $AP(:,i)$ followed by synchronization to obtain the full W. Then, since $P^tAP = P^tW$ is a symmetric $t \times t$ matrix, only the upper triangular part needs to be computed. This work is split between the t agents followed by a synchronization. Then, each agent i computes $R^tP(:,i)$, solves for $\alpha(:,i)$, updates $X(:,i)$ and $R(:,i)$, computes $W^tR(:,i)$, solves for $\beta(:,i)$, updates $P(:,i)$, and computes $||R(:,i)||_2^2$. Then, a synchronization is needed to find ρ and check for convergence. A total of $3log(t)$ messages are sent with $O(nt)$ words.

We can further reduced communication in the above parallel implementation. First, every agent fetches the matrix A and P. Then agent i performs $W(:,i) = AP(:,i)$, followed by $P^tW(:,i)$. So a communication is avoided by computing the full matrix P^tAP rather than the upper triangular part only. After that, a communication is needed so that every agent i has the full $t \times t$ matrix P^tAP needed for finding $\alpha(:,i)$ and $\beta(:,i)$, and then updating $X(:,i)$, $R(:,i)$ and $P(:,i)$ in the order indicated in the previous paragraph. A total of $2log(t)$ messages are sent with $O(nt)$ words.

3.2.4.2 MSD-CG

The multiple search directions CG (MSD-CG), introduced by Gu et al. [44], solves the system $Ax = b$, and starts by having a partitioned domain and by defining at each iteration k a search direction p_i^k on each of the t subdomains (δ_i, $i = 1, 2, ..., t$) such that $p_i^k(\delta_j) = 0$ for all $j \neq i$. Then, the approximate solution at the k^{th} iteration is defined as $x_k = x_{k-1} + P_k\alpha_k$ where $P_k = [p_1^k\ p_2^k\ p_3^k\ ...\ p_t^k]$ is a matrix containing all the k^{th} search directions, and α_k is a vector of size t.

Given an initial guess x_0, the residual is defined as $r_k = b - Ax_k$ for $k \geqslant 0$. The first set of domain search directions is defined by the initial residual r_0, such that $p_i^1(\delta_i) = r_0(\delta_i)$ for

$i = 1, 2, ..., t$ and zero otherwise. Then, for $k > 1$ the domain search directions are defined as follows, $p_i^k = T_i(r_{k-1}) + \beta_i^k p_i^{k-1}$ for $i = 1, 2, ..., t$ where β_i^k is a scalar and T_i is an operator that projects a vector onto the subdomain δ_i ($[T_i(x)](\delta_j) = 0$ for $j \neq i$ and $[T_i(x)](\delta_i) = x(\delta_i)$). The search directions block has the following sparsity pattern for all k,

$$P_k = \begin{pmatrix} * & 0 & & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ * & 0 & & 0 & 0 \\ 0 & * & & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & * & & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & & * & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & & * & 0 \\ 0 & 0 & & 0 & * \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & & 0 & * \end{pmatrix}_{n \times t}.$$

As for $\alpha_k = (P_k^t A P_k)^{-1} P_k^t r_{k-1}$, it is chosen such that it minimizes $\phi(x_k) = min\{\phi(x_{k-1} + P_k\alpha), \forall \alpha \in \mathbb{R}^t\}$. Unlike CG, block CG and coop-CG, MSD-CG does not have the A-orthogonality condition of the search directions, i.e. $P_k^t A P_i$ is not equal to zero for all $i \neq k$. Thus, $\beta_k = (P_{k-1}^t A P_{k-1})^{-1} P_{k-1}^t A r_{k-1}$ is chosen so that the global search direction $p^k = \sum_{i=1}^t p_i^k$ is A-orthogonal to the previous domain search direction p_i^{k-1}, i.e. $(p^k)^t A P_{k-1} = 0$, for $i = 1, 2, .., t$. As for the convergence, it is shown that the rate of convergence of MSD-CG is at least as fast as that of the steepest descent method. Yet, steepest descent is known for its slow "zig-zagging" convergence. This causes the MSD-CG to have slower convergence than CG, and in some cases it does not converge at all with respect to the given stopping criteria as shown in section 4.4.

Similarly to coop-CG, the parallel implementation of MSD-CG Algorithm 33 starts by computing $W = AP$ where each processor i performs $W(:, i) = AP(:, i)$ followed by followed by $C(:, i) = P^t W(:, i)$ and $P^t(i, :,)r$. After that, a communication is needed so that every agent i has the full $t \times t$ matrix $C = P^t AP$ needed for finding α and β, the full $P^t r$ vector, and $W(Adj(\delta_i), :)$. Then, solving for α can be done using iterative or direct methods in parallel where some communicating might be needed depending on the choice of the method. After finding α, every processor i can compute $x(\delta_i) = x(\delta_i) + P(\delta_i, :)\alpha$.

As for $r = r - AP\alpha$, it can be computed similarly to x. But then there would be a need for communication before computing $P^t Ar = W^t r$. To avoid this communication, we compute $r(\gamma_i)$, where $\gamma_i = Adj(G(A), \delta_i)$ rather than $r(\delta_i)$, i.e. $r(\gamma_i) = r(\gamma_i) - W(\gamma_i, :)\alpha$. Then each processor i can compute $W^t(i, :)r = W^t(i, :)r(\gamma_i)$ independently. Then, the processors solve for β using some iterative or direct methods with some communication. Finally, each processor i updates its p_i. In this parallelization scheme, there are 3 global communications, two of which are when solving for α and β.

Algorithm 33 The MSD-CG Algorithm

Input: A, the $n \times n$ symmetric positive definite matrix
Input: b, the $n \times 1$ right-hand side; x_0, the initial guess or iterate
Input: ϵ, the stopping tolerance; k_{max}, the maximum allowed iterations
Output: x_k, the approximate solution of the system $Ax = b$

$r_0 = b - Ax_0$, $\rho = ||r_0||_2^2$, $k = 1$
for $i = 1, .., t$ **do** Let $P(:, i) = T_i(r_0)$
end for
while ($\sqrt{\rho} > \epsilon ||b||_2$ and $k < k_{max}$) **do**
    $\alpha = (P^t AP)^{-1}(P^t r)$
    $x = x + P\alpha$
    $r = r - AP\alpha$
    $\beta = (P^t AP)^{-1}(P^t Ar)$
    for $i = 1, .., t$ **do** Let $P(:, i) = T_i(r) + \beta(i)P(:, i)$
    end for
    $\rho = ||r||_2^2$
    $k = k + 1$
end while

In [44], the authors proposed to solve the $t \times t$ α and β systems inaccurately by using Jacobi method which for regular structured matrices would need local communication with neighboring processors. They call this version global inner products free CG, GIPF-CG method. Given the system $C\alpha = f$, where $C = D + R$ with D being the diagonal of C and $LU = C - D$ the remainder, then at iteration k, the Jacobi method approximates the solution by $\alpha_k = D^{-1}(f - LU\alpha_{k-1})$. At each iteration, processor i computes $\alpha_k(i) = D^{-1}(i, i)(f(i) - LU(i, :)\alpha_{k-1})$ where only $\alpha_{k-1}(Adj(G(LU), i))$ is needed. Then the processors send their part to check for convergence and communicate with neighboring processors to fetch $\alpha_k(opAdj(G(LU), i))$.

We present another alternative than the usual Jacobi method which needs local communication after each iteration. We present the s-step communication-avoiding Jacobi method which consists of perform s iterations and then checking for convergence where each processor fetches $\alpha_0(\Omega_s)$, $LU(\Omega_{s-1}, :)$, $f(\Omega_{s-1})$, and $D^{-1}(\Omega_{s-1}, \Omega_{s-1})$ where $\Omega_s = R(G(U), i, s)$. Then at iteration $1 \leqslant k \leqslant s$, processor i computes $\alpha_k(\Omega_{s-k}) = D^{-1}(\Omega_{s-k}, \Omega_{s-k})(f(\Omega_{s-k}) - LU(\Omega_{s-k}, :)\alpha_{k-1}(\Omega_{s-k+1}))$. After computing α_s, every processor sends its part of α_s to the main processor that checks for convergence. If the method did not converge then each processor fetches $\alpha_s(\Omega_s)$ and starts over. By replacing the Jacobi method with the s-step communication-avoiding version, we reduce the communication by a factor of s.

3.3 Preconditioners

Preconditioning is a process in which the original system of equation $Ax = b$ is transformed into a new system with the same solution by applying a preconditioner M, where A is an $n \times n$ matrix. There are three types of preconditioning.

1. Left preconditioning: $M^{-1}Ax = M^{-1}b$
2. Right preconditioning: $AM^{-1}y = b$, where $y = Mx$
3. Split preconditioning with $M = M_1M_2$: $M_1^{-1}AM_2^{-1}y = M_1^{-1}b$ where $y = M_2x$

The preconditioned system should have a faster rate of convergence than the original system, when solved using iterative methods. This is often realized by choosing M such that the condition number $cond_2(M^{-1}A) \approx 1$ for left preconditioners, $cond_2(AM^{-1}) \approx 1$ for right preconditioners and $cond_2(M_1^{-1}AM_2^{-1}) \approx 1$ for split preconditioners, where $M^{-1} \approx A^{-1}$. Moreover, building the preconditioner M should be cheap in terms of flops and communication. And the preconditioner M must be chosen such that the application of M^{-1} to an $n \times 1$ vector is inexpensive, $z = M^{-1}x$, or alternatively the solution of $Mz = x$ should be inexpensive in case M^{-1} is not computed. Note that it is not necessary to compute the full matrices M and M^{-1}, they could be operators on vectors.

Finding a preconditioner M, for some sparse linear system, that satisfies the above conditions is not an easy task. For systems obtained from the discretization of PDE's, it is possible to build preconditioners based on the geometry of the original problem. However, we will only discuss algebraic preconditioners that are defined by the matrix A only. The simplest algebraic preconditioners in terms of construction and application to a vector are those preconditioner based on the classical iterative methods like Jacobi, Gauss-Seidel, successive over relaxation (SOR), and symmetric successive over relaxation (SSOR) methods. These preconditioners are based on splitting the matrix A into $A = D - E - F$ where $-E$ is the strict lower triangular part of A, $-F$ is the strict upper triangular part of A, D is the diagonal part of A, and nnz is the number of nonzero entries in A. Then, the preconditioners are defined as follows, where we show the cost of $z = M^{-1}v$:

- $M = D$, Jacobi preconditioner where $z = M^{-1}v$ costs n flops
- $M = D - E$, forward Gauss Seidel preconditioner where solving the upper triangular system $Mz = v$ costs nnz flops
- $M = D - F$, backward Gauss Seidel preconditioner where solving the lower triangular system $Mz = v$ costs nnz flops
- $M = (D - E)D^{-1}(D - F)$, symmetric Gauss Seidel preconditioner where solving the lower and upper triangular systems $Mz = v$ costs 2nnz + n flops

- $M = \frac{1}{\omega}D - E$, successive over relaxation (SOR) preconditioner where solving the upper triangular system $Mz = v$ costs nnz + n flops

- $M = \frac{1}{2-\omega}(\frac{1}{\omega}D - E)(\frac{1}{\omega}D)^{-1}(\frac{1}{\omega}D - F)$ symmetric successive over relaxation (SSOR) preconditioner where solving the lower and upper triangular systems $Mz = v$ costs 2nnz + $3n$ flops

A second type of preconditioners is based on an approximate factorization of A, like incomplete LU preconditioner and incomplete Cholesky preconditioner. The ILU preconditioner is based on the ILU factorization of $A = LU + R$. Here $M = LU$, L is sparse lower triangular , U is sparse upper triangular, and R is the residual $R = A - M$. Incomplete Cholesky (IC) preconditioner for a symmetric positive definite A is based on the IC factorization $A = LL^T + R$, where $M = LL^t$, L is sparse lower triangular, and R is the residual $R = A - M$ (refer to [70] and references therein) .

A third type of preconditioners is sparse approximate inverse (SPAI) . It is known that the inverse of a sparse matrix is a full matrix. Thus, SPAI preconditioner is based on the idea of choosing some sparse matrix $T \in \mathbb{S}$ that minimizes $||I - TA||_F = ||I - A^tT^t||_F$ for left preconditioning, or minimizes $||I - AT||_F$ for right preconditioning, where $\mathbb{S}$ is a set of sparse matrices, and $||I - AT||_F^2 = \sum_{i=1}^{n} ||e_i - AT(:,i)||_2^2$ is the Frobenius norm. Thus finding the $M^{-1} = T$ is equivalent to solving n independent least square problems since

$$min_{T\in\mathbb{S}}(\sum_{i=1}^{n} ||e_i - AT(:,i)||_2^2) = \sum_{i=1}^{n}(min_{T\in\mathbb{S}}||e_i - AT(:,i)||_2^2)$$

For an overview of different SPAI techniques and of their parallelization, refer to [7, 16, 15] and references heirin.

A fourth type of preconditioners is based on domain decomposition of the unknowns. The subdomains can be overlapping like restricted additive Schwarz (RAS) or non-overlapping like block Jacobi preconditioner (BJ). Then the preconditioner is equal to the blocks of A restricted to the subdomains.

Deflation is used to accelerate the convergence of Krylov subspace methods, and it can be done through preconditioning or augmenting the Krylov subspace by some vectors [32]. Deflation as a preconditioning method was first introduced for speeding up the convergence of CG [62] in 1987. Since then a lot of work has been done on the subject of deflated CG [67, 73, 31, 54], deflated GMRES [27, 29, 53, 14], and augmented Krylov methods [28, 61].

There are other types of preconditioners that we do not describe in this thesis, like algebraic multigrid preconditioners and algebraic multilevel preconditioners. For an introduction to algebraic multigrid preconditioners, which are based algebraic multigrid methods [17] that solve the problem on a coarser grid of unknown x and then interpolates the solution back to the initial fine grid, refer to [80] and references herein. For an introduction to algebraic multilevel preconditioners, refer to [2, 68, 9].

The preconditioned versions of CG, GMRES, and the other parallelizable variants are slightly different from the original methods. The matrix A is replaced by $M^{-1}A$, AM^{-1} or $M_1^{-1}AM_2^{-1}$ and b by $M^{-1}b$ or $M_1^{-1}b$ depending on the preconditioning type. In all the Krylov methods discussed in this section, the matrix A is either multiplied by a vector or a block of vectors. Thus, there is no need to multiply the matrices A and M^{-1}, it should only be possible to apply the preconditioner to a vector $M^{-1}v$. For the full preconditioned versions refer to [65, 50, 63, 44, 8].

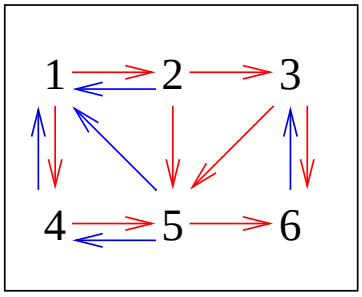
In this section, ILU preconditioner (section 3.3.1), block Jacobi preconditioner (section 3.3.2), and RAS preconditioner (section 3.3.3), are briefly described. For a survey on preconditioning techniques refer to [5].

3.3.1 Incomplete LU preconditioner

Incomplete LU preconditioners is based on incomplete LU factorizations where $A = LU + R$ and $M = LU$. The complete LU factorization is a Gaussian elimination, where the obtained L and U factors have more nonzero entries than the input sparse matrix A. There are several incomplete LU factorizations that drop some entries of the L and U matrices to obtain sparse factors . The dropping process is based on some condition, that can be a sparsity pattern or some drop tolerence. Some of the ILU factorizations are zero fill-in ILU(0), level of fill ILU(p), threshold ILUT, and modified ILU (MILU), and other variants. For a full description of the different ILU factorizations refer to [65].

$$\begin{pmatrix} \times & \times & 0 & \times & 0 & 0 \\ \times & \times & \times & 0 & \times & 0 \\ 0 & 0 & \times & 0 & \times & \times \\ \times & 0 & 0 & \times & \times & 0 \\ \times & 0 & 0 & \times & \times & \times \\ 0 & 0 & \times & 0 & 0 & \times \end{pmatrix}$$

(a) Matrix A



(b) Graph of A

Figure 3.1: The figure shows the sparsity pattern of a matrix A and its graph. The lower triangular blue part along with the diagonal represents the sparsity pattern of the L matrix obtained from the ILU(0) factorization of A. Whereas the upper triangular red part along with the diagonal represents the sparsity pattern of the U matrix. Similarly, the blue edges in Figure 3.1(b) represent the graph of L, whereas the red ones represent that of U.

In this section, we will briefly discuss the ILU(0) factorization and preconditioner. Although we do not address the issue of parallelizing the factorization, there has been a lot of work on parallelizing ILU factorization based on applying some reordering to obtain a set of rows that can

be eliminated in parallel. For a brief overview of the different approaches for parallelizing the ILU factorization, refer to [18] and references herein. Recently, Chow et al. [18] presented a new approach for parallelizing the ILU factorization which is not based on reordering the matrix, but it is based on reformulating the ILU factorization as a solution of a set of bilinear equations.

The ILU(0) factorization of $A = LU + R$ produces L and U factors that have the same sparsity pattern as the lower and upper triangular part of A as shown in Figure 3.1. This is obtained by performing an LU factorization, where only the nonzero entries of A are modified as shown in Algorithm 34. The obtained L matrix has ones on the diagonals. As it is clear in figure 3.1, the graph of L has all the edges of $G(A)$ connecting vertex i to j where $j < i$. Whereas, the graph of U has all the edges of $G(A)$ connecting vertex j to i where $j < i$. Thus, in chapter 5, the figures with the graph of A also represent the graph of L and U obtained from the ILU(0) factorization of A.

Algorithm 34 ILU(0) factorization

Input: A, the $n \times n$ matrix; b, the $n \times 1$ right-hand side
Output: L, U, the lower and upper triangular matrices from the ILU(0) factorization of A

```
1:  L = I
2:  for i = 2 : n do
3:      for k = 1 : i − 1 and A(i, k) ≠ 0 do
4:          A(i, k) = A(i, k)/A(k, k)
5:          for j = k + 1 : n and A(i, j) ≠ 0 do
6:              A(i, j) = A(i, j) − A(i, k)A(k, j)
7:          end for
8:      end for
9:      Let L(i, 1 : i − 1) = A(i, 1 : i − 1) and U(i, i : n) = A(i, i : n)
10: end for
```

The complete LU factorization of a dense $n \times n$ matrix, where the $A(i,k) \neq 0$ and $A(i,j) \neq 0$ conditions in Algorithm 34 are dropped, costs $\frac{2}{3}n^3 + 2n^2$ flops. Whereas the ILU(0) factorization of A costs at most 2nnz flops where nnz is the number of nonzero entries in A.

On the other hand, the multiplication of the ILU(0) preconditioner to a vectors $z = M^{-1}v = (LU)^{-1}v$ is equivalent to solving an upper and lower triangular system $LUz = v$, where $Ly = v$ and $Uz = y$. Given that L and U have the same sparsity pattern as the lower and upper part A, the cost of computing $z = (LU)^{-1}v$ is 2nnz flops.

3.3.2 Block Jacobi preconditioner

Block Jacobi preconditioner can be considered as a domain decomposition preconditioner, where the unknowns are partitioned or alternatively the graph of A is partitioned into p subgraphs that are connected by a few edges. The matrix A can be permuted and partitioned using k-way graph

partitioning or other partitioning techniques with edge separators. Let $\delta = \{1, 2, .., n\}$ be the set of indices associated with the vertices of permuted A's graph. Then $\delta = \cup_{i=1}^{p} \delta_i$, where the δ_i's are the set of indices associated with the subgraphs' vertices, $\delta_i \cap \delta_j = \phi$ for $j \neq i$, and δ_i is a set of consecutive indices. Then, the block Jacobi preconditioner M is defined as $M(\delta_i, \delta_i) = A(\delta_i, \delta_i)$ and zero elsewhere, which is equivalent to a block diagonal matrix.

Each of the blocks is factorized using some incomplete factorization, like ILU(0) or incomplete Cholesky factorization for SPD matrices. If the blocks are small then it is possible to use the complete factorizations. A four blocks Jacobi preconditioner with ILU factorization has the following form, where $M_i = M(\delta_i, \delta_i) = A(\delta_i, \delta_i) = L_i U_i$ for $i = 1, 2, 3, 4$.

$$M = \begin{pmatrix} M_1 & 0 & 0 & 0 \\ 0 & M_2 & 0 & 0 \\ 0 & 0 & M_3 & 0 \\ 0 & 0 & 0 & M_4 \end{pmatrix} = \begin{pmatrix} L_1 & 0 & 0 & 0 \\ 0 & L_2 & 0 & 0 \\ 0 & 0 & L_3 & 0 \\ 0 & 0 & 0 & L_4 \end{pmatrix} \begin{pmatrix} U_1 & 0 & 0 & 0 \\ 0 & U_2 & 0 & 0 \\ 0 & 0 & U_3 & 0 \\ 0 & 0 & 0 & U_4 \end{pmatrix} = L_{BJ} U_{BJ}$$

The cost of performing the p independent ILU factorizations of M_i is less than the cost of performing the ILU factorization of the matrix A. The multiplication $z = M^{-1}v = (L_{BJ}U_{BJ})^{-1}v$ is naturally parallelizable due to the block format of L_{BJ} and U_{BJ}. Each processor i solves $L_i U_i z(\delta_i) = v(\delta_i)$ by solving $L_i y(\delta_i) = v(\delta_i)$ and $U_i z(\delta_i) = y(\delta_i)$ without any communication with the other processors. And its cost is less than solving the L and U systems obtained from the ILU factorization of A. For example, the cost of the ILU(0) factorization of all $M_i = L_i U_i$'s is less than 2nnz flops, where nnz is the number of nonzero entries in A. And solving the corresponding upper and lower triangular system costs less than 2nnz flops. The exact cost depends on the size of the blocks. The smaller the blocks are, the cheaper the preconditioner is. But it becomes less efficient as a preconditioner, since a lot of information has been dropped out.

3.3.3 Restricted additive Schwarz preconditioner

Restricted additive Schwarz (RAS) is a domain decomposition method. Similarly to block Jacobi method, the graph of A is partitioned into p subgraphs with $\delta = \cup_{i=1}^{p} \delta_i = \{1, 2, .., n\}$ the set of indices associated with the vertices of the permuted A or the permuted unknowns. Unlike BJ, RAS has overlapping subdomains, and the size of the overlap defines the different preconditioners.

Let $\delta_i^1 = Adj(G(A), \delta_i) = R(G(A), \delta, 1)$ and in general $\delta_i^j = R(G(A), \delta, j)$. The classical additive Schwarz AS(j) preconditioner is defined as follows:

$$M^{-1} = \sum_{i=1}^{p} R_i^j A_i^{-1} R_i^j$$

where R_i^j is an $n \times n$ restriction matrix with $R_i^j(\delta_i^j, \delta_i^j) = I$ and zero elsewhere, and $A_i = R_i^j A R_i^j$ with $A_i(\delta_i^j, \delta_i^j) = A(\delta_i^j, \delta_i^j)$ and zero elsewhere. When multiplying $z = M^{-1}v$ in parallel, each processor can fetch A_i and $v(\delta_i^j)$ and compute $z_i(\delta_i^j) = [R_i^j A_i^{-1} R_i^j v](\delta_i^j)$. But $z(\delta_i^j) = \sum_{i=1}^{p} z_i(\delta_i^j) \neq$

$z_i(\delta_i^j)$ due to the fact that $\delta_i^j \cap \delta_h^j \neq \phi$ for some $h \neq i$. Thus there is a need for communication to get $z(\delta_i^j)$. The RAS preconditioner avoids this communication.

The RAS(j) preconditioner, introduced in [10], is defined as follows:

$$M^{-1} = \sum_{i=1}^{p} R_i^0 A_i^{-1} R_i^j$$

where R_i^0 is an $n \times n$ restriction matrix with $R_i^0(\delta_i, \delta_i) = I$ and zero elsewhere, and $A_i = A(\delta_i^j, \delta_i^j)$. When computing $z = M^{-1}v = \sum_{i=1}^{p} R_i^0 A_i^{-1} R_i^j v$ in parallel, each processor i fetches A_i and $v(\delta_i^j)$ and computes $z(\delta_i) = z_i(\delta_i) = [R_i^0 A_i^{-1} R_i^j v](\delta_i)$. This is due to the replacement of R_i^j by R_i^0. A_i is not invertible, however $A(\delta_i^j, \delta_i^j)$ is. Thus, processor i computes $y(\delta_i^j) = A^{-1}(\delta_i^j, \delta_i^j)v(\delta_i^j)$ by solving the system $A(\delta_i^j, \delta_i^j)y(\delta_i^j) = v(\delta_i^j)$ similarly to block Jacobi. Then, $z(\delta_i) = y(\delta_i)$ where the overlapping entries of $y(\delta_i^j)$ are dropped.

Chapter 4

Enlarged Krylov Subspace (EKS) Methods

In this chapter we introduce the new enlarged Krylov subspace (section 4.1) which is based on domain decomposition. The purpose of enlarging the Krylov subspace is to obtain enlarged Krylov subspace (EKS) methods that converge faster than the classical Krylov methods when solving the system $Ax = b$. Moreover, we would like the EKS methods to be better parallelizable, than the Krylov subspace methods, while avoiding communication, similarly to block methods (section 3.2.1), s-step methods (section 3.2.2), and communication avoiding methods (section 3.2.3) that replace BLAS 1 and BLAS 2 computations by BLAS2 and BLAS 3. This chapter is based on the article [41] which is in preparation for submission.

We introduce two enlarged Krylov projection methods. The first is called Multiple Search Direction with Orthogonalization Conjugate Gradient (MSDO-CG) method which is based on the idea of using multiple search directions at each iteration. This idea is not new, it was introduced in [44]. However, in MSDO-CG (section 4.2) after defining the t search directions, they are A-orthonormalized against previous search directions and against each others, to obtain a projection method. The second method is called Long Recurrence Enlarged Conjugate Gradient (LRE-CG) method where rather than defining search directions, an orthonormal basis for the Enlarged Krylov subspace is built (section 4.3). At each iteration k, t new basis vectors are computed for the enlarged Krylov subspace. Then, rather than having short recurrences, the approximate solution x_k is defined by all the basis vectors as in GMRES.

By enlarging the Krylov subspace, the MSDO-CG and LRE-CG converge faster than CG in exact precision. In section 4.4, we present the convergence results of both methods in finite precision and compare them to existing methods, like CG (section 3.1.4), coop-CG (section 3.2.4.1), MSD-CG (section 3.2.4.2). Both methods, MSDO-CG and LRE-CG, require saving at most tk vectors versus one search direction in CG. Yet LRE-CG converges faster than MSDO-CG (section 4.4) at the expense of solving growing systems of size tk. Several remedies to this problem are discussed in section 4.3.1. In section 4.5, we present possible parallel versions of the methods and their expected performance. Finally, in section 4.6, we present the preconditioned versions of the methods and their convergence behavior.

Although we only discuss in this thesis EKS conjugate gradient versions, it is possible to derive other enlarged Krylov methods, like EKS-GMRES which has been derived but not tested yet.

4.1 The enlarged Krylov subspace

The enlarged Krylov subspace and methods are based on a partition of the unknowns, or alternatively the rows of the $n \times n$ matrix A. Assume that the index domain $\delta = \{1, 2, .., n\}$ is divided into t distinct subdomains δ_i, where $\delta = \cup_{i=1}^{t} \delta_i$. Note that the partitioning of the index domain δ can be obtained by partitioning the graph of A, $G(A) = (V, E)$ into t subgraphs $\{\Omega_1, \Omega_2, ..., \Omega_t\}$ as discussed in section 2.2 , where $\delta_i = V(\Omega_i)$ and $\delta = V(G(A))$.

We define $T_i(x)$ to be the operator that projects the vector x onto the subdomain δ_i. Let $y = T_i(x)$, then $y(\delta_i) = x(\delta_i)$ and zero elsewhere. Then, we define $T(x)$ to be an operator that transforms the $n \times 1$ vector x into t vectors of size $n \times 1$ that correspond to the projection of x onto the subdomains δ_i for $i = 1, 2, .., t$. If the obtained t vectors are assembled in increasing order into a block vector X, then we have $X(\delta_i, i) = x(\delta_i)$ for all i and zero elsewhere. We will refer to R_0 as the block containing the t vectors obtained from $T(r_0)$. Note that $R_0 \neq T(r_0)$ since R_0 is a matrix, whereas $T(r_0) = \{T_1(r_0),\ T_2(r_0),\ ....\ , T_t(r_0)\}$ is a set of vectors. But $R_0 = [T_1(r_0)\, T_2(r_0)\, ...\, T_t(r_0)]$, where the brackets $[..]$ denote a matrix format.

Definition 4.1.1. *Let*

$$\begin{aligned} \mathcal{K}_{t,k} &= span\{T(r_0), AT(r_0), A^2T(r0), ..., A^{k-1}T(r_0)\} \\ &= span\{T_1(r_0), T_2(r_0), ..., T_t(r_0), AT_1(r_0), AT_2(r_0), ..., AT_t(r_0), ..., A^{k-1}T_1(r_0), ..., A^{k-1}T_t(r_0)\} \end{aligned}$$

be an enlarged Krylov subspace of dimension $k \leqslant z \leqslant tk$ generated by the matrix A and the vector r_0, and associated to a given partition defined by δ_i for $i = 1, 2, .., t$.

The enlarged Krylov subspaces $\mathcal{K}_{t,k}(A, r_0)$ are increasing subspaces, yet bounded. We denote by k_{max} the upper bound k for which the dimension of the enlarged Krylov subspace $\mathcal{K}_{t,k}(A, r_0)$ stops increasing. For simplicity, we will denote the enlarged Krylov subspace generated by A and r_0, $\mathcal{K}_{t,k}(A, r_0)$, by $\mathcal{K}_{t,k}$, and the Krylov subspace generated by A and r_0, $\mathcal{K}_k(A, r_0)$ by $\mathcal{K}_k$.

Theorem 4.1.2. *The Krylov subspace $\mathcal{K}_k$ is a subset of the enlarged Krylov subspace $\mathcal{K}_{t,k}$ ($\mathcal{K}_k \subset \mathcal{K}_{t,k}$).*

Proof. Let $y \in \mathcal{K}_k$ where $\mathcal{K}_k = span\{r_0, Ar_0, .., A^{k-1}r_0\}$. Then

$$y = \sum_{j=0}^{k-1} a_j A^j r_0 = \sum_{j=0}^{k-1} a_j A^j R_0 * \mathbb{1}_t = \sum_{j=0}^{k-1} \sum_{i=1}^{t} a_j A^j T_i(r_0) \in \mathcal{K}_{t,k}$$

since $r_0 = R_0 * \mathbb{1}_t = [T_1(r_0)\, T_2(r_0)\, ...\, T_t(r_0)] * \mathbb{1}_t$. □

Krylov subspace methods search for an approximate solution $x_k \in x_0 + \mathcal{K}_k$. A corollary of theorem 4.1.2 is that we can search for an approximate solution x_k in $x_0 + \mathcal{K}_{t,k}$ instead, since $\mathcal{K}_k \subset \mathcal{K}_{t,k}$.

In theorem 4.1.3, we do not use the direct sum $\oplus$ since it is not guaranteed that the intersection of the two subspaces, $\mathcal{K}_{t,k}$ and $span\{A^kT_1(r_0), A^kT_2(r_0), ..., A^kT_t(r_0)\}$, is empty.

Theorem 4.1.3. *By definition 4.1.1 of the enlarged Krylov subspace,*

$$\mathcal{K}_{t,k+1} = \mathcal{K}_{t,k} \ + \ span\{A^kT_1(r_0), A^kT_2(r_0), ..., A^kT_t(r_0)\}.$$

If $A^kT_v(r_0) \in \mathcal{K}_{t,k}$ for all $1 \leqslant v \leqslant t$, then $A^{k+q}T_i(r_0) \in \mathcal{K}_{t,k}$ for some $1 \leqslant i \leqslant t$ and for some $q > 0$.

Proof. We prove this by induction.
Base Case:
Given that $A^kT_v(r_0) \in \mathcal{K}_{t,k}$ for all $1 \leqslant v \leqslant t$, we show that $A^{k+1}T_i(r_0) \in \mathcal{K}_{t,k}$, where $1 \leqslant i \leqslant t$. $A^kT_i(r_0) = \sum_{u=0}^{k-1}\sum_{v=1}^{t} \alpha_{u,v}A^uT_v(r_0)$ since $A^kT_i(r_0) \in \mathcal{K}_{t,k}$. Then

$$\begin{aligned}
A^{k+1}T_i(r_0) &= \sum_{u=0}^{k-1}\sum_{v=1}^{t} \alpha_{u,v}A^{u+1}T_v(r_0) = \sum_{u=0}^{k-2}\sum_{v=1}^{t} \alpha_{u,v}A^{u+1}T_v(r_0) + \sum_{v=1}^{t} \alpha_{k-1,v}A^kT_v(r_0) \\
&= \sum_{u=0}^{k-2}\sum_{v=1}^{t} \alpha_{u,v}A^{u+1}T_v(r_0) + \sum_{v=1}^{t} \alpha_{k-1,v}(\sum_{u=0}^{k-1}\sum_{y=1}^{t} \beta_{u,y}A^uT_y(r_0)) \\
&= \sum_{u=0}^{k-1}\sum_{v=1}^{t} \gamma_{u,v}A^uT_v(r_0) \in \mathcal{K}_{t,k}
\end{aligned}$$

Assume true for q:
Assume that $A^{k+q}T_i(r_0) \in \mathcal{K}_{t,k}$ where $1 \leqslant i \leqslant t$, that is $A^{k+q}T_i(r_0) = \sum_{u=0}^{k-1}\sum_{v=1}^{t} \alpha_{u,v}A^uT_v(r_0)$
Prove true for q+1:
Show that $A^{k+q+1}T_i(r_0) \in \mathcal{K}_{t,k}$

$$\begin{aligned}
A^{k+q+1}T_i(r_0) &= \sum_{u=0}^{k-1}\sum_{v=1}^{t} \alpha_{u,v}A^{u+1}T_v(r_0) = \sum_{u=0}^{k-2}\sum_{v=1}^{t} \alpha_{u,v}A^{u+1}T_v(r_0) + \sum_{v=1}^{t} \alpha_{k-1,v}A^kT_v(r_0) \\
&= \sum_{u=0}^{k-2}\sum_{v=1}^{t} \alpha_{u,v}A^{u+1}T_v(r_0) + \sum_{v=1}^{t} \alpha_{k-1,v}(\sum_{u=0}^{k-1}\sum_{y=1}^{t} \beta_{u,y}A^uT_y(r_0)) \\
&= \sum_{u=0}^{k-1}\sum_{v=1}^{t} \gamma_{u,v}A^uT_v(r_0) \in \mathcal{K}_{t,k}
\end{aligned}$$

□

Given that $\mathcal{K}_{t,k} \neq \mathcal{K}_{t,k-1}$, then a corollary of Theorem 4.1.3 is that $\mathcal{K}_{t,k} = \mathcal{K}_{t,k+q}$ for all $q > 0$, where $k_{max} = k$ is the upper bound for which the dimension of the enlarged Krylov subspace stops increasing. Assume that $A^k T_v(r_0) \in \mathcal{K}_{t,k}$ for all $1 \leqslant v \leqslant t$, then by Theorem 4.1.3 $A^{k+q} T_i(r_0) \in \mathcal{K}_{t,k}$ for all $q > 0$ and for some $1 \leqslant i \leqslant t$. Then for all $1 \leqslant i \leqslant t$ and for all $q > 0$, $A^{k+q} T_i(r_0) \in \mathcal{K}_{t,k}$. Thus no new vector is added to the basis of $\mathcal{K}_{t,k+q}$ for all $q > 0$ and $\mathcal{K}_{t,k} = \mathcal{K}_{t,k+q}$. Moreover, since $\mathcal{K}_{t,k} \neq \mathcal{K}_{t,k-1}$ then $k_{max} = k$.

Theorem 4.1.4. *If* $A^k T_i(r_0) \in \mathcal{K}_{t,k} + span\{A^k T_1(r_0), ..., A^k T_{i-1}(r_0), A^k T_{i+1}(r_0), ..., A^k T_t(r_0)\}$, *then*
$A^{k+q} T_i(r_0) \in \mathcal{K}_{t,k+q} + span\{A^{k+q} T_1(r_0), ..., A^{k+q} T_{i-1}(r_0), A^{k+q} T_{i+1}(r_0), ..., A^{k+q} T_t(r_0)\}$ *for all* $1 \leqslant i \leqslant t$ *and* $q > 0$.

