parGeMSLR: a Parallel Multilevel Schur Complement Low-Rank Preconditioning and Solution Package for General Sparse Matrices

Tianshi Xu\textsuperscript{a,∗}, Vassilis Kalantzis\textsuperscript{b}, Ruipeng Li\textsuperscript{c}, Yuanzhe Xi\textsuperscript{d}, Geoffrey Dillon\textsuperscript{e}, Yousef Saad\textsuperscript{a}

\textsuperscript{a}Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455
\textsuperscript{b}Thomas J. Watson Research Center, IBM Research, Yorktown Heights, NY 10598
\textsuperscript{c}Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, P.O. Box 808, L-561, Livermore, CA 94551
\textsuperscript{d}Department of Mathematics, Emory University, Atlanta, GA 30322
\textsuperscript{e}Department of Mathematics, University of South Carolina, Columbia, SC 29208

Abstract

This paper discusses parGeMSLR, a C++/MPI software library for the solution of sparse systems of linear algebraic equations via preconditioned Krylov subspace methods in distributed-memory computing environments. The preconditioner implemented in parGeMSLR is based on algebraic domain decomposition and partitions the symmetrized adjacency graph recursively into several non-overlapping partitions via a p-way vertex separator, where p is an integer multiple of the total number of MPI processes. From a numerical perspective, parGeMSLR builds a Schur complement approximate inverse preconditioner as the sum between the matrix inverse of the interface coupling matrix and a low-rank correction term. To reduce the cost associated with the computation of the approximate inverse matrices, parGeMSLR exploits a multilevel partitioning of the algebraic domain. The parGeMSLR library is implemented on top of the Message Passing Interface and can solve both real and complex linear systems. Furthermore, parGeMSLR can take advantage of hybrid computing environments with in-node access to one or more Graphics Processing Units. Finally, the parallel efficiency (weak and strong scaling) of parGeMSLR is demonstrated on a few model problems arising from discretizations of 3D Partial Differential Equations.

Keywords: Schur complement, low-rank correction, distributed-memory preconditioner, sparse non-Hermitian linear systems, Graphics Processing Units

1. Introduction

This paper discusses a distributed-memory library for the iterative solution of systems of linear algebraic equations of the form

\[ Ax = b, \]  

where the matrix \( A \in \mathbb{C}^{n \times n} \) is large, sparse, and (non-)Hermitian. Problems of this form typically originate from the discretization of a Partial Differential Equation in 2D or 3D domains.

Iterative methods solve (1) by a preconditioned Krylov subspace iterative methods \([1,2]\), e.g., preconditioned Conjugate Gradient \([1]\), if \( A \) is Hermitian and positive-definite, or GMRES \([3]\) if \( A \) is non-Hermitian. The role of the preconditioner is to cluster the eigenvalues in an effort to accelerate the convergence of Krylov subspace method. For example, an efficient right preconditioner \( M \) transforms (1) into the preconditioned system \( M^{-1}(Ax) = M^{-1}b \), where \( M^{-1} \) can be applied inexpensively. An additional requirement is that the setup and application of the operator \( M^{-1} \) should be easily parallelizable.

Similarly to Krylov subspace methods, algebraic multigrid (AMG) methods are another widely-used class of iterative solvers \([4]\). AMG uses the ideas of interpolation and restriction to build multilevel preconditioners that eliminate the smooth error components. AMG is provably optimal for Poisson-like problems on regular meshes where the number of iterations to achieve convergence almost stays constant as the problem size increases. This property leads to appealing weak scaling results of AMG in distributed-memory computing environments \([5,6,7]\). However, AMG can fail when applied either to indefinite problems or irregular meshes. It is worth mentioning that AMG can also be used as a preconditioner in the context of Krylov subspace methods.