Proof. If $A^k T_i(r_0) \in \mathcal{K}_{t,k} + span\{A^k T_1(r_0), ..., A^k T_{i-1}(r_0), A^k T_{i+1}(r_0), ..., A^k T_t(r_0)\}$, then $A^k T_i(r_0) = \sum_{u=0}^{k-1} \sum_{v=1}^{t} \alpha_{u,v} A^u T_v(r_0) + \sum_{\substack{v=1 \\ v \neq i}}^{t} \alpha_{k,v} A^k T_v(r_0)$. Thus,

$$\begin{aligned} A^{k+q} T_i(r_0) &= \sum_{u=0}^{k-1} \sum_{v=1}^{t} \alpha_{u,v} A^{u+q} T_v(r_0) + \sum_{\substack{v=1 \\ v \neq i}}^{t} \alpha_{j,v} A^{k+q} T_v(r_0) \\ &\in \mathcal{K}_{t,k+q} + span\{A^{k+q} T_1(r_0), ..., A^{k+q} T_{i-1}(r_0), A^{k+q} T_{i+1}(r_0), ..., A^{k+q} T_t(r_0)\} \end{aligned}$$

□

A corollary of Theorem 4.1.4 is that if $t - i_k$ vectors of the form $A^k T_y(r_0)$ with $y = i_k + 1, ..., t$ belong to the subspace $\mathcal{K}_{t,k} + span\{A^k T_1(r_0), A^k T_2(r_0), ..., A^k T_{i_k}(r_0)\}$, then the $t - i_k$ vectors of the form $A^{k+q} T_y(r_0)$ belong to the subspace $\mathcal{K}_{t,k+q} + span\{A^{k+q} T_1(r_0), A^{k+q} T_2(r_0), ..., A^{k+q} T_{i_j}(r_0)\}$.

Theorem 4.1.5. *Let* k_{max} *be the smallest integer such that* $\mathcal{K}_{t,k_{max}} = \mathcal{K}_{t,k_{max}+q}$ *for all* $q > 0$. *Then, for all* $k < k_{max}$ *the dimension of the enlarged Krylov subspaces* $\mathcal{K}_{t,k}$ *and* $\mathcal{K}_{t,k+1}$ *is strictly increasing by some number* i_k *and* i_{k+1} *respectively, where* $1 \leqslant i_{k+1} \leqslant i_k \leqslant t$.

Proof. By definition of k_{max}, we have that for all $q > 0$

$$\mathcal{K}_{t,1} \subsetneq ... \subsetneq \mathcal{K}_{t,k_{max}-1} \subsetneq \mathcal{K}_{t,k_{max}} = \mathcal{K}_{t,k_{max}+q}.$$

Then for all $k < k_{max}$, the dimension of the enlarged Krylov subspaces $\mathcal{K}_{t,k}$ is strictly increasing by some number $i_k \neq 0$ with respect to the dimension of $\mathcal{K}_{t,k-1}$.

If the t new vectors are linearly independent and none of them belongs to $\mathcal{K}_{t,k-1}$, then the t vectors are added to the basis of $\mathcal{K}_{t,k}$ and $dim(\mathcal{K}_{t,k}) = dim(\mathcal{K}_{t,k-1}) + t$, where $i_k = t$ and $dim()$ is the dimension of a subspace . In case the t new vectors are linearly dependent and none of them belongs to $\mathcal{K}_{t,k-1}$, then only one vector is added to the basis of $\mathcal{K}_{t,k}$ and $dim(\mathcal{K}_{t,k}) = dim(\mathcal{K}_{t,k-1}) + 1$ that is $i_k = 1$. There are many other cases where $1 < t - i_k < t$ of the t vectors belong to $\mathcal{K}_{t,k-1}$ or are linearly dependant on the other i_k vectors and $\mathcal{K}_{t,k-1}$. Then i_k vectors are added to the basis of $\mathcal{K}_{t,k}$ and $dim(\mathcal{K}_{t,k}) = (\mathcal{K}_{t,k-1}) + i_k$, where $1 < i_k < t$.

In general, $dim(\mathcal{K}_{t,k}) = dim(\mathcal{K}_{t,k-1}) + i_k$, where $1 \leqslant i_k \leqslant t$. Similarly, $dim(\mathcal{K}_{t,k+1}) = dim(\mathcal{K}_{t,k}) + i_{k+1}$, where $1 \leqslant i_{k+1} \leqslant t$. Moreover, in $\mathcal{K}_{t,k}$'s basis we added i_k new vectors of the form $A^{k-1}T_i(r_0)$, while the other $t - i_k$ either belong to $\mathcal{K}_{t,k-1}$ or are linearly dependant on the i_k vectors and $\mathcal{K}_{t,k-1}$. In both cases, the $t - i_k$ vectors of the form $A^{k-1}T_i(r_0)$ belong to the subspace $\mathcal{K}_{t,k-1} + span\{A^{k-1}T_1(r_0), ..., A^{k-1}T_{i_k}(r_0)\}$. Then by Theorem 4.1.4 and its corollary, the $t - i_k$ vectors of the form $A^{k+q}T_i(r_0)$ belong to the subspace $\mathcal{K}_{t,k+q} + span\{A^{k+q}T_1(r_0), A^{k+q}T_2(r_0), ..., A^{j+q}T_{i_k}(r_0)\}$ for $q > 0$.Therefore, we have at least $t-i_k$ linearly dependent vectors added to $\mathcal{K}_{t,k+1}$, hence i_{k+1} can never be greater than i_k. □

Theorem 4.1.6. *Let p_{max} and k_{max} be such that $\mathcal{K}_{p_{max}} = \mathcal{K}_{p_{max}+q}$ and $\mathcal{K}_{t,k_{max}} = \mathcal{K}_{t,k_{max}+q}$ for $q > 0$. Then $k_{max} \leqslant p_{max}$.*

Proof. Let $\mathcal{K}_{p_{max}} = \mathcal{K}_{p_{max}+q}$ and $A^{p_{max}+q-1}r_0 \in \mathcal{K}_{p_{max}+q}$ where $q > 0$. Then $A^{p_{max}+q-1}r_0 \in \mathcal{K}_{p_{max}}$ with $\mathcal{K}_{p_{max}} \subset \mathcal{K}_{t,p_{max}}$, implying that $A^{p_{max}+q-1}r_0 = \sum_{j=1}^{p_{max}} \sum_{i=1}^{t} \alpha_{j,i}A^{j-1}T_i(r_0)$. Thus,

$$A^{p_{max}+q-1} \sum_{i=1}^{t} T_i(r_0) = \sum_{j=1}^{p_{max}} \sum_{i=1}^{t} \alpha_{j,i}A^{j-1}T_i(r_0)$$

Suppose that $A^{p_{max}+q-1}T_i(r_0) \notin \mathcal{K}_{t,p_{max}}$ for all $1 \leqslant i \leqslant t$. Then

$$A^{p_{max}+q-1} \sum_{i=1}^{t} T_i(r_0) = \sum_{j=1}^{p_{max}+q-1} \sum_{i=1}^{t} \alpha_{j,i}A^{j-1}T_i(r_0).$$

We may assume that there exists at least one $\alpha_{j,i} \neq 0$ for $j > p_{max}$, then this leads to a contradiction. This implies that $A^{p_{max}+q-1}T_i(r_0) \in \mathcal{K}_{t,p_{max}}$ for all $1 \leqslant i \leqslant t$.

Thus by definition of the $T(.)$ operator and since $\mathcal{K}_p$ is a subset of $\mathcal{K}_{t,p}$, if $\mathcal{K}_{p_{max}} = \mathcal{K}_{p_{max}+q}$, then $\mathcal{K}_{t,p_{max}} = \mathcal{K}_{t,p_{max}+q}$. However, if $\mathcal{K}_{t,k_{max}} = \mathcal{K}_{t,k_{max}+q}$ this does not imply that $\mathcal{K}_{k_{max}} = \mathcal{K}_{k_{max}+q}$. Since $\mathcal{K}_{t,k}$ is a much larger subspace than $\mathcal{K}_k$, it is possible to reach stagnation earlier. Therefore $k_{max} \leqslant p_{max}$. □

Theorem 4.1.7. *The solution of the system $Ax = b$ belongs to the subspace $x_0 + \mathcal{K}_{t,k_{max}}$, where $\mathcal{K}_{t,k_{max}+q} = \mathcal{K}_{t,k_{max}}$, for $q > 0$.*

Proof. The solution $x_{sol} \in x_0 + \mathcal{K}_{p_{max}}$, where $\mathcal{K}_{p_{max}} = span\{r_0, Ar_0, .., A^{p_{max}-1}r_0\}$ and $\mathcal{K}_{p_{max}} = \mathcal{K}_{p_{max}+q}$ for $q > 0$. Since $\mathcal{K}_{p_{max}} \subset \mathcal{K}_{t,p_{max}}$, the solution $x_{sol} \in x_0 + \mathcal{K}_{t,p_{max}}$, where $p_{max} \geqslant k_{max}$ by Theorem 4.1.6.

Suppose that $x_{sol} \in x_0 + \mathcal{K}_{t,p_{max}}$, but $x_{sol} \notin x_0 + \mathcal{K}_{t,k_{max}}$. This implies that $\mathcal{K}_{t,k_{max}} \neq \mathcal{K}_{t,p_{max}}$. However, by definition of k_{max} and since $k_{max} \leqslant p_{max}$, we have that $\mathcal{K}_{t,k_{max}} = \mathcal{K}_{t,p_{max}}$. This is a contradiction. □

4.1.1 Krylov projection methods

The Krylov projection methods find a sequence of approximate solutions x_k ($k > 0$) of the system $Ax = b$ from the subspace $x_0 + \mathcal{K}_k \subseteq \mathbb{R}^n$ (or $\subseteq \mathbb{C}^n$) by imposing the Petrov-Galerkin constraint on the k^{th} residual $r_k = b - Ax_k$, that is r_k is orthogonal to some well-defined subspace of dimension k.

We define our new enlarged Krylov projection methods based on CG by the subspace $\mathcal{K}_{t,k}$ and the following two conditions:

1. Subspace condition: $x_k \in x_0 + \mathcal{K}_{t,k}$
2. Orthogonality condition: $r_k \perp \mathcal{K}_{t,k}$

$$\iff (r_k)^t y = 0, \quad \text{for all} \quad y \in \mathcal{K}_{t,k}$$

where $\mathcal{K}_{t,k}$ is a well-defined subspace of dimension $k \leqslant z \leqslant tk$.

4.1.2 The minimization property

The new enlarged CG methods find the new approximate solution by minimizing the function $\phi(x)$ over the subspace $x_0 + \mathcal{K}_{t,k}$.

Theorem 4.1.8. *If $r_k \perp \mathcal{K}_{t,k}$, then $\phi(x_k) = min\{\phi(x), \forall x \in x_0 + \mathcal{K}_{t,k}\}$.*

Proof. By the Petrov-Galerkin condition we have that $r_k \perp \mathcal{K}_{t,k}$

$$\begin{aligned} \implies (r_k)^t y &= 0, \ \forall y \in \mathcal{K}_{t,k} \\ (b - Ax_k)^t y &= 0, \ \forall y \in \mathcal{K}_{t,k} \\ b^t y - (x_k)^t A y &= 0, \ \forall y \in \mathcal{K}_{t,k} \end{aligned}$$

Let $y = x_k - x_0 \in \mathcal{K}_{t,k}$

$$\begin{aligned} &\implies (x_k)^t A(x_k - x_0) - b^t(x_k - x_0) = 0 \\ &\implies (x_k)^t A x_k - b^t x_k = (x_k)^t A x_0 - b^t x_0 \\ &\implies \phi(x_k) = \frac{1}{2}(x_k)^t A x_k - b^t x_k = -\frac{1}{2}(x_k)^t A x_k + (x_k)^t A x_0 - b^t x_0 \end{aligned}$$

By showing that $\phi(x) \geqslant \phi(x_k)$, for all $x \in x_0 + \mathcal{K}_{t,k}$ then we have proven that

$$\phi(x_k) = min\{\phi(x), \forall x \in x_0 + \mathcal{K}_k\}. \tag{4.1}$$

$$\begin{aligned}
\phi(x) - \phi(x_k) &= \frac{1}{2}x^tAx - b^tx - [-\frac{1}{2}(x_k)^tAx_k + (x_k)^tAx_0 - b^tx_0] \\
&= \frac{1}{2}x^tAx - b^tz + \frac{1}{2}(x_k)^tAx_k - (x_k)^tAx_0, \ where\, z = x - x_0 \in \mathcal{K}_{t,k} \\
&= \frac{1}{2}x^tAx - (x_k)^tAz + \frac{1}{2}(x_k)^tAx_k - (x_k)^tAx_0, \ since\, b^tz = (x_k)^tAz \\
&= \frac{1}{2}x^tAx - (x_k)^tAx + \frac{1}{2}(x_k)^tAx_k \\
&= \frac{1}{2}(x - x_k)^tA(x - x_k) \geqslant 0, \ since\ A\ is\ positive\ definite \quad \square
\end{aligned}$$

Theorem 4.1.9. *$\phi(x_k) = min\{\phi(x), \forall x \in x_0 + \mathcal{K}_{t,k}\}$ if and only if $||x^* - x_k||_A = min\{||x^* - x||_A, \forall x \in x_0 + \mathcal{K}_{t,k}\}$, where x^* is the exact solution of* (3.1).

Proof. $f(x) = ||x^* - x||_A = (x^*)^tAx^* - 2(x^*)^tAx + x^tAx = b^tx^* - 2b^tx + x^tAx = b^tx^* + 2\phi(x)$. The minimum of $f(x)$ is given by $f'(x) = \bigtriangledown\phi(x) = 0$. $\square$

4.1.3 Convergence analysis

The conjugate gradient method of Hestenes and Stiefel is known to converge in $\overline{K}$ iterations where $\overline{K} \leqslant n$, if the matrix $A \in \mathbb{R}^{n,n}$ is SPD. Moreover, the k^{th} error of CG $\overline{e}_k = ||x^* - \overline{x}_k|| \leqslant 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k ||\overline{e}_0||_A$ where $\kappa = ||A||_2||A^{-1}||_2 = \frac{\lambda_{max}}{\lambda_{min}}$ is the L2-condition number of the matrix A.

Assuming that the k^{th} residual of the new conjugate gradient methods $r_k \perp \mathcal{K}_{t,k}$, then by Theorem 4.1.8 and Theorem 4.1.9 we have that

$$\begin{aligned}
||e_k||_A = ||x^* - x_k||_A &= min\{||x^* - x||_A, \forall x \in x_0 + \mathcal{K}_{t,k}\} && (4.2) \\
&\leqslant min\{||x^* - \overline{x}||_A, \forall \overline{x} \in x_0 + \mathcal{K}_k\} \text{ since } \mathcal{K}_k \subset \mathcal{K}_{t,k} && (4.3) \\
&\leqslant ||\overline{e}_k||_A. && (4.4)
\end{aligned}$$

Therefore, our methods converge at least as fast as the Classical Conjugate Gradient method, assuming that the Petrov Galerkin condition is respected ($r_k \perp \mathcal{K}_{t,k}$). Hence, the enlarged Krylov subspace CG methods will converge in K iterations, where $K \leqslant \overline{K} \leqslant n$.

4.2 Multiple search direction with orthogonalization conjugate gradient (MSDO-CG) method

The MSD-CG method (section 3.2.4.2) introduced by Gu et al [44], can be viewed as an enlarged Krylov method where $P_0 = [T(r_0)]$, and at the k^{th} iteration $p_i^k = T_i(r_{k-1}) + \beta_i^k p_i^{k-1}$ for $i = 1, 2, ..., t$, $P_k = [p_1^k, p_2^k, ..., p_t^k]$, $x_k = x_{k-1} + P_k\alpha_k$ and $r_k = r_{k-1} - AP_k\alpha_k$ with

$\alpha_k = (P_k^t A P_k)^{-1} P_k^t r_{k-1}$ and $\beta_k = [\beta_1^k, \beta_2^k, ..., \beta_t^k] = (P_{k-1}^t A P_{k-1})^{-1} P_{k-1}^t A r_{k-1}$. However, the P_k's are not A-orthogonal implying that $r_k \not\perp \mathcal{K}_{t,k}$. Thus, MSD-CG is not a projection method.

The multiple search directions with orthogonalization CG (MSDO-CG) is an enlarged Krylov projection method that solves the system $Ax = b$, by approximating the solution at the k^{th} iteration with the vector $x_k = x_{k-1} + P_k \alpha_k$ such that

$$\phi(x_k) = min\{\phi(x), \forall x \in \mathcal{K}_{t,k}\},$$

where $P_k \alpha_k \in \mathcal{K}_{t,k}$, P_k is an $n \times t$ block vector containing the t subdomain search directions, and α_k is a vector of size t.

The minimum of $\phi(x)$ is given by $\bigtriangledown \phi(x) = 0$, which is equivalent to $Ax - b = 0$. Thus, by minimizing $\phi(x)$, we are solving the system $Ax = b$. Note that since $\phi(x_k) = min\{\phi(x), \forall x \in x_0 + \mathcal{K}_{t,k}\}$, then

$$\phi(x_k) = \phi(x_{k-1} + P_k \alpha_k) = min\{\phi(x_{k-1} + P_k \alpha), \forall \alpha \in \mathbb{R}^t\}. \tag{4.5}$$

Once x_k has been chosen, either x_k is the desired solution of $Ax = b$, or t new domain search direction vectors P_{k+1} and a new approximation $x_{k+1} = x_k + P_{k+1}\alpha_{k+1}$ are computed. Similarly to MSD-CG, $P_{k+1} = [p_1^{k+1}\ p_2^{k+1}\ ...\ p_t^{k+1}]$, where $p_i^1 = T_i(r_0)$ and $p_i^{k+1} = T_i(r_k) + \beta_i^{k+1} p_i^k$ for $i = 1, 2, ..., t$. But unlike MSD-CG, MSDO-CG is a projection method. Hence, we A-orthonormalize all the search directions, P_{k+1}, to ensure that $r_{k+1} \perp \mathcal{K}_{t,k+1}$ as discussed in section 4.2.2. By imposing the orthogonality condition, $r_{k+1} \perp \mathcal{K}_{t,k+1}$, it is guaranteed that MSDO-CG converges at least as fast as CG as proven in section 4.1.3.

This procedure is repeated until convergence. Thus, we need to find the recursion relations of r_k, P_k, α_k, and $\beta_k = [\beta_1^k, \beta_2^k, ..., \beta_t^k]^t$.

4.2.1 The residual r_k

By definition, the residual $r_k = b - Ax_k$, where $x_k \in \mathcal{K}_{t,k}$. Thus $r_k \in \mathcal{K}_{t,k+1}$. As for the recursion relation of r_k, we simply replace x_k by its expression and obtain the following:

$$\begin{aligned} r_k &= b - Ax_k \\ &= b - A(x_{k-1} + P_k \alpha_k) \\ &= r_{k-1} - AP_k \alpha_k \end{aligned}$$

Moreover, if the orthogonality condition, $r_k \perp \mathcal{K}_{t,k}$, is ensured, then $(r_k)^t r_i = 0$, for all $i < k$. Hence, the residuals form an orthogonal set.

Theorem 4.2.1. *The orthogonality condition $(r_k)^t y = 0$ for all $y \in \mathcal{K}_{t,k}$, implies the A-orthogonality of the block search directions $P_i^t A P_j = 0$, for all $i \neq j$, and $i, j \leqslant k$.*

Proof. By definition, $P_i \in \mathcal{K}_{t,i}$ and $\mathcal{K}_{t,i} \subset \mathcal{K}_{t,i+1}$. Thus $P_i \in \mathcal{K}_{t,i+c}$ for $c \geqslant 0$. By the Petrov-Galerkin condition $r_{k-1}^t P_i = 0$ for $i \leqslant k-1$ and $r_k^t P_i = 0$. Thus, $r_k^t P_i = r_{k-1}^t P_i - \alpha_k^t P_k^t A P_i = 0$ for $i \leqslant k-1$. This implies that $P_k^t A P_i = 0$ for $i \leqslant k-1$ since $\alpha_k \neq 0$ by definition. Therefore, the A-orthogonality of the search directions. □

4.2.2 The domain search direction P_k

By definition, the domain search direction is $P_k = [p_1^k \ p_2^k \ ... \ p_t^k]$ where $p_i^1 = T_i(r_0)$ and $p_i^k = T_i(r_{k-1}) + \beta_i^k p_i^{k-1}$ for $i = 1, 2, ..., t$. $p_i^k \in \mathcal{K}_{t,k}$ for $i = 1, 2, ..., t$ and $P_k \alpha_k \in \mathcal{K}_{t,k}$.

The recursion relation of P_k is defined as follows

$$P_k = T(r_{k-1}) + P_{k-1} diag(\beta_k), \tag{4.6}$$

where $diag(\beta_k)$ is a $t \times t$ matrix with the vector β_k on the diagonal.

The domain search directions defined in (4.6) are not A-orthogonal to each others. To ensure that the orthogonality condition is valid, at each iteration k the block vector P_k is A-orthonormalized against all the previous P_i, where $i = 1, 2, .., k-1$. Then the column vectors of P_k are A-orthonormalized against each others. Thus, the obtained search directions $\widetilde{P}_k$ satisfy $(\widetilde{P}_k)^t A \widetilde{P}_i = 0$ for all $i \neq k$. Moreover, $(\widetilde{P}_k)^t A \widetilde{P}_k = I$, where I is the identity matrix, assuming that the column vectors of P_k are linearly independent with respect to each others and the previous directions, or alternatively none of the column vectors of $\widetilde{P}_k$ is zero. Note that, once $P_k = T(r_{k-1}) + P_{k-1} diag(\beta_k)$ is defined, it is directly A-orthonormalized. Thus, in the sections that follow, we denote by P_k the A-orthonormalized search directions and we do not use the $\widetilde{P}_k$ notation to be consistent with the initial definitions in the previous sections.

There are several A-orthonormalization methods. First, for A-orthonormalizing P_k against all the previous P_i, where $i = 1, 2, .., k-1$, one can use classical Gram Schmidt (CGS), modified Gram Schmidt (MGS), or classical Gram Schmidt with reorthogonalization (CGS2) where we apply the CGS algorithm twice for numerical stability reasons. As for A-orthonormalizing P_k, there are many methods that are discussed in [59, 64], but we will only refer to CGS, CGS2, MGS, A-CholQR and Pre-CholeQR. We seek a combination of both A-orthonormalizations that is stable and parallelizable with reduced communication. For that reason, in section 4.4 we test the MSDO-CG method with the different combinations of the A-orthonormalization methods and we conclude that the MSDO-CG is numerically most stable when we use MGS, CGS2+A-CholQR, or CGS2+Pre-CholQR. In section 4.5.1 we discuss the parallelization of the MSDO-CG algorithm with the stable A-orthonormalization methods.

Note that in section 2.4, we discuss the A-orthonormalization using modified Gram Schmidt and classical Gram Schmidt. We also present versions of the algorithms that reduce communication along with their parallelizations. For example, Algorithm 13 is a block Gram Schmidt A-orthonormalization based on classical Gram Schmidt that A-orthonormalizes P_k against previous vectors. And Algorithm 16 A-orthonormalizes P_k's vectors against each others using a classical Gram Schmidt.

4.2.3 Finding the expression of α_{k+1} and β_{k+1}

At each iteration the step α_{k+1} is chosen such that

$$\phi(x_{k+1}) = min\{\phi(x_k + P_{k+1}\alpha), \forall \alpha \in \mathbb{R}^t\}$$

Let $F(\alpha) = \phi(x_k + P_{k+1}\alpha)$ where $\phi(x) = \frac{1}{2}x^tAx - x^tb$.

$$\begin{aligned}
\text{Then, } F(\alpha) &= \frac{1}{2}(x_k + P_{k+1}\alpha)^tA(x_k + P_{k+1}\alpha) - (x_k + P_{k+1}\alpha)^tb \\
&= \phi(x_k) + \frac{1}{2}[(x_k)^tAP_{k+1}\alpha + \alpha^t(P_{k+1})^tAx_k + \alpha^t(P_{k+1})^tAP_{k+1}\alpha] - \alpha^t(P_{k+1})^tb \\
&= \phi(x_k) + \frac{1}{2}[(x_k)^tAP_{k+1}\alpha - \alpha^t(P_{k+1})^tAx_k] + \frac{1}{2}\alpha^t(P_{k+1})^tAP_{k+1}\alpha - \alpha^t(P_{k+1})^tr_k \\
&= \phi(x_k) + \frac{1}{2}\alpha^t(P_{k+1})^tAP_{k+1}\alpha - \alpha^t(P_{k+1})^tr_k, \quad since \ \ A \ is \ SPD
\end{aligned}$$

The minimum of $F(\alpha)$ is given by $F'(\alpha) = 0$.

$$\Rightarrow F'(\alpha) = (P_{k+1})^tAP_{k+1}\alpha - (P_{k+1})^tr_k = 0$$

Therefore, $\boxed{\alpha_{k+1} = (P_{k+1}^tAP_{k+1})^{-1}(P_{k+1}^tr_k)}$.

As for β_{k+1}, it should be chosen to ensure that P_{k+1} is A-orthogonal to P_k. $P_{k+1} = T(r_k) + P_kdiag(\beta_{k+1})$ and $P_k^tAP_{k+1} = P_k^tAT(r_k) + P_k^tAP_kdiag(\beta_{k+1})$. Since P_k is an A-orthonormal matrix, $P_k^tAP_k = I$, $diag(\beta_{k+1})$ should be equal to $-P_k^tAT(r_k)$. But nothing guarantees that $P_k^tAT(r_k)$ is a diagonal matrix. So we choose $\boxed{\beta_{k+1} = (P_k^tAP_k)^{-1}P_k^tAr_k}$ which guarantees that $P_{k+1} * \mathbb{1}_t$ is A-orthogonal to P_k, similarly to MSD-CG. Moreover, in case $P_k^tAT(r_k)$ is a diagonal matrix, then our choice of β_{k+1} implies that P_{k+1} is A-orthogonal to P_k. If $t = 1$, then MSDO-CG is reduced to the classical conjugate gradient.

Note that, since the vectors of P_{k+1} are A-orthonormalized ($P_{k+1}^tAP_{k+1} = I$), then α_{k+1} and β_{k+1} systems are reduced to $\boxed{\alpha_{k+1} = P_{k+1}^tr_k}$ and $\boxed{\beta_{k+1} = -P_k^tAr_k}$.

4.2.4 The MSDO-CG algorithm

After deriving the recurrence relations of x_k, r_k, P_k, α_k, and β_k, we present the MSDO-CG algorithm in Algorithm 35. We do not specify the A-orthonormalization methods, since this choice will be based first on the numerical stability of the method (section 4.4), then on its parallelization with the least communication possible (section 4.5.1).

Thus, we present the MSDO-CG algorithm (Algorithm 35) and the computed flops per iteration except for the A-orthonormalizations. To reduce communication and computation in the A-orthonormalizations, be it MGS (Algorithms 10 and 11), CGS (Algorithms 14 and 17), A-CholQR

(Algorithm 21), or Pre-CholQR (Algorithm 23), we replace $W_{k+1} = AP_{k+1}$ by

$$\left\{ \begin{array}{rcl} W_1 & = & AP_1 \\ W_{k+1} & = & AT(r_k) + AP_k diag(\beta) \quad \forall k \geqslant 1 \\ & = & AT(r_k) + W_k diag(\beta) \end{array} \right.$$

and update it accordingly in the A-orthonormalization algorithms, as discussed in section 2.4.

Algorithm 35 MSDO-CG algorithm — Flops

Input: A, the $n \times n$ symmetric positive definite matrix
Input: b, the $n \times 1$ right-hand side; x_0, the initial guess or iterate
Input: ϵ, the stopping tolerance; k_{max}, the maximum allowed iterations
Output: x_k, the approximate solution of the system $Ax = b$

1:	$r = b - Ax_0$, $\rho = \|\|r\|\|_2^2$, $k = 1$	$2\text{nnz} + 2n - 1$
2:	Let $P_1 = T(r_0)$ and $W_1 = AP_1$	$2\text{nnz} - (t-1)n$
3:	A-orthonormalize P_1	not included here
4:	**while** ($\sqrt{\rho} > \epsilon \|\|b\|\|_2$ and $k < k_{max}$) **do**	$2n$
5:	$\alpha = (P_k^t W_k)^{-1}(P_k^t r) = P_k^t r$	$(2n-1)t$
6:	$x = x + P_k \alpha$	$(2t-1)n + n$
7:	$r = r - W_k \alpha$	$(2t-1)n + n$
8:	$\rho = \|\|r\|\|_2^2$	$2n - 1$
9:	$\beta = -(P_k^t W_k)^{-1}(W_k^t r) = -W_k^t r$	$(2n-1)t$
10:	$P_{k+1} = T(r) + P_k diag(\beta)$	$2nt$
11:	$W_{k+1} = AT(r_k) + W_k diag(\beta)$	$2\text{nnz} - (t-1)n + 2nt$
12:	A-orthonormalize P_{k+1} against all P_i's for $i \leqslant k$ and update W_{k+1}	not included here
13:	A-orthonormalize P_{k+1} and update W_{k+1}	not included here
14:	$k = k + 1$	1
15:	**end while**	

The total number of flops computed sequentially after k_c iterations, except for the A-orthonormalizations, is

$$\begin{array}{rcl} \text{Total Flops} & = & 4\text{nnz} - nt + 5n - 1 + k_c[11nt - 2t + 2n - 1 + 2\text{nnz} + n + 1] \\ & = & 4\text{nnz} - nt + 5n - 1 + k_c[11nt + 3n - 2t + 2\text{nnz}] \\ & \approx & 4\text{nnz} + 5n + k_c[11nt + 2\text{nnz}], \end{array}$$

which is of the order of $\text{nnz}k_c + ntk_c$ flops, where nnz is the number of nonzero entries in the $n \times n$ matrix A and t is the number of search directions computed at each iteration.

It must be noted that since the P_i's are A-orthonormal to each others, then the $t \times t$ matrix $P_k^t W_k = P_k^t A P_k$ is the identity matrix. Hence, solving for α_k and β_k is simply performing matrix vector multiplication.

4.3 Long recurrence enlarged conjugate gradient (LRE-CG) method

In this section, we introduce the long recurrence enlarged conjugate gradient (LRE-CG) method which is an enlarged Krylov projection method that solves the system $Ax = b$ by approximating the solution at the k^{th} iteration with the vector $x_k = x_{k-1} + Q_k\alpha_k \in x_0 + \mathcal{K}_{t,k}$ such that

$$\phi(x_k) = min\{\phi(x), \forall x \in x_0 + \mathcal{K}_{t,k}\},$$

where $Q_k\alpha_k \in \mathcal{K}_{t,k}$ and Q_k is an $n \times tk$ matrix containing the orthonormal basis vectors of $\mathcal{K}_{t,k}$ and $\phi(x) = \frac{1}{2}x^tAx - x^tb$. The LRE-CG method does not have short recurrences as MSDO-CG, but it has similarities with GMRES in that the whole basis is used to define the new approximate solution rather than t search directions. As mentioned earlier, the minimum of $\phi(x)$ is given by $\bigtriangledown\phi(x) = 0$ which is equivalent to $Ax - b = 0$. Thus, by minimizing $\phi(x)$ we are solving the system $Ax = b$. Since $\phi(x_k) = min\{\phi(x), \forall x \in x_0 + \mathcal{K}_{t,k}\}$, then

$$\phi(x_k) = \phi(x_{k-1} + Q_k\alpha_k) = min\{\phi(x_{k-1} + Q_k\alpha), \forall\alpha \in \mathbb{R}^{tk}\}. \tag{4.7}$$

Once x_k has been chosen, either x_k is the exact solution of $Ax = b$, or t new basis vectors and the new approximation $x_{k+1} = x_k + Q_{k+1}\alpha_{k+1}$ are computed. This procedure is repeated until convergence.

Thus, we need to find the recursion relations of r_k and α_k. By definition, the residual $r_k = b - Ax_k$ where $x_k \in x_0 + \mathcal{K}_{t,k}$. Thus $r_k \in \mathcal{K}_{t,k+1}$. The recursion relation of r_k can be simply obtained by replacing x_k by its expression as follows

$$\begin{aligned} r_k &= b - Ax_k \\ &= b - A(x_{k-1} + Q_k\alpha_k) \\ &= r_{k-1} - AQ_k\alpha_k. \end{aligned}$$

At each iteration the step α_{k+1} is chosen such that

$$\phi(x_{k+1}) = min\{\phi(x_k + Q_{k+1}\alpha), \forall\alpha \in \mathbb{R}^t(k+1)\}.$$

Let $F(\alpha) = \phi(x_k + Q_{k+1}\alpha)$ where $\phi(x) = \frac{1}{2}x^tAx - x^tb$. Then,

$$\begin{aligned} F(\alpha) &= \frac{1}{2}(x_k + Q_{k+1}\alpha)^tA(x_k + Q_{k+1}\alpha) - (x_k + Q_{k+1}\alpha)^tb \\ &= \phi(x_k) + \frac{1}{2}[(x_k)^tAQ_{k+1}\alpha + \alpha^t(Q_{k+1})^tAx_k + \alpha^t(Q_{k+1})^tAQ_{k+1}\alpha] - \alpha^t(Q_{k+1})^tb \\ &= \phi(x_k) + \frac{1}{2}[(x_k)^tAQ_{k+1}\alpha - \alpha^t(Q_{k+1})^tAx_k] + \frac{1}{2}\alpha^t(Q_{k+1})^tAQ_{k+1}\alpha - \alpha^t(Q_{k+1})^tr_k \\ &= \phi(x_k) + \frac{1}{2}\alpha^t(Q_{k+1})^tAQ_{k+1}\alpha - \alpha^t(Q_{k+1})^tr_k, \quad since\ \ A\ is\ SPD. \end{aligned}$$

The minimum of $F(\alpha)$ is given by $F'(\alpha) = 0$

$$\Rightarrow F'(\alpha) = (Q_{k+1})^t A Q_{k+1}\alpha - (Q_{k+1})^t r_k = 0.$$

Therefore, $\boxed{\alpha_{k+1} = (Q_{k+1}^t A Q_{k+1})^{-1}(Q_{k+1}^t r_k)}$.

By minimizing $\phi(x)$, the Petrov-Galerkin condition, $r_k \perp \mathcal{K}_{t,k}$, is ensured (Theorem 4.3.1). Therefore, $(r_k)^t r_i = 0$, for all $i < k$, and the residuals form an orthogonal set.

Theorem 4.3.1. *The Petrov-Galerkin condition in LRE-CG, $r_k \perp \mathcal{K}_{t,k}$, is equivalent to x_k being the minimum of $\phi(x)$ in $x_0 + \mathcal{K}_{t,k}$.*

Proof.

1. x_k is the minimum of $\phi(x)$ in $x_0 + \mathcal{K}_{t,k}$ implies $r_k \perp \mathcal{K}_{t,k}$
   The minimum of $F(\alpha) = \phi(x_k) = \phi(x_{k-1} + Q_k\alpha)$ is given by
   $F'(\alpha) = (Q_k)^t A Q_k \alpha - (Q_k)^t r_{k-1} = 0$. Since x_k is the minimum, then $\alpha = \alpha_k$ and $F'(\alpha) = -Q_k^t r_k = 0$. Thus $r_k \perp \mathcal{K}_{t,k}$.

2. $r_k \perp \mathcal{K}_{t,k}$ implies x_k is the minimum of $\phi(x)$ in $x_0 + \mathcal{K}_{t,k}$ (Proof by contradiction)
   Assume that $r_k \perp \mathcal{K}_{t,k}$ and x_k is not the minimum of $\phi(x)$ in $x_0 + \mathcal{K}_{t,k}$. Then $F'(\alpha_k) \neq 0$. Hence $Q_k^t r_k \neq 0$ and r_k is not orthogonal to $\mathcal{K}_{t,k}$. This contradicts our assumption. Thus x_k is the minimum of $\phi(x)$.

□

4.3.1 The LRE-CG algorithm

After deriving the expressions and the recursion relations of

$$\begin{aligned} x_k &= x_{k-1} + Q_k\alpha_k, \\ r_k &= r_{k-1} - AQ_k\alpha_k, \\ \alpha_k &= (Q_k^t A Q_k)^{-1}(Q_k^t r_{k-1}), \end{aligned}$$

we present in Algorithm (36) the LRE-CG algorithm and the performed flops, except for the orthonormalization. We refer to the cost of solving the $tk \times tk$ linear system from step 5 in Algorithm 36 as $Solve_\alpha(tk)$.