For general sparse linear systems, a well-known class of general-purpose preconditioners is that of Incomplete LU (ILU) factorization preconditioners \([8,9,1]\). Here, the matrix \( A \) is approximately factored as \( A \approx LU \) where \( L \) is lower triangular and \( U \) is upper triangular, and the preconditioner is defined as \( M = LU \). Applying \( M^{-1} \) then consists of two triangular substitutions. ILU preconditioners can be applied to a greater selection of problems than AMG, including indefinite problems such as discretized Helmholtz equations \([10,11]\), and their robustness can be improved by modified/shifted ILU strategies \([12,13,14]\). On the other hand, the scalability of ILU preconditioned Krylov subspace methods is typically inferior com-
pared to AMG. In particular, even for Poisson-like problems, the number of iterations to achieve convergence by ILU preconditioned Krylov subspace methods increases with respect to the matrix size. Moreover, the sequential nature of triangular substitutions limit the parallel efficiency of ILU preconditioners implemented on distributed-memory systems, and recent efforts have been focusing on improving their scalability, e.g., see [15][16][17].

The parallel efficiency of ILU preconditioners can be enhanced by domain decomposition (DD), where the original problem is decomposed into several subdomains which correspond to different blocks of rows of the coefficient matrix A. The simplest DD-based ILU approach is the block-Jacobi ILU preconditioner, where a local ILU is performed on each local submatrix. Since this method ignores all of the off-diagonal matrices corresponding to inter-domain couplings, its convergence rate tends to become slower as the number of subdomains increases, and several strategies have been proposed to handle the inter-domain couplings in order to improve the convergence rate. Restricted Additive Schwarz (RAS) methods expand the local matrix by a certain level to gain a faster convergence rate at the cost of losing some memory scalability [18]. Global factorization ILU methods factorize local rows corresponding to interior unknowns first, after which a global factorization of the couplings matrix is applied based on some graph algorithms [19][20]. These methods use partial ILU techniques with dropping [21][22], incomplete triangular solve [23], and low-rank approximation [24] to form the Schur complement system and can be generalized into multilevel ILU approaches [21][22][24]. When the Finite Element method is used and the elements are known, two-level DD methods including BDDC [25] and FEITI-DP [26][27], as well as the GenEO preconditioner [28] are also have been shown to be effective approaches. We note that an additional strategy is to combine approximate direct factorization techniques with low-rank representation of matrix blocks [29], ParTix [30], and DDLR [31]. When the matrix A is SPD, it is possible to reduce the size of the Schur complement matrix without introducing any fill-in, e.g., see SpaND [32].

Other preconditioning strategies that can be implemented on distributed-memory environments include the (factorized) sparse approximate inverse preconditioners [33][34][35][36][37], polynomial preconditioners [38], and rank-structured preconditioners [39][40][41][42]; see also [43] for a distributed-memory hierarchical solver. Some of the these techniques can be further compounded with AMG, as smoothers”, or ILU-based preconditioners. For example, a combination of SLR [44] and polynomial preconditioning is discussed in [45].

1.1. Contributions of this paper

This paper discusses the implementation of a distributed-memory library, termed parGeMSLR, for the iterative solution of sparse systems of linear algebraic equations in large-scale distributed-memory computing environments. parGeMSLR is written in C++, and communication among different processor groups is achieved by means of the Message Passing Interface standard (MPI). The parGeMSLR library is based on the Generalized Multilevel Schur complement Low-Rank (GeMSLR) algorithm described in [24]. GeMSLR applies a multilevel partitioning of the algebraic domain, and the variables associated with each level are divided into either interior or interface variables. The multilevel structure is built by applying a p-way graph partitioner to partition the induced subgraph associated with the interface variables of the preceding level. Once the multilevel partitioning is completed, GeMSLR creates a separate Schur complement approximate inverse at each level. Each approximate inverse is the sum of two terms, with the first term being an approximate inverse of the interface coupling matrix, and the second term being a low-rank correction which aims at bridging the gap between the first term and the actual Schur complement matrix inverse associated with that level. Below, we summarize the main features of the parGeMSLR library:

1. Scalability. parGeMSLR extends the capabilities of low-rank-based preconditioners, such as GeMSLR, by recursively partitioning the algebraic domain into levels which have the same number of partitions. In turn, this leads to enhanced scalability when running on distributed-memory environments.