Algorithm 36 The LRE-CG algorithm — Flops

Input: A, the $n \times n$ symmetric positive definite matrix
Input: b, the $n \times 1$ right-hand side; x_0, the initial guess or iterate
Input: ϵ, the stopping tolerance; k_{max}, the maximum allowed iterations
Output: x_k, the approximate solution of the system $Ax = b$

1:	$r_0 = b - Ax_0$, $\rho_0 = \|\|r_0\|\|_2^2$, k = 1	$2\text{nnz} + 2n - 1$
2:	Let $W = T(r_0)$, normalize its vectors and then let $Q = W$	$3n$
3:	**while** ($\sqrt{\rho_{k-1}} > \epsilon\|\|b\|\|_2$ and $k < k_{max}$) **do**	$2n$
4:	$G = (Q^tAQ)$	$(2\text{nnz} - n)tk + (2n - 1)t^2k^2$
5:	$\alpha = G^{-1}(Q^tr)$	$(2n-1)tk + Solve_\alpha(tk)$
6:	$x = x + Q\alpha$	$2tkn$
7:	$r = r - AQ\alpha$	$2tkn$
8:	$\rho_k = \|\|r\|\|_2^2$	$2n - 1$
9:	Let $W = AW$	$(2\text{nnz} - n)t$
10:	Orthonormalize the vectors of W against the vectors of Q	not included in here
11:	Orthonormalize the vectors of W and let $Q = [Q\ W]$	not included in here
12:	k = k+1	1
13:	**end while**	

The cost of the LRE-CG, using Algorithm 36, except for the orthonormalization in steps 10 and 11, is

$$\begin{aligned}
\text{Total Flops} &= 2\text{nnz} + 7n - 1 + k_c\big[(2\text{nnz} + 5n - 1)t\tfrac{k_c+1}{2} + 2n + 2\text{nnz}t - nt\big] \\
&\quad + \textstyle\sum_{k=1}^{k_c} Solve_\alpha(tk) + (2n-1)\tfrac{t^2}{6}(k_c+1)(2k_c+1) \\
&= 2\text{nnz} + 7n - 1 + k_c\big[(2\text{nnz} + 5n - 1)t\tfrac{k_c+1}{2} + 2n + 2\text{nnz}t - nt\big] + \textstyle\sum_{k=1}^{k_c} Solve_\alpha(tk) \\
&\quad + (2n-1)\tfrac{t^2}{6}(2k_c^2 + 3k_c + 1) \\
&= 2\text{nnz} + 7n - 1 + (2n-1)\tfrac{t^2}{6} + k_c\big[(2\text{nnz} + 5n - 1)t\tfrac{k_c+1}{2} + 2n + 2\text{nnz}t - nt \\
&\quad + (2n-1)\tfrac{t^2}{6}(2k_c + 3)\big] + \textstyle\sum_{k=1}^{k_c} Solve_\alpha(tk) \\
&\approx \text{nnz}tk_c^2 + nt^2k_c^2 + \textstyle\sum_{k=1}^{k_c} Solve_\alpha(tk),
\end{aligned}$$

where the first term $\text{nnz}tk_c^2$ corresponds to the multiplication AQ and the second term $nt^2k_c^2$ corresponds to the orthonormalization with respect to previous vectors. As for the memory requirements, we have to store the matrix A and $tk_c + 2$ vectors of size $n \times 1$. And there should be enough memory for the $tk_c \times tk_c$ matrix Q^tAQ.

However, the multiplication $Q_k^tAQ_k$ can be reduced since $Q_k = [Q_{k-1}W_k]$. Let $Z_k = AW_k$, then $D_k = AQ_k = [AQ_{k-1}Z_k]$. At iteration $k-1$, $D_{k-1} = AQ_{k-1}$ is computed. Thus at iteration

k, only $Z_k = AW_k$ is computed. As for $G_k = Q_k^t A Q_k$, it is equal to

$$G_k = Q_k^t A Q_k = Q_k^t D_k = \begin{pmatrix} Q_{k-1}^t D_{k-1} & Q_{k-1}^t Z_k \\ W_k^t D_{k-1} & W_k^t Z_k \end{pmatrix} = \begin{pmatrix} G_{k-1} & Q_{k-1}^t Z_k \\ Z_k^t Q_{k-1} & W_k^t Z_k \end{pmatrix} = \begin{pmatrix} G_{k-1} & F_k \\ F_k^t & E_k \end{pmatrix},$$

where G_{k-1} is computed at iteration $k-1$, $F_k = Q_{k-1}^t Z_k$, and $E_k = W_k^t Z_k$. Thus computing $G_k = Q_k^t A Q_k$ can be reduced to computing F_k and E_k.

Algorithm 37 The LRE-CG Algorithm — Flops

Input: A, the $n \times n$ symmetric positive definite matrix
Input: b, the $n \times 1$ right-hand side; x_0, the initial guess or iterate
Input: ϵ, the stopping tolerance; k_{max}, the maximum allowed iterations
Output: x_k, the approximate solution of the system $Ax = b$

1:	$r_0 = b - Ax_0$, $\rho_0 = \|\|r_0\|\|_2^2$, k = 1	2nnz $+ 2n - 1$
2:	Let $W = T(r_0)$, normalize its vectors and then let $Q = W$	$3n$
3:	**while** ($\sqrt{\rho_{k-1}} > \epsilon \|\|b\|\|_2$ and $k < k_{max}$) **do**	$2n$
4:	$Z = AW$	$(2\text{nnz} - n)t$
5:	$E = W^t Z$	$(2n-1)t^2$
6:	**if** $k == 1$ **then**	
7:	$D = Z$ and $G = E$	
8:	**else**	
9:	$F = Q(:, 1 : t(k-1))^t Z$	$(2n-1)t^2(k-1)$
10:	$G = \begin{pmatrix} G & F \\ F^t & E \end{pmatrix}$ and $D = [D \;\; Z]$	
11:	**end if**	
12:	$\alpha = G^{-1}(Q^t r)$	$(2n-1)tk + Solve_\alpha(tk)$
13:	$x = x + Q\alpha$	$2tkn$
14:	$r = r - D\alpha$	$2tkn$
15:	$\rho_k = \|\|r\|\|_2^2$	$2n - 1$
16:	Let $W = Z$	
17:	Orthonormalize the columns of W against those of Q	not included in here
18:	Orthonormalize the vectors of W and let $Q = [Q \; W]$	not included in here
19:	k = k+1	1
20:	**end while**	

Then, the cost of k_c iterations of LRE-CG using Algorithm 37, except for the orthonormaliza-

tion, is

$$\begin{aligned}\text{Total Flops} &= 2\text{nnz} + 7n - 1 + k_c[2\text{nnz}t - nt + (6n-1)t\tfrac{k_c+1}{2} + (2n-1)t^2\tfrac{k_c+1}{2} + 2n] \\ &\quad + \textstyle\sum_{k=1}^{k_c} Solve_\alpha(tk) \\ &= 2\text{nnz} + 7n - 1 + k_c[(2\text{nnz} + 2n - \tfrac{1}{2})t + (n-\tfrac{1}{2})t^2 + (6n-1)t\tfrac{k_c}{2} + (2n-1)t^2\tfrac{k_c}{2} \\ &\quad + 2n] + \textstyle\sum_{k=1}^{k_c} Solve_\alpha(tk) \\ &\approx 3ntk_c^2 + nt^2k_c^2 + nt^2k_c + \textstyle\sum_{k=1}^{k_c} Solve_\alpha(tk)\end{aligned}$$

Note that tk_c should be much smaller than n, or otherwise the cost of Algorithm 37 would be $O(n^3 + Solve_\alpha(n))$, and n vectors of size n have to be stored in addition to the $n \times n$ system $Q_{k_c}^t A Q_{k_c}$.

One remedy to this problem, that we do not address in this thesis, is to restart LRE-CG after some iterations. But this restart might have an effect on convergence as in restarted GMRES. Another alternative is to choose a linearly independent subset of the t computed vectors at each iteration i. This reduces the size of the system solved at each iteration. A third alternative is to compute at each iteration i, t_i vectors and then choose a linearly independent set of cardinality $\widehat{t}_i$, where $t_0 = t$, $t_i \leqslant t$, $\widehat{t}_i \leqslant t_i$, and $\widehat{t}_i = t_{i+1}$. This reduces not only the size of the system solved at each iteration, but also the memory requirements and the number of computed vectors per iteration. In exact precision, the second and third alternatives are equivalent by Theorem 4.1.4, since if a vector $T_j(r_0)$ is linearly dependent on $\{T_1(r_0), .., T_{j-1}(r_0), T_{j+1}(r_0), ..., T_t(r_0)\}$ then $AT_j(r_0)$ is linearly dependent on $\{AT_1(r_0), .., AT_{j-1}(r_0), AT_{j+1}(r_0), ..., AT_t(r_0)\}$. However, this has to be tested in finite precision. Note that there is an additional cost for choosing a linearly independent subset of the t or t_i vectors.

The $tk \times tk$ α system can be solved using iterative methods like Jacobi method or Krylov subspace methods. Moreover, the s-step or communication avoiding Krylov subspace methods can be used to reduce communication. We use matlab's backslash to solve the α systems in the convergence tests that follow.

4.4 Convergence results

After introducing the new CG methods, MSDO-CG and LRE-CG, we compare their convergence behavior with respect to different A-orthonormalization and orthonormalization schemes respectively, on several matrices for different number of partitions (2, 4, 8, 16, 32 and 64 partitions). Then we compare the convergence behavior of both methods with repect to CG, Coop-CG, and MSD-CG. Recall that coop-CG (section 3.2.4.1) solves the system $Ax = b$ t times in parallel by starting with t distinct initial guesses. The matrices are first reordered using Metis's k-way partitioning [49] that defines the subdomains δ_i. Then x is chosen randomly using matlab's rand function and $b = A * x$, except for the ELASTICITY3Dmatrix where A and b are available. In tables 4.1, 4.2, and 4.3, "Iter" is the number of iterations k_c needed for convergence, "Err" is the relative error $\frac{||x - x_{k_c}||_2}{||x||_2}$

at convergence, and Pa is the number of partitions for all the methods except for coop-CG where it refers to the number of initial guesses.

Table 4.1: Comparison of the convergence of MSDO-CG with different A-orthonormalization schemes, with respect to number of partitions (Pa) with $x_0 = 0$.

		MSDO-CG with different A-Orthonormalization Methods									
		MGS		CGS+A-CholQR		CGS+Pre-CholQR		CGS2+A-CholQR		CGS2+Pre-CholQR	
	Pa	Iter	Err	Iter	Err	Iter	Err	Iter	Err	Iter	Err
POISSON2D $tol = 10^{-6}$	2	200	4E-5	204	3E-5	204	3E-5	204	3E-5	204	3E-5
	4	167	2E-5	167	2E-5	167	2E-5	167	2E-5	167	2E-5
	8	139	1E-5	139	1E-5	139	1E-5	139	1E-5	139	1E-5
	16	121	5E-6	121	5E-6	121	5E-6	121	5E-6	121	5E-6
	32	94	2E-6	94	2E-6	94	2E-6	94	2E-6	94	2E-6
	64	69	2E-6	69	2E-6	69	2E-6	69	2E-6	69	2E-6
NH2D $tol = 10^{-8}$	2	256	1E-7	256	1E-7	256	1E-7	256	1E-7	256	1E-7
	4	208	1E-7	208	1E-7	208	1E-7	208	1E-7	208	1E-7
	8	169	8E-8	169	8E-8	169	8E-8	169	8E-8	169	8E-8
	16	138	6E-8	138	6E-8	138	6E-8	138	6E-8	138	6E-8
	32	107	2E-8	107	2E-8	107	2E-8	107	2E-8	107	2E-8
	64	77	1E-8	77	1E-8	77	1E-8	77	1E-8	77	1E-8
SKY2D1 $tol = 10^{-8}$	2	1559	8E-4	–	–	–	–	1562	8E-4	1559	9E-4
	4	917	4E-4	–	–	–	–	917	4E-4	917	4E-4
	8	532	3E-4	–	–	–	–	531	2E-4	534	2E-4
	16	307	1E-4	–	–	–	–	307	1E-4	307	1E-4
	32	178	6E-5	–	–	–	–	178	6E-5	178	6E-5
	64	126	3E-6	–	–	–	–	124	2E-6	124	2E-6
SKY3D1 $tol = 10^{-8}$	2	610	4E-5	611	4E-5	611	4E-5	611	4E-5	638	1E-5
	4	420	2E-5	–	–	–	–	424	1E-5	418	2E-5
	8	228	1E-5	–	–	–	–	230	1E-5	228	2E-5
	16	134	1E-5	–	–	–	–	134	1E-5	134	1E-5
	32	87	1E-6	–	–	–	–	83	1E-5	83	1E-5
	64	53	6E-6	–	–	–	–	51	1E-5	51	1E-5
ANI3D $tol = 10^{-8}$	2	893	6e-5	893	6e-5	893	6e-5	893	6e-5	893	6e-5
	4	749	8e-5	749	8e-5	749	8e-5	749	8e-5	749	8e-5
	8	498	8e-5	506	9e-5	511	8e-5	498	7e-5	503	7e-5
	16	328	1e-4	–	–	–	–	326	1e-4	326	1e-4
	32	192	2e-4	–	–	–	–	192	1e-4	192	1e-4
	64	122	5e-5	–	–	–	–	122*	4e-5	122*	4e-5

In table 4.1 we compare the convergence behvior of the MSDO-CG method (Algorithm 35) with different A-orthonormalization schemes for A-orthonormalizing P_k against previous P_i's

(MGS, CGS, CGS2) and then A-orthonormalizing P_k against itself (MGS, CGS, CGS2, A-CholQR, Pre-CholQR) and for different number of partitions $t = 2, 4, 8, 16, 32, 64$ that correspond to the maximum number of vectors added at each iteration to the enlarged Krylov subspace, $\mathcal{K}_{t,k}$. We have tested different combinations of A-orthonormalization, but we only show MGS (MGS+MGS), CGS+A-CholQR, CGS+Pre-CholQR, CGS2+A-CholQR, and CGS2+Pre-CholQR. Note that MSDO-CG with CGS A-orthonormalization (CGS+CGS) did not converge neither with CGS2 A-orthonormalization (CGS2+CGS2) nor with CGS2+CGS or CGS+CGS2 A-orthonormalization. The reason is that the search directions are not A-orthogonal to satisfactory precision. And by Theorem 4.2.1, this implies that $r_k \not\perp \mathcal{K}_{t,k}$. Thus, nothing guarantees convergence since we have shown in section 4.1.3 that MSDO-CG converges faster than CG if $r_k \perp \mathcal{K}_{t,k}$. Moreover, we did not test combinations of MGS and QR factorizations since MGS is expensive in terms of communication compared to the other methods (section 4.5.1). But we tested MSDO-CG with MGS for comparison purposes since MGS is known for its numerical stability.

As shown in table 4.1, MSDO-CG with MGS A-orthonormalization converges for all the tested matrices and as we increase t, the number of iterations needed for convergence decreases. As we mentioned earlier, MSDO-CG with CGS A-orthonormalization did not converge. Therefore, we replaced CGS with CGS+A-CholQR and with CGS+Pre-CholQR A-orthonormalization. We notice that MSDO-CG with CGS+A-CholQR A-orthonormalization and MSDO-CG with CGS+Pre-CholQR A-orthonormalization have the same convergence behavior. For the matrices POISSON2Dand NH2D, both methods converge with the same number of iterations as MSDO-CG with MGS A-orthonormalization. However, for the matrix SKY2D1, both methods did not converge. As for the matrices SKY3D1and ANI3D, both methods converged only for $t = 2$ partitions, and $t = 2, 4, 8$ partitions respectively. The reason for this difference in behavior for different matrices is the condition number ($cond_2 = ||A||_2||A^{-1}||_2$). The condition number of the matrices POISSON2Dand NH2Dis 6×10^3, whereas that of the matrices SKY3D1, ANI3Dand SKY2D1is 1×10^6 , 2×10^6, and 3×10^7 respetively. Although it was shown in [59] that Pre-CholQR A-orthonormalization is more stable than A-CholQR, however MSDO-CG with CGS+A-CholQR A-orthonormalization and MSDO-CG with CGS+Pre-CholQR A-orthonormalization are both numerically unstable.

Thus, we replace CGS with CGS2 where the A-orthonormalization is performed twice for numerical stability. Then, the MSDO-CG with CGS2+A-CholQR A-orthonormalization and MSDO-CG with CGS2+Pre-CholQR A-orthonormalization converge as fast as MSDO-CG with MGS A-orthonormalization for all t and all the tested matrices. Hence, we concude that MGS, CGS2+A-CholQR, and CGS2+Pre-CholQR A-orthonormalizations are stable enough to be used in the MSDO-CG method (Algorithm 35).

In table 4.2, we compare the convergence behavior of the LRE-CG method (Algorithm 37) with different orthonormalization schemes for orthonormalizing W against the $n \times tk$ matrix Q (MGS, CGS) and then orthonormalizing W against itself (MGS, CGS, TSQR) and for different number of partitions $t = 2, 4, 8, 16, 32, 64$ that correspond to the maximum number of vectors added at each iteration to the enlarged Krylov subspace, $\mathcal{K}_{t,k}$. We start by testing the convergence of LRE-CG with MGS (MGS+MGS) orthonormalization. It converges for all the tested matrices since it is

Table 4.2: Comparison of the convergence of the LRE-CG method with different orthonormalization schemes, with respect to number of partitions Pa, with $x_0 = 0$.

		LRE-CG with different Orthonormalization Methods					
		MGS+MGS		CGS+CGS		CGS+TSQR	
	Pa	Iter	Err	Iter	Err	Iter	Err
POISSON2D $tol = 10^{-6}$	2	193	2E-5	193	2E-5	193	2E-5
	4	153	1E-5	153	1E-5	153	1E-5
	8	123	8E-6	123	8E-6	123	8E-6
	16	95	4E-6	95	4E-6	95	4E-6
	32	70	2E-6	70	2E-6	70	2E-6
	64	52	1E-6	52	1E-6	52	1E-6
NH2D $tol = 10^{-8}$	2	245	1E-7	245	1E-7	245	1E-7
	4	188	1E-7	188	1E-7	188	1E-7
	8	149	5E-8	149	5E-8	149	5E-8
	16	112	3E-8	112	3E-8	112	3E-8
	32	82	2E-8	82	2E-8	82	2E-8
	64	60	1E-8	60	1E-8	60	1E-8
SKY2D1 $tol = 10^{-8}$	2	1415	5E-04	1415	8E-4	1415	5E-04
	4	757	1E-4	(140)	–	754	1E-4
	8	398	1E-4	(112)	–	398	1E-4
	16	220	9E-5	(70)	–	220	1E-4
	32	126	5E-5	(51)	–	126	5E-5
	64	75	3E-5	(29)	–	75	4E-5
SKY3D1 $tol = 10^{-8}$	2	557	2E-5	570	1E-5	563	1E-5
	4	373	2E-5	(140)	–	377	1E-5
	8	211	1E-5	(54)	–	211	1E-5
	16	119	9E-6	(37)	–	119	9E-6
	32	69	9E-6	(18)	–	69	9E-6
	64	43	4E-6	(15)	–	42	1E-5
ANI3D $tol = 10^{-8}$	2	875	7e-5	875	7E-5	875	7e-5
	4	673	8e-5	(185)	–	673	8e-5
	8	449	1e-4	(116)	–	449	1e-4
	16	253	2e-4	(16)	–	253	2e-4
	32	148	2e-4	(9)	–	148	2e-4
	64	92	1e-4	(13)	–	92	1e-4

numerically stable, and the number of iterations needed for convergence decreases when increasing the number of partitions t. However, as mentioned in section 4.5.1, MGS is expensive in terms of communication ($O(tklog(t))$ messages per iteration, where t processors A-orthonormalized t vectors against tk vectors). Thus, we tested the LRE-CG method with Classical Gram Schmidt (CGS) orthogonalization which requires sending $O(tlog(t))$ messages per iteration. The LRE-CG with CGS converges in the same number of iterations as LRE-CG with MGS for the matrices POISSON2Dand NH2D. However, for the other matrices, it does not converge for the given stopping criteria except for $t = 2$ as shown in table 4.2. The reason is that the the matrix $C = Q^tAQ$ is becoming close to singular, with $rank(C) < tk$, as the iterations proceed due to the loss of orthogonality in the CGS orthogonalization. The number of iterations in parentheses in table 4.2 is not the number of iterations for convergence but it denotes the iteration at which the C matrix becomes close to singular.

In CA-GMRES [60], the authors use a parallelizable tall and skinny QR (TSQR) factorization [21] for orthonormalizing the $n \times t$ tall and skinny matrix instead of CGS. They have shown that the combination of CGS for orthonormalizing W against Q and TSQR for orthonormalizing W is stable. We have tested LRE-CG with CGS and TSQR (CGS+TSQR) orthonormalization, and it has the same convergence behavior as LRE-CG with MGS (MGS+MGS) orthonormalization (table 4.2). Thus, we conclude that MGS, and CGS+TSQR orthonormalizations are stable enough to be used in the LRE-CG method from Algorithm 37.

In tables 4.3 and 4.4, we compare the convergence behavior of MSDO-CG with MGS A-orthonormalization, LRE-CG with MGS orthonormalization, Coop-CG and MSD-CG with respect to CG for several matrices with different number of partitions $t = 2, 4, 8, 16, 32, 64$. The MSDO-CG, COOP-CG and LRE-CG have better convergence than CG, and LRE-CG has the best convergence. The MSD-CG converges, but requires more iterations than CG, three times more iterations for the matrices SKY2D1, SKY3D1, ANI3D, and Electricity3D. As for Coop-CG, which starts with t different initial guesses and solves two systems of fixed size $t \times t$, its convergence is slightly better than MSDO-CG for the matrices POISSON2D, NH2D, and Electricity3D. But it requires much more iterations than both MSDO-CG and LRE-CG for the other matrices (SKY2D1, SKY3D1, ANI3D). Moreover, the results may vary depending on the t initial guesses that are used for the different matrices.

For the tested matrices, LRE-CG has slightly better convergence than MSDO-CG, since it uses the whole basis to define the new approximate solution rather than t search directions. For the matrices POISSON2Dand NH2D, LRE-CG and MSDO-CG have almost the same convergence as CG for $t = 2$, and then as t is doubled the iterations needed for convergence is decreased by 20% to 30%. For $t = 2$, LRE-CG requires 35% and 40% less iterations than CG for the matrices ELASTICITY3Dand SKY3D1respectively. And as t is doubled the number of iterations needed for convergence is decreased by 25% to 30%, and 32% to 45% respectively. For $t = 2$, LRE-CG requires 60% and 80% less iterations than CG for the matrices SKY2D1and ANI3Drespectively. And as t is doubled, the number of iterations needed for convergence is decreased by 45% to 50% and 25% to 40% respectively.

Table 4.3: Comparison between the convergence of the different CG versions with respect to number of partitions or initial guesses for Coop-CG with $x_0 = 0$.

		CG		Coop-CG		MSD-CG		MSDO-CG		LRE-CG	
	Pa	Iter	Err	Iter	Err	Iter	Err	Iter	Err	Iter	Err
poisson(100,100) $tol = 10^{-6}$	2	195	2E-5	206	2E-7	235	3E-1	200	4E-5	193	2E-5
	4	195	2E-5	171	1E-7	252	7E-1	167	2E-5	153	1E-5
	8	195	2E-5	137	1E-7	245	7E-1	139	1E-5	123	8E-6
	16	195	2E-5	106	3E-8	249	7E-1	121	5E-6	95	4E-6
	32	195	2E-5	80	1E-8	240	7E-1	94	2E-6	70	2E-6
	64	195	2E-5	59	1E-8	253	7E-1	69	2E-6	52	1E-6
matvf2dnh100100 $tol = 10^{-8}$	2	259	4E-7	206	2E-7	363	3E-1	256	1E-7	245	1E-7
	4	259	4E-7	179	1E-7	343	7E-1	208	1E-7	188	1E-7
	8	259	4E-7	157	2.02E-5	372	7E-1	169	8E-8	149	5E-8
	16	259	4E-7	107	2E-8	373	7E-1	138	6E-8	112	3E-8
	32	259	4E-7	81	2E-8	324	7E-1	107	2E-8	82	2E-8
	64	259	4E-7	59	1E-8	457	7E-1	77	1E-8	60	1E-8
sky100100 $tol = 10^{-8}$	2	5951	4E-4	4893	2E-4	17907	3E-1	1559	8E-4	1415	5E-04
	4	5951	4E-4	3737	9E-5	66979	7E-1	917	4E-4	757	1E-4
	8	5951	4E-4	3391	1E-5	25298	7E-1	532	3E-4	398	1E-4
	16	5951	4E-4	2437	9E-6	23486	7E-1	307	1E-4	220	9E-5
	32	5951	4E-4	1406	4E-6	15448	7E-1	178	6E-5	126	5E-5
	64	5951	4E-4	802	2E-6	23981	7E-1	126	3E-6	75	3E-5

4.5 Parallel model and expected performance

In this section we describe the parallelization of the MSDO-CG method (section 4.5.1) and the LRE-CG method (section 4.5.2) with computed flops, number of messages and words sent and the estimated parallel runtime.

For simplicity, we assume that the algorithms are executed on a distributed memory machine formed by t processors, where t corresponds to the number of vectors computed at each iteration. Recall that on a distributed-memory architecture, the estimated runtime of an algorithm with a total of z computed flops and s sent messages each of size k is $\gamma_c z + \alpha_c s + \beta_c$, where γ_c is the inverse floating-point rate (seconds per floating-point operation), α_c is the latency (with units of seconds) and β_c is the inverse bandwidth (seconds per word).

We partition the graph of A into t subdomains using k-way partitioning or another graph partitioning. We denote by δ_i, for $i = 1, 2, .., t$ the subsets of indices obtained from the partitioning. That is $\delta_i \cap \delta_h = \phi$ for all $i \neq h$, $\cup_{h=1}^{t} \delta_h = \{1, 2, 3, ..., n\}$, and $|\delta_i| \approx \frac{n}{t}$. Then each processor i is assigned the $\frac{n}{t} \times n$ rowwise part of the matrix A ($A(\delta_i, :) = A(:, \delta_i)$ since A is SPD), the $\frac{n}{t} \times 1$

Table 4.4: Comparison between the convergence of the different CG versions with respect to number of partitions or initial guesses for Coop-CG with $x_0 = 0$.

		CG		Coop-CG		MSD-CG		MSDO-CG		LRE-CG	
	Pa	Iter	Err	Iter	Err	Iter	Err	Iter	Err	Iter	Err
sky202020 $tol = 10^{-8}$	2	902	1E-5	795	8E-6	3070	2E-1	610	4E-5	557	2E-5
	4	902	1E-5	627	1E-5	11572	6E-1	420	2E-5	373	2E-5
	8	902	1E-5	542	4E-6	3207	7E-1	228	1E-5	211	1E-5
	16	902	1E-5	414	3E-6	4225	7E-1	134	1E-5	119	9E-6
	32	902	1E-5	290	1E-6	3149	7E-1	87	1E-6	69	9E-6
	64	902	1E-5	183	8E-7	2719	7E-1	53	6E-6	43	4E-6
ANI202020 $tol = 10^{-8}$	2	4187	4e-5	3584	5e-5	12404	2e-1	893	6e-5	875	7e-5
	4	4146	4e-5	3371	4e-5	17311	6e-1	749	8e-5	673	8e-5
	8	4146	4e-5	2865	4e-5	22339	7e-1	498	8e-5	449	1e-4
	16	4146	4e-5	2314	3e-5	21989	7e-1	328	1e-4	253	2e-4
	32	4146	4e-5	1615	2e-5	17042	7e-1	192	2e-4	148	2e-4
	64	4146	4e-5	1002	1e-5	19257	1e-4	122	5e-5	92	1e-4
ELASTICITY3D $tol = 10^{-8}$	2	987	4e-12	718	3e-12	3065	8e-1	764	3e-12	634	4e-12
	4	987	4e-12	534	8e-12	3497	8e-1	622	4e-12	480	2e-12
	8	987	4e-12	425	6e-11	3101	8e-1	472	1e-12	334	1e-12
	16	987	4e-12	348	6e-11	4239	1e-0	343	1e-12	235	1e-12
	32	987	4e-12	294	9e-12	–	–	234	1e-12	170	1e-12
	64	987	4e-12	235	1e-11	–	–			117	7e-13

rowwise part of the vector b ($b(\delta_i)$), and the vector $x_0(\overline{\delta}_i)$, where $\overline{\delta}_i = Adjacent(G(A), \delta_i)$ is the adjacent of δ_i in the graph of A. Processor i computes $x_k(\delta_i)$.

However, for performance reasons and due to the multicore nature of most architectures, it is possible to use a number of processors greater than t, preferably a multiple of t. In this case, we start by partitioning the graph of A into t subdomains using k-way partitioning or another graph partitioning, where δ_i for $i = 1, 2, .., t$ are the subsets of indices obtained from the partitioning. This partitioning is used to define the $T(.)$ operator and eventually the enlarged Krylov subspace. Assuming that we have ct processors, then every c processors are assigned an $\frac{n}{t} \times n$ rowwise part of the matrix A, $A(\delta_i, :)$, $\frac{n}{t} \times 1$ rowwise part of the vector b ($b(\delta_i)$) and the vector $x_0(\overline{\delta}_i)$, and should output $x_k(\delta_i)$. In other words, we partition each of our t subdomains into c non-overlapping subdomains to obtain a total of ct subdomains with set of indices $\delta_{i,j}$, where $i = 1, 2, .., t$, $j = 1, 2, .., c$, and $\delta_i = \cup_{j=1}^{c} \delta_{i,j}$. Then, in Algorithms 38 and 39, $log(t)$ is replaced by $log(ct)$, and $\frac{n}{t}$ is replaced by $\frac{n}{ct}$.

4.5.1 MSDO-CG

In this section we describe the parallelization of the MSDO-CG algorithm and we estimate its runtime in terms of flops, number of messages, and words sent. As mentioned in section 4.4, MGS, CGS2+A-CholQR, and CGS2+Pre-CholQR A-orthonormalizations are numerically the most stable and allow the convergence of MSDO-CG for the matrices in our test set. As discussed in section 2.4, the most parallelizable versions of MGS, Algorithms 10 and 11, require sending $(tk+1)log(t)$ and $2(t-1)log(t)$ messages respectively. Whereas CGS2, Algorithm 18, requires sending $4log(t)$ messages. On the other hand, Algorithm 21 of A-CholQR requires sending $log(t)$ messages, and Pre-CholQR Algorithm 23 requires sending $3log(t)$ messages. The CGS2+A-CholQR and CGS2+Pre-CholQR A-orthonormalizations can be called communication avoiding since they require sending $5log(t)$ and $7log(t)$ messages respectively, unlike the MGS A-orthonormalization. Since both methods are stable and CGS2+A-CholQR requires less communication, we present the Parallel MSDO-CG with CGS2+A-CholQR A-orthonormalization in Algorithm 38.

In Algorithm 38 we have two types of communication. The first is an "all reduce" communication that requires synchronization between all the processors and is equivalent to $log(t)$ messages, each of the same size (refer to [74]). For example, in line 10 of Algorithm 38, the "all reduce" is equivalent to $log(t)$ messages each of size t words, since β is a vector of size t.

The second type of communication is a point-to-point communication between each processor i and its m_i neighboring processors for computing a matrix - block of vectors multiplication, specifically $A[T(r)]$. We denote by $m_{MB} = max\{m_i| \quad i = 1, 2, .., t\}$ the largest number of neighboring processors where $m_i \leqslant m_{MB} \leqslant (t-1)$ for all i. Note that processor i has to compute $A(\delta_i, \overline{\delta}_i)[T(r)](\overline{\delta}_i, :)$, where $\overline{\delta}_i = Adjacent(G(A), \delta_i)$. Then, the neighboring processors of a given processor i are defined as all the processors j from which processor i needs some rows of $[T(r)]$ to compute its part of $A[T(r)]$. In other words, neighboring processors are all the processors j for which $\overline{\delta}_i \cap \delta_j \neq \phi$. Moreover, $[T(r)](\delta_i, :)$ is all zeros except for the i^{th} column which is equal to $r(\delta_i)$. Thus, processor i sends $r(\delta_i)$ of size $\frac{n}{t} \times 1$ to its neighboring processors once $r(\delta_i)$ is computed at step 8. Since $r(\delta_i)$ is used in the computation at step 12, this communication is overlapped with the computations from step 9 to 11. Simultaneously, processor i receives $r(\delta_j)$ from all its neighboring processors j for $j = 1, 2, .., m_i$. Then it computes $A(\delta_i, \overline{\delta}_i)[T(r)](\overline{\delta}_i)$ by performing approximately $\frac{2\text{nnz}-n}{t}$ flops.

The scheduling of the communications and computations in Algorithm 38 can be done as follows:

1. Processor i computes $r(\delta_i)$ and sends it to its neighboring processors.
2. Processor i overlaps the computation of $r(\delta_i)^t r(\delta_i)$ with the reception of $r(\delta_j)$ from all neighboring processors j
3. Processor i overlaps the computation of $W_1(\delta_i)$ with ρ's "all reduce"
4. Call algorithm 11 to A-orthonormalize P_1

Algorithm 38 Parallel MSDO-CG with CGS2+A-CholQR A-orthonormalization | Estimated Time

Input: δ_i, the set of indices assigned to processor i
Input: $A(\delta_i, :)$, the $\frac{n}{t} \times n$ row part of A
Input: $b(\delta_i)$, the $\frac{n}{t} \times 1$ row part of b; $x_0(\overline{\delta}_i)$, the $|\overline{\delta}_i| \approx \frac{n}{t} + c_i$ row part of r_0
Input: ϵ, the stopping tolerance; k_{max}, the maximum allowed iterations
Output: $x_k(\delta_i)$, the row part of the approximate solution of $Ax = b$

1:	**for** each processor $i = 1 : t$ in parallel **do**	
2:	Processor i computes $r(\delta_i) = b(\delta_i) - A(\delta_i, \overline{\delta}_i)x_0(\overline{\delta}_i)$ and let $k = 1$	$\gamma_c(2\frac{\text{nnz}}{t})$
3:	Processor i computes $r(\delta_i)^t r(\delta_i)$ and receives the full $\rho = \|\|r\|\|_2^2$	$\gamma_c(2\frac{n}{t})$
	via an all reduce (overlapped with the next computation)	$+(\alpha_c + \beta_c)log(t)$
4:	Processor i sends $P_1(\delta_i, :) = [T(r)](\delta_i, :) = [0, .., 0, r(\delta_i), 0, .., 0]$ to its	$\gamma_c(2\frac{\text{nnz}}{t})$
	m_{MB} neighboring processors and receives from them the corresponding	$+\alpha_c m_{MB}$
	blocks to obtain $P_1(\overline{\delta}_i, :)$. Then it computes $W_1(\delta_i) = A(\delta_i, \overline{\delta}_i)P_1(\overline{\delta}_i, :)$	$+\beta_c \frac{n}{t} m_{MB}$
5:	Call A-CholQR algorithm 21 to A-orthonormalize P_1	$\gamma_c 4nt + (\alpha_c + \beta_c t^2)log(t)$
6:	**while** ($\sqrt{\rho} > \epsilon\|\|b\|\|_2$ and $k < k_{max}$) **do**	
7:	Processor i computes $P_k(\delta_i, :)^t r(\delta_i)$ and receives the full	$\gamma_c(2\frac{n}{t} - 1)t$
	$\alpha = P_k^t r$ via an all reduce	$+(\alpha_c + t\beta_c)log(t)$
8:	Processor i computes $x(\delta_i) = x(\delta_i) + P_k(\delta_i, :)\alpha$	$4\gamma_c \frac{n}{t} t$
	and $r(\delta_i) = r(\delta_i) - W_k(\delta_i, :)\alpha$	
9:	Processor i computes $r(\delta_i)^t r(\delta_i)$ and receives $\rho = \|\|r\|\|_2^2$	$\gamma_c(2\frac{n}{t} - 1)$
	via an all reduce (overlapped with the next communication)	$+(\alpha_c + \beta_c)log(t)$
10:	Processor i computes $-W_k(\delta_i, :)^t r(\delta_i)$ and receives the full	$\gamma_c(2\frac{n}{t} - 1)t$
	$\beta = -W_k^t r$ via an all reduce	$+(\alpha_c + t\beta_c)log(t)$
11:	Processor i computes $P_{k+1}(\delta_i, :) = [T(r)](\delta_i, :) + P_k(\delta_i, :)diag(\beta)$	$2\gamma_c \frac{n}{t} t$
12:	Processor i sends $[T(r)](\delta_i, :)$ to its m_{MB} neighboring processors	$\gamma_c(2\frac{\text{nnz}}{t})$
	and receives from them the corresponding blocks to obtain	$+\alpha_c m_{MB}$
	$[T(r)](\overline{\delta}_i, :)$. Then it computes $Z(\delta_i) = A(\delta_i, \overline{\delta}_i)[T(r)](\overline{\delta}_i, :)$	$+\beta_c \frac{n}{t} m_{MB}$
13:	Processor i computes $W_{k+1}(\delta_i, :) = Z(\delta_i, :) + W_k(\delta_i, :)diag(\beta)$	$2\gamma_c \frac{n}{t} t$
14:	Call CGS2 Algorithm 18 to A-orthonormalize P_{k+1} against P_i	$12\gamma_c ntk$
	for all $i \leqslant k$	$+2(2\alpha_c + \beta_c t^2 k)log(t)$
15:	Call A-CholQR Algorithm 21 to A-orthonormalize P_{k+1}	$\gamma_c 4nt + (\alpha_c + \beta_c t^2)log(t)$
16:	$k = k + 1$	
17:	**end while**	
18:	**end for**	

Then at each iteration:

5. Processor i computes $P_k(\delta_i, :)^t r(\delta_i)$ and receives the full $\alpha = P_k^t r$ via an all reduce
6. Processor i computes $r(\delta_i)$ and send it to its neighboring processors.
7. Processor i overlaps the computation of $x(\delta_i)$, $r(\delta_i)^t r(\delta_i)$, and $-W_k(\delta_i, :)^t r(\delta_i)$ with the the

reception of $r(\delta_j)$ from all neighboring processors j

8. Processor i overlaps the computation of $Z(\delta_i) = A(\delta_i, \overline{\delta}_i)[T(r)](\overline{\delta}_i, :)$ and the reception $\rho = ||r||_2^2$ and $\beta = -W_k^t r$ via the same "all reduce" that costs $log(t)$ messages and $(t+1) * log(t)$ words

9. Processor i computes $P_{k+1}(\delta_i, :) = [T(r)](\delta_i, :) + P_k(\delta_i, :)diag(\beta)$ and $W_{k+1}(\delta_i, :) = Z(\delta_i, :) + W_k(\delta_i, :)diag(\beta)$

10. Call algorithm 18 to A-orthonormalize P_{k+1} against all P_i's for $i \leqslant k$

11. Call algorithm 21 to A-orthonormalize P_{k+1}

In summary, without the A-orthogonalization at steps 14 and 15, the estimated time of k_c iterations of Algorithm 38, where we ignore lower order terms, is

$$\gamma_c(11n + 2n\frac{\text{nnz}}{t})k_c + \alpha_c(2log(t) + m_{MB})k_c + \beta_c(\frac{n}{t}m_{MB} + 2tlog(t))k_c.$$

At iteration k, the CGS2+A-CholQR A-orthonormalization requires sending $5log(t)$ messages with $(t + 2tk + 2)tlog(t)$ words and performing approximately $12ntk + 4nt + 6n$ flops. After k_c iterations the estimated time for the A-orthonormalization is $\gamma_c(12ntk_c + 16nt + 6n)k_c + \alpha_c(5log(t))k_c + \beta_c(t + 2t\frac{(k_c+1)}{2} + 2)tlog(t)k_c$. Thus, the estimated time of k_c iterations of algorithm 38 is

$$\begin{aligned}\text{Time}_{\text{MSDO-CG}}(k_c) \approx\ & \gamma_c(2n\tfrac{\text{nnz}}{t} + 12ntk_c + 10nt + 17n)k_c + \alpha_c(7log(t) + m_{MB})k_c \\ & +\beta_c(\tfrac{n}{t}m_{MB} + t^2k_clog(t))k_c.\end{aligned}$$

4.5.2 LRE-CG

In this section we describe the parallelization of the LRE-CG algorithm and we estimate its runtime in terms of flops, number of messages, and words sent. As mentioned in section 4.4, MGS and the CGS+TSQR orthonormalizations are numerically the most stable and allow the convergence of LRE-CG for the matrices in our test set. The parallel version of MGS orthonormalization, Algorithms 4 and 5, is similar to that of the A-orthonormalization discussed in section 2.4, and requires sending $(tk + 1)log(t)$ and $2(t - 1)log(t)$ messages respectively. Whereas the CGS orthonormalization, Algorithm 2, can be parallelized in a block format like Algorithm 14, and requires sending $2log(t)$ messages. On the other hand, the TSQR orthonormalization (section 2.4.2.2) using binary trees as discussed in [21] requires sending $log(t)$ messages. The combination of BCGS and TSQR was discussed in [60] and it requires sending only $3log(t)$ messages as compared to the $(tk + 2t - 1)log(t)$ messages of MGS. We present the Parallel LRE-CG with BCGS+TSQR orthonormalization in Algorithm 39.

Algorithm 39 Parallel LRE-CG with BCGS+TSQR Algorithm — Estimated time

Input: δ_i, the set of indices assigned to processor i; $A(\delta_i, :)$, the $\frac{n}{t} \times n$ row part of A
Input: $b(\delta_i)$, the $\frac{n}{t} \times 1$ row part of b; $x_0(\overline{\delta}_i)$, the $|\overline{\delta}_i| \times 1$ row part of r_0
Input: ϵ, the stopping tolerance; k_{max}, the maximum allowed iterations
Output: $x_k(\delta_i)$, the row part of the approximate solution of $Ax = b$

1: **for** each processor $i = 1 : t$ in parallel **do**
2: Processor i computes $r(\delta_i) = b(\delta_i) - A(\delta_i, \overline{\delta}_i)x_0(\overline{\delta}_i)$ — $\gamma_c(2\frac{\text{nnz}}{t})$
3: Processor i computes $r(\delta_i)^t r(\delta_i)$ and receives the full $\rho = ||r||_2^2$ via an all reduce (overlapped with the next computation) — $\gamma_c(2\frac{n}{t} - 1) + (\alpha_c + \beta_c)log(t)$
4: Let $Q(\delta_i, :) = W(\delta_i, :) = [T(r)](\delta_i, :)$, and normalize its vectors — $\gamma_c(2\frac{n}{t}t)$
5: Processor i sends $W(\delta_i, :) = [T(r)](\delta_i, :)$ to its m_i neighboring processors and receives from them the corresponding blocks to obtain $W(\overline{\delta}_i, :)$. Let $k = 1$ — $+\alpha_c m_{MB} + \beta_c \frac{n}{t} m_{MB}$
6: **while** ($\sqrt{\rho_{k-1}} > \epsilon||b||_2$ and $k < k_{max}$) **do**
7: Processor i computes $Z(\delta_i, :) = A(\delta_i, \overline{\delta}_i)W(\overline{\delta}_i, :)$ — $2\gamma_c \frac{\text{nnz}}{t} t$
8: Processor i computes $W(\delta_i, :)^t Z(\delta_i, :)$ and receives $E = W^t Z$ via an "all reduce" — $\gamma_c 2\frac{n}{t}t^2 + (\alpha_c + t^2\beta_c)log(t)$
9: **if** $k == 1$ **then**
10: $D = Z$ and $G = E$
11: **else**
12: Processor i computes $Q(\delta_i, 1 : t(k-1))^t Z(\delta_i, :)$ and receives $F = Q(:, 1 : t(k-1))^t Z$ via an "all reduce" — $\gamma_c 2\frac{n}{t}t^2 k + \alpha_c log(t)$
13: $G = \begin{pmatrix} G & F \\ F^t & E \end{pmatrix}$ and $D = [D \;\; Z]$ — $+\beta_c t^2(k-1)log(t)$
14: **end if**
15: Processor i computes $Q(\delta_i, :)^t r(\delta_i)$ and receives $Q^t r$ via an "all reduce" — $\gamma_c 2\frac{n}{t}tk + (\alpha_c + tk\beta_c)log(t)$
16: $\alpha = G^{-1}(Q^t r)$ — $Time_\alpha(tk)$
17: Processor i computes $x(\delta_i) = x(\delta_i) + Q(\delta_i, :)\alpha$ — $\gamma_c(2tk\frac{n}{t})$
18: Processor i computes $r(\delta_i) = r(\delta_i) - D(\delta_i, :)\alpha$ — $\gamma_c(2tk\frac{n}{t})$
19: Processor i computes $r(\delta_i)^t r(\delta_i)$ and receives $\rho_k = ||r||_2^2$ via an "all reduce" (overlapped with the next computation) — $\gamma_c(2\frac{n}{t} - 1) + (\alpha_c + \beta_c)log(t)$
20: Let $W = Z$ and orthogonalize the columns of W against those of Q using BCGS Algorithm 2 — $\gamma_c 4ntk + 2\alpha_c log(t) + t^2 k\beta_c log(t)$
21: Orthonormalize the vectors of W using TSQR (section 2.3.3) and let $Q = [Q\ W]$ — $\gamma_c(2\frac{n}{t}t^2 + \frac{2}{3}t^3 log(t)) + (\alpha_c + \beta_c\frac{t}{2})log(t)$
22: Processor i sends $W(\delta_i, :)$ to its m_i neighboring processors and receives from them the corresponding blocks to obtain $W(\overline{\delta}_i, :)$ — $+\alpha_c m_{MB} + \beta_c \frac{n}{t} t m_{MB}$
23: k = k+1
24: **end while**
25: **end for**

In Algorithm 39 there are two types of communication, similarly to Algorithm 38. The first type of communication is a point-to-point communication between each processor i and its m_i neighboring processors for computing the matrix - block of vectors multiplication $Z = AW$ in line 7. We denote by $m_{MB} = max\{m_i|\ \ i = 1, 2, .., t\}$ the largest number of neighboring processors, where $m_{MB} \leqslant (t-1)$ for all i. Note that processor i has to compute $A(\delta_i, \overline{\delta}_i)W(\overline{\delta}_i, :)$, where $\overline{\delta}_i = Adjacent(G(A), \delta_i)$. Then, the neighboring processors of a given processor i are defined as all the processors j from which processor i needs some rows of W to compute its part of AW. In other words, neighboring processors are all the processors j for which $\overline{\delta}_i \cap \delta_j \neq \phi$. Note that for the first iteration, $W(\delta_i, :) = [T(r)](\delta_i, :)$ is all zeros except for the i^{th} column which is equal to $r(\delta_i)$. Thus, processor i sends $r(\delta_i)$ of size $\frac{n}{t} \times 1$ to its neighboring processors once $r(\delta_i)$ is computed. However, for the next iteration the W is no longer sparse, therefore $W(\delta_i, :)$ of size $\frac{n}{t} \times t$ is sent.

The second type of communication is an "all reduce" that requires synchronization between all the processors, and it is equivalent to $log(t)$ messages each of the same size (refer to [74]). For example, in lines 3 and 19 of Algorithm 39, the "all reduce" is equivalent to $log(t)$ messages each of size 1 word. As mentioned, this communication can be overlapped with the next computation. The reception of $E = W^tZ$, $F = Q(:, 1 : t(k-1))^tZ$ and Q^tr via an "all reduce" in lines 8, 12 and 15 of Algorithm 39 is equivalent to $log(t)$ messages each of size t^2 words, $log(t)$ messages each of size $t^2(k-1)$ words, and $log(t)$ messages each of size tk words respectively. However, the three computations are independent. Thus, each processor can compute its part of the three aforementioned computations and then receive the full matrices and vectors via $log(t)$ messages each of size $kt(t+1)$ words, assuming that it is possible to send t^2k words in one message. Another alternative is to compute $Q(\delta_i, 1 : t(k-1))^tZ(\delta_i, :)$ in several steps and overlap the communication with the next computation. The number of steps depends on the machine's architecture and on the values of t and k.

The scheduling of the communications and computations in Algorithm 39 can be done as follows:

1. Processor i computes $r(\delta_i)$ and sends it to its m_i neighboring processors.
2. Processor i overlaps the computation of $r(\delta_i)^tr(\delta_i)$ with the reception of $r(\delta_j)$ from all neighboring processors j
3. Processor i overlaps the normalization of $W(\delta_i, :)$ with ρ's "all reduce"

   Then at each iteration:

4. Processor i computes $Z(\delta_i, :) = A(\delta_i, \overline{\delta}_i)W(\overline{\delta}_i, :), W(\delta_i, :)^tZ(\delta_i, :)$, $Q(\delta_i, 1 : t(k-1))^tZ(\delta_i, :)$, and $Q(\delta_i, :)^tr(\delta_i)$. Then it receives $E = W^tZ$, $F = Q(:, 1 : t(k-1))^tZ$ and Q^tr via an "all reduce" that costs $log(t)$ messages and $kt(t+1) * log(t)$ words. Then update G and D
5. Solve $\alpha = G^{-1}(Q^tr)$ where each processor i receives the full vector α

6. Processor i computes $x(\delta_i)$, $r(\delta_j)$, and $r(\delta_i)^t r(\delta_i)$. Then overlaps the reception $\rho = ||r||_2^2$ with the next computation.

7. Orthogonalize the columns of W against those of Q using BCGS

8. Orthonormalize the vectors of W using TSQR

In Algorithm 39, we show the estimated time for each computation and communication, where $Time_\alpha(tk)$ is the estimated time for solving the $tk \times tk$ α system in line 16. At the k^{th} iteration of Algorithm 39 the total flops, except for the α system, is 2nnz $+ (6nt - 2t^2 + 6n - t)k + 2nt + \frac{2}{3}t^3 log(t) + 2n + 2\frac{n}{t} - 1$. And $4log(t) + m_{MB}$ messages are sent with $((2t^2 + t)k + \frac{t^2}{2} + t)log(t) + nm_{MB}$ words.

Then, by ignoring the lower order terms the estimated time of k_c iterations of Algorithm 39, where $t > 1$, $k_c > 1$, is

$$\begin{aligned} Time_{\text{LRE-CG}}(k_c) \approx\ & \gamma_c(2\text{nnz} + 3ntk_c + \tfrac{2}{3}t^3 log(t))k_c + \alpha_c(4log(t) + m_{MB})k_c \\ & + \beta_c[t^2 k_c log(t) + \tfrac{3}{2}t^2 log(t) + m_{MB}n]k_c + \textstyle\sum_{k=1}^{k_c} Time_\alpha(tk) \end{aligned}$$

4.6 Preconditioned enlarged Krylov subspace methods

After introducing the enlarged Krylov subspace methods and proving, theoretically and numerically, that these methods converge, we describe the preconditioned enlarged Krylov methods. A system $Ax = b$ can be left, right, or split preconditioned. In the case of conjugate gradient methods, the matrix A is symmetric positive definite (SPD). Hence, the preconditioned matrix should also be SPD. For left and right preconditioning, it is not easy to find some matrix M such that $M^{-1}A$ or AM^{-1} is SPD. But assuming that $M = LL^t$, then the split preconditioned matrix $L^{-1}AL^{-t}$ is SPD with $L^{-t} = (L^t)^{-1}$.

Given an $n \times n$ SPD matrix A, $n \times 1$ vector b and some preconditioner $M = LL^t$, then the split preconditioned enlarged Krylov subspace corresponding to the system $L^{-1}AL^{-t}y = L^{-1}b$ with $y = L^t x$ and $M = LL^t$, is defined by

$$\begin{aligned} \mathcal{K}_{t,k}(L^{-1}AL^{-t}, r_0) &= span\{T(r_0), L^{-1}AL^{-t}T(r_0), (L^{-1}AL^{-t})^2 T(r0), ..., (L^{-1}AL^{-t})^{k-1}T(r_0)\} \\ &= span\{T_1(r_0), T_2(r_0), ..., T_t(r_0), L^{-1}AL^{-t}T_1(r_0), L^{-1}AL^{-t}T_2(r_0), ..., \\ &\quad L^{-1}AL^{-t}T_t(r_0), ..., (L^{-1}AL^{-t})^{k-1}T_1(r_0), ..., (L^{-1}AL^{-t})^{k-1}T_t(r_0)\}, \end{aligned}$$

where $r_0 = L^{-1}(b - AL^{-t}y_0) = L^{-1}(b - Ax_0)$, $y_0 = L^t x_0$, and x_0 is the initial guess.

Consequently, the split preconditioned enlarged conjugate gradient methods are defined by the orthogonality condition and the subspace condition associated with preconditioned enlarged Krylov subspace. For example, given a split preconditioned system $L^{-1}AL^{-t}y = L^{-1}b$ with $y = L^t x$ and $M = LL^t$, the enlarged CG Krylov projection methods are defined by $y_k \in y_0 + \mathcal{K}_{t,k}(L^{-1}AL^{-t}, r_0)$ (the subspace condition), and $r_k \perp \mathcal{K}_{t,k}(L^{-1}AL^{-t}, r_0)$ (orthogonality

condition), where $r_k = L^{-1}(b - AL^{-t}y_k) = L^{-1}(b - Ax_k)$. Assuming $\widehat{A} = L^{-1}AL^{-t}$, $\widehat{b} = L^{-1}b$, and $\widehat{x} = y$, then all the theorems and properties discussed in section 4.1 are valid for the system $\widehat{A}\widehat{x} = \widehat{b}$.

Algorithm 40 Split preconditioned MSDO-CG with CGS2+$\widehat{A}$-CholQR $\widehat{A}$-orthonormalization

Input: A, the $n \times n$ symmetric positive definite matrix; L, $n \times n$ split preconditioner
Input: b, the $n \times 1$ right-hand side; x_0, the initial guess or iterate
Input: ϵ, the stopping tolerance; k_{max}, the maximum allowed iterations
Output: x, the approximate solution of $Ax = b$ by solving for $L^{-1}AL^{-t}y = L^{-1}b$ and $y = L^t x$

$y = L^t x_0$, $r = L^{-1}(b - Ax_0)$, $\rho = ||r||_2^2$, $nb = ||L^{-1}b||_2$, $k = 1$
Let $P_1 = T(r)$ and $L^{-1}AL^{-t}$-orthonormalize it using Algorithm 22 which outputs $W_1 = L^{-1}AL^{-t}P_1$
while ($\sqrt{\rho} > \epsilon(nb)$ and $k < k_{max}$) **do**
    $\alpha = (P_k^t W_k)^{-1}(P_k^t r) = P_k^t r$
    $y = y + P_k\alpha$ and $r = r - W_k\alpha$
    $\rho = ||r||_2^2$
    $\beta = -(P_k^t W_k)^{-1}(W_k^t r) = -W_k^t r$
    $P_{k+1} = T(r) + P_k diag(\beta)$
    $L^{-1}AL^{-t}$-orthonormalize P_{k+1} against all P_i's for $i \leqslant k$ using CGS2 Algorithm 19
    $L^{-1}AL^{-t}$-orthonormalize P_{k+1} using $\widehat{A}$-CholQR Algorithm 22 which outputs $W_{k+1} = L^{-1}AL^{-t}P_{k+1}$
    $k = k + 1$
end while
Solve $L^t x = y$

Given an SPD matrix M, then the Cholesky factorization $M = LL^t$ can be used for split preconditioning the system $Ax = b$, where the matrix $L^{-1}AL^{-t}$ is SPD. As the Cholesky factorization of an $n \times n$ matrix can be expensive, another alternative is to use block Jacobi preconditioner (section 3.3.2) with Cholesky factorization of the diagonal blocks.

We present in Algorithm 40 the split preconditioned MSDO-CG with CGS2+$\widehat{A}$-CholQR $\widehat{A}$-orthonormalization of the system $\widehat{A}\widehat{x} = \widehat{b}$, where $\widehat{A} = L^{-1}AL^{-t}$, $\widehat{b} = L^{-1}b$, $\widehat{x} = y$, and $y = L^t x$. We omit the W recursion due to numerical errors since $W = \widehat{A}P = L^{-1}AL^{-t}P$ consists of performing backward and forward substitution in addition to the matrix vector multiplication. Thus, we use a version of the CGS2 $\widehat{A}$-orthonormalization (Algorithm 19) that computes $W = \widehat{A}P = L^{-1}AL^{-t}P$ and outputs it. As for the $\widehat{A}$-CholQR, by assuming that $W = \widehat{A}P = L^{-1}AL^{-t}P$ is computed, then we can use Algorithm 22 with input $W = \widehat{A}P$. The additional cost of preconditioning is computing at each iteration k, four times $W_{k+1} = L^{-1}AL^{-t}P_{k+1}$ which is equivalent to a backward and forward substitution with t right hand sides and a matrix vector

Algorithm 41 Split preconditioned LRE-CG with BCGS+TSQR orthonormalization Algorithm

Input: A, the $n \times n$ symmetric positive definite matrix; L, $n \times n$ split preconditioner
Input: b, the $n \times 1$ right-hand side; x_0, the initial guess or iterate
Input: ϵ, the stopping tolerance; k_{max}, the maximum allowed iterations
Output: x, the approximate solution of $Ax = b$ by solving for $L^{-1}AL^{-t}y = L^{-1}b$ and $y = L^t x$

1: $y = L^t x_0$, $r = L^{-1}(b - Ax_0)$, $\rho_0 = ||r||_2^2$, $nb = ||L^{-1}b||_2$,k = 1
2: Let $W = T(r_0)$, normalize its vectors and then let $Q = W$
3: **while** ($\sqrt{\rho_{k-1}} > \epsilon(nb)$ and $k < k_{max}$) **do**
4: $\alpha = (Q^t L^{-1}AL^{-t}Q)^{-1}(Q^t r)$
5: $y = y + Q\alpha$
6: $r = r - L^{-1}AL^{-t}Q\alpha$ and $\rho_k = ||r||_2^2$
7: Let $W = L^{-1}AL^{-t}W$
8: Orthonormalize the vectors of W against the vectors of Q using BCGS Algorithm 2
9: Orthonormalize the vectors of W using TSQR (section 2.3.3) and let $Q = [Q\ W]$ and $k = k + 1$
10: **end while**
11: Solve $L^t x = y$

multiplication. The difference between the preconditioned and unpreconditioned MSDO-CG is in the A-orthonormalization, the computation of W, and the backward substitution $L^t x = y$. Note that the split preconditioned MSDO-CG with MGS $\widehat{A}$-orthonormalization did not converge. This might be due to numerical errors in solving the tk backward and forward substitutions in the MGS $\widehat{A}$-orthonormalization. However, the MSDO-CG with CGS2+$\widehat{A}$-CholQR $\widehat{A}$-orthonormalization converges very well as shown in section 4.6.1.

In Algorithm 41, we present the split preconditioned LRE-CG algorithm with BCGS+TSQR orthonormalization of the system $\widehat{A}\widehat{x} = \widehat{b}$, where $\widehat{A} = L^{-1}AL^{-t}$, $\widehat{b} = L^{-1}b$, $\widehat{x} = y$, and $y = L^t x$. At first glance, it might appear to the reader that the additional cost at iteration k in Algorithm 41 is solving a forward and backward substitution with tk right hand sides ($L^{-1}AL^{-t}Q$) and a forward and backward substitution with t right hand sides ($L^{-1}AL^{-t}W$). However, by taking a quick look at Algorithm 42, it is clear that the additional cost of preconditioning at iteration k is solving only a forward and backward substitution with t right hand sides ($L^{-1}AL^{-t}W$). And this is the only difference with the unpreconditioned version in addition to the backward substitution $L^t x = y$ at the end.

4.6.1 Convergence

We compare the convergence of split preconditioned MSDO-CG with CGS2+CholQR $\widehat{A}$-orthonormalization and split preconditioned LRE-CG with CGS+TSQR orthonormalization to CG and split precon-

Algorithm 42 Split preconditioned LRE-CG with BCGS+TSQR orthonormalization Pseudo Code

Input: A, the $n \times n$ symmetric positive definite matrix; b, the $n \times 1$ right-hand side
Input: x_0, the initial guess; ϵ, the stopping tolerance; k_{max}, the maximum allowed iterations
Output: x_k, the approximate solution of the system $Ax = b$

$y = L^t x_0$, $r = L^{-1}(b - Ax_0)$, $\rho_0 = ||r||_2^2$, $nb = ||L^{-1}b||_2$,k = 1
Let $W = T(r_0)$, normalize its vectors and then let $Q = W$
while ($\sqrt{\rho_{k-1}} > \epsilon(nb)$ and $k < k_{max}$) **do**
    $Z = L^{-1}AL^{-t}W$, $E = W^tZ$
    if $k == 1$ **then**
        $D = Z$ and $G = E$
    else
        $D = [D \;\; Z]$, $F = Q(:, 1 : t(k-1))^tZ$, and $G = \begin{pmatrix} G & F \\ F^t & E \end{pmatrix}$
    end if
    $\alpha = G^{-1}(Q^tr)$
    $y = y + Q\alpha$
    $r = r - D\alpha$ and $\rho_k = ||r||_2^2$
    Let $W = Z$
    Orthonormalize the columns of W against those of Q using BCGS Algorithm 2
    Orthonormalize the vectors of W using TSQR (section 2.3.3) and let $Q = [Q \;\; W]$ and $k = k + 1$
end while
Solve $L^tx = y$

ditioned CG (PCG). We use Block Jacobi with Cholesky factorization of the block diagonals as a preconditioner.

In table 4.5, we use a different Block Jacobi preconditioner for the different partitions. First, the graph of A is partitioned into t parts that define the enlarged Krylov subspace using Metis's k-way edge separator where $t = 2, 4, 8, 16, 32, 64$. Then the Block Jacobi preconditioner M is defined as the t diagonal blocks of the permuted matrix A. Each of the t blocks is factorized using Cholesky decomposition to obtain a block L. The preconditioned LRE-CG converges faster than the preconditioned MSDO-CG and PCG for the different configurations. As the number of partitions or the maximum basis vectors added at each iteration is doubled, the Block Jacobi preconditioned CG needs more iterations to converge. However, for the matrices POISSON2D, NH2D1, and SKY2D, the number of iteration of the preconditioned LRE-CG and MSDO-CG decreases. As for the matrices SKY3D and ANI3D, the number of iterations of LRE-CG increases then decreases back to the same number of iterations for $t = 2$, unlike preconditioned MSDO-CG.

In table 4.6, we use a fixed Block Jacobi preconditioner for all the partitions to compare the

Table 4.5: Comparison of the convergence of the split preconditioned CG, MSDO-CG with CGS2+CholQR A-orthonormalization, and LRE-CG with CGS+TSQR orthonormalization method with varying Block Jacobi preconditioners, with respect to number of partitions Pa, with $x_0 = 0$.

				Split Preconditioned Methods					
		CG		PCG		MSDO-CG		LRE-CG	
	Pa	Iter	Err	Iter	Err	Iter	Err	Iter	Err
POISSON2D $tol = 10^{-6}$	2	195	2E-5	35	1E-5	30	2E-6	30	2E-6
	4			40	1E-5	28	4E-6	28	2E-6
	8			48	2E-5	30	6E-6	27	2E-6
	16			50	1E-5	28	1E-6	25	1E-6
	32			57	2E-5	26	8E-7	23	5E-7
	64			66	2E-5	23	1E-6	20	3E-7
NH2D1 $tol = 10^{-8}$	2	259	4E-7	47	3E-8	37	6E-9	37	6E-9
	4			55	7E-8	34	2E-8	34	1E-8
	8			65	1E-7	36	1E-8	33	1E-8
	16			71	3E-7	33	1E-8	30	8E-9
	32			83	1E-7	29	1E-8	27	4E-9
	64			88	5E-7	26	5E-9	23	4E-9
SKY2D $tol = 10^{-8}$	2	5855	4E-4	74	3E-7	40	4E-7	40	4E-7
	4			80	2E-6	43	1E-7	36	5E-7
	8			144	2E-5	48	3E-7	31	3E-7
	16			162	1E-4	46	1E-7	27	2E-7
	32			210	3E-4	39	1E-7	23	2E-7
	64			260	2E-7	31	8E-8	20	2E-7
SKY3D $tol = 10^{-8}$	2	902	2E-5	37	2E-6	24	2E-7	24	2E-7
	4			113	2E-5	54	1E-7	43	1E-7
	8			120	8E-6	54	7E-8	33	9E-8
	16			154	1E-5	49	1E-7	28	5E-8
	32			208	1E-5	60	2E-8	30	4E-8
	64			213	1E-5	46	1E-8	22	3E-8
ANI3D $tol = 10^{-8}$	2	4184	4e-5	26	1E-5	31	3E-7	31	3e-7
	4			43	4E-6	39	5e-7	39	6E-7
	8			47	5E-7	39	6E-7	39	5E-7
	16			54	7E-7	43	1E-6	41	6E-7
	32			61	2E-7	47	4e-7	41	1E-6
	64			66	8E-7	46	2E-7	38	4E-7

Table 4.6: Comparison of the convergence of the split preconditioned CG, MSDO-CG with CGS2+CholQR $\widehat{A}$-orthonormalization, and LRE-CG with CGS+TSQR orthonormalization method with a fixed Block Jacobi preconditioner , with respect to number of partitions Pa, with $x_0 = 0$.

				Split Preconditioned Methods					
		CG		PCG		MSDO-CG		LRE-CG	
	Pa	Iter	Err	Iter	Err	Iter	Err	Iter	Err
POISSON2D $tol = 10^{-6}$	2	195	2E-5	66	2E-5	62	2E-5	61	7E-6
	4					54	9E-6	50	8E-6
	8					47	4E-6	41	4E-6
	16					39	3E-6	33	1E-6
	32					31	2E-6	25	8E-7
	64					25	8E-7	20	3E-7
NH2D1 $tol = 10^{-8}$	2	259	4E-7	88	5E-7	82	1E-7	76	7E-8
	4					67	5E-8	63	5E-8
	8					57	3E-8	57	1E-8
	16					46	1E-8	39	2E-8
	32					36	2E-8	36	4E-9
	64					28	7E-9	23	4E-9
SKY2D $tol = 10^{-8}$	2	5773	5E-04	261	2E-4	223	2E-5	184	6E-7
	4					152	4E-7	99	5E-7
	8					109	2E-7	66	4E-7
	16					72	1E-7	44	4E-7
	32					52	5E-8	29	1E-7
	64					34	7E-8	20	2E-7
SKY3D $tol = 10^{-8}$	2	902	2E-5	225	4E-6	191	3E-6	181	5E-6
	4					163	6E-6	135	1E-6
	8					126	2E-6	78	1E-7
	16					94	8E-8	48	9E-8
	32					61	7E-8	28	1E-7
	64					47	3E-8	21	1E-7
ANI3D $tol = 10^{-8}$	2	4184	4E-5	69	8E-7	68	8E-7	66	7e-7
	4					66	4e-7	63	4E-7
	8					61	4E-7	57	3E-7
	16					58	5E-7	52	6E-7
	32					53	6e-7	46	1E-6
	64					45	3E-7	37	8E-7

convergence of the methods with respect to the doubling of the number of partitions t. First, the graph of A is partitioned into 64 parts using Metis's k-way edge separator. Then the Block Jacobi preconditioner M is defined as the 64 block diagonals of the permuted matrix A. Each of the 64 blocks is factorized using Cholesky decomposition to obtain a block L. Then the matrix A is partitioned once again using k-way into $t = 2, 4, 8, 16, 32$ or 64 parts that define the enlarged Krylov subspace. The preconditioner is permuted accordingly. The preconditioned LRE-CG converges faster than the preconditioned MSDO-CG and PCG. As the number of partitions or the maximum basis vectors added at each iteration is doubled, the preconditioned LRE-CG and MSDO-CG converge faster. However, for some matrices, like POISSON2D, NH2D1, and ANI3D, as the number of partitions is doubled, the number of iterations till convergence decreases by only $10\% - 20\%$. Thus, it is efficent to use a maximum of $t = 4$ partitions which correspond to adding at most t vectors to the basis of the enlarged Krylov subspace. For the matrices SKY2D and SKY3D, the number of iterations decreases by $33\% - 45\%$ and $25\% - 45\%$ respectively. Hence it is possible to use a maximum of $t = 8$ partitions or at most $t = 16$.

4.7 Summary

In this chapter we introduced two new iterative methods, MSDO-CG and LRE-CG, which are based on the enlarged Krylov subspace. We defined the properties of the enlarged Krylov subspace, derived the new methods in the context of projection CG versions, provided parallel versions that reduce communication, and shown that the methods converge at least as fast as Classical CG in exact precision arithmetic. The convergence results show that they also converge faster than CG in finite precision arithmetic. We have also presented the preconditioned versions and tested their convergence with block Jacobi preconditioner.

MSDO-CG is a variation of the MSD-CG version, where we A-orthonormalize the t search directions against previous directions and against each others. Due to the A-orthonormalization, we lose the short recurrence property of CG and we are obliged to save all the tk_c search directions, where k_c is the number of iterations till convergence. In LRE-CG we start by building an orthonormal basis for the enlarged Krylov subspace, then we use the whole basis to update the solution. The main difference between both methods in terms of performance, is that at each iteration of MSDO-CG, we use t search directions to update the new approximate solution. Whereas in LRE-CG, at each iteration i, we use the entire basis formed by ti vectors, to update the approximate soltion and we solve a $ti \times ti$ system. However, this use of the whole basis leads to a relatively faster convergence than MSDO-CG. One way to limit this increasing cost is by restarting LRE-CG after some iterations. Another alternative is to choose at each iteration i, a linearly independent subset of the t computed vectors. This adds an extra cost, but reduces the size of the system that has to be solved at each iteration. A third alternative is to compute t_i vectors at each iteration i, where $t_0 = t$, and $t_i \leqslant t$. Then choose $\widehat{t}_i$ linearly independant vectors where $\widehat{t}_i \leqslant t_i$, and $t_{i+1} = \widehat{t}_i$.

Although each iteration of the MSDO-CG and LRE-CG methods is at least t times more ex-

pensive than the CG iteration in terms of flops, as shown in section 4.5, both methods use less communication, and Blas2 and Blas3 operations that can be parallelized in a more efficient way than the dot products in CG. Moreover, the MSDO-CG and LRE-CG methods converge faster than CG in terms of iterations as shown in section 4.4.

Chapter 5

Communication Avoiding Incomplete LU(0) Preconditioner

In this chapter, we introduce a communication avoiding version of the ILU(0) preconditioner that can be used with preconditioned communication avoiding Krylov methods, preconditioned s-step Krylov methods, preconditioned classical Krylov methods, and any method that uses a preconditioned version of the matrix powers kernel (Algorithm 25). Recall that the matrix powers kernel, introduced in section 2.5, computes s vectors of the Krylov subspace of the form $y_i = Ay_{i-1}$ for $i = 1, ..., s$ in parallel without any communication, by ghosting some data and performing redundant flops. This chapter is based on some parts of a revised version of the technical report [40], and on the article [39], which was submitted to SIAM journal on scientific computing (SISC) and is in revision.

Our goal is to design communication avoiding preconditioners that are efficient in accelerating the iterative methods and also minimize communication. In other words, given a preconditioner M, the preconditioned system with its communication avoiding version $M_{ca}^{-1}Ax = M_{ca}^{-1}b$ should have the same order of convergence as the original preconditioned system $M^{-1}Ax = M^{-1}b$, and also reduce communication. This is a challenging problem, since applying a preconditioner on its own may, and in general will, require extra communication. Thus we restrict our work to ILU(0) preconditioner. Since the construction of $M = LU$ represents typically an important part of the overall runtime of the linear solver, we focus on both minimizing communication during the construction of M (i.e, the ILU(0) factorization), and during its application to a vector ($z = M^{-1}y = (LU)^{-1}y$)) at each iteration of the linear solver.

We introduce CA-ILU0, a communication avoiding ILU(0) preconditioner for left preconditioned systems. With a few modifications discussed in section 5.3, CA-ILU0 preconditioner can be applied to right and split preconditioned systems. We start by adapting the matrix powers kernel (Algorithm 25) to the ILU preconditioned system to obtain the ILU matrix powers kernel in section 5.1. Each vector of this kernel is obtained by computing ($(LU)^{-1}Ay$), that is in addition to the matrix-vector multiplication Ay, it uses a forward and a backward substitution. The ILU matrix

powers kernel, which is designed for any given LU decomposition, does not allow to avoid communication by itself. That is, if we want to compute s vectors of this kernel with no communication through ghosting some of the data, there are cases when one processor performs an important part of the entire computation. We restrain then our attention to the ILU0 factorization, which has the property that the L and U factors have the same sparsity pattern as the lower triangular part of A and the upper triangular part of A respectively. To obtain a communication avoiding ILU0 preconditioner, we introduce in section 5.2 a reordering, alternating min-max layers AMML(s), of the input matrix A, which is reflected in the L and U matrices. First, the graph of A is partitioned using some known graph partitioning technique that minimize the size of edge separators like k-way, or vertices separators like nested dissection (section 2.2). Then each of the obtained subdomains is further reordered. We present two versions of AMML(s), the first based on nested dissection and the second on k-way. Note that it is possible to adapt the AMML(s) reordering to any other graph or hypergraph partitioning technique.

The AMML(1) reordering allows avoiding communication when computing $s = 1$ matrix vector multiplication $(LU)^{-1}Ax$. In other words, the matrix vector multiplication Ax, and the backward and forward substitution are parallelized with only one communication before starting the whole computation.Thus, the CA-ILU0 preconditioner can be used with classical preconditioned Krylov methods like GMRES to reduce communication. Moreover, it can be used with preconditioned block Krylov methods and preconditioned enlarged Krylov subspace methods.

In general, the AMML(s) reordering allows to avoid communication for s-steps of the matrix vector multiplication $((LU)^{-1}Ax)$. In other words, s backward and forward solves corresponding to a submatrix of A can be performed when $s \geqslant 1$, without needing any data from other submatrices. Thus with our reordering it is sufficient to communicate once at the beginning of the first multiplication. This is possible since the CA-ILU0 $(L)^{-1}$ and $(U)^{-1}$ factors are sparse, unlike those of ILU(0), as shown in Figures 5.4 and 5.7 . Thus, the CA-ILU0 preconditioner can also be used with s-step Krylov methods like s-step GMRES and CA-GMRES to reduce communication. In section 5.4.2, we discuss the reduction in communication introduced by our method for $s \geqslant 1$.