2. Robustness and complex arithmetic. In contrast to ILU preconditioners, the numerical method implemented in parGeMSLR is less sensitive to indefiniteness and can be updated on-the-fly without discarding previous computational efforts. Additionally, parGeMSLR supports complex arithmetic and thus can be utilized to solve complex linear systems such as those originating from the discretization of Helmholtz equations.

3. Hybrid hardware acceleration. GPU acceleration is supported in several iterative solver libraries aiming to speed-up the application of preconditioners such as AMG or ILU, e.g., hypre [45], PARALUTION [46], ViennaCL [47], HIFLOW [48], PETSc [49], and Trilinos [50]. A number of direct solver libraries including STRUMPACK [51][52][53] and SuperLU_DIST [54] also provide GPU support. Similarly, parGeMSLR can exploit one or more GPUs by offloading any computation for which the user provides a CUDA interface.

This paper is organized as follows. Section 2 discusses low-rank correction preconditioners and provides an algorithmic description of parGeMSLR. Section 3 provides details on the multilevel reordering used by parGeMSLR. Section 4 presents in-depth discussion and details related to the implementation and parallel performance aspects of parGeMSLR. Section 5 demonstrates the performance of parGeMSLR on distributed-memory environments. Finally, our concluding remarks are presented in Section 6.

---

1The abbreviation of the library is derived by the complete name parallel Generalized multilevel Schur complement Low-Rank preconditioner.

2The source code can be found in [https://github.com/Hitenze/pargemslr](https://github.com/Hitenze/pargemslr)
2. Schur complement approximate inverse preconditioners via low-rank corrections

This section discussed the main idea behind (multilevel) Schur complement preconditioners enhanced by low-rank corrections, e.g., see [55, 44, 21, 56].

2.1. The Schur complement viewpoint

Let the linear system $Ax = b$ be permuted as

$$A_0 x = P^T A P (P^T x) = P^T b,$$

where $P$ is an $n \times n$ permutation matrix such that

$$A_0 = \begin{bmatrix} B & F \\ E & C \end{bmatrix} = \begin{bmatrix} B^{(1)} & F^{(1)} \\ B^{(2)} & F^{(2)} \\ \vdots & \vdots \\ B^{(p)} & F^{(p)} \\ E^{(1)} & E^{(2)} & \ldots & E^{(p)} & C \end{bmatrix},$$

and the matrices $B^{(i)}, F^{(i)},$ and $E^{(i)}$ are of size $d_i \times d_i, d_i \times s,$ and $s \times d_i,$ respectively. The matrix $C$ is of size $s \times s,$ and the matrix partitioning satisfies $d + s = \sum_{i=1}^{p} d_i + s_i = n.$ Such matrix partitions can be computed by partitioning the adjacency graph of the matrix $|A| + |A^T|$ into $p \in \mathbb{N}$ non-overlapping partitions and reordering the unknowns/equations such that the variables associated with the $d$ interior nodes across all partitions are ordered before the variables associated with the $s$ interface nodes.

Following the above notation, the linear system in (2) can be written in a block form

$$\begin{bmatrix} B & F \\ E & C \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix},$$

where $u, f \in \mathbb{R}^d$ and $v, g \in \mathbb{R}^s.$ Once the solution in (3) is computed, the solution $x$ of the original, non-permuted system of linear algebraic equations $Ax = b$ can be obtained by the inverse permutation $x = P \begin{bmatrix} u \\ v \end{bmatrix}.$ Throughout the rest of this section we focus on the solution of the system in (3).