In this chapter we portray our CA-ILU0 preconditioner (section 5.3) and its performance (section 5.4) using GMRES, but it can be used with other Krylov subspace methods as well. Although we focus on structured matrices arising from the discretization of partial differential equations on regular grids, it must be noted that the method also works for sparse unstructured matrices whose graphs can be well partitioned (small edge or vertex separators). The AMML(s) reordering can be used to avoid communication not only in parallel computations (between processors, shared-memory cores, or between CPU and GPU) but also in sequential computations (between different levels of the memory hierarchy). Thus in this chapter we will use the term *processor* to indicate the component performing the computation and *fetch* to indicate the movement of data (read, copy, or receive message) as discussed in section 2.1.1. In section 5.4 we show that our reordering does not affect the convergence of ILU0 preconditioned GMRES, and we model the expected performance of our preconditioner based on the needed memory and the redundant flops introduced to reduce communication.

5.1 ILU matrix powers kernel

The algorithm for solving a left-preconditioned system by using Krylov subspace methods is the same as a non-preconditioned system, with the exception of the matrix vector multiplications. For example GMRES requires computing $y = Ax$, while the preconditioned version computes $y = M^{-1}Ax$, where M is a preconditioner. Similarly, a preconditioned CA-GMRES relies on a preconditioned matrix powers kernel. And constructing a communication avoiding preconditioner is equivalent to building a preconditioned matrix powers kernel which computes the set of s basis vectors $\{M^{-1}Ay_0, (M^{-1}A)^2y_0, ..., (M^{-1}A)^{s-1}y_0, (M^{-1}A)^s y_0\}$ while minimizing communication, where y_0 is a starting vector and $s \geqslant 1$. In this section we first define the partitioning problem (section 5.1.1) associated with computing s vectors of the preconditioned matrix powers kernel. Then we identify dependencies involved in the computation of the preconditioned matrix powers kernel (section 5.1.2).

In this section we focus on the incomplete LU preconditioner $M = LU$ (section 3.3.1), which consists of finding a lower triangular matrix L and an upper triangular matrix U that approximate the input matrix A up to some error, i.e. $A = LU + R$, where R is the residual matrix.

5.1.1 The partitioning problem

In the following, we consider that the matrix A and the factors L and U of the preconditioner are distributed block row-wise over P processors. A communication avoiding preconditioner can be obtained if we are able to ghost some data and perform some redundant computation such that s basis vectors $\{M^{-1}Ay_0, (M^{-1}A)^2y_0, ..., (M^{-1}A)^{s-1}y_0, (M^{-1}A)^s y_0\}$ can be computed with no communication. To obtain an efficient preconditioner, ideally we would like to minimize the size of the ghost data while balancing the load among processors.

We are interested in ILU preconditioners where $M = LU$ is obtained from the ILU factorization of A. In practice, $(LU)^{-1}A$ is never computed explicitly. In fact, determining $y_i = (LU)^{-1}Ay_{i-1}$ during the computation of the preconditioned matrix powers kernel is equivalent to performing the three steps:

1. Compute $f = Ay_{i-1}$

   Solve $LUy_i = f$ i.e.

2. Solve $Lz = f$ by forward substitution

3. Solve $Uy_i = z$ by backward substitution

Thus, we use a heuristic which starts by partitioning the graph of A using either the vertex separator, nested dissection (section 2.2.2), or the edge separator, k-way (section 2.2.3) provided in Metis's library [49]. Then, the same partition, that satisfies the balance criterion (2.1) and minimize the cutsize (2.2), is applied for L and U. Then, we introduce heuristics to reduce the

redundant data and computation needed to perform the forward and backward substitution by each processor without communication.

Note that other graph partitioning techniques and even hypergraph partitioning techniques can be used to partition A. But for simplicity we base our work on graphs.

5.1.2 ILU preconditioned matrix powers kernel

Our ILU matrix powers kernel is based on the matrix powers kernel, with the exception that A is replaced by $(LU)^{-1}A$, since we have a preconditioned system. Given a partition $\pi = \{\Omega_1, \Omega_2, .., \Omega_{p-1}, \Omega_p\}$, that partitions the graph of $(LU)^{-1}A$ into p subgraphs, where p is the number of processors. Let $\alpha_0^{(j)} = V(\Omega_j)$ be the set of vertices assigned to processor j, where processor j must compute $\{y_1(\alpha_0^{(j)}), y_2(\alpha_0^{(j)}), ..., y_{s-1}(\alpha_0^{(j)}), y_s(\alpha_0^{(j)})\}$ with $y_i = (LU)^{-1}Ay_{i-1}$. However, since $(LU)^{-1}A$ is not available explicitly, the ILU matrix powers kernel is different than the matrix powers kernel in structure.

In the following we describe an algorithm that allows a processor j to perform s steps with no communication, by ghosting parts of A, L, U, and y_0 on processor j before starting the s iterations. Consider that at some step i, processor j needs to compute $y_i(\alpha)$. The last operation that leads to the computation of $y_i(\alpha)$ is the backward substitution $Uy_i(\alpha) = z$. Due to the dependencies in the backward substitution, the equations α are not sufficient for computing $y_i(\alpha)$. Gilbert and Peierls showed in [35] that the set of equations that need to be solved in addition to α is the set of reachable vertices form α in the graph of U. Thus, the equations $\beta = R(G(U), \alpha)$ are necessary and sufficient for solving the equations α. In other words, if the processor j has in its memory $U(\beta, \beta)$ and $z(\beta)$, then it can solve with no communication the reduced system $U(\beta, \beta)y_i(\beta) = z(\beta)$. This is because by definition of reachable sets, there are no edges between the vertices in the set β and other vertices. Thus all the columns in $U(\beta, :)$, except the β columns, are zero columns. To solve the reduced system $U(\beta, \beta)y_i(\beta) = z(\beta)$, processor j needs to have in his memory $z(\beta)$ beforehand. And this is equivalent to solving the set of equations $\gamma = R(G(L), \beta)$ of $Lz = f$. Similarly, processor j solves the reduced system $L(\gamma, \gamma)z_i(\gamma) = f(\gamma)$, where $f(\gamma)$ must be available. Computing $f(\gamma)$ is equivalent to computing $A(\gamma, :)y_{i-1}$. However, it must be noted that the entire vector y_{i-1} is not used, since for computing this subset of matrix vector multiplication, processor j only needs $y_{i-1}(\delta)$, where $\delta = Adj(G(A), \gamma)$. Therefore, it computes $A(\gamma, \delta)y_{i-1}(\delta)$.

Hence to compute the first step, $y_1 = (LU)^{-1}Ay_0$, processor j fetches $y_0(\delta_1^{(j)})$, $A(\gamma_1^{(j)}, \delta_1^{(j)})$, $L(\gamma_1^{(j)}, \gamma_1^{(j)})$ and $U(\beta_1^{(j)}, \beta_1^{(j)})$. To perform another step, we simply let $\alpha_1^{(j)} = \delta_1^{(j)}$ and do the same analysis. This procedure is summarized in Algorithm 43, where the superscript (j), referring to processor j, is dropped for better readability. Thus to compute s steps of $y_i(\alpha_0^{(j)}) = [(LU)^{-1}Ay_{i-1}](\alpha_0^{(j)})$, processor j fetches $y_0(\delta_s^{(j)})$, $A(\gamma_s^{(j)}, \delta_s^{(j)})$, $L(\gamma_s^{(j)}, \gamma_s^{(j)})$, and $U(\beta_s^{(j)}, \beta_s^{(j)})$. Note that $\alpha_{i-1}^{(j)} \subseteq \beta_i^{(j)} \subseteq \gamma_i^{(j)} \subseteq \delta_i^{(j)} \subseteq \alpha_i^{(j)}$, for $i = 1$ until s. After fetching all the data needed, processor j computes its part using Algorithm 44. Thus Algorithm 43 has to output all the subsets

Algorithm 43 s-step Dependencies

Input: $G(A)$, $G(L)$, $G(U)$; s, number of steps; α_0, subset of unknowns assigned to processor j
Output: Sets β_i, γ_i and δ_i for all $i = 1$ till s

1: **for** $i = 1$ *to* s
2: Find $\beta_i = R(G(U), \alpha_{i-1})$
3: Find $\gamma_i = R(G(L), \beta_i)$
4: Find $\delta_i = Adj(G(A), \gamma_i$
5: Set $\alpha_i = \delta_i$
6: **end for**

Algorithm 44 ILU Matrix Powers Kernel ($A(\gamma_s, \delta_s)$, $L(\gamma_s, \gamma_s)$, $U(\beta_s, \beta_s)$, s, α_0)

Input: $A(\gamma_s, \delta_s)$, $L(\gamma_s, \gamma_s)$, $U(\beta_s, \beta_s)$, s: number of steps, α_0: subset of unknowns assigned to processor j
Output: $y_i(\alpha_0)$, where $1 \leqslant i \leqslant s$

1: **for** $i = s$ *to* 1
2: Compute $f(\gamma_i) = A(\gamma_i, \delta_i)y_{j-s}(\delta_i)$
3: Solve $L(\gamma_i, \gamma_i)z_{i-s+1}(\gamma_i) = f(\gamma_i)$
4: Solve $U(\beta_i, \beta_i)y_{i-s+1}(\beta_i) = z(\beta_i)$
5: Save $y_{i-s+1}(\alpha_0)$, which is the part that processor j has to compute
6: **end for**

$\beta_i^{(j)}$, $\gamma_i^{(j)}$ and $\delta_i^{(j)}$ for $1 \leqslant i \leqslant s$ which will be used in Algorithm 44. Note that although processor j needs to compute only $y_i(\alpha_0^{(j)})$, where $1 \leqslant i \leqslant s$, it computes some redundant flops in order to avoid communication.

The ILU matrix powers kernel presented in here is general and works for any matrices L and U. However, it is not sufficient to reduce or avoid communication, since the reachable sets $\beta_i^{(j)}$ and $\gamma_i^{(j)}$ might be much larger than $\alpha_{i-1}^{(j)}$ and $\beta_i^{(j)}$ respectively. A communication avoiding method is efficient if there is a good trade-off between the number of redundant flops and the amount of communication which was reduced. This reflects in the runtime of the algorithm. In other words, if performing three or four steps of a CA-ILU preconditioned iterative solver, each processor ends up needing all the data and computing almost entirely the vectors y_i, then either we are not exploiting the parallelism of our problem efficiently or the problem is not fit for communication avoiding techniques. This is indeed the case if Algorithm 44 is applied to the 2D 5 point stencil matrix whose graph, presented in figure 5.5(a), is partitioned into 4 subdomains by using k-way partitioning. To perform only one step of an iterative solver preconditioned by CA-ILU with no communication, processor 1 (which computes Domain 1 in the figure) ends up computing the entire vector y_i and fetching all the matrices A, L, and U where the L and U matrices are obtained from the ILU0 factorization. This cancels any possible effect of the parallelization of the problem, and shows that what works for the matrix powers kernel of the form $y_i = Ay_{i-1}$ does not work for the

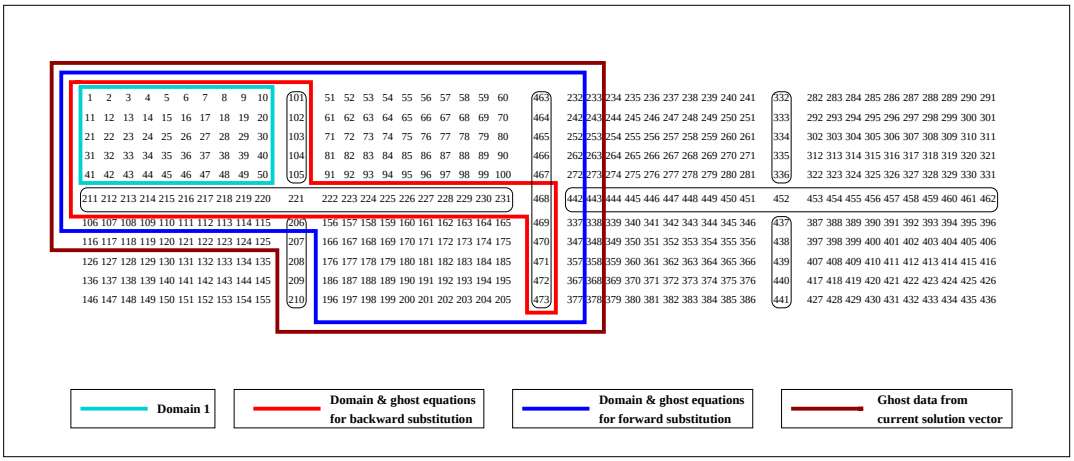


Figure 5.1: An 11 by 43 5-point stencil, partitioned into 8 subdomains using 7 separators. The data needed to compute one $y_i = (LU)^{-1}Ay_{i-1}$ on Domain 1 is shown, where L and U matrices are obtained from ILU(0). The bidirectional edges connecting each vertex to its north, south, east and west neighboring vertices are omitted in this figure.

same kernel where the multiplication is $y_i = (LU)^{-1}Ay_{i-1}$. Nested dissection might look like a better solution since it splits the domain into independent subdomains that interact only with the separators. However it is not sufficient either to obtain a communication avoiding preconditioner. This can be seen in figure 5.1. To compute one matrix-vector multiplication of the form $y_i = (LU)^{-1}Ay_{i-1}$, processor 1 has to fetch half of matrix A, half of matrix L, half of the vector y_{i-1}, almost the quarter of matrix U, and perform the associated computation.

This shows that partitioning the graph of the input matrix A by using techniques as nested dissection or k-way is not sufficient to reduce communication in the preconditioned matrix powers kernel. This is because both the matrix vector multiplication and the forward/backward substitutions need to be performed in a communication avoiding manner. In the next section, we introduce a new reordering that reduces the communication. Note that in Metis library [49], the subdomains with number of vertices greater than 200 do not have a natural ordering as shown in Figure 5.1, but they are partitioned recursively using nested dissection into smaller subdomains and separators. And this tends to reduce communication and the computed redundant flops, as we will detail later in the experimental section 5.4.

5.2 Alternating min-max layers (AMML(s)) reordering for ILU(0) matrix powers kernel

In this section we describe a reordering that allows to compute and apply an ILU(0) preconditioner in parallel in a preconditioned Krylov subspace method with no communication. This precondi-

tioner produces L and U matrices that have the same sparsity pattern as A. Hence the graphs of L and U are known before the numerical values of the factors L and U are computed, and this allows to reorder these graphs in order to avoid communication during the computation of the factors. This is not the case in drop-tolerance ILU, where the graphs of L and U are not known before the numerical computation of the factors, and hence avoiding communication is a more complex task.

Let A be a matrix whose graph is partitioned into P subgraphs or subdomains $\pi = \{\Omega_1, \Omega_2, ..., \Omega_P\}$ using nested dissection (overlapping subdomains) or k-way graph partitioning (nonoverlapping subdomains). To compute and apply in parallel the preconditioner, each processor j is assigned one subdomain Ω_j over which it should compute the s multiplications $y_{i+1}(\alpha_0^{(j)}) = ((LU)^{-1}Ay_i)(\alpha_0^{(j)})$, where $0 \leqslant i \leqslant s-1$, $\alpha_0^{(j)} = V(\Omega_j)$, L and U are obtained from the ILU(0) factorization of A. The goal of our reordering algorithm is to renumber the vertices of each subdomain, $\alpha_0^{(j)} = V(\Omega_j)$, such that processor j can compute its assigned part of the s multiplications without any communication. As explained in the previous section, k-way partitioning and nested dissection alone are not sufficient to reduce data movement and redundant flops in the ILU(0) matrix powers kernel. However our new reordering, which is applied locally on the vertices of the subdomains Ω_j, for $j = 1, 2, .., P$, obtained after a graph partitioning, reduces the ghost zones in Figures 5.1 and 5.5(a) not only for performing 1 step, but also for performing $2, 3, .., s$ steps of the ILU(0) matrix powers kernel. In this section we focus on the reordering after applying k-way graph partitioning, we refer to this reordering as alternating min-max layers (AMML(s)) reordering (section 5.2.2). However, we first introduce AMML(s) reordering based on nested dissection (section 5.2.1), which might seem as a better choice due to the reduction of dependencies as shown in Figure 5.1.

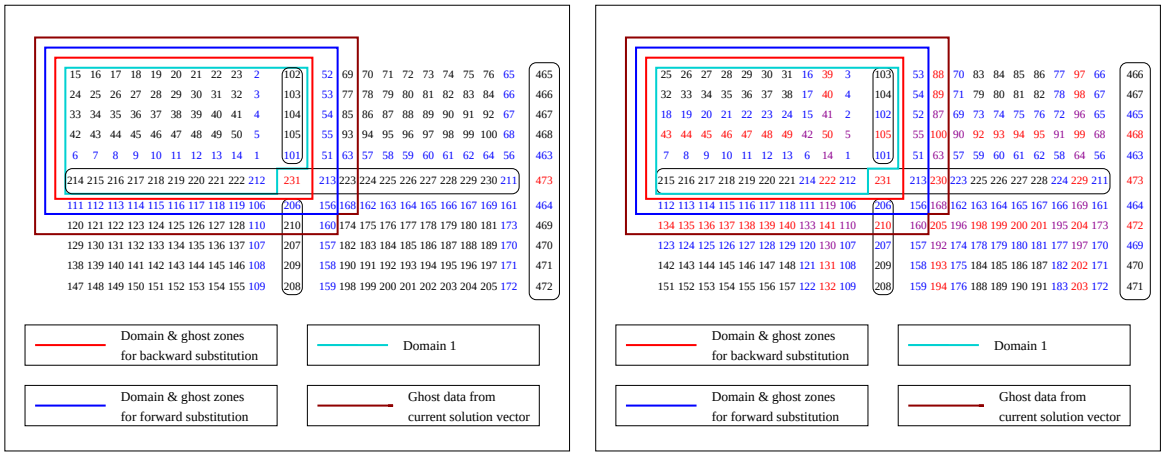
The k-way partitioning assigns to the vertices of every subdomain a set of consecutive distinct indices or numbers, num. The AMML(s) reordering does not change the set of indices assigned to every subdomain, but it changes the order of the vertices within each subdomain. Similarly for nested dissection, where the order of the vertices in the subdomains and separators is altered by AMML(s) reordering.

5.2.1 Nested dissection + AMML(s) reordering of the matrix A

The graph of A is partitioned using nested dissection (section 2.2.2) to obtain P subdomains and $P-1$ separators. We denote the P subdomains' subgraphs by $\Omega_{t,l}$ where $t = log(P)$ levels are performed and $l \leqslant P$. We denote the separators subgraphs introduced at level i of the nested dissection by $\Sigma_{i,j}$ where $j \leqslant 2^{i-1}$, and $i \leqslant t$. We also denote the vertices of the separators and the final subdomains by $\mathbf{S_{i,j}} = \mathbf{V}(\mathbf{\Sigma_{i,j}})$ and $\mathbf{D_l} = \mathbf{V}(\mathbf{\Omega_{t,l}})$ respectively.

Our ND + AMML(s) reordering consists of two algorithms. The first rearranges the vertices of subdomains D_l ($l = 1, 2, ..., P$) obtained in the last level (t^{th} level) of Nested Dissection (Algorithm 45). The second rearranges the vertices of the separators $S_{j,m}$ where j is the level of nested dissection and m is the separator's order within this level of nested dissection ($m = 1, 2, ..., 2^{j-1}$) (Algorithm 46). In this manner, even the reordering can be performed in parallel, where each pro-

cessor reorders its subdomain and then each can reorder a separator. Note that we do not change neither the order of the domains and separators nor the set of indices assigned to each (refer to Appendix D). For example, in nested dissection, the indices 1 till 50 are assigned to the first domain (Figure 5.1). In ND+AMML(s) reordering, the indices 1 till 50 are still assigned to the first domain, however their ordering is changed (Figures 5.2(a) and 5.2(b)).



(a) The graph of the ND+AMML(1) rearranged matrix A with the data needed to compute one matrix-vector multiplication on domain1

(b) The graph of the ND+AMML(2) rearranged matrix A with the data needed to compute one matrix-vector multiplication on domain1

Figure 5.2: Half of an 11-by-43 5-point stencil grid, partitioned into 8 subdomains using 7 separators.

In a classic computation based on nested dissection, the computation on the subdomains is done in parallel, followed by the computation associated with the separators. This requires $\log(P)$ messages to be exchanged during the forward and the backward solves performed at each iteration of a Krylov subspace method. To be able to avoid communication, we first merge the computation of the separators to the subdomains. Therefore, each processor computes a set $\alpha_0^{(j)} = Adj(G(A), D_j) \cap (\cup_{\forall j} S_{j,m}) = Adj(G(A), D_j)$. Without going into details, the algorithm ensures that all the vertices of the separators belong to some $\alpha_0^{(j)}$. For example in Figure 5.2(a), nodes 231 and 473 are added to some $\alpha_0^{(i)}$.

The reordering is designed to isolate as much as possible the sets of vertices $\alpha_0^{(j)}$, for all j, in the graphs of L and U. In other words, the goal is to minimize the number of vertices in the sets $\beta_1^{(j)} = R(G(U), \alpha_1^{(j)})$ and $\gamma_j = R(G(L), \beta_1^{(j)})$. For the U matrix, this means that the set $\beta_1^{(j)}$ should be equal to the set $\alpha_0^{(j)} \cup h_{U,j,1}$, where $h_{U,j,1} = opAdj(G(U), \alpha_0^{(j)})$. The data ghosted represents at most one layer of vertices around $\alpha_0^{(j)}$. For this, the set $h_{U,j,1}$ is numbered with the largest possible numbers. By doing so, in 2D 5-point stencil and 9-point stencil grids, $h_{U,j,1}$ contains at most 4 vertices. For 3D 7-point stencil and 27-point stencil grids, $h_{U,j,1}$ is at most $12 \times (n/P)^{1/3} + 8$ vertices, where we assume in the first case that the subdomain is a cube containing n/P vertices and in the second case that $\alpha_0^{(j)}$ is a cube containing n/P vertices. Similarly, for the L matrix, the

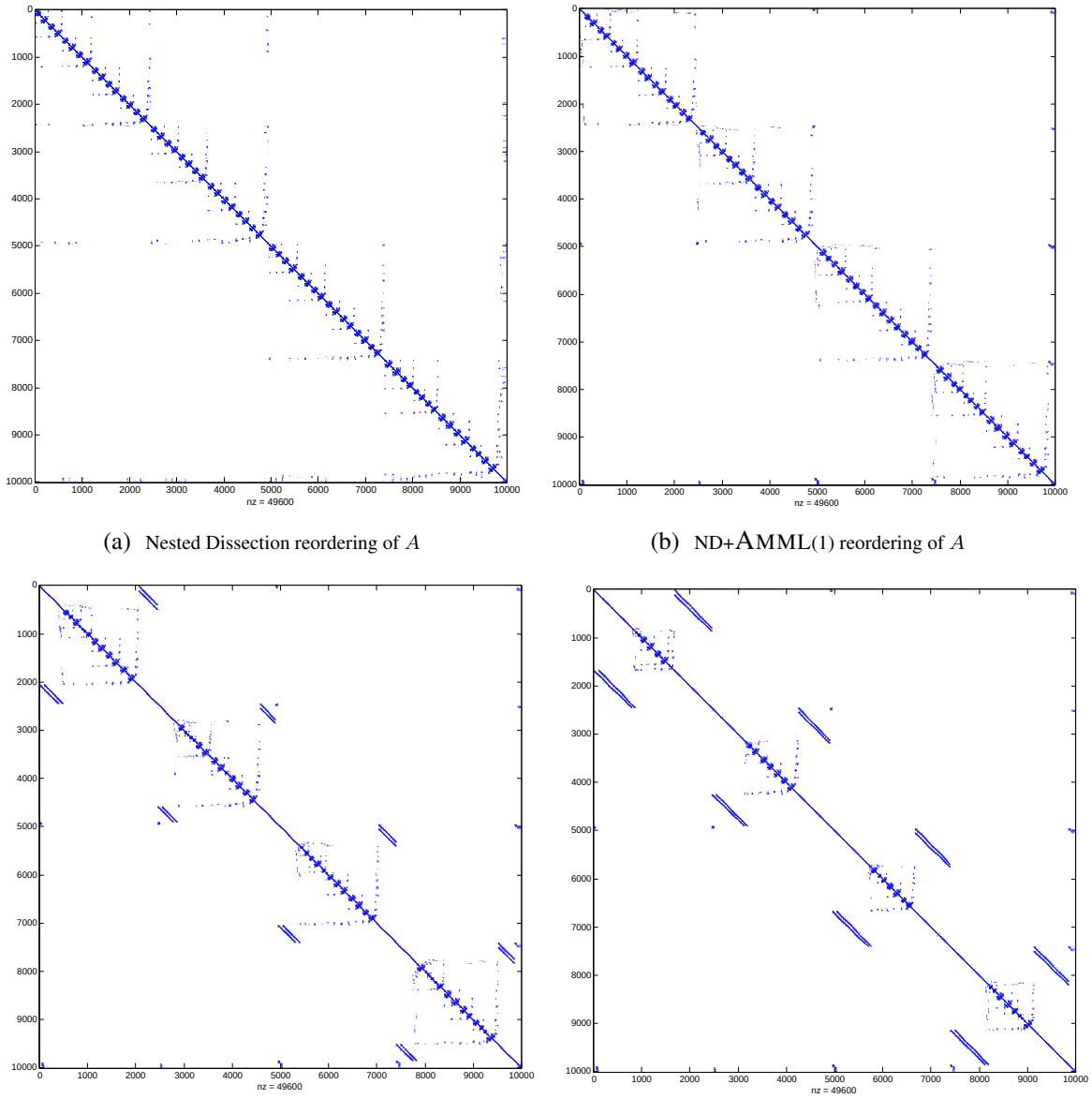
goal is to have the set $\gamma_1^{(j)}$ to be as close as possible to the set $\beta_1^{(j)} \cup h_{L,j,1}$ (if possible equal), where $h_{L,j,1} = opAdj(G(L), \beta_1^{(j)})$. Thus, the set $h_{L,j,1}$ is numbered with the smallest numbers possible. Hence one layer of ghosted data is added around β_j. By generating all these conditions for all $\alpha_0^{(j)}$ with $j = 1, 2, ..., p = 2^t$ and by taking into consideration the structure of a nested dissection graph, the reordering for the subdomains and the separators is presented in Algorithms 45 and 46 respectively, where the parameter s is set to 1. As it can be seen in Figure 5.2(a), this alternating reordering reduces the ghost zones as compared to Figure 5.1. Thus to compute one matrix-vector multiplication of the form $y_i = (LU)^{-1}Ay_{i-1}$ on 8 processors, processor 1 has to fetch one eighth of matrix U, a bit more than one eighth of matrices L and A and of the vector y_{i-1}.

Algorithm 45 ND+AMML(s) Subdomain (D_l, $S_{j,m}$ $\forall j, s, evenodd, num$)

Input: D_l, the set of vertices to be reordered; G(A), the graph of A
Input: $S_{j,m}$ $\forall (j, m)$, the vertices of separators
Input: s, the number of multiplications to be performed without communication
Input: $evenodd$, a tag that can be either even or odd; num, the set of numbers/indices assigned to the vertices D_l
Output: the reordered set of indices assigned to the vertices D_l

if $s == 0$ **then**
  Number D_l in any order, preferably in the natural order.
else
  for $j = 1$ to t **do** Find the vertices $bv_j = D_l \cap Adj_A(S_{j,m})$
  for $j = 1$ to t and $i = 1$ to t, $i \neq j$ **do** Find the vertices $cor_{j,i}$ = $bv_j \cap bv_i$, if they exist.
  for $j = 1$ to t , **do** let $corners_j = \cup_{\forall i}\, cor_{j,i}$
  for $j = 1$ to t **do**
    if evenodd = odd **then** Assign to the unnumbered vertices of bv_j, the smallest numbers in num, num_{bv_j}
    else Assign to the unnumbered vertices of bv_j, the largest numbers in num, num_{bv_j}
    end if
    Remove the numbers num_{bv_j} from num ($num = num - num_{bv_j}$)
    if $corners_j = \phi$ **then** Number the unnumbered vertices of bv_j with the indices num_{bv_j}, in any order.
    else Call **ND+AMML(s) Subdomain** ($\boldsymbol{bv_j}$, $\boldsymbol{Sep_{i,m}}$ $\forall \boldsymbol{i \neq j, s, evenodd, num_{bv_j}}$)
    end if
  end for
  Let $D_l = D_l - \cup_{\forall j} bv_j$
  if evenodd = even **then** Call **ND+AMML(s) Subdomain** ($\boldsymbol{D_l}$, $\boldsymbol{bv_j}$ $\forall \boldsymbol{j, s, odd, num}$)
  else Call **ND+AMML(s) Subdomain** ($\boldsymbol{D_l}$, $\boldsymbol{bv_j}$ $\forall \boldsymbol{j, s-1, even, num}$)
  end if
end if

To perform $s - step$ of the multiplication $y_i = (LU)^{-1}Ay_{i-1}$ in a communication-avoiding



(a) Nested Dissection reordering of A

(b) ND+AMML(1) reordering of A

(c) ND+AMML(5) reordering of A

(d) ND+AMML(10) reordering of A

Figure 5.3: Comparison of the sparsity patterns of the ND, ND+AMML(1), ND+AMML(5) and ND+AMML(10) reordered matrix NH2D of size $10^4 \times 10^4$ with P=4

manner, our goal is to minimize the number of vertices in the sets $\beta_i^{(j)} = R(G(U), \alpha_{i-1}^{(j)})$ and $\gamma_i^{(j)} = R(G(L), \beta_i^{(j)})$, for $i = 1, 2, .., s$, $j = 1, .., 2^t$ where $\alpha_0^{(j)} = Adj(G(A), D_j)$ and for $i > 0$, $\alpha_i^{(j)} = Adj(G(A), \gamma_i^{(j)})$. We perform the same analysis as for the case of $s = 1$, but for $s - steps$. The set $h_{U,j,i} = opAdj(G(A), \alpha_{i-1}^{(j)})$ is numbered with the largest possible numbers, and

Algorithm 46 ND+AMML(s) Separator (S_{i,m_0}, $S_{j,m}$ $\forall(j,m)$,s,num)

Input: $S_{i,m}$, the set of vertices to be reordered; G(A), the graph of A
Input: D_o $\forall o$, the vertices of the subdomains; $S_{j,m}$ $\forall(j,m) \neq (i,m)$, the vertices of separators
Input: s, the number of multiplications to be performed without communication
Input: $evenodd$, a tag that can be either even or odd; num, the set of numbers/indices assigned to the vertices $S_{i,m}$
Output: the reordered set of indices assigned to the vertices $S_{i,m}$

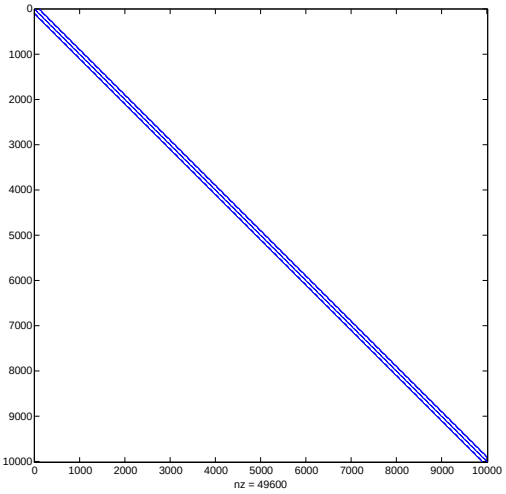
1: Find all the interacting separators of $S_{i,m}$, iS_j where $j = 1, 2, ..., s$.
iS_j is the set of all boundary separators of D_o, $bS_{o,j} = Adj(G(A), D_o) \cap S_{j,m}$, where there is at least one vertex, $vert \in bS_{o,j}$, such that $vert \in Adj_A(S_{i,m})$.

$$iS_j = \{bS_{o,j}\ \forall o, s.t. \exists vert \in bS_{o,j}\ and\ vert \in Adj_A(S_i(m))\}$$

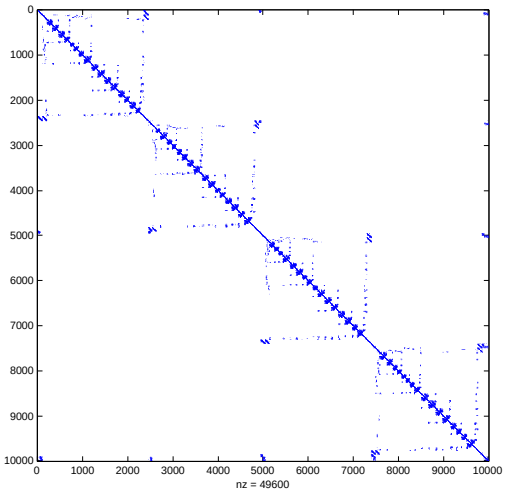
2: **for** $j = 1$ to s, Find the set of vertices $int(i,m,j) = \begin{cases} S_i(m) \cap Adj_A(iS_j) & if\ iS_j \nsubseteq S_i(m) \\ S_i(m) \cap opAdj_A(iS_i) & if\ iS_j \subseteq S_i(m) \end{cases}$
3: Number the set $first = \{int(i,m,j); \forall\ j < i\}$ with the smallest numbers in num, num_1 and let $num = num - num_1$
4: **for** $j = i, i+1, ..., s$
5: **If** for some $k < i$, $comm = int(i,m,j) \cap int(i,m,k) \neq \phi$ **then** Let $last1 = last1 \cup \{opAdj_A(comm), \forall comm \neq \phi\}$
6: **end if**
7: Number $last1$ with the largest numbers in num, num_{last1} and let $num = num - num_{last1}$
8: **end for**
9: Find the set $last2 = \{v \in int(i,m,j), \forall j > i \ \ \& \ \ v \notin int(i,m,j), \forall j \leqslant i\}$ and number it with the largest numbers in num, num_{last2}
10: Number the set of vertices $near = S_i(m) \cap opAdj_A(\ last1 \cup last2\ )$ with the smallest numbers in num, num_2 and let $num = num - num_2 - num_{last2}$
11: Let $bSep = \{near \cup last2 \cup last1 \cup first\}$
12: Let $Block = S_i(m) - bSep$
13: Call **ND+AMML(s) Subdomain** ($Block$, $bSep$, $s-1$, odd, num)

$h_{L,j,i} = opAdj(G(A), \beta_{j,i})$ is numbered with the smallest possible numbers, for $i = 1$ till s. This leads to $2s-1$ alternating layers reordering from the separators as shown in Figure 5.2(b), where $s = 2$. Figure 5.3 shows the sparsity pattern of the matrix NH2D when reordered using ND, ND + AMML(1), ND + AMML(5), and ND + AMML(10) with $P = 4$. The reason AMML(s) reordering avoids communication in the ILU matrix powers kernel is that it reduces the $\beta_i^{(j)}$, and $\gamma_i^{(j)}$ sets. In matrix format, this is equivalent to reducing the fill-ins in the U^{-1} and L^{-1} matrices, specifically the off-block-diagonal entries, as shown in Figure 5.4 where NH2D is a symmetric matrix.

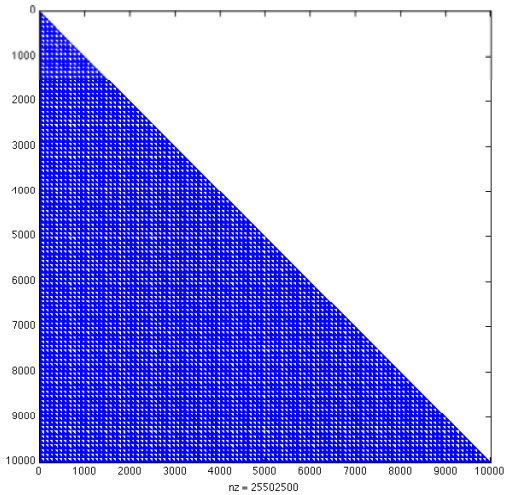
Algorithm 45 takes as input the graph of A, the vertices of the subdomain to be rearranged, the vertices of the separators. Note that to reorder a given subdomain D_j, the algorithm needs one separator from each level of nested dissection, specifically the separator which was part of a



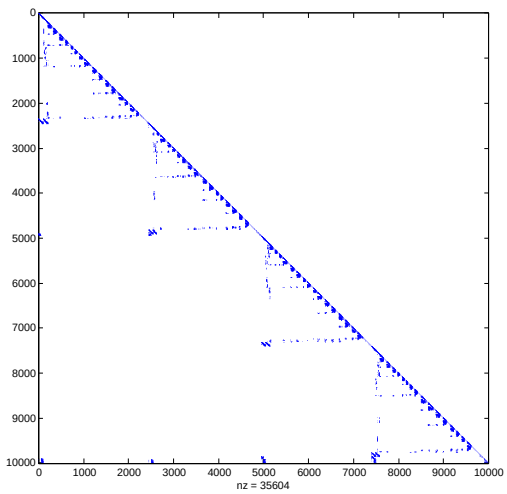
(a) 2D matrix A of size $10^4 \times 10^4$



(b) A_{ca}, ND + AMML(2) reordering of A



(c) L^{-1} matrix from the ILU(0) factorization of A



(d) L_{ca}^{-1} matrix from the ILU(0) factorization of A_{ca}

Figure 5.4: Comparison of the fill-ins in the L^{-1} obtained from the ILU(0) factorization of the matrix $A =$NH2D and its ND+AMML(2) reordered version A_{ca}, where $U = L^t$

parent subdomain. The algorithm also takes as input s, the number of steps to be performed, and $evenodd$ which defines in which order we want to number our nodes "first, last, first,.. " (odd) or "last, first, last, first, ..." (even). Note that the first call to the algorithm to reorder a subdomain, the initial parameters are set to $evenodd = odd$ and num to be the set of indices assigned to the subdomain by nested dissection. Algorithm 45 is a recursive algorithm that starts by looping over the separators $S_{j,m}$ and finding their adjacent sets in D_l, bv_j. The aim is to number the bv_j's first

(smallest indices) or last (largest indices) depending on the $evenodd$ tag which specifies if we are reducing $\beta_{j,k}$ or $\gamma_{j,k}$ ($k = 1, .., s$). In case some other separator $S_{i,m}$ depends on some vertices of bv_j ($corners_j \neq \phi$) then we treat bv_j as a block, its separators being $S_{i,m}$, where $i \neq j$. Then Algorithm 45 is called recursively to limit the size of $\beta_{i,k}$ or $\gamma_{i,k}$. In case there is no separator that depends on bv_j, then we number it in any order. Finally, Algorithm 45 is called recursively on the remaining part of the subdomain $D_l - \cup_{\forall j} bv_j$ with the separators being bv_j, the appropriate value of $evenodd$ and s.

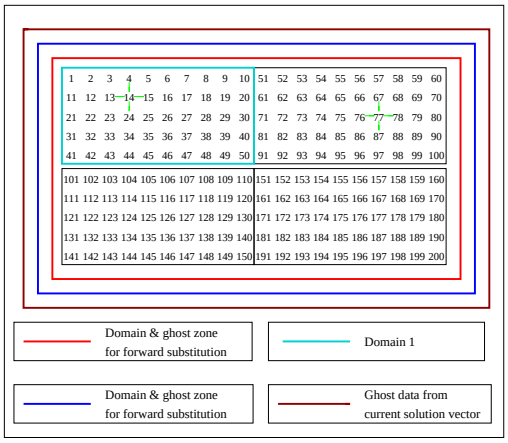
Algorithm 46 takes as input, the graph of A, the vertices of the separator to be rearranged S_{i,m_0}, the vertices of other separators $S_{j,m}$, and s. The aim is to find the vertices of S_{i,m_0} that belong to $h_{U,o,1}$ and $h_{L,o,1}$ for all o. Then the algorithm numbers $h_{U,o,1}$ last, $opAdj(G(A), h_{U,o,1})$ first, and $h_{L,o,1}$ first. This is done by looping over the separators from the same level j, iS_j that interact with S_{i,m_0} rather than the subdomains $\alpha_0^{(o)}$. And we find $int(i, m, j)$, where $\cup_{\forall j} int(i, m, j) = \cup_{\forall o}(h_{U,o,1} \cup h_{L,o,1})$. After finding the vertices $int(i, m, j)$ and numbering them accordingly with $opAdj(G(A), last1 \cup last2)$ numbered last. In this way the vertices have been numbered for performing 1 step with no communication. Then Algorithm 45 is called to rearrange the remaining vertices of S_{i,m_0} alternatively.

5.2.2 K-way + AMML(s) reordering of the matrix A

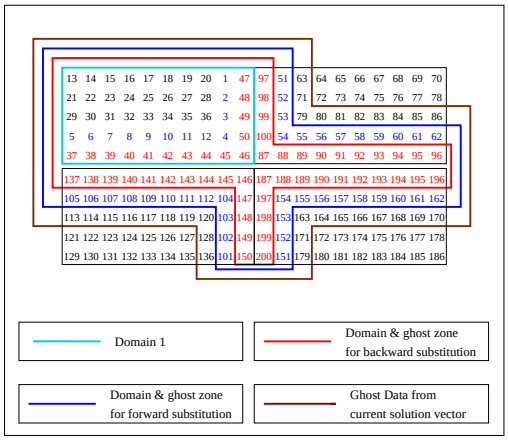
The graph of A is partitioned using k-way graph partitioning (section 2.2.3), to obtain P non-overlapping subdomains Ω_i for $i = 1, 2, ..., P$. Then, we introduce in this section the AMML(s) reordering based on k-way, which differs slightly than the ND+AMML(s) reordering presented in section 5.2.1.

We present two versions of k-way+AMML(s) reordering, Algorithms 47 and 48. Both algorithms take as input the graph of A, the vertices of the subdomain to be reordered $\overline{\alpha}^{(j)} = \overline{\alpha}_0^{(j)} = V(\Omega_j)$, D the set of d neighboring subdomains $\overline{\alpha}^{(i)}$ that depend on $\overline{\alpha}^{(j)}$ (there exists at least one directed edge connecting a vertex in $\overline{\alpha}^{(i)}$ to a vertex in $\overline{\alpha}^{(j)}$ in the graph of A) , the number of steps s to be performed, the set of indices num assigned to $\overline{\alpha}^{(j)}$, and $evenodd$ which defines in which order we want to number our nodes "first, last, first,.. " (odd) or "last, first, last, first, ..." (even). During the first call to the algorithm to reorder a subdomain, the initial parameters are set to $evenodd = even$ and num to the set of indices assigned to the subdomain by k-way partitioning. Note that, in this section we use the term $\overline{\alpha}_0^{(j)}$ to denote $V(\Omega_j)$ rather than $\alpha_0^{(j)}$ which is used in section 5.1. The reason is that after reordering some vertices of $\overline{\alpha}_0^{(j)}$, the remaining vertices $\overline{\alpha}_1^{(j)}$ are a subset of $\overline{\alpha}_0^{(j)}$ ($\overline{\alpha}_s^{(j)} \subset ... \subset \overline{\alpha}_1^{(j)} \subset \overline{\alpha}_0^{(j)}$). Whereas in section 5.1, $\alpha_1^{(j)} = \delta_1^{(j)}$ is a superset of $\alpha_0^{(j)}$ ($\alpha_0^{(j)} \subset \alpha_1^{(j)} \subset ... \subset \alpha_s^{(j)}$).

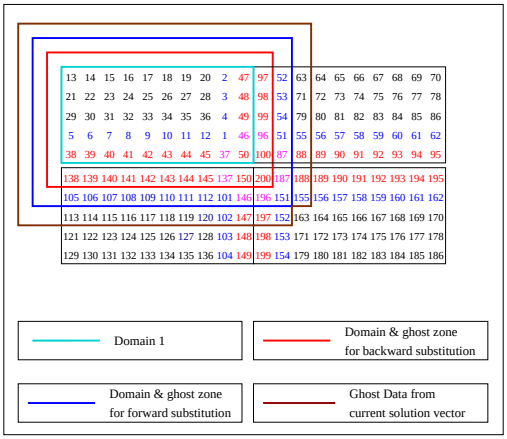
We explain first the reordering for applying the ILU(0) preconditioner during one iteration of a Krylov subspace solver ($s = 1$), where the goal is to reduce the number of vertices in the sets $\beta_1^{(i)} = R(G(U), \overline{\alpha}_0^{(i)})$ and $\gamma_1^{(i)} = R(G(L), \beta_1^{(i)})$ for each subdomain Ω_i, where $\overline{\alpha}_0^{(i)} = V(\Omega_i)$. In the following, the figures 5.5(c) and 5.5(b) are used to explain the alternating reordering for one



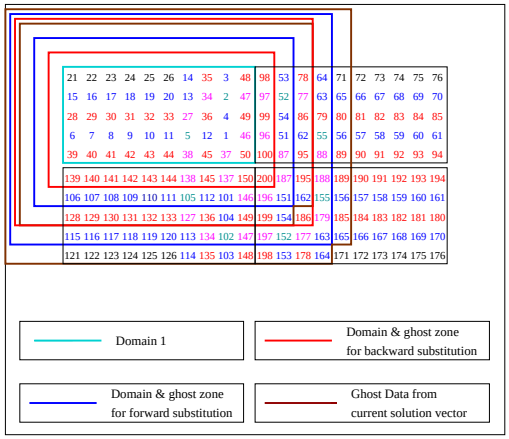
(a) Natural reordering



(b) Numbering the $bv_i^{(U)}$ and $bv_i^{(L)}$ with the largest and smallest indices, without any special reordering within each



(c) AMML(1) reordering



(d) AMML(2) reordering

Figure 5.5: Data needed to compute $y_i = (LU)^{-1}Ay_{i-1}$ on Domain 1 using ILU Matrix Powers Kernel for different reorderings where $i = 1$ in figures (a), (b) and (c) and $i = 1, 2$ in figure (d).

step ($s = 1$), while figure 5.5(d) is used to display the reordering for two steps ($s = 2$). To reduce globally the number of reachable vertices of interest in the graphs of L and U, the alternating reordering renumbers the vertices of each subdomain Ω_j, such that locally on this subdomain the set of reachable vertices $\beta_1^{(i)} \cap \overline{\alpha}_0^{(j)}$ and $\gamma_1^{(i)} \cap \overline{\alpha}_0^{(j)}$ from all the other subdomains Ω_i is reduced. To do so, Algorithms 47 and 48 identify first the boundary vertices of each neighboring subdomain Ω_i in subdomain Ω_j, $bvU_0^{(j,i)} = \overline{\alpha}_0^{(j)} \cap Adj(G(A), \overline{\alpha}_0^{(i)})$ and assign to the sets $bvU_0^{(j,i)} = \overline{\alpha}_0^{(j)} \cap Adj(G(A), \overline{\alpha}_0^{(i)})$ the largest possible numbers in num. Then the algorithms identify the adjacent vertices of $bvU_0^{(j,i)}$ in the graph of A, $bvL_0^{(j,i)} = \overline{\alpha}_0^{(j)} \cap opAdj(G(A), bvU_0^{(j,i)})$, and assign to these

sets the smallest numbers possible in num. Figure 5.5(b) displays the reordered graph obtained after this reordering, where the vertices of the sets $bvU_0^{(j,i)}$ and $bvL_0^{(j,i)}$ are kept in their natural ordering. The set $\beta_1^{(i)}$ is the set of vertices bounded by the red polygon and the set $\gamma_1^{(i)}$ is the set of vertices bounded by the blue polygon for $i = 1$. In the worst case, the reachable set $\beta_1^{(i)}$ is equal to the union of the set $\overline{\alpha}_0^{(i)}$ with all the sets $bvU_0^{(j,k)}$ for $i \neq j$ and $j \neq k$. Similarly, the reachable set $\gamma_1^{(i)}$ is equal to $\beta_1^{(i)}$ and all the $bvL_0^{(j,k)}$.That is,

$$\beta_1^{(i)} \subseteq \overline{\alpha}_0^{(i)} \cup \bigcup_{\substack{j=1 \\ j\neq i}}^{P} bvU_0^{(j)} \quad \text{and} \quad \gamma_1^{(i)} \subseteq \beta_1^{(i)} \cup \bigcup_{\substack{j=1 \\ j\neq i}}^{P} bvL_0^{(j)}$$

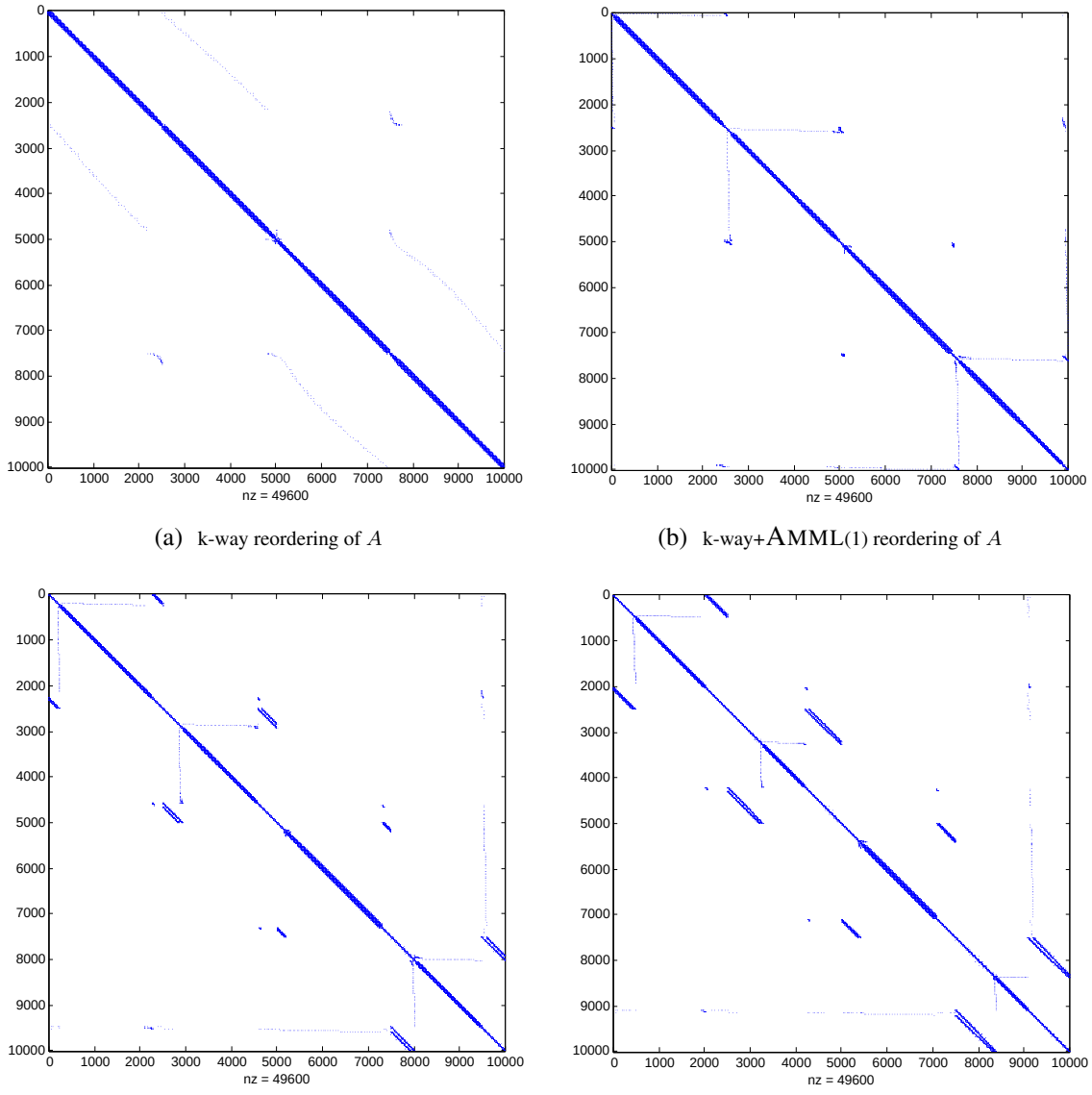
where $bvU_0^{(j)} = \cup_{\substack{k=1 \\ k\neq j}}^{P} bvU_0^{(j,k)}$ and $bvL_0^{(j)} = \cup_{\substack{k=1 \\ k\neq j}}^{P} bvL_0^{(j,k)}$. However, for each subdomain Ω_i, the reachable sets can be further reduced by reordering the vertices within the sets $bvU_0^{(j,i)}$ and $bvL_0^{(j,i)}$, for all the neighboring subdomains Ω_j. Algorithms 47 and 48 differ only in the approach used for reordering the vertices within the sets $bvU_0^{(j,i)}$ and $bvL_0^{(j,i)}$.

The remaining numbers in num are assigned to the remaining vertices $\overline{\alpha}_1^{(j)} = \overline{\alpha}_0^{(j)} - bvL_0^{(j)} - bvU_0^{(j)}$, where the $\alpha_1^{(j)}$ vertices are kept in their natural ordering, the $bvU_0^{(j)}$ vertices are the vertices in subdomain Ω_j that all the other subdomains $i \neq j$ depend on and $bvL_0^{(j)} = Adj(G(A), bvU_0^{(j)})$. Then we get,

$$\beta_1^{(i)} \subset Adj(G(A), \overline{\alpha}_0^{(i)}) \cup \zeta \quad \text{and} \quad \gamma_1^{(i)} \subset Adj(G(A), \beta_1^{(i)}) \cup \zeta^*$$

where $\beta_1^{(i)} = R(G(U), \overline{\alpha}_0^{(i)})$, $\gamma_1^{(i)} = R(G(L), \beta_1^{(i)})$, $|\zeta| \ll |Adj(G(A), \overline{\alpha}_0^{(i)})|$ and $|\zeta^*| \ll |Adj(G(A), \beta_1^{(i)}|$. The sets ζ and ζ^* represent additional vertices that belong to the reachable set in addition to the adjacent set. For example, in figure 5.5(c), the vertices 200 and 151 belong to the sets $\beta_1^{(1)} = R(G(U), \overline{\alpha}_0^{(1)})$ and $\gamma_1^{(1)} = R(G(L), \beta_1^{(1)})$ respectively. However, these vertices do not belong to the sets $Adj(G(A), \overline{\alpha}_0^{(1)})$ and $Adj(G(A), \beta_1^{(1)})$. Note that for matrices arising from 1D 3-point stencil, 2D 9-point stencil, and 3D 27-point stencil discretizations, we have $\zeta = \zeta^* = \phi$.

When computing $s > 1$ multiplications of the form $y_i = (LU)^{-1}Ay_{i-1}$ for $i = 1, 2, .., s$, the goal of the alternating reordering is to reduce the number of vertices not only in the sets $\beta_1^{(j)}$ and $\gamma_1^{(j)}$, but also in the sets $\beta_i^{(j)}$ and $\gamma_i^{(j)}$, for $i = 1, 2, .., s$. Thus we perform the same analysis as for the case of $s = 1$. We obtain a recursive reordering on the given set of vertices $\overline{\alpha}_0^{(j)}$ such that the two layers $bvU_0^{(j,i)} = \overline{\alpha}_0^{(j)} \cap Adj(G(A), \overline{\alpha}_0^{(i)})$ and $bvL_0^{(j,i)} = \overline{\alpha}_0^{(j)} \cap opAdj(G(A), bvU_0^{(j,i)})$ for all $i \neq j$ are assigned with the largest and smallest numbers respectively. The remaining numbers are assigned to the unnumbered vertices $\overline{\alpha}_1^{(j)} = \overline{\alpha}_0^{(j)} - bvU_0^{(j)} - bvL_0^{(j)}$. But unlike the case of $s = 1$, the vertices $\overline{\alpha}_1^{(j)}$ are reordered recursively, to minimize the cardinality of the sets $\beta_i^{(j)}$ and $\gamma_i^{(j)}$, for $i = 2, 3.., s$. First $bvU_1^{(j,i)} = \overline{\alpha}_1^{(j)} \cap opAdj(G(A), bvL_0^{(j,i)})$ and $bvL_1^{(j,i)} = \overline{\alpha}_1^{(j)} \cap opAdj(G(A), bvU_1^{(j,i)})$ for all $i \neq j$, are assigned with the largest and smallest numbers respectively. Then if $s = 2$, the



(a) k-way reordering of A

(b) k-way+AMML(1) reordering of A

(c) k-way+AMML(5) reordering of A

(d) k-way+AMML(10) reordering of A

Figure 5.6: Comparison of the sparsity patterns of the k-way, k-way+AMML(1), k-way+AMML(5) and k-way+AMML(10) reordered matrix NH2D of size $10^4 \times 10^4$ with P=4

remaining numbers are assigned to $\alpha_2^{(j)} = \overline{\alpha}_1^{(j)} - bvU_1^{(j)} - bvL_1^{(j)}$, where the order of the vertices $\alpha_2^{(j)}$ is unchanged. If $s > 2$ then $\alpha_2^{(j)}$ is also reordered recursively for $s-2$ to minimize the cardinality of the sets $\beta_i^{(j)}$ and $\gamma_i^{(j)}$ for $i = 3, 4, .., s$. The reordering is performed in s recursive steps. At each step, two layers ($bvU_t^{(j)} = \cup_{\substack{i=1 \\ i \neq j}}^{P} bvU_t^{(j,i)}$ and $bvL_t^{(j)} = \cup_{\substack{i=1 \\ i \neq j}}^{P} bvL_t^{(j,i)}$) are reordered,

Algorithm 47 K-way+AMML(s) Subdomain ($\overline{\alpha}^{(j)}$, D, s, $evenodd$, $G(A)$, num)

Input: $\overline{\alpha}^{(j)}$, the set of vertices of the subdomain to be reordered; G(A), the graph of A
Input: $D = \{\overline{\alpha}^{(i)} | Adj(G(A), \overline{\alpha}^{(i)}) \cap \overline{\alpha}^{(j)} \neq \phi, i = 1 : P, i \neq j\}$, set of d neighboring subdomains; s, the number of steps to be performed in the ILU matrix powers kernel
Input: $evenodd$, a tag that can be 'even' or 'odd'; num, set of indices assigned to $\overline{\alpha}^{(j)}$
Let $d = |D|$ be the cardinality of the set D
if $s == 0$ **then** Number $\overline{\alpha}^{(j)}$ in any order
else
  for $i = 1$ to d **do** Find the vertices $bv_i = \overline{\alpha}^{(j)} \cap Adj(G(A), \overline{\alpha}^{(i)})$ **end for**
  for $i = 1$ to d **do** let $corners_i = \cup_{k=1}^{d} (bv_i \cap bv_k)$ **end for**
  for $i = 1$ to d **do**
    if evenodd = odd **then** Assign to the unnumbered vertices of bv_i, the smallest numbers in num, num_{bv_i}
    else Assign to the unnumbered vertices of bv_i, the largest numbers in num, num_{bv_i}
    end if
    Remove the numbers num_{bv_i} from num ($num = num - num_{bv_i}$)
    if $corners_i = \phi$ **then** Number the unnumbered vertices of bv_i with the indices num_{bv_i}, in any order
    else Call K-way+AMML(s) Subdomain ($\boldsymbol{bv_i}$, $\boldsymbol{D}$, $\boldsymbol{s}$, $\boldsymbol{evenodd}$, $\boldsymbol{G(A)}$,$\boldsymbol{num_{bv_j}}$)
    end if
  end for
  Let $\overline{\alpha}^{(j)} = \overline{\alpha}^{(j)} - \cup_{i=1}^{d} bv_i$
  if evenodd = even **then** Call K-way+AMML**(s)** Subdomain ($\overline{\boldsymbol{\alpha}}^{(j)}$, $\{\boldsymbol{bv_i} \mid \boldsymbol{i = 1 : d}\}$, $\boldsymbol{s}$, $\boldsymbol{odd}$, $\boldsymbol{G(A)num}$)
  else Call **K-way+**AMML**(s) Subdomain** ($\overline{\boldsymbol{\alpha}}^{(j)}$, $\{\boldsymbol{bv_i} \mid \boldsymbol{i = 1 : d}\}$, $\boldsymbol{s - 1}$, $\boldsymbol{even}$, $\boldsymbol{G(A)}$, $\boldsymbol{num}$)
  end if
end if

and this produces an alternating min-max $2s$ layers reordering. In figure 5.5(d) where the graph is reordered for performing two multiplications, there are four alternating layers starting from the boundary vertices in every subdomain. Note that, similarly to the case of $s = 1$, the vertices of $bvU_t^{(j,i)}$ and $bvU_t^{(j,i)}$ for $t = 0, 1, ..., s-1$ have to be reordered to reduce the addition of unnecessary vertices to the reachable sets.

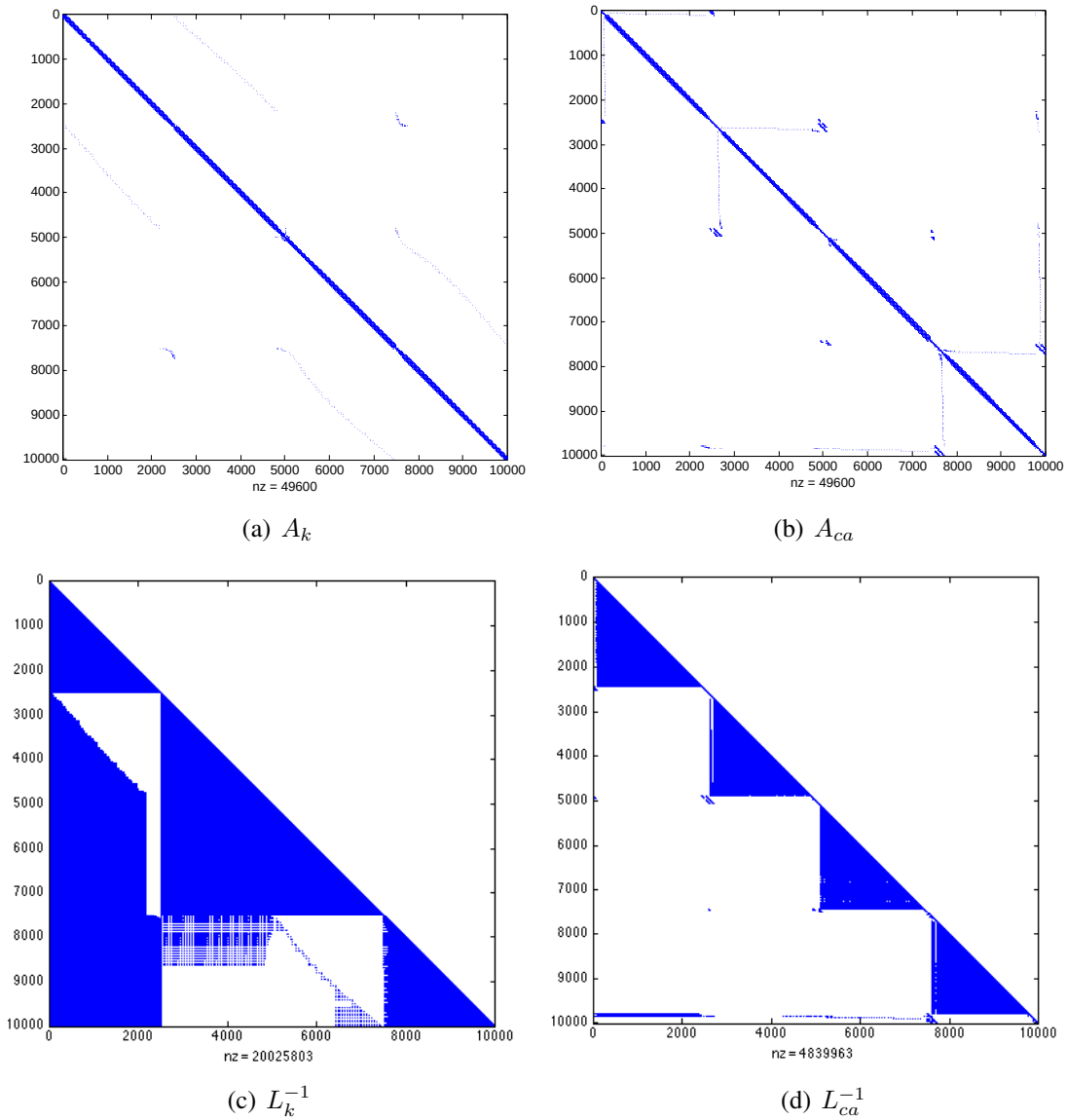
At each recursive call in the Algorithms 47 and 48, either $bvU_t^{(j,i)}$ or $bvL_t^{(j,i)}$, denoted by bv_i, is reordered. The tag $evenodd$ is used to decide which one to reorder. If $evenodd = even$, then the largest available numbers in the set num are assigned to $bvU_t^{(j,i)}$. Otherwise, the smallest numbers in the set num are assigned to $bvL_t^{(j,i)}$. In algorithm 47, the $bvU_t^{(j,i)}$ and the $bvL_t^{(j,i)}$ are reordered by calling the algorithm recursively to ensure an alternating reordering within each. In case the vertices of $bvU_t^{(j,i)}$ ($bvL_t^{(j,i)}$) do not belong to any other $bvU_t^{(j,k)}$ ($bvL_t^{(j,k)}$) where $k \neq i$, i.e. $corners_j = \phi$, then the vertices of $bvU_t^{(j,i)}$ ($bvL_t^{(j,i)}$) are kept in their natural ordering. The

reordering for one and two steps is shown in figures 5.5(c) and 5.5(d). The only difference in algorithm 48 is that we let $bvU_t^{(j)} = \cup_{\substack{i=1 \\ i\neq j}}^{P} bvU_t^{(j,i)}$ and $bvL_t^{(j)} = \cup_{\substack{i=1 \\ i\neq j}}^{P} bvL_t^{(j,i)}$, denoted by bv_j, and then reorder each using nested dissection. Since nested dissection assigns to the vertices of the separators larger numbers than the two subdomains, and then continues partitioning each till the final subdomains are very small, then the obtained reordering of the $bvU_t^{(j)}$ and $bvL_t^{(j)}$ is very similar to an alternating reordering. If $s > 1$, the remaining vertices of $\overline{\alpha}^{(j)}$ are reordered recursively as shown in Algorithms 47 and 48.

Algorithm 48 K-way + AMML**(s) Subdomain V2 ($\overline{\boldsymbol{\alpha}}^{(j)}$, $\boldsymbol{D}$, $\boldsymbol{s}$, $\boldsymbol{evenodd}$, $\boldsymbol{G(A)}$, $\boldsymbol{num}$)**

Input: $\overline{\alpha}^{(j)}$, the set of vertices of the subdomain to be reordered; G(A), the graph of A
Input: $D = \{\overline{\alpha}^{(i)} | Adj(G(A), \overline{\alpha}^{(i)}) \cap \overline{\alpha}^{(j)} \neq \phi, i = 1 : P, i \neq j\}$, set of d neighboring subdomains; s, the number of steps to be performed in the ILU matrix powers kernel
Input: $evenodd$, a tag that can be 'even' or 'odd'; num, set of indices assigned to $\overline{\alpha}^{(l)}$
1: Let $d = |D|$ be the cardinality of the set D
2: **if** $s == 0$ **then** Number $\overline{\alpha}^{(j)}$ in any order
3: **else**
4: **for** $i = 1$ to d **do** Find the vertices $bv_i = \overline{\alpha}^{(j)} \cap Adj(G(A), \overline{\alpha}^{(i)})$
5: Let $bv_j = \cup_{i=1}^{d} bv_i$
6: **if** evenodd = odd **then** Assign to the vertices of bv_j, the smallest numbers in num, num_{bv_j}
7: **else** Assign to the vertices of bv_j, the largest numbers in num, num_{bv_j}
8: **end if**
9: Remove the numbers num_{bv_j} from num ($num = num - num_{bv_j}$)
10: Reorder bv_j using Nested Dissection to obtain an alternating reordering
11: Let $\overline{\alpha}^{(j)} = \overline{\alpha}^{(j)} - bv_j$
12: **if** evenodd = even **then** Call **K-way+**AMML**(s) Subdomain V2** ($\overline{\boldsymbol{\alpha}}^{(j)}$, $\boldsymbol{bv_j}$, $\boldsymbol{s}$, $\boldsymbol{odd}$, $\boldsymbol{G(A)}$, $\boldsymbol{num}$)
13: **else** Call **K-way+**AMML**(s) Subdomain V2** ($\overline{\boldsymbol{\alpha}}^{(j)}$, $\boldsymbol{bv_j}$, $\boldsymbol{s-1}$, $\boldsymbol{even}$, $\boldsymbol{G(A)}$, $\boldsymbol{num}$)
14: **end if**
15: **end if**

Figure 5.6 shows the sparsity pattern of the k-way, k-way+AMML(1), k-way+AMML(5), and k-way+AMML(10) reordered A. Similarly to ND+AMML(s) reordering, the reason that the k-way+AMML(s) reordering reduces communication is that it reduces the reachable sets in the L and U matrices. In matrix format, this is equivalent to reducing the fill-ins in the L^{-1} and U^{-1} matrices. As shown in Figure 5.7, the L_{ca}^{-1} has an almost block diagonal format. The difference between the L_{ca}^{-1} obtained from the ND+AMML(s) reordered A (Figure 5.4) and that from the k-way+AMML(s) reordered A is that the first further reduces the fill-ins, specifically in the block diagonals. This is due to the fact that Metis's nested dissection performed much more than $t = 2$ levels of Nested Dissection. Thus it would be a good to reorder the vertices $\overline{\alpha}_j$ in line 2 of Algorithms 47 and 48 using nested dissection or some other reordering that reduced the fill-ins of the block diagonal entries in L^{-1} and U^{-1}. However, in our tests we keep the same ordering as obtained from k-way



(a) A_k (b) A_{ca}

(c) L_k^{-1} (d) L_{ca}^{-1}

Figure 5.7: The fill-ins in the inverse of L obtained from the ILU(0) factorization of a k-way reordered matrix A and its AMML(2) reordered version A_{ca} where $U^{-1} = (L^{-1})^t$

partitioning.

5.2.3 Complexity of AMML(s) Reordering

We define the complexity of our Alternating Min-Max Layers (AMML(s)) reordering as being the number of times the vertices and the edges of the graph of A are visited in order to perform

the reordering. In section 5.2.3.1 and 5.2.3.2 we show that ND+AMML(s) and k-way+AMML(s) reordering are both of linear complexity.

5.2.3.1 ND+AMML(s)

Recall that nested dissection partitions the graph of A into P subgraphs $\{\Omega_1, \Omega_2, ..., \Omega_P\}$ and $P-1$ separators $\Sigma_{i,m}$. Let $D_l = V(\Omega_l)$ and $S_{i,m} = V(i,m)$.

We start by finding the complexity of rearranging the vertices D_l for $s - steps$. To define the alternating layers from the separators, we will have to read a maximum of $\sum_{\forall i} |R(G(A), bv_i, 2s - 2) \cap D_l|$ vertices and their edges where $bv_i = Adj(G(A), S_{i,m}) \cap D_l$. Thus, at most D_l vertices and their corresponding $E(\Omega_l)$ edges are visited.

In case $corners_i \neq \phi$, we will have to read the bv_i vertices and its edges again. So we will have to read some fraction of $|\cup_{\forall i} R(G(A), bv_i, 2s-2)|$ nodes and its edges. In the worst case, we may assume that $corners(i) \neq \phi, \forall i$. Thus, we might read up to $\sum_{\forall i} |R(G(A), bv_i, 2s-2)|$ nodes and their edges.

So in D_l we will read at most $2|D_l| << 2|D_{max}|$ vertices and their edges, where $D_{max} = V(\Omega_{max})$, $|D_{max}| > |D_l|$, for all $l = 1, 2, .., P$ and $E(\Omega_{max}) > E(\Omega_l)$, for all l.

But before that, we need to read the separators' vertices $S_{i,m}, \forall i$ and find $bvi = Adj(G(A), S_i) \cap D_l$. In other words, we will visit the vertices $S_{i,m}, \forall i$ and their edges. Each subdomain will read $log(P) = t$ separators where each separator is from a different level of nested dissection and $P = 2^t$ is the total number of subdomains.

So in total, $S_{1,m}$ will be read P times, $S_{2,m}$ will be read $\frac{P}{2}$ times,..., $S_i(m)$ will be read $\frac{P}{2^{i-1}}$ times ($\sum_{\forall i=1}^{t} \frac{P}{2^{i-1}} \sum_{\forall m} |S_{i,m}|$).

Let $S_{max} = V(\Sigma_{max})$ where $|S_{max}| > |Si, m|$ for all (i, m) and $|E(\Sigma_{max})| > |E(\Sigma_{i,m})|$ for all (i, m). Then

$$\sum_{\forall i=1}^{t} \frac{P}{2^{i-1}} \sum_{\forall m} |S_{i,m}| < tP|S_{max}|$$

Thus to rearrange all the P subdomains Ω_l we will need to visit/read at most $2P|D_{max}| + tP|S_{max}|$ vertices and $2P|E(\Omega_{max})| + tP|E(\Sigma_{max})|$ edges.

As for the complexity of rearranging $S_{i,m}$, we have to read the interacting separators vertices $iSep(j), \forall j \neq i$ and their edges. Based on the nested dissection structure, $S_{i,m}$ can interact at most with one separator from each level. Then, we have to read the separator itself to rearrange it for $s - steps$. So in total, to rearrange $S_{i,m}$ we have to read $log(P) = t$ separators where each separator is from a different level of nested dissection and $P - 1 = 2^t - 1$ is the total number of separators.

Thus, for rearranging all the $P - 1$ separators, $S_{1,m}$ will be read $p - 1$ times, $S_{2,m}$ will be read $\frac{P-2}{2} + 1$ times,..., $S_{i,m}$ will be read $\frac{P-2^{i-1}}{2^{i-1}} + (i-1) = \frac{P}{2^{i-1}} + (i-2)$ times.