Following a block-LDU factorization of the matrix $A_0$, the permuted linear system in (2) can be written as

$$\begin{bmatrix} I & 0 \\ EB^{-1} I & S \end{bmatrix} \begin{bmatrix} B \\ S \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix},$$

where $S = C - EB^{-1}F$ denotes the Schur complement matrix. The solution of (3) is then equal to

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} I & 0 \\ EB^{-1} I & S^{-1} \end{bmatrix} \begin{bmatrix} B^{-1} & I \\ I & -EB^{-1} \end{bmatrix} \begin{bmatrix} f \\ g \end{bmatrix},$$

which requires: a) the solution of two linear systems with the block-diagonal matrix $B$, and $b)$ the solution of one linear system with the the Schur complement matrix $S$. Note that since the matrix $B$ is block-diagonal, the associated linear systems are decoupled into $p$ independent systems of linear algebraic equations. Assuming a distributed-memory computing environment with $p$ separate processor groups, each system of linear algebraic equations can be solved in parallel by means of applying a direct solver locally in each separate process.

In several real-world applications, e.g., those involving the discretization of PDEs on three-dimensional domains, solving the systems of linear algebraic equations with matrices $B$ and $S$ through a direct solver is generally impractical, primarily due to the large computational and memory cost associated with forming and factorizing the Schur complement matrix. An alternative then is to solve the linear systems with matrices $B$ and $S$ inexactly. For example, the solution of linear systems with matrix $B$ can be computed approximately by replacing its exact LU factorization with an incomplete threshold LU (ILUT) [8]. Likewise, the exact Schur complement can be sparsified by discarding entries below a certain threshold value or located outside a pre-determined pattern [22, 57].

The approximate factorizations of the matrices $B$ and $S$ can be combined to form an approximate LDU factorization of (4) which can be then used as a preconditioner in a Krylov subspace iterative solver such as GMRES.

2.2. Schur complements and low-rank corrections

One of the main drawbacks associated with incomplete factorizations is that they can not be easily updated if one needs a more accurate preconditioner unless the iterative ParILUT [15, 16] works for the problem and is used. Moreover, their robustness can be limited when the matrix $A$ is indefinite. For such scenarios, it has been advocated to add a low-rank correction term to enhance the efficiency of the Schur complement preconditioner, without discarding the previously computed incomplete factorizations. The low-rank enhancement implemented in parGeMSLR follows the GeMSLR multilevel preconditioner [23], a non-Hermitian extension of [44, 50]. Other approaches based on low-rank corrections can be found in [55, 58].

The GeMSLR preconditioner expresses the Schur complement matrix as

$$S = (I - EB^{-1}F^{-1})C = (I - G)C,$$

where $G = EB^{-1}F^{-1}$. Consider now the complex Schur decomposition $G = WRW^H$, where the $s \times s$ matrix $W$ is unitary and the $s \times s$ matrix $R$ is upper-triangular such that its diagonal entries contain the eigenvalues of matrix $G$. Plugging the latter in (5) results to

$$S = (I - WRW^H)C = W(I - R)W^H C,$$

from which we can write the inverse of the Schur complement matrix as (Sherman-Morrison-Woodbury formula):

$$S^{-1} = C^{-1} + C^{-1}W[(I - R)^{-1} - I]W^H.$$

Following (6), a system of linear equations with the Schur complement matrix requires the solution of a system of linear equations with matrix $C$, as well as matrix-vector multiplications and triangular matrix inversions with matrices $W^H$ and $(I - R)^{-1}$, respectively. The product of matrices $W(I - R)^{-1} -
$I\omega H$ is a Schur decomposition by itself, with corresponding eigenvalues $\gamma_i/(1 - \gamma_i)$, $i = 1, \ldots, s$, where $\gamma_i$ denotes the $i$-th eigenvalue of the matrix $G$. Therefore, as long as the eigenvalues of the latter matrix are not located close to one, the matrix $C(S^{-1} - C^{-1}) = W(I - R^{-1}) - I\omega H$ can be approximated by a low-rank matrix, i.e., $S^{-1}$ is approximately equal to $C^{-1}$ plus some low-rank correction.

The expression in (5) can be transformed into a practical preconditioner if the matrix $W(I - R^{-1}) - I\omega H$ is replaced by a rank-$k$ approximation, where $k \in \mathbb{N}$ is generally a user-given parameter. More specifically, let $W_k$ denote the $s \times k$ matrix which holds the leading $k$ Schur vectors of matrix $G$, and let $R_k$ denote the $k \times k$ leading principal submatrix of matrix $R$. Then, the GeMSLR approximate inverse preconditioner is equal to

$$M^{-1} = C^{-1} + C^{-1}W_k[(I - R_k)^{-1} - I]\omega H_k \approx S^{-1}. \quad (7)$$

### 2.3. Computations with an incomplete factorization of $B$

For large-scale problems, computing an exact factorization of the block-diagonal matrix $B$ can be quite expensive. Instead, what is typically available is an ILUT factorization $LU \approx B$. Therefore, instead of computing a rank-$k$ Schur decomposition of matrix $G$, in practice we approximate a truncated Schur decomposition of the matrix $G = E(U^{-1}L^{-1})F^{-1}$. Let then

$$\hat{G} = \hat{V}n\hat{H} + \beta l\hat{v}m+1\hat{H},$$

denote an $m$-length Arnoldi relation obtained with matrix $\hat{G}$, where $\hat{V}_m\hat{v}_{m+1}\hat{H} = I$, and $\hat{H}$ is upper-Hessenberg. Moreover, let $\hat{H}_m = QTQ^H$ denote the truncated Schur decomposition of matrix $\hat{H}_m$. The low-rank correction term used in GeMSLR is of the form $\hat{W}_k[(I - \hat{R}_k)^{-1} - I]W_k^H$, where $T_k \in \mathbb{R}^{k \times k}$ denotes the $k \times k$ leading principal submatrix of matrix $T$, and $\hat{W}_k = \hat{V}_mQ_k$, where $Q_k \in \mathbb{R}^{s \times k}$ denotes the matrix holding the $k$ leading Schur vectors of matrix $\hat{H}_m$.

### 2.4. Multilevel extensions

For large-scale, high-dimensional problems, the application of the matrix $C^{-1}$ by means of an LU factorization of matrix $C$ can still be expensive; especially when the value of $p$ is too large, leading to large vertex separators. The idea suggested in [55, 51], and employed by GeMSLR, is to take advantage of the purely algebraic formulation developed in the previous section and apply $C^{-1}$ exactly by using the Schur complement low-rank preconditioner described in the previous section. In fact, this approach can be repeated more than once, leading to a multilevel preconditioner.

More specifically, let $l_{ev} \in \mathbb{N}$ denote the number of levels, and define the sequence of matrices

$$A_l = P_{l-1}C_{l-1}P_{l-1} = \begin{bmatrix} B_l & F_l \\ E_l & C_l \end{bmatrix}, \quad C_{l-1} = A, \quad l = 0, 1, \ldots, l_{ev} - 1,$$  

where the matrix $B_l$ is block-diagonal with $p$ on-diagonal matrix blocks. The $2 \times 2$ block matrix partition of each matrix $A_l$ is obtained by partitioning the adjacency graph of the matrix $|C_{l-1}| + |C_{l-1}^T|$ into $p$ non-overlapping partitions and reordering the unknowns/equations such that the variables associated with the interior nodes across all partitions are ordered before the variables associated with the interface nodes of the adjacency graph. The matrix $C_{l-1}$ is then permuted in-place through the $s_{l-1} \times s_{l-1}$ permutation matrix $P_{l-1}$, where $s_{l-1}$ denotes the size of the matrix $C_{l-1}$.

The solution of a system of linear algebraic equations with matrix $A_l$ as the coefficient matrix and $\begin{bmatrix} f_l^T & g_l^T \end{bmatrix}$ as the right-hand side, can be computed as

$$\begin{bmatrix} w_l \\ v_l \end{bmatrix} = \begin{bmatrix} I & -B_l^{-1}F_l \\ I & B_l^{-1} \\ I & -E_lB_l^{-1}I \end{bmatrix} \begin{bmatrix} f_l \\ g_l \end{bmatrix},$$

where $S_l = C_l - E_lB_l^{-1}F_l$ denotes the $s_l \times s_l$ Schur complement matrix associated with the $l$-th level, where $s_l \in \mathbb{N}$ denotes the size of the matrix $C_l$. Instead of computing the exact LU factorizations of matrices $R_l$ and $S_l$, the preconditioner implemented in the parGeMSLR library substitutes $B_l^{-1} \approx (L_lU_l)^{-1}$, where $L_lU_l$ denotes an ILUT factorization of matrix $B_l$.

$$S_{l-1}^{-1} = C_{l-1}^{-1} + C_{l-1}^{-1}W_k[(I - R_k)^{-1} - I]W_k^H. \quad (9)$$

where $\hat{W}_k$ denotes the matrix which holds the approximate leading $k$ Schur vectors of the matrix $\hat{G}_l = E_lU_l^{-1}L_l^{-1}F_lC_l^{-1}$, and $\hat{R}_k$ denotes the approximation of the $k \times k$ leading principal submatrix of the matrix $\hat{R}_k$ that satisfies the Schur decomposition $\hat{G}_l = \hat{V}_k\hat{R}_k\hat{W}_k$. Algorithm 1 summarizes the above discussion (setup phase) in the form of an algorithm. Notice that the recursion stops at level $l_{ev} - 1$, and an ILUT of the matrix $C_{l_{ev} - 1}$ is computed explicitly.

### Algorithm 1 Parallel GeMSLR Setup

1: procedure Parallel GeMSLR Setup($A$, $l_{ev}$)
2: Generate $l_{ev}$-level structure by Algorithm 2
3: for $l$ from 0 to $l_{ev} - 1$ do
4: Compute ILU factorization $L_lU_l \approx B_l$,
5: Compute matrices $\hat{W}_k$ and $\hat{R}_k$,
6: If $l = l_{ev} - 1$, compute an ILUT factorization $L_{l_{ev} - 1}U_{l_{ev} - 1} \approx C_{l_{ev} - 1}$, exit.
7: end for
8: end procedure

Algorithm 2 outlines the procedure associated with the application of the GeMSLR preconditioner (solve phase). At each level, the preconditioning step consists of a forward and backward substitution with the ILUT triangular factors of $B_l$, followed by the application of the rank-$k$ correction term. When $l = l_{ev} - 1$, there is no low-rank correction term applied, since this is the last level. Moreover, when $l = 0$ (root level), it is possible to enhance the GeMSLR preconditioner by applying a few steps of right preconditioned GMRES. Note though that these iterations are performed with the inexact Schur complement $\hat{S}_l = C_l - E_l(U_l^{-1}L_l^{-1})F_l$.

### 3. Multilevel reordering

This section outlines the multilevel reordering approach implemented in the parGeMSLR library. For simplicity, we focus
Algorithm 2 Standard Parallel GeMSLR Solve

1: procedure rGeMSLR\(\text{Solve}(b, l)\)
2: \hspace{1em} Apply reordering \(b_1 = P_{1-}b,\)
3: \hspace{1em} Solve \(z_1 = U_1^{-1}L_1^{-1}b_1,\)
4: \hspace{1em} Compute \(z_2 = b_2 - E_zz_1,\)
5: \hspace{1em} if \(l = 0\) then
6: \hspace{2em} Solve \(S_1y_2 = z_2\) by right preconditioned GMRES.
7: \hspace{1em} else
8: \hspace{2em} Compute \(u_2 = \bar{W}_{l,k}((I - \bar{R}_{l,k})^{-1} - I)\bar{W}_{l,k}^Hz_2\)
9: \hspace{2em} Call \(y_2 = p\text{GeMSLR}\text{Solve}(u_2 + z_2, l + 1).\)
10: \hspace{1em} end if
11: \hspace{1em} Compute \(y_1 = z_1 - U_1^{-1}L_1^{-1}F_1y_2,\)
12: \hspace{1em} Apply reordering \(q = P_{l-1}^\top y_1\)
13: \hspace{1em} return \(y\)
14: \hspace{1em} end procedure

Algorithm 3 Parallel GeMSLR Reordering

1: procedure pGeMSLR\text{REORDERING}(A, \(l_{ev}\))
2: \hspace{1em} Set \(C_{-1} \equiv A.\)
3: \hspace{1em} for \(l = 0\) to \(l_{ev} - 1\) do
4: \hspace{2em} Apply \(p\)-way partitioning to the graph associated with the matrix \(|C_{-1}| + |C_{-2}|,\)
5: \hspace{2em} Set \(A_l = P_{l-1}C_{l-1}P_{l-1} = \begin{bmatrix} B_l & F_l \\ E_l & C_l \end{bmatrix}.\)
6: \hspace{2em} end for
7: \hspace{1em} return
8: \hspace{1em} end procedure

3.2. Multilevel partitioning through \(p\)-way vertex separators

In contrast to low-rank correction preconditioners such as MSLR and GeMSLR \(\cite{56, 24},\) the main goal of parGeMSLR is to sustain good parallel efficiency, and thus HID is not appropriate. Instead, the default approach in parGeMSLR is to partition the adjacency graph by a multi-level partitioner where each level consists of \(p\) partitions and a vertex separator. The latter choice results to a fixed number of \(p\) partitions at each level, and thus load balancing is generally much better than that obtained using HID.

A high-level description can be found in Algorithm 3. At the root level \((l = 0)\), the graph associated with the matrix \(|A| + |A^T|,\) is partitioned into \(p\) subdomains with a \(p\)-way vertex separator, resulting to \(p\) non-overlapping connected components and their associated vertex separator. The multilevel partitioner then proceeds to the next level, \(l = 1,\) and applies the \(p\)-way vertex partitioner to the induced subgraph associated with the vertex separator at level \(l = 0\). This leads to a second set of \(p\) non-overlapping connected components and a new, albeit smaller vertex separator. The \(p\)-way vertex partitioner is then applied again to the induced subgraph associated with the vertex separator obtained at level \(l = 1,\) etc. The procedure continues until either level \(l_{ev} - 1\) is reached, or the vertex separator at the current level \(l\) has so few vertices that it can not be further partitioned into \(p\) non-overlapping partitions.

An illustration of a three-level, four-way partitioner applied to a three-dimensional algebraic domain (a unit cube) is shown in Figure 4. The leftmost subfigure shows the \(p = 4\) separate partitions obtained by the application of the four-way vertex partitioner as well as the vertex separator itself (shown in white color) at level \(l = 0\). This vertex separator, which consists of four two-dimensional faces, forms the algebraic object to be partitioned at level \(l = 1,\) and the partitioning is shown in the middle subfigure, where this time the vertex separator is a one-dimensional object. Finally, at level \(l = 2,\) the most recent vertex separator is further partitioned into four independent partitions, leading to a new vertex separator which consists of only three vertices; see the rightmost subfigure.

In addition to the above illustration, Figure 2 plots the sparsity pattern of a Finite Difference discretization of the Laplace operator on a three-dimensional domain, after reordering its rows and columns according to a \(p\)-way, multilevel reordering with \(l_{ev} = 4\) and \(p = 4\) (left). A zoom-in of the submatrix associated with the permutation of the vertex separators is also shown (right). Note that in this particular example, the last level has already too few variables to be partitioned any further. In addition to the global, multilevel permutation, each matrix \(B_l\) can be further permuted locally by a reordering scheme such as reverse Cuthill-McKee (RCM) algorithm or approximate minimum degree algorithm (AMD) \(\cite{64, 65}\) to reduce the fill-ins.

4. Implementation details of parGeMSLR

The parGeMSLR library consists of three main modules: \(a\) a distributed-memory reordering scheme, \(b\) a Krylov subspace

\footnote{Nonetheless, HID is offered in parGeMSLR.}
4.1. Distributed-memory operations in Krylov accelerators

Standard, non-preconditioned Krylov iterative methods are built on top of simple linear algebraic operations such as matrix-vector multiplication, vector scaling and additions, and DOT products. Iterative solvers such as GMRES or FGMRES also require the solution of small-scale ordinary linear-least squares problems which are typically solved redundantly in each MPI process.

Assuming that the data associated with the system of linear algebraic equations we wish to solve is already distributed across the different MPI process via 1D row distribution, AXPY operations can be executed locally and involve no communication overhead. On the other hand, sparse matrix-vector multiplications and DOT products involve either point-to-point or collective communication. In particular, assume \( n_p \in \mathbb{N} \) MPI processes. A DOT product then requires a collective operation, i.e., \( \text{MPI}_\text{Allreduce} \), to sum the \( n_p \) local DOT products. The cost of this operation is roughly \( O(\log(n_p)\alpha) \), where \( \alpha \in \mathbb{R} \) denotes the maximum latency between two MPI process. On the other hand, a matrix-vector multiplication with the coefficient matrix of the linear system requires point-to-point communication, where the local matrix-vector product in each MPI process consists of operations using local data, as well as data
associated with MPI processes which are assigned to neighboring subdomains, e.g., see \[66\] for additional details and recent advances.

4.2. Preconditioner setup and application

The main module of parGeMSLR is the setup of the GeMSLR preconditioner, followed by the application of the latter at each iteration of the Krylov subspace iterative solver of choice. Following a multilevel partition into \(l_e\) levels (see Section 3), the setup phase of the GeMSLR preconditioner associated with each level \(l = 0, 1, \ldots, l_e - 1\), is further divided into two separate submodules: a) computation of an ILUT factorization \(B_l \approx L_lU_l\), and b) computation of an approximate rank-\(k\) Schur decomposition of the matrix \(G_l = E_l^1U_l^{-1}L_l^{-1}F_lC_l^{-1}\).

Let us consider each one of the above two tasks separately. Recall that the data matrix at each level \(0 \leq l \leq l_e - 1\) has the following pattern

\[
A_l = P_lC_{l-1}P_l = \begin{bmatrix} B_l^1 & F_l^1 \\ E_l & C_l \\ \vdots & \vdots \\ B_l^p & F_l^p \\ E_l^p & C_l \end{bmatrix}.
\]

Now, without loss of generality, assume that each partition is assigned to a separate MPI process. Figure 3 (left) plots a graphical illustration of the data layout of matrix \(A_l\) obtained by a permutation using \(p = 4\), across four different MPI processes. Data associated with separate MPI processes are presented with a different color. Notice that the right-bottom submatrix denotes the matrix \(C_l\) representing the coupling between variables of the vertex separator at level \(l\). Computing an ILUT factorization of the matrix \(B_l\) decouples into \(p\) independent ILUT subproblems \(B_l^{(j)} = L_l^{(j)}U_l^{(j)}, j = 1, \ldots, p\), and thus no communication overhead is enabled. On the other hand, the computation of the low-rank correction term requires the application of several steps of the Arnoldi iteration, and requires communication overhead.

More specifically, the Arnoldi iteration requires communication among the various MPI processes to compute matrix-vector multiplications with the iteration matrix \(G_l\), as well as to maintain orthogonality of the Krylov basis. When the latter is achieved by means of standard Gram-Schmidt, Arnoldi requires one MPI\_Allreduce operation at each iteration. Similarly, the matrix-vector multiplication between \(G_l\) and a vector \(z\) is equal to

\[
E_l^1 \quad \cdots \quad E_l^p \quad \begin{bmatrix} L_l^{(1)}U_l^{(1)} & \ldots & F_l^1 \\ \vdots & \vdots & \vdots \\ L_l^{(p)}U_l^{(p)} & \ldots & F_l^p \end{bmatrix}^{-1} \begin{bmatrix} C_l \end{bmatrix}^T z.
\]

The computation of the product \(C_l^{-1}z\) requires access to the incomplete ILUT factorizations and rank-\(k\) correction terms associated with all levels \(l < l_e\). Once the matrix-vector multiplication \(C_l^{-1}z\) is computed, the matrix-vector multiplication with matrix \(F_l\) is computed with trivial parallelism among the MPI processes, and the same holds for the linear system solutions with matrices \(L_l^{(j)}U_l^{(j)}, j = 1, \ldots, p\). Finally, the matrix-vector multiplication