The total number of vertices to be read for rearranging all the separators' vertices can be expressed as:

$$\begin{aligned}
\sum_{\forall i=1}^{t}(\frac{P}{2^{i-1}}+i-2)\sum_{\forall m}|S_{i,m}| &< \sum_{\forall i=1}^{t}(\frac{P}{2^{i-1}}+i-2)2^{i-1}|S_{max,m_{max}}| \\
&< |S_{max}|(tP+\sum_{\forall i=1}^{t}(i-2)2^{i-1}) \\
&< |S_{max}|(tP+3+(t-3)2^t) \\
&< |S_{max}|(tP+3+(t-3)P) \\
&< |S_{max}|(2tP+3-3P))
\end{aligned}$$

where $|E(\Sigma_{max})|(etp-3p+3)$ edges are read.

In total, for rearranging the matrix A we will have to read/visit at most $2P|D_{max}|+(3tP-3P+3)|S_{max}|$ vertices and $2P|E(\Omega_{max})|+(3tP-3P+3)|E(\Sigma_{max})|$ edges. Since the ND+AMML(s) reordering is done in parallel on P processors, then this complexity is divided by P.

The parallel complexity of the ND+AMML(s) rearrangement is less than $2|D_{max}|+(3t-2)|S_{max}|+2|E(\Omega_{max})|+(3t-2)|E(\Sigma_{max})|$ which is of linear complexity with respect to $|D_{max}|+|S_{max}|+|E(\Omega_{max})|+|E(\Sigma_{max})|$.

5.2.3.2 K-way+AMML(s)

The complexity of Algorithms 47 and 48 for reordering the vertices of a subdomain Ω_j for $s-steps$ is equivalent to the complexity of finding the alternating layers from vertices $\overline{\alpha}^{(i)}$ of each neighboring subdomain Ω_i, and then reordering each of these layers, where $\overline{\alpha}^{(i)}=V(\Omega_i)$

Both algorithms take as input all the sets of vertices of the d neighboring subdomains $\overline{\alpha}^{(i)}$ that depend on the vertices $\overline{\alpha}^{(j)}$ and find $bv_i=Adj(G(A),\overline{\alpha}^{(i)})\cap\overline{\alpha}^{(j)}$ where $i=1,2,..,d$. This means that for finding the sets bv_i for each neighboring subdomain Ω_i, all the vertices and edges of the subdomain Ω_i are visited. In other words, for each subdomain Ω_j, $d\times|V(\Omega_{max})|$ vertices and $d\times|E(\Omega_{max})|$ edges have to be visited, where $|\overline{\alpha}^{(i)}|\leqslant|V(\Omega_{max})|$ and $|E(\Omega_i)|\leqslant|E(\Omega_{max})|$ for $i\neq j$, and d is the number of neighboring subdomains. However, this is not necessary. We can perform a preprocessing step where each processor Ω_i finds its subdomain's boundary vertices, S_i, that depend on some other vertex in $\overline{\alpha}^{(j)}$ where $i\neq j$. Then the algorithms can use S_i instead of $\overline{\alpha}^{(i)}$, where $|S_i|\ll|\overline{\alpha}^{(i)}|$. But to keep the presentation of the algorithms simple, $\overline{\alpha}^{(i)}$ was used instead. To find S_i for each subdomain Ω_i, $|\overline{\alpha}^{(i)}|$ vertices and $|E(\Omega_i)|$ edges are visited.

For each subdomain Ω_j, finding the alternating layers from S_i for each neighboring subdomain Ω_i requires visiting at most $\sum_{i=1}^{d}|R(G(A),S_i,2s)|$ vertices and their associated edges. In algorithm 47, in case $corners_i\neq\phi$, the vertices of the set bv_i and associated edges need to be visited again. So some fraction of $|\cup_{i=1}^{d}(R(G(A),S_i,2s)\cap\overline{\alpha}^{(j)})|$ vertices and their associated edges are visited in this case. In the worst case, where $corners(i)\neq\phi$, for all the d neighboring subdomains

$i = 1 : d, i \neq j$, the algorithm visits at most $|\cup_{i=1}^{d} (R(G(A), S_i, 2s) \cap \overline{\alpha}^{(j)})|$ vertices and associated edges. Hence reordering the vertices of a subdomain Ω_j, $\overline{\alpha}^{(j)}$ requires visiting at most

$$2\sum_{i=1}^{d} |R(G(A), S_i, 2s) \cap \overline{\alpha}^{(j)}| + \sum_{i=1}^{d} |S_i| \ll 2|\overline{\alpha}^{(j)}| + \sum_{i=1}^{d} \xi|\overline{\alpha}^{(i)}| \ll (2 + d\xi)|V(\Omega_{max})|$$

vertices and $(2 + d\xi)|E(\Omega_{max})|$ edges. Note that $|S_i| = \xi|\overline{\alpha}^{(i)}|$, where $0 \leqslant \xi \ll 1$ is the ratio of the cardinality of the boundary vertices with respect to the cardinality of the subdomain's vertices. The quantity ξ should be very small since the number of boundary vertices in a subdomain is at most equal to the edge-cuts of that subdomain, and k-way partitioning aims at minimizing the edge-cuts.

Thus, in total to reorder $\overline{\alpha}^{(j)}$, at most $(3 + d\xi)|V(\Omega_{max})|$ vertices and $(3 + d\xi)|E(\Omega_{max})|$ edges are visited. Since the AMML(s) reordering is done in parallel on P processors, its parallel complexity is upper bounded by $(3 + d\xi)(|V(\Omega_{max})| + |E(\Omega_{max})|)$. Hence our algorithm is of linear complexity with respect to $(|V(\Omega_{max})| + |E(\Omega_{max})|)$. In case the AMML(s) reordering is done sequentially, one processor loops over the vertices of the subdomains and reorders them, then the complexity is upper bounded by $(3 + d\xi)P(|V(\Omega_{max})| + |E(\Omega_{max})|)$ where P is the number of subdomains.

5.3 CA-ILU0 preconditioner

In this section we summarize the different steps required for constructing the CA-ILU0 preconditioner, presented in Algorithm 49. The algorithm first reorders the input matrix by using a graph partitioning technique, and in this thesis we consider the usage of ND and k-way partitioning. The obtained matrix is further reordered using AMML(s) reordering. Note that the permutations applied to the matrix A are also applied to the vector b. Then, the redundant data needed by each processor is identified using Algorithm 43. The final step is to compute the ILU(0) factorization of the reordered matrix to obtain the L and U matrices for preconditioning the system $Ax = b$. The ILU(0) factorization can be done sequentially on one processor, where the needed parts of the L and U matrices have to be fetched by the processors before starting the computations in Krylov subspace solver, or it can be done in parallel where each processor performs the ILU(0) factorization of the augmented part of A to obtain the needed parts of L and U.

The ILU(0) factorization of $A(\gamma_s^{(j)}, :)$ can be performed in parallel without any communication for the following reasons. Performing the ILU(0) factorization of $A(\rho, :)$ requires computing the factorization of $A(\omega, :)$ beforehand, where $\omega = R(G(L), \rho)$. And by definition of reachable sets, we have that $R(G(L), \gamma_s^{(j)}) = \gamma_s^{(j)}$. Hence, processor j has all the needed rows of A to perform the ILU0 factorization in parallel without any synchronization or communication, at the expense of doing some redundant computation.

All the steps of the CA-ILU0 preconditioner construction are done in parallel. In addition, steps 2 and 6 in Algorithm 49 can be done in parallel without communication. Note that, after

Algorithm 49 Construction of CA-ILU0 preconditioner

1: Partition the graph of A into P subdomains by using neste dissection or k-way graph partitioning
2: Find a permutation using ND+AMML(s) reordering (Algorithms 45 and 46) or k-way+AMML(s) reordering (Algorithm 47 or 48)
3: Apply the permutation to matrix A
4: Find the redundant/ghost data that each processor needs using Algorithm 43
5: Each processor i fetches its corresponding $A(\gamma_s^{(i)}, :)$
6: Each processor i performs the CA-ILU(0) factorization of $A(\gamma_s^{(i)}, :)$ to obtain the corresponding $L(\gamma_s^{(i)}, :)$ and $U(\gamma_s^{(i)}, :)$ matrices

fetching the matrix $A(\gamma_s^{(j)}, :)$ and $y_0(\delta_s^{(j)})$, each processor can compute its part of the L and U factors of the preconditioner and the first s multiplication without any communication with other processors.

The CA-ILU0 preconditioner can be used with a classic Krylov subspace solver, in which case it allows to apply the left-preconditioner without any communication. In this case AMML(s) reordering has the parameter s set to 1.

On the other hand, the CA-ILU0 preconditioner can be used for ILU0 right-preconditioned systems by slightly modifying the order of the operations in ILU matrix powers kernel where AMML(s) reordering is unchanged. As for the split preconditioned system of the form $L^{-1}AU^{-1}y = L^{-1}b$ with $x = U^{-1}y$, CA-ILU0 preconditioner can also be used by slightly modifying the ILU matrix powers kernel and AMML(s) reordering. In this case, when the tag $evenodd = even$, we assign to the layer at hand the smallest indices in num and the largest indices otherwise. This produces $2s$ alternating layers starting from small indices.

5.4 Expected numerical efficiency and performance of CA-ILU0 preconditioner

The numerical efficiency and performance of CA-ILU0 preconditioner depends on the effect of AMML(s) reordering on the convergence of GMRES for the CA-ILU0 preconditioned system, on the complexity of AMML(s) reordering of the input matrix A, and on the additional memory requirements and redundant flops of the ILU(0) matrix powers kernel. We discuss the convergence of the CA-ILU0 preconditioned system using GMRES in section 5.4.1, the memory requirements and redundant flops in section 5.4.2. While we do not present here results of a parallel implementation of CA-ILU0, the results presented in this section show that CA-ILU0 can be expected to be faster in practice than implementations of the classic ILU0 based on different reordering strategies.

We use in our experiments METIS [49] for graph partitioning purposes. METIS provides two

versions of multilevel k-way partitioning. We use PartGraphKway version that minimizes edge-cuts that is referred to as Kway in the plots of iterations, redundant flops, and memory. In addition, METIS provides a multilevel nested dissection version called NodeNDP. The Kway version has the fastest convergence but requires more memory and redundant flops than nested dissection versions. Hence the Kway version is a good candidate for our CA-ILU0 preconditioner if we choose appropriately the number of partitions and number of steps to obtain a good trade-off between the amount of communication reduced and the amount of redundant computation. For this reason, in sections 5.4.1 and 5.4.2 the plots correspond to the CA-ILU0 preconditioner based on Kway partitioning.

Finally, we compare our CA-ILU0 preconditioner with block Jacobi preconditioner in section 5.4.3.

5.4.1 Convergence

It is known that the convergence of ILU0 preconditioned systems depends on the ordering of the input matrix. The best convergence is often observed when the matrix is ordered using reverse Cuthill-McKee (RCM) or natural ordering (NO), while the usage of k-way partitioning or nested dissection tends to lead to a slower convergence (see for example [24]). Hence, we first discuss the effect of our reordering on the convergence of ILU(0) preconditioned system. Ideally, we would like our ND+AMMLand k-way+AMML(s) reordering to have a negligible effect on the convergence of ILU(0) preconditioner with respect to the most efficient orderings (RCM or NO). Our goal is to compare the convergence of the CA-ILU0 preconditioner. Since s-step GMRES can lead by itself to a slower convergence with respect to a classic GMRES method, we use in our experiments the classic GMRES method. For a brief description of the used test matrices and the nature of the problems they arise from refer to section 2.6.

Table 5.1 compares the convergence behavior the ILU0 preconditioned restarted GMRES on ND, and ND+AMML(s) reordered matrix NH2D with respect to the natural ordering of NH2D, where $s = 1, 5, 10$ and $P = 16, 32, 64$. The effect of the ND+AMML(s) reordering with respect to nested dissection reordering on the convergence of ILU(0) preconditioned GMRES is negligible for the matrix NH2D (few iterations). However, the effect of nested dissection reordering on the convergence of ILU(0) preconditioned GMRES with respect to natural ordering, is almost doubling the iterations (82 iterations versus 146). The more iterations are performed, the more communication is needed. Thus, nested dissection might not be the optimal partitioning technique to be used for the CA-ILU0 preconditioner.

We compare the convergence of GMRES for the ILU0 preconditioned system where the matrix A is reordered using k-way + AMML(s) version1 reordering (Algorithm 47), k-way + AMML(s) version2 reordering (Algorithm 48), k-way and the natural ordering of A for different number of partitions and for $s = 1, 2, 5, 10$. We set the GMRES tolerance to 10^{-8} and the maximum number of iterations to 500. Figure 5.8 shows the convergence behavior for the matrices NH2D1, UTM3060, BO1, BO2, SKY2D, and SKY3D.

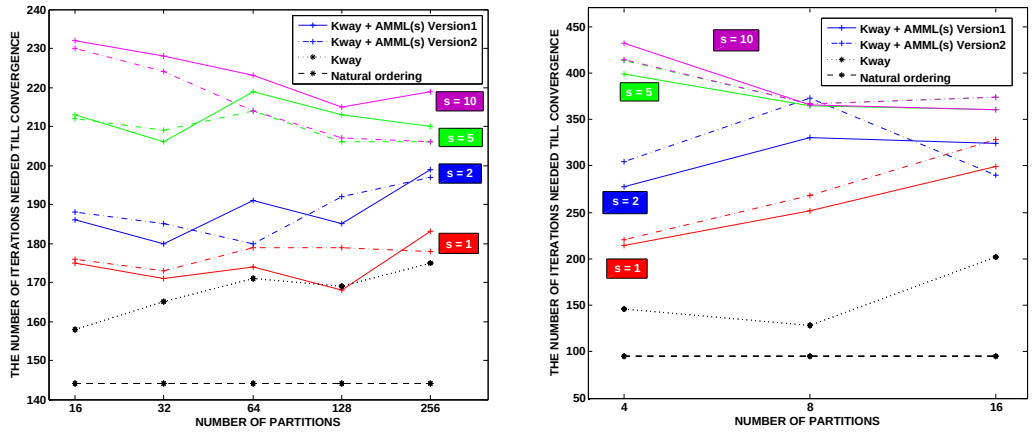
Table 5.1: Convergence of the ILU0 preconditioned restarted GMRES on NO, ND, ND+AMML(s) reordered matrix NH2D. tol = 10^{-8}, maximum iterations = 200, number of restarts = 2

Ordering	**Real error** $\frac{\mathbf{norm(x_{sol}-x_{app})}}{\mathbf{norm(x_{sol})}}$	**Relative residual** $\frac{\mathbf{norm(b-Ax_{app})}}{\mathbf{norm(b)}}$	**Number of iterations**
NO	1.09×10^{-7}	9.80×10^{-9}	82
ND 16	8.19×10^{-7}	9.30×10^{-9}	148
ND 32	1.13×10^{-6}	8.80×10^{-9}	146
ND 64	1.45×10^{-6}	9.50×10^{-9}	142
ND+AMML(1) 16	8.14×10^{-7}	9.50×10^{-9}	148
ND+AMML(1) 32	1.31×10^{-6}	9.30×10^{-9}	147
ND+AMML(1) 64	1.87×10^{-6}	9.70×10^{-9}	144
ND+AMML(5) 16	1.43×10^{-6}	9.90×10^{-9}	147
ND+AMML(5) 32	2.35×10^{-6}	9.10×10^{-9}	152
ND+AMML(5) 64	2.47×10^{-6}	9.70×10^{-9}	149
ND+AMML(10) 16	9.46×10^{-7}	9.40×10^{-9}	146
ND+AMML(10) 32	2.44×10^{-6}	9.50×10^{-9}	152
ND+AMML(10) 64	2.48×10^{-6}	9.70×10^{-9}	149

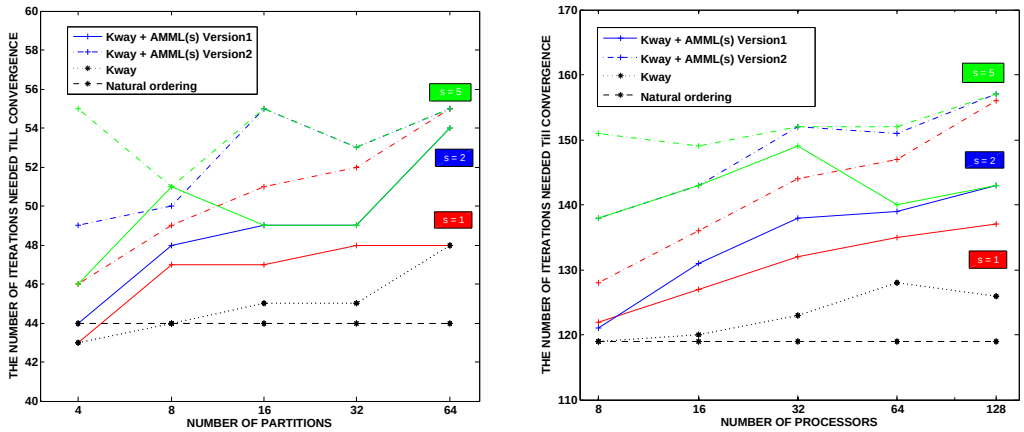
As expected, the ILU0 preconditioned system with the natural ordering of A converges faster than when A is reordered using k-way and the two versions of AMML(s). The convergence of the RCM ILU0 preconditioned system is not shown in the plots, but for the symmetric matrices NH2D1, BO1, BO2, BCSSTK18 and SKY2D it has the same convergence as natural ordering, while for UTM3060 it converges with 2 iterations less than natural ordering. For the matrix NH2D1, when A is reordered using k-way and k-way plus AMML(1), the preconditioned GMRES has the same rate of convergence. But as s increases and the more we reorder the matrix, the more iterations are needed for convergence. We notice the same behavior for the matrices UTM3060, BO1 and BO2 (Figures 5.8(b), 5.8(c), and 5.8(d)). But for the matrix BCSSTK18, GMRES converges in a maximum of 9 iterations for the different reorderings of A as shown in figure 5.8(e). It must be noted that without preconditioning, the BCSSTK18 system does not converge for the given tolerance, while for $tol = 10^{-6}$ it converges in 909 iterations.

For the matrices BO1 and BO2, k-way plus AMML(s) version1 (Algorithm 3) has a better convergence than version 2 (Algorithm 4). However for the matrices NH2D1 and UTM3060 there is no clear winner. But for $s = 1$, version1 converges slightly better, since in version2 nested dissection reorders two layers of vertices.

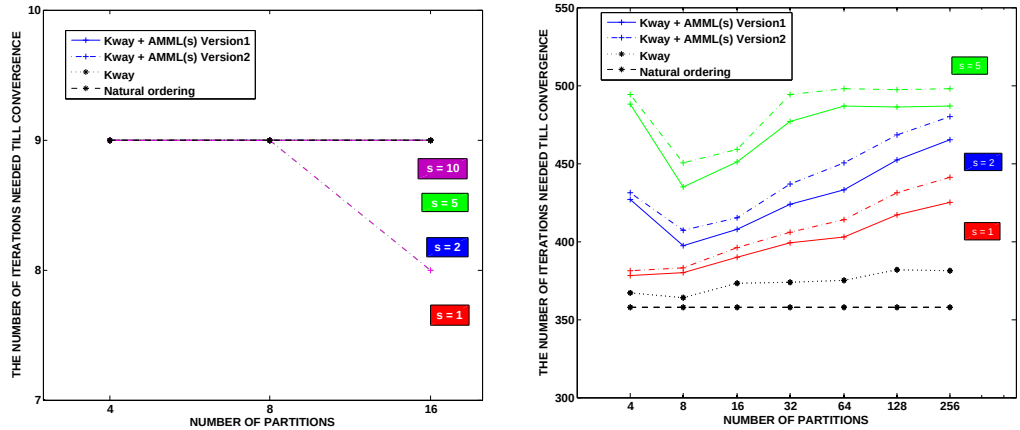
Table A.1 in Appendix B, shows the convergence behavior of ILU(0) preconditioned GMRES with the different reorderings for WATT2 and the other seven matrices. We observe similar convergence behaviors with respect to number of partitions, steps s, and the two versions of AMML(s). Thus s has to be chosen such that the total reduced communication in the ILU(0) matrix powers kernel is much greater than time needed to compute the extra iterations resulting from the k-way+AMML(s) reordering with respect to natural ordering.



(a) Matrix NH2D1 (b) Matrix UTM3060



(c) Matrix BO1 (d) Matrix BO2



(e) Matrix BCSSTK18 (f) Matrix SKY2D

Figure 5.8: The number of iterations needed for convergence for 6 matrices as a function of the number of partitions and steps s. The matrices are either in natural ordering or reorderd using k-way partitioning (*Kway*), k-way partitioning followed by AMML(s) based on Algorithm 3 (Version 1) or Algorithm 4 (Version 2). The number of partitions vary from $4, 8, 16, 32, 64, 128$ to 256 depending on the size of the matrix. The number of steps s is either 1 (red), 2 (blue), 5 (green) and 10 (magenta).

We can conclude that our CA-ILU0 preconditioned system where the matrix A is reordered using k-way and AMML(s) has a very similar convergence behavior to the ILU(0) preconditioned system where the matrix A is only reordered using k-way graph partitioning technique. Thus, our additional AMML(s) reordering of the matrix does not affect much its convergence, while it enhances its communication avoiding parallelizability.

5.4.2 Avoided communication versus memory requirements and redundant flops of the ILU0 matrix powers kernel

The ILU(0) matrix powers kernel avoids communication by performing redundant flops and storing more vectors and data. Table 5.2 compares the needed memory and performed flops for s matrix vector multiplications on one subdomain/processor when using the non-preconditioned CA-GMRES ($Ax, A^2x, .., A^sx$) and the CA-ILU0 preconditioned CA-GMRES ($(LU)^{-1}Ax$, ... ,$((LU)^{-1}A)^sx$) on 2D 9-point stencils and 3D 27-point stencils. We assume that each processor j has to compute the part $\alpha_0^{(j)} = V(\Omega_j)$ of the s matrix vector multiplication, where A is an $n \times n$ matrix, $|\alpha_0^{(j)}| \approx n/P = w^d$, $d = 2$ for 2D matrices and 3 for 3D matrices, and $w = (n/P)^{\frac{1}{d}}$ is the width of the square or cube subdomain. For simplicity, we refer to $\alpha_0^{(j)}$ as α in table 5.2.

Table 5.2: Memory and computational cost required for performing s matrix vector multiplication on one subdomain α, for the non-preconditioned CA-GMRES and for the CA-ILU0 preconditioned CA-GMRES.

	CA-GMRES		**CA-ILU0 CA-GMRES**	
Stencil	**Memory**	**Flops**	**Memory**	**Flops**
2D 9-pt	$(s + 10)\|\alpha\| + \frac{4}{3}s^3 + 38s^2 - \frac{214}{3}s + 36 + 2(s^2 + 19s - 18)\|\alpha\|^{\frac{1}{2}}$	$17s\|\alpha\| - 34s^2 + \frac{17}{3}(4s^3 + 2s) + 34(s^2 - s)\|\alpha\|^{\frac{1}{2}}$	$(s + 21)\|\alpha\| + \frac{16}{3}s^3 + 328s^2 - \frac{184}{3}s + 24 + 4(s^2 + 41s - 4)\|\alpha\|^{\frac{1}{2}}$	$35s\|\alpha\| + \frac{560}{3}s^3 + 208s^2 + \frac{172}{3}s + 4s(35s + 26)\|\alpha\|^{\frac{1}{2}}$
3D 27-pt	$(s + 28)\|\alpha\| + 2s^4 + 220s^3 - 646s^2 + 648s - 216 + 3(s^2 + 55s - 54)\|\alpha\|^{\frac{2}{3}} + [4s^3 + 330s^2 - 646s + 324]\|\alpha\|^{\frac{1}{3}}$	$53s\|\alpha\| + 106s^4 + 106(-2s^3 + s^2) + 159(s^2 - s)\|\alpha\|^{\frac{2}{3}} + 106[2s^3 - 3s^2 + s]\|\alpha\|^{\frac{1}{3}}$	$(s + 57)\|\alpha\| + 16s^4 + 8s[452s^2 - 154s + 90] - 88 + (6s^2 + 678s - 78)\|\alpha\|^{\frac{2}{3}} + 4[4s^3 + 678s^2 - 154s + 45]\|\alpha\|^{\frac{1}{3}}$	$107s\|\alpha\| + 1064s^2 + 2560s^3 + 6s[107s + 80]\|\alpha\|^{\frac{2}{3}} + 2s[856s^2 + 960s + 266]\|\alpha\|^{\frac{1}{3}} + 1712s^4$

CA-GMRES requires storing s vectors of size $|R(G(A), \alpha_0^{(j)}, i)| \approx |(w + 2i)^d|,\ i = 1, 2, ..., s$,

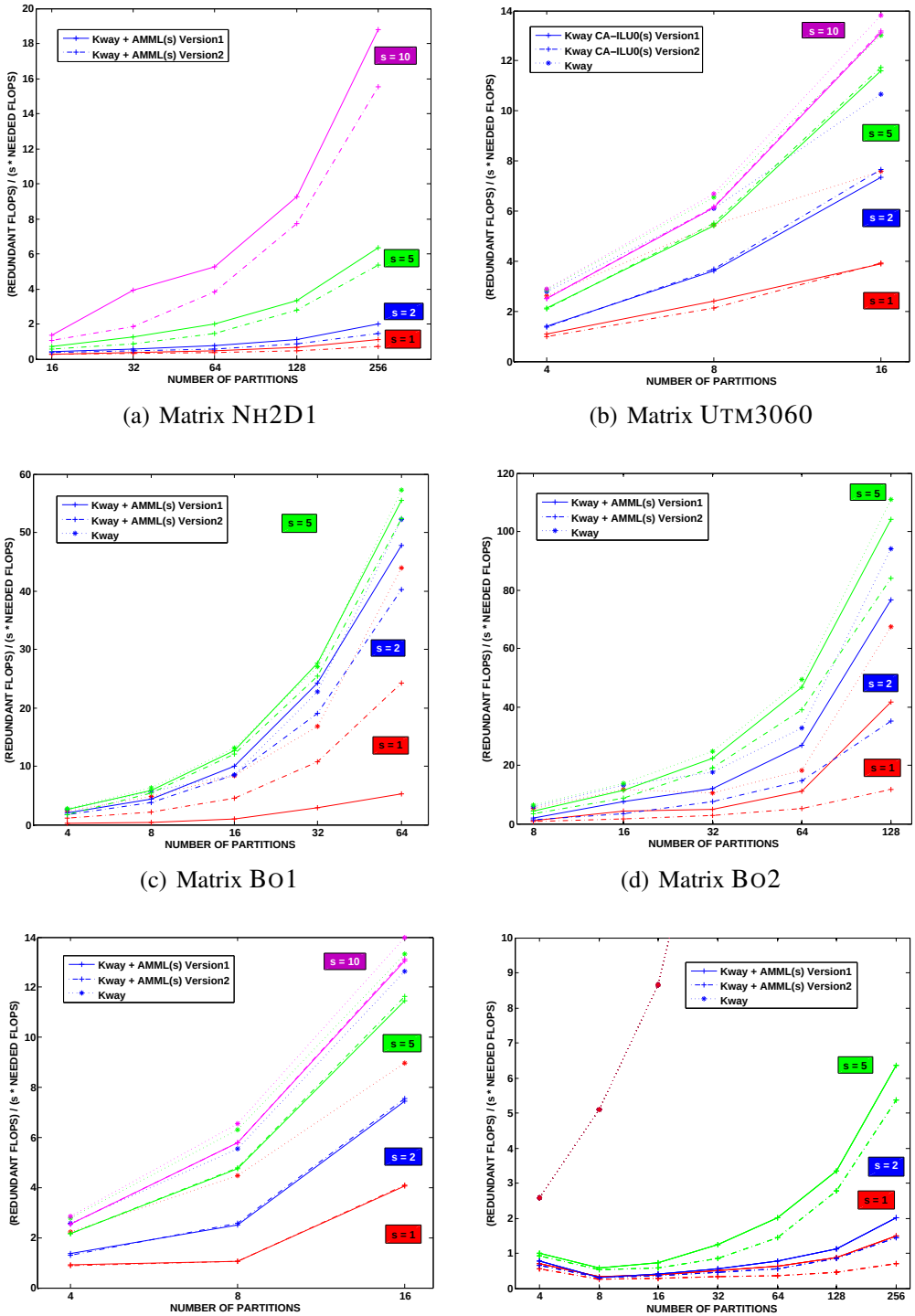
one vector of size $|\alpha_0^{(j)}|$ and the corresponding $|R(G(A), \alpha_0^{(j)}, s-1)|$ rows of the matrix A. Then, it performs $\sum_{i=1}^{s}((w+2(i-1))^d)(2 \times nnz - 1)$ flops where nnz is the number of nonzeros per row (9 and 27). CA-ILU0 preconditioned CA-GMRES requires storing s vectors of size $|R(G(A), \alpha_0^{(j)}, 2(i-1))| \approx |(w+4(i-1))^d|,\ i = 1, 2, ..., s$, one vector of size $|R(G(A), \alpha_0^{(j)}, 2s+1)|$, one vector of size $|R(G(A), \alpha_0^{(j)}, 2s)|$, the corresponding $|R(G(A), \alpha_0^{(j)}, 2s)|$ rows of the matrices A and L, and the $|R(G(A), \alpha_0^{(j)}, 2s-1)|$ rows of the matrix U. Then it performs $\sum_{i=1}^{s}(2 \times nnz-1)((w+4i)^d)$ flops to compute Ax. Solving the s lower triangular systems ($Lz = f$) requires $\sum_{i=1}^{s}[1 + 2 \times (nnz-1)/2]((w+4i)^d)$ flops. Similarly, solving the s upper triangular systems requires $\sum_{i=1}^{s}[1 + 2 \times (nnz-1)/2]((w+4i-2)^d)$ flops. Note that the memory and flops of CA-GMRES and CA-ILU0 preconditioned CA-GMRES are governed by the same big O function.

Figure 5.9 plots the ratio of the total redundant flops in the ILU(0) matrix powers kernel for $s = 1, 2, 5, 10$ with respect to the needed flops for computing s matrix vector multiplication in the sequential ILU0 preconditioned GMRES for six matrices in our set that are reordered using k-way, k-way+AMML(s) Version1 and k-way+AMML(s) Version2. Figures 5.10 plots the ratio of the ghost data that has to be saved in memory in the ILU0 matrix powers kernel for $s = 1, 2, 5, 10$ with respect to the needed memory in the matrix vector multiplication of the sequential ILU0 preconditioned GMRES for six matrices in our set that are reordered using k-way, k-way+AMML(s) Version1 and k-way+AMML(s) Version2. In figures 5.9(a) and 5.10(a) we do not show the ratio of redundant flops with respect to needed flops for the k-way reordered matrix NH2D1, since it is at least 10 times more than that of AMML(s) reordering. Hence the AMML(s) reordering leads to at least 90% less redundant flops and ghost memory in the ILU0 matrix powers kernel than METIS's k-way partitioning. This leads to a reduction of the volume of the communicated data at the end of the s steps.

In figures 5.9(b) and 5.9(e), AMML(s) reordering performs from 10 to 50% less redundant flops than k-way partitioning in the ILU0 matrix powers kernel. On the other hand, in figures 5.10(b) and 5.10(e), AMML(s) reordering needs 50% and 25% less ghost memory for $s = 1$ and $s = 2$ respectively. Whereas for $s = 5$ and 10, AMML(s) reordering and k-way partitioning ratios are equal to $P - 1$, where P is the number of processors or partitions. This means that each processor ends up needing all the matrices A, L, U and computing almost everything for a number of steps. Hence for matrices UTM3060 and BCSSTK18, s has to be less than 5.

We compare the ratio of redundant flops and ghost data of the two versions of AMML(s) reordering for the above three matrices. For matrix NH2D1, figures 5.9(a) and 5.10(a), version2 performs less redundant flops. For matrix UTM3060, figures 5.9(b) and 5.10(b), version1 has a slightly better performance for $s > 1$. As for the matrix BCSSTK18, figures 5.9(e) and 5.10(e), the two versions have almost the same performance.

It is clear that as s or the number of partitions increase, the redundant flops and ghost memory increase. Thus, one has to choose the appropriate number of partitions and steps s with respect to the problem at hand, to obtain the best performance. In other words, one has to find a balance between the redundant flops and communication (number of messages) while taking into consid-



(a) Matrix NH2D1

(b) Matrix UTM3060

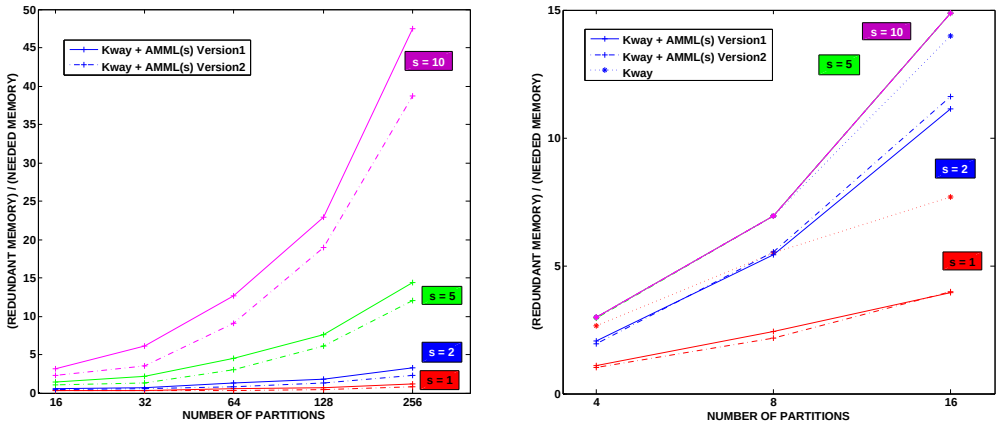
(c) Matrix BO1

(d) Matrix BO2

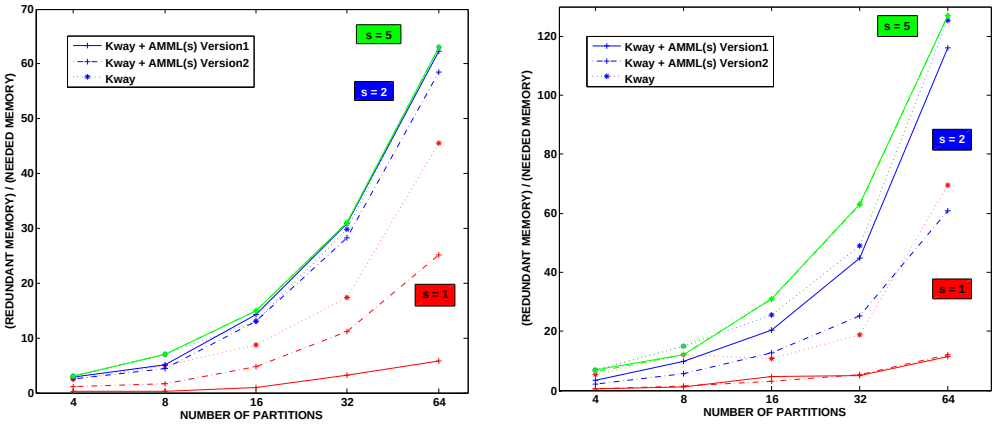
(e) Matrix BCSSTK18

(f) Matrix SKY2D

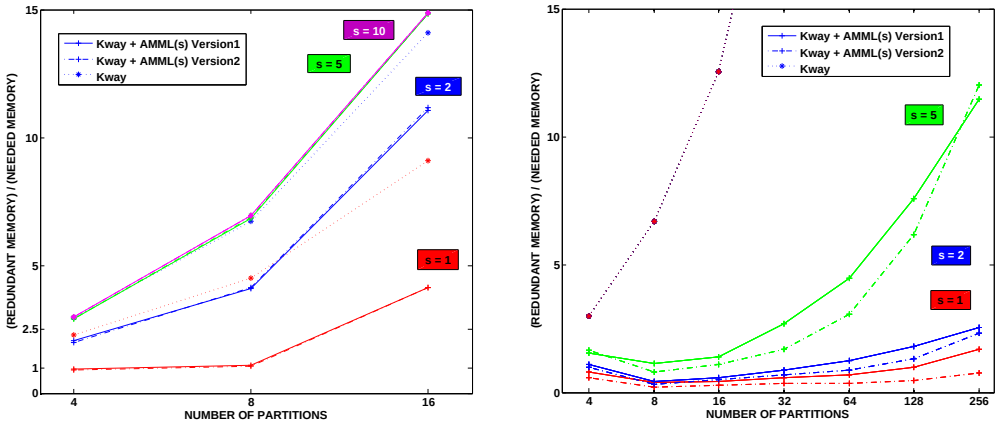
Figure 5.9: The ratio of redundant flops w.r.t needed flops in the ILU0 matrix powers kernel as a function of the number of partitions and steps s. The matrices are either reordered using k-way partitioning (*Kway*), or k-way partitioning followed by AMML(s) based on Algorithm 3 (Version 1) or Algorithm 4 (Version 2). The number of partitions vary from 4,8,16,32,64,128 to 256 depending on the size of the matrix. The number of steps s is either 1 (red), 2 (blue), 5 (green) and 10 (magenta).



(a) Matrix NH2D1 (b) Matrix UTM3060



(c) Matrix BO1 (d) Matrix BO2



(e) Matrix BCSSTK18 (f) Matrix SKY2D

Figure 5.10: The ratio of redundant data w.r.t needed data in the ILU0 matrix powers kernel as a function of the number of partitions and steps s. The matrices are either reordered using k-way partitioning (*Kway*), or k-way partitioning followed by AMML(s) based on Algorithm 3 (Version 1) or Algorithm 4 (Version 2). The number of partitions vary from 4,8,16,32,64,128 to 256 depending on the size of the matrix. The number of steps s is either 1 (red), 2 (blue), 5 (green) and 10 (magenta).

eration the available memory. The choice of the number of partitions P is related to the concept of surface-to-volume ratio discussed in [48] which is an indicator of data dependencies. In other words, the ratio of a subdomain's vertices with edge-cuts with respect to those without edge-cuts should be relatively small. On the other hand, values of s should be chosen so that the processor communicate at most with his neighbors and some factor of his neighbor's neighbors. And the smaller the subdomains are (large P) the smaller s should be and vice versa.

For example, for matrix NH2D1of size $40,000 \times 40,000$, which corresponds to a graph of size 200×200, for $p = 256$ with subdomain of size 12×13 the surface-to-volume ratio is around 0.3, which is not very small. Thus for $s = 1$ or $s = 2$ the redundant flops computed are 1 or 2 times the flops needed to perform the multiplication sequentially, which is reasonable (Figure 5.9(a)). But for $s = 5$ or $s = 10$ it is prohibitive (6 and 18 times). Similarly, for $s = 10$ with $p = 16$ (50×50 subdomains), $p = 32$ (35×36 subdomains), and $p = 64$ (25×25 subdomains) the redundant flops are 1, 2, and 4 times the sequential version, which is reasonable. Note that an increase in the computed redundant flops is equivalent to an increase in the needed memory and the volume of communicated data after computing s basis vectors. Thus, small values of s might be used in practice.

Table 5.3: Messages and number of words received for performing $s = 1$ multiplication per iteration, on one subdomain α_j of a 2D 5-point stencil matrix, for GMRES and CA-ILU0 preconditioned GMRES.

	GMRES $y = Ax$	**CA-ILU0 GMRES** $y = (LU)^{-1}Ax$
Each processor j	receives one message from each of its 4 neighbors of size $w = (\frac{n}{P})^{\frac{1}{2}}$ words	receives one message from each of its 4 neighbors of size $w = (\frac{n}{P})^{\frac{1}{2}}$ words
		receives one message from 4 other processors each of size 4 words

In the case of $s = 1$, the CA-ILU0 preconditioner can be used with the classical preconditioned GMRES where the parallelized multiplication of the form $y_1 = (LU)^{-1}Ay_0$ is replaced by the $s = 1$ version of the ILU0 matrix powers kernel. At the beginning of the first iteration, each processor fetches its corresponding parts of A and y_0 and then factorizes its part of A and computes its part of y_1. Then, before every iteration of GMRES, one communication phase is needed when y_0 is fetched . Table 5.3 shows the messages and number of words received by processor j on domain α_j of a 2D 5-point stencil, for computing $y = Ax$ in GMRES and $y = (LU)^{-1}Ax$ in CA-ILU0 preconditioned GMRES, where the communication pattern in both is similar. We did not compare CA-ILU0 preconditioned GMRES to ILU0 preconditioned GMRES since parallelizing the backward and forward substitution can be implemented by using different approaches. Consider, for example, that the implementation uses Nested Dissection. For each of the $log(P)$ levels of nested dissection, there is need for one communication phase between processors, in both forward and backward substitution. Thus, at least $log(P)$ messages are sent of different sizes. In

summary, the communication cost of parallelizing the $y = (LU)^{-1}Ax$ in ILU0 preconditioned GMRES is at least $2log(p) + 4$ messages in $2log(P) + 1$ communication phases. Whereas in CA-ILU0 GMRES, it is of the order of 8 messages in 1 communication phase before the computations. Thus the communication is reduced by a factor of $O(2log(P))$. In general, for $s > 1$, the CA-ILU0 preconditioner reduces communication by at least a factor of $O(2slog(P) + s)$.

5.4.3 Comparison between CA-ILU0 preconditioner and block Jacobi preconditioner

The block Jacobi preconditioner is one of the simplest parallel preconditioners which avoids communication when performing one multiplication of the form $y = M^{-1}Ax$ where

$$M = \begin{pmatrix} A_{1,1} & 0 & ... & 0 \\ 0 & A_{2,2} & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & ... & 0 & A_{P,P} \end{pmatrix} = \begin{pmatrix} L_{1,1}U_{1,1} & 0 & ... & 0 \\ 0 & L_{2,2}U_{2,2} & 0 & 0 \\ 0 & 0 & \ddots & 0 \\ 0 & ... & 0 & L_{P,P}U_{P,P} \end{pmatrix} = LU$$

is constructed from the diagonal blocks of A. The block Jacobi preconditioner starts by partitioning the graph of A into P well balanced partitions $\pi = \{\Omega_1, \Omega_2, ..., \Omega_P\}$. Then each processor i is assigned the set of vertices $\alpha_0^{(i)} = V(\Omega_i)$ and has to compute $y(\alpha_0^{(i)})$. Processor i fetches $A(\alpha_0^{(i)}, :)$ and $x(\delta_1^{(i)})$ where $\delta_1^{(i)} = Adj(G(A), \alpha_0^{(i)})$. Then, processor i factorizes $A_{i,i} = A(\alpha_0^{(i)}, \alpha_0^{(i)})$ into $L_{i,i}$ and $U_{i,i}$ matrices by using complete or incomplete LU factorization. Since the diagonal blocks of M are independent, it is possible to perform the LU factorization, and the backward and forward solves in parallel without communication. Thus our CA-ILU0 preconditioner is very similar to block Jacobi in the communication pattern for $s = 1$ only. But for $s > 1$, the block Jacobi preconditioner can't be used with the ILU matrix powers kernel since the reachable sets $\beta_j^{(i)} = R(G(U), \alpha_{j-1}^{(i)})$, $\gamma_j^{(i)} = R(G(L), \beta_j^{(i)})$, and $\delta_j^{(i)} = Adj(G(A), \gamma_j^{(i)})$ can grow in size rapidly where $1 < j \leqslant s$.

We compare the convergence behavior of the block Jacobi preconditioner, with k-way reordering and LU or ILU0 block diagonal factorization, to the CA-ILU0 preconditioner, where the input matrix A is reordered using k-way plus AMML(1) reordering. Table 5.4 shows the ratio of the norm of the error (Err) between the real solution and the approximate solution obtained by the different preconditioned GMRES versions for $tol = 10^{-8}$ ($norm(x - x_{app})/norm(x)$), the number of iterations (Iter) needed till convergence, the correctness(LUErr) of the different factorizations ($norm(A - LU)/norm(A)$), and the introduced fill-in ratio (Fill) of the block Jacobi-LU ($(nnz(L) + nnz(U))/nnz(A)$, nnz is the number of nonzero entries). The input matrix is partitioned into 16,32, 128,256, 512, 1024 or 2048 parts (Pa) using k-way.

For all the matrices CA-ILU0 preconditioner has better convergence than block Jacobi-ILU0. However, block Jacobi-LU preconditioner has the best convergence when the number of partitions is relatively small, since it is then very similar to a complete LU preconditioner. But when the

Table 5.4: Comparison between the convergence of CA-ILU0 preconditioner with k-way+AMMLreordering and block Jacobi preconditioner.

		CA-ILU0			Block Jacobi-ILU0			Block Jacobi-LU			
	Pa	Iter	Err	LUErr	Iter	Err	LUErr	Iter	Err	LUErr	Fill
BO1	16	51	2E-8	7E-10	62	7E-8	1E-9	49	5E-8	1E-9	6.0
	32	52	4E-8	7E-10	67	8E-8	1E-9	57	5E-8	1E-9	3.6
BO2	32	144	6E-7	8E-10	151	4E-7	7E-10	110	13E-7	6E-10	15.1
	128	156	4E-7	1E-9	170	5E-7	4E-9	144	4E-7	4E-9	5.7
	256	154	8E-7	1E-9	197	1E-6	4E-9	176	6E-7	4E-9	3.4
NH2D1	32	173	8E-7	9E-6	193	1E-6	1E-5	116	4E-7	1E-5	14.1
	128	179	1E-6	8E-6	196	1E-6	1E-5	139	6E-7	1E-5	7.0
	512	184	1E-6	1E-5	221	1E-6	1E-5	181	9E-7	1E-5	3.6
	1024	191	1E-6	1E-5	236	2E-6	1E-5	217	1E-6	1E-5	2.5
NH2D2	32	301	5E-6	2E-6	322	3E-6	2E-6	154	1E-6	2E-6	29.3
	128	308	5E-6	2E-6	339	6E-6	2E-6	201	9E-7	2E-6	13.7
	1024	314	8E-6	2E-6	369	8E-6	2E-6	292	2E-6	2E-6	5.0
	2048	322	4E-6	2E-6	372	5E-6	2E-6	315	3E-6	2E-6	3.6
SKY3D	128	594	9E-5	1E-3	643	1E-4	1E-3	526	1E-4	1E-3	15.5
	256	576	1E-4	1E-3	674	2E-4	1E-3	569	1E-4	1E-3	9.5
	512	563	8E-5	1E-3	723	3E-4	1E-3	627	1E-4	1E-3	5.9
	1024	597	9E-5	1E-3	775	2E-4	1E-3	729	3E-4	1E-3	3.7

fill-in ratio decreases as the number of partitions increases, the convergence behavior of CA-ILU0 and Block Jacobi-LU preconditioner become very similar. For example, the CA-ILU0 preconditioner converges faster for the BO1matrix with 32 partitions, the BO2matrix with 256 partitions, NH2D1matrix with 1024 partitions, and SKY3Dwith 512 and 1024 partitions.

5.5 Summary

In this chapter, we have introduced CA-ILU0, a communication avoiding ILU0 left-preconditioner. First, we have adapted the matrix powers kernel to the ILU preconditioned system to obtain the ILU matrix powers kernel. Then we have introduced AMML, a reordering of the matrix A which is applied once the input matrix was partitioned using k-way graph partitioning with edge separators or nested dissection with vertex separator. AMML reorders the matrix A such that s-steps of a Krylov subspace solver based on multiplications of the form $y_i = (LU)^{-1}Ay_{i-1}$ can be performed with no communication. The difference between the k-way+AMML(s) and ND+AMML(s) reordering is in the subdomain's local reordering. When using nested dissection to partition the graph of A, the obtained subdomains are reordered with AMML(s) reordering that produces $2s-1$

layers by setting the tag $evenodd = odd$ in the algorithm, since the separators are numbered larger than the subdomains. Whereas, when using k-way graph partitioning, the obtained subdomains are reordered with AMML(s) reordering that produces $2s$ layers by setting the tag $evenodd = even$ in the algorithm.

We have shown that the reordering does not affect much the convergence of the ILU0 preconditioned GMRES, once the matrix A was reordered using k-way partitioning. Then, we have shown that the complexity of the CA-ILU0(s) reordering is linear with respect to the number of vertices of largest subdomain. We have also shown that the memory requirements and redundant flops are limited by the same big O function in both CA-GMRES and CA-ILU0 preconditioned GMRES. For all these reasons, we expect that our parallel CA-ILU0 preconditioner will be faster in practice than implementations of ILU0 preconditioners based on other reordering strategies. It will be faster than block Jacobi preconditioner for relatively small partitions where the dropped data in the block Jacobi preconditoner is no longer negligible.

The AMML(s) reordering allows to both compute and apply the preconditioner in parallel with no communication, once some ghost data was stored redundantly on each processor. CA-ILU0 can be used with a classic Krylov subspace solver, in which case applying the left preconditioner at each iteration can be done in parallel with no communication. It can also be used with s-step methods, where the ILU0 matrix powers kernel allows to avoid communication during s iterations of the Krylov subspace solver. In addition, CA-ILU0 preconditioner can be used for ILU0 right preconditioned and split preconditioned systems by slightly modifying the ILU Matrix Powers Kernel and the AMML(s) reordering.

Chapter 6

Conclusion and Future work

The work presented in this thesis focused on introducing communication avoiding methods in numerical linear algebra, specifically for solving sparse systems of linear equations. First, we briefly discussed Krylov subspace methods, block methods, s-step methods, communication avoiding methods, and preconditioners that are related to our work. Then, we introduced a new class of Krylov subspace methods, the enlarged Krylov subspace methods. We defined the new enlarged Krylov subspace, described its properties, and gave a general framework for the enlarged Krylov projection methods. The idea is to enlarge the subspace by replacing r_0 by t vectors, obtained from the projection of r_0 on t distinct subdomains, where the sum of these t vectors is r_0. Then, rather than computing one basis vector by multiplying r_0 by the powers of A, we compute t vectors by multiplying $\{T_1(r_0), T_2(r_0), ..., T_t(r_0)\}$ by the same power of A. This guarantees that the Krylov subspace is a subset of the enlarged Krylov subspace. Moreover, the enlarged Krylov subspace methods converge faster than the classical methods in exact precision, and are better parallelizable while reducing communication.

In this thesis we have introduced two enlarged conjugate gradient methods, multiple search direction with orthonormalization CG (MSDO-CG) and long recurrence enlarged CG (LRE-CG). We have shown that in finite precision both methods converge faster than CG. The main difference between both methods in terms of performance, is that at each iteration of MSDO-CG, we use t search directions to update the new approximate solution. Whereas in LRE-CG, at each iteration i, we use the entire basis formed by ti vectors, to update the approximate solution and we solve a $ti \times ti$ system. The use of the whole basis leads to a relatively faster convergence than MSDO-CG. However, this comes at the cost of performing more flops as the iterations proceed. One way to limit this increasing cost is by restarting LRE-CG after some iterations. Another alternative is to choose at each iteration i, a linearly independent subset of the t computed vectors. This adds an extra cost, but reduces the size of the system that has to be solved at each iteration. A third alternative is to compute t_i vectors at each iteration i, where $t_0 = t$, and $t_i \leqslant t$. Then choose $\widehat{t}_i$ linearly independent vectors where $\widehat{t}_i \leqslant t_i$, and $t_{i+1} = \widehat{t}_i$.

Although each iteration of the MSDO-CG and LRE-CG methods is at least t times more ex-

pensive than the CG iteration in terms of flops, as shown in section 4.5, both methods use less communication, and Blas2 and Blas3 operations that can be parallelized in a more efficient way than the dot products in CG, as shown in section 4.5. Moreover, the MSDO-CG and LRE-CG methods can be preconditioned with the classical preconditioners, since we have a matrix-block of vectors multiplication at each iteration, and not s matrix powers as in s-step methods.

In the second part of the thesis we have introduced a communication avoiding ILU(0) preconditioner that allows the computation of s multiplications of the form $y_i = (LU)^{-1}Ay_{i-1}$ without any communication at the expense of performing redundant computations, for $i = 1, .., s$. In other words, it is possible to perform s backward solves, s forward solves, and s matrix vector multiplications without communication. The CA-ILU0 preconditioner can be used with communication avoiding methods, s-step methods, block methods ($s = 1$), classical Krylov methods ($s = 1$), enlarged Krylov subspace methods $s = 1$, and any other methods that use the preconditioned matrix powers kernel. The building blocks of the CA-ILU0 preconditioner are the ILU matrix powers kernel and the Alternating Min-Max Layers AMML(s) reordering. The ILU matrix powers kernel is an adaptation of the matrix powers kernel to the case of ILU preconditioned systems. The AMML(s) reordering reduces the data dependencies needed for solving the s upper and lower triangular systems, and for performing the ILU(0) factorization in parallel. We assume that we have P subdomains, obtained by nested dissection, kway graph partitioning, or any other graph or hypergraph partitioning. Then, the subdomains are reordered to obtain alternating layers. We presented the ND+AMML(s) reordering and the k-way+AMML(s) reordering. We have tested the effect of the ND+AMML(s) and k-way+AMML(s) reordering on the convergence of ILU(0) preconditioned GMRES, and shown that the more we reorder the domains, i.e. as s and P increase, slightly more iterations are needed for convergence. It is possible to modify the presented algorithms in section 5.2 to obtain an AMML(s) reordering based on other partitioning techniques, if needed.

Moreover, we modeled the expected performance of the CA-ILU0 preconditioner based on the complexity of the AMML(s) reordering, on the effect of the AMML(s) reordering, and on the redundant computations and memory requirements needed to avoid communication. Thus, the number of partitions or processors P and s should be chosen wisely to obtain the best performance. In other words, the avoided communication should be much more expensive than the redundant flops and the additional iterations introduced by the AMML(s) reordering. This would lead to a CA-ILU0 preconditioner that is faster than ILU(0) preconditioner in a parallel environment.

In this thesis, all the algorithms were implemented and tested in matlab. Currently, Sebastien Cayrols, a PhD student of Laura Grigori, is implementing CA-ILU0 preconditioner on distributed-memory architectures for $s = 1$ with k-way+AMML(s) reordering, and comparing it to Petsc's ILU(0) implementation, block Jacobi preconditioner, and RAS preconditioner.

Our future work on CA-ILU0 preconditioner will focus on implementing it for $s > 1$ in a parallel environment to evaluate the improvements with respect to existing implementations of ILU(0) with s-step methods. We will also extend the method to more general incomplete LU factorizations. An interesting idea would be to test the convergence of a modified version of CA-ILU0 where the same procedure is applied as in Algorithm 49, except that the ILU(0) factorization is replaced with

a more accurate incomplete factorization such as ILU(1) factorization. We would also like to derive communication avoiding versions of ILU(k) and ILU(drop tolerance) preconditioners, similarly to the CA-ILU0 preconditioner. As for the enlarged Krylov subspace methods, our future work will focus on testing the LRE-CG versions discussed above, that are less expensive in terms of flops and memory requirements than LRE-CG, like restarted LRE-CG or LRE-CG with selected basis vectors. Then, the most stable version will be implemented in a parallel environment. We will also test LRE-CG on other real applications' matrices, and with different preconditioners. Moreover, we would also like to compare the runtime of the LRE-CG version with the MSDO-CG method on a parallel environment. We will also derive and test other enlarged Krylov methods, like enlarged GMRES which has been derived but not tested yet.

Bibliography

[1] P. Amestoy, J.Y. L'Excellent, F.H. Rouet, and M. Sid-Lakhdar. Modeling 1D distributed-memory dense kernels for an asynchronous multifrontal sparse solver (regular paper). In and, editor, *High-Performance Computing for Computational Science, VECPAR 2014, Eugene, Oregon, USA, 30/06/2014-03/07/2014*, http://www.laas.fr, 2014. LAAS.

[2] O. Axelsson and P. S. Vassilevski. Algebraic multilevel preconditioning methods, II. *SIAM Journal on Numerical Analysis*, 27(6):1569–1590, 1990.

[3] C. Aykanat, B. B. Cambazoglu, and B. Uçar. Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *J. Parallel Distrib. Comput.*, 68(5):609–625, May 2008.

[4] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in linear algebra. *CoRR*, abs/0905.2485, 2009.

[5] M. Benzi. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182(2):418 – 477, 2002.

[6] M. Benzi, D. Szyld, and A. van Duin. Orderings for incomplete factorization preconditioning of nonsymmetric problems. *SIAM Journal on Scientific Computing*, 20(5):1652–1670, 1999.

[7] M. Benzi and M. Tůma. A comparative study of sparse approximate inverse preconditioners. *Appl. Numer. Math.*, 30(2-3):305–340, June 1999.

[8] A. Bhaya, P. Bliman, G. Niedu, and F. Pazos. A cooperative conjugate gradient method for linear systems permitting multithread implementation of low complexity. In *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, pages 638–643, Dec 2012.

[9] M. Bollhöfer, M. Grote, and O. Schenk. Algebraic multilevel preconditioner for the Helmholtz equation in heterogeneous media. *SIAM Journal on Scientific Computing*, 31(5):3781–3805, 2009.

[10] X. Cai and M. Sarkis. A restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21(2):792–797, 1999.

[11] E. Carson, J. Demmel, and N. Knight. Hypergraph partitioning for computing matrix powers, 2010.

[12] E. Carson, N. Knight, and J. Demmel. Avoiding communication in two-sided Krylov subspace methods. Technical Report UCB/EECS-2011-93, EECS Department, University of California, Berkeley, Aug 2011.

[13] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Parallel and Distributed Computing*, 10:673–693, 1999.

[14] A. Chapman and Y. Saad. Deflated and augmented Krylov subspace techniques. *Numer. Linear Algebra Appl*, 4:43–66, 1996.

[15] E. Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM Journal on Scientific Computing*, 21(5):1804–1822, 2000.

[16] E. Chow. Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns. *Int. J. High Perf. Comput. Appl*, 15:56–74, 2001.

[17] E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro, and U. M. Yang. *10. A Survey of Parallelization Techniques for Multigrid Solvers*, chapter 10, pages 179–201.

[18] E. Chow and A. Patel. Fine-grained parallel incomplete LU factorization. 2014.

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

[20] 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.

[21] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Sci. Comput.*, 34(1):206–239, February 2012.

[22] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in sparse matrix computations. In *In Proceedings of International Parallel and Distributed Processing Symposium*, 2008.

[23] J. W. Demmel, M. T. Heath, and H. A. van der Vorst. Parallel numerical linear algebra. Technical Report UCB/CSD-92-703, EECS Department, University of California, Berkeley, Oct 1992.

[24] S. Doi and T. Washio. Ordering strategies and related techniques to overcome the trade-off between parallelism and convergence in incomplete factorizations. *Parallel Computing*, 25(1314):1995 – 2014, 1999.

[25] H. C. Elman. Relaxed and stabilized incomplete factorizations for non-self-adjoint linear systems. *BIT Numerical Mathematics*, 29(4):890–915, 1989.

[26] J. Erhel. A parallel GMRES version for general sparse matrices. *Electronic Transactions on Numerical Analysis*, 3:160–176, 1995.

[27] J. Erhel, K. Burrage, and B. Pohl. Restarted GMRES preconditioned by deflation. *Journal of Computational and Applied Mathematics*, 69(2):303 – 318, 1996.

[28] J. Erhel and F. Guyomarc'h. An augmented conjugate gradient method for solving consecutive symmetric positive definite linear systems. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1279–1299, 2000.

[29] Y. Erlangga and R. Nabben. Deflation and balancing preconditioners for Krylov subspace methods applied to nonsymmetric matrices. *SIAM Journal on Matrix Analysis and Applications*, 30(2):684–699, 2008.

[30] R. Fletcher. Conjugate gradient methods for indefinite systems. In G.Alistair Watson, editor, *Numerical Analysis*, volume 506 of *Lecture Notes in Mathematics*, pages 73–89. Springer Berlin Heidelberg, 1976.

[31] J. Frank and C. Vuik. On the construction of deflation-based preconditioners. *SIAM J. Sci. Comput.*, 23(2):442–462, February 2001.

[32] A. Gaul, M. Gutknecht, J. Liesen, and R. Nabben. A framework for deflated and augmented Krylov subspace methods. *SIAM Journal on Matrix Analysis and Applications*, 34(2):495–518, 2013.

[33] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[34] P. Ghysels, T. Ashby, K. Meerbergen, and W. Vanroose. Hiding global communication latency in the GMRES algorithm on massively parallel machines. *SIAM Journal on Scientific Computing*, 35(1):C48–C71, 2013.

[35] J. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal on Scientific and Statistical Computing*, 9(5):862–874, 1988.

[36] L. Giraud, J. Langou, and M. Rozloznik. The loss of orthogonality in the Gram Schmidt orthogonalization process. *Computers & Mathematics with Applications*, 50(7):1069 – 1075, 2005. Numerical Methods and Computational Mechanics.

[37] S. L. Graham, M. Snir, and C. A. Patterson, Editors, Committee on the Future of Supercomputing, National Research Council. *Getting Up to Speed: The Future of Supercomputing*. The National Academies Press, 2004.

[38] L. Grigori, J. W. Demmel, and H. Xiang. CALU: A Communication Optimal LU Factorization Algorithm. *SIAM J. Matrix Anal. Appl.*, 32(4):1317–1350, November 2011.

[39] L. Grigori and S. Moufawad. Communication Avoiding ILU0 Preconditioner. Submitted to SIAM Journal on Scientific Computing.

[40] L. Grigori and S. Moufawad. Communication Avoiding ILU0 Preconditioner. Research Report RR-8266, INRIA, March 2013.

[41] L. Grigori, S. Moufawad, and F. Nataf. Enlarged Krylov Subspace Conjugate Gradient methods for Reducing Communication. In preparation for submission.

[42] L. Grigori, F. Nataf, and S. Yousef. Robust algebraic Schur complement preconditioners based on low rank corrections. Rapport de recherche RR-8557, INRIA, July 2014.

[43] W. Gropp. Update on libraries for blue waters. http://jointlab.ncsa.illinoisedu/events/workshop3/pdf/presentations/Gropp-Update-on-Libraries.pdf.

[44] T. Gu, X. Liu, Z. Mo, and X. Chi. Multiple search direction conjugate gradient method I: methods and their propositions. *Int. J. Comput. Math.*, 81(9):1133–1143, 2004.

[45] M. H. Gutknecht. Block krylov space methods for linear systems with multiple right-hand sides: an introduction. 2006.

[46] F. Hecht. New development in FreeFem++. *J. Numer. Math.*, 20(3-4):251–265, 2012.

[47] M. R. Hestenes and E. Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, December 1952.

[48] M. Hoemmen. *Communication-Avoiding Krylov Subspace Methods*. PhD thesis, EECS Department, University of California, Berkeley, 2010.

[49] G. Karypis and V. Kumar. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0. http://www.cs.umn.edu/ metis, 2009.

[50] C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. Society for Industrial and Applied Mathematics, 1995.

[51] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[52] M. S. Khaira, G. L. Miller, and T. J. Sheffler. Nested dissection: A survey and comparison of various nested dissection algorithms. Technical report, 1992.

[53] S. A. Kharchenko and A. Yu. Yeremin. Eigenvalue translation based preconditioners for the GMRES(k) method. *Numer. Linear Algebra Appl.*, 2:51–77, 1995.

[54] L.Yu Kolotilina. Twofold deflation preconditioning of linear algebraic systems. I. Theory. *Journal of Mathematical Sciences*, 89(6):1652–1689, 1998.

[55] A. Kuzmin, M. Luisier, and O. Schenk. Fast methods for computing selected elements of the greens function in massively parallel nanoelectronic device simulations. In F. Wolf, B. Mohr, and D. an Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 533–544. Springer Berlin Heidelberg, 2013.

[56] C. Lanczos. Solution of systems of linear equations by minimized iterations. *J. Res. Natl. Bur. Stand*, 49:33–53, 1952.

[57] Xiaoye S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *toms*, 31(3):302–325, September 2005.

[58] X.S. Li, J .W. Demmel, J.R. Gilbert, L. Grigori, M. Shao, and I. Yamazaki. SuperLU Users' Guide. Technical Report LBNL-44289, Lawrence Berkeley National Laboratory, September 1999. url:http://crd.lbl.gov/ xiaoye/SuperLU/. Last update: August 2011.

[59] B. R. Lowery and J. Langou. Stability Analysis of QR factorization in an Oblique Inner Product. *ArXiv e-prints*, January 2014.

[60] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 36:1–36:12, New York, NY, USA, 2009. ACM.

[61] R. Morgan. A restarted GMRES method augmented with eigenvectors. *SIAM Journal on Matrix Analysis and Applications*, 16(4):1154–1171, 1995.

[62] R. Nicolaides. Deflation of conjugate gradients with applications to boundary value problems. *SIAM Journal on Numerical Analysis*, 24(2):355–365, 1987.

[63] D. P. O'Leary. The block conjugate gradient algorithm and related methods. *Linear Algebra and its Applications*, 29(0):293 – 322, 1980. Special Volume Dedicated to Alson S. Householder.

[64] M. Rozložník, M. Tuma, A. Smoktunowicz, and J. Kopal. Numerical stability of orthogonalization methods with a non-standard inner product. *BIT Numerical Mathematics*, 52(4):1035–1058, 2012.

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

[66] Y. Saad and M. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[67] Y. Saad, M. Yeung, J. Erhel, and F. Guyomarc'h. A deflated version of the conjugate gradient algorithm. *SIAM J. Sci. Comput.*, 21(5):1909–1926, December 1999.

[68] Y. Saad and Jun Zhang. A multi-level preconditioner with applications to the numerical simulation of coating problems. In *Iterative Methods in Scientific Computing II*, pages 437–449. IMACS, 1998.

[69] O. Schenk and K. Gartner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20:475–487, 2004.

[70] J. Scott and M. Tma. On positive semidefinite modification schemes for incomplete Cholesky factorization. *SIAM Journal on Scientific Computing*, 36(2):A609–A633, 2014.

[71] A. Smoktunowicz, J. L.. Barlow, and J. Langou. A note on the error analysis of classical Gram Schmidt. *Numerische Mathematik*, 105(2):299–313, 2006.

[72] A. Stathopoulos and K. Wu. A block orthogonalization procedure with constant synchronization requirements. *SIAM Journal on Scientific Computing*, 23(6):2165–2182, 2002.

[73] J.M. Tang, R. Nabben, C. Vuik, and Y.A. Erlangga. Comparison of two-level preconditioners derived from deflation, domain decomposition and multigrid methods. *Journal of Scientific Computing*, 39(3):340–370, 2009.

[74] R. Thakur. Improving the performance of collective operations in mpich. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number 2840 in LNCS, Springer Verlag (2003) 257267 10th European PVM/MPI Users Group Meeting*, pages 257–267. Springer Verlag, 2003.

[75] H. Van der Vorst. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644, 1992.

[76] H. Van der Vorst and C. Vuik. The superlinear convergence behaviour of GMRES. *Journal of Computational and Applied Mathematics*, 48(3):327 – 341, 1993.

[77] J. van Rosendale. Minimizing inner product data dependence in conjugate gradient iteration. In *IEEE International Conference on Parallel Processing*, 1983.

[78] B. Vital. *Étude de quelques méthodes de résolution de problèmes linéaires de grande taille sur multiprocesseur*. PhD thesis, Université de Rennes, 1990.

[79] H. F. Walker. Implementation of the GMRES method using Householder transformations. *SIAM J. Sci. Stat. Comput.*, 9(1):152–163, January 1988.

[80] U. M. Yang. Parallel algebraic multigrid methods high performance preconditioners. In A. M. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 209–236. Springer Berlin Heidelberg, 2006.

Appendix A

ILU(0) preconditioned GMRES convergence for different reorderings

We compare the convergence of the ILU(0) preconditioned GMRES where the matrix A is reordered using nested dissection, kway, kway+AMML(1)V1, kway+AMML(5)V1, kway+AMML(1)V2, kway+AMML(5)V2, to the case where A is reordered using RCM or in natural ordering for the matrices .

Table A.1: ILU(0) preconditioned GMRES convergence for different reorderings with respect to number of partitions for the initial guess $x_0 = 0$ and $tol = 10^{-8}$.

		NO		RCM		ND		Kway		AMML(1)V1		AMML(5)V1		AMML(1)V2		AMML(5)V2	
	Pa	Iter	Err	Iter	Err	Iter	Err	Iter	Err	Iter	Err	Iter	Err	Iter	Err	Iter	Err
WATT2	2	50	3E-8	47	3E-8	64	2E-7	50	4E-8	51	3E-8	52	1E-7	50	3E-8	54	1E-7
	4					64	7E-8	50	3E-8	52	3E-8	53	3E-8	53	5E-8	58	6E-8
	8					65	4E-8	50	2E-8	51	8E-8	54	1E-7	56	2E-8	57	8E-8
	16					65	1E-7	51	3E-8	54	3E-8	54	1E-8	57	8E-8	58	8E-8
	32					65	1E-7	51	3E-8	52	1E-7	52	1E-7	56	1E-7	56	1E-7
	64					63	5E-8	54	1E-7	53	9E-8	52	3E-7	57	1E-7	57	1E-7
CD20P1	2	59	2E-8	62	4E-8	94	1E-7	59	3E-8	60	4E-8	83	4E-8	60	4E-8	83	8E-8
	4					95	9E-8	60	2E-8	65	4E-8	85	1E-7	67	3E-8	89	1E-7
	8					91	1E-7	59	3E-8	72	9E-8	97	1E-7	73	8E-8	98	1E-7
	16					94	1E-7	64	7E-8	76	9E-8	92	1E-7	76	4E-8	92	9E-8
	32					91	1E-7	63	6E-8	83	6E-8	89	1E-7	85	5E-8	93	1E-7
	64					89	5E-8	63	4E-8	87	5E-8	90	5E-8	90	5E-8	92	5E-8
CD50P1	2	64	4E-8	66	1E-7	96	1E-7	64	2E-8	66	5E-8	79	1E-7	67	3E-8	79	7E-8
	4					98	7E-8	64	5E-8	69	5E-8	79	1E-7	71	4E-8	83	1E-7
	8					96	1E-7	64	3E-8	71	7E-8	94	8E-8	77	1E-7	98	1E-7
	16					94	9E-8	71	2E-8	81	5E-8	98	1E-7	83	8E-8	97	7E-8
	32					96	1E-7	72	4E-8	87	7E-8	98	1E-7	92	5E-8	94	1E-7
	64					93	7E-8	71	5E-8	87	1E-7	95	1E-7	94	1E-7	95	1E-7
CD100P1	2	69	8E-8	71	3E-8	102	1E-7	70	1E-7	68	8E-8	78	1E-7	69	7E-8	80	1E-7
	4					97	2E-7	71	8E-8	75	1E-7	96	1E-7	74	1E-7	96	1E-7
	8					100	1E-7	71	8E-8	81	1E-7	107	2E-7	82	1E-7	101	1E-7
	16					105	1E-7	77	7E-8	89	6E-8	110	1E-7	89	8E-8	105	1E-7
	32					97	1E-7	73	4E-8	92	5E-8	106	1E-7	94	9E-8	105	1E-7
	64					96	7E-8	76	6E-8	96	7E-8	98	1E-7	97	7E-8	103	1E-7
CD500P1	2	92	1E-7	118	1E-7	186	4E-7	93	1E-7	94	1E-7	98	1E-7	94	1E-7	99	1E-7
	4					176	7E-7	93	1E-7	98	1E-7	113	2E-7	98	2E-7	119	2E-7
	8					174	7E-7	93	1E-7	102	1E-7	117	1E-7	105	1E-7	115	2E-7
	16					184	2E-7	94	1E-7	107	1E-7	167	1E-7	108	1E-7	180	1E-7
	32					190	6E-7	99	1E-7	139	1E-7	188	1E-7	144	1E-7	190	1E-7
	64					198	1E-7	102	1E-7	167	2E-7	177	1E-7	161	1E-7	178	1E-7
CD20P2	2	146	1E-7	155	2E-7	162	3E-7	147	2E-7	151	2E-7	158	2E-7	151	2E-7	154	3E-7
	4					163	3E-7	143	2E-7	147	2E-7	156	2E-7	147	2E-7	153	3E-7
	8					160	3E-7	142	2E-7	148	2E-7	167	3E-7	148	2E-7	161	2E-7
	16					158	3E-7	150	2E-7	158	2E-7	170	3E-7	154	2E-7	151	2E-7
	32					161	3E-7	150	3E-7	156	2E-7	164	3E-7	152	2E-7	154	3E-7
	64					162	3E-7	149	2E-7	163	2E-7	169	2E-7	156	3E-7	159	3E-7
CD100P2	2	157	2E-7	170	3E-7	177	2E-7	158	2E-7	159	2E-7	159	2E-7	159	3E-7	160	3E-7
	4					177	2E-7	152	2E-7	155	1E-7	155	1E-7	159	3E-7	154	3E-7
	8					183	2E-7	156	2E-7	164	4E-7	178	2E-7	163	2E-7	162	3E-7
	16					175	4E-7	153	3E-7	165	3E-7	171	2E-7	163	3E-7	163	3E-7
	32					167	3E-7	155	3E-7	164	2E-7	182	2E-7	158	3E-7	162	2E-7
	64					178	3E-7	162	3E-7	176	3E-7	184	3E-7	176	3E-7	169	2E-7
CD400P2	2	278	4E-7	244	2E-7	369	2E-7	277	4E-7	278	4E-7	276	4E-7	278	4E-7	280	4E-7
	4					339	3E-7	278	5E-7	279	5E-7	302	3E-7	280	4E-7	282	5E-7
	8					354	3E-7	275	4E-7	300	5E-7	305	2E-7	301	5E-7	304	5E-7
	16					391	2E-7	280	4E-7	308	4E-7	313	2E-7	313	3E-7	293	2E-7
	32					353	2E-7	287	4E-7	317	4E-7	327	3E-7	312	5E-7	363	4E-7
	64					335	3E-7	279	2E-7	316	3E-7	314	5E-7	289	4E-7	330	3E-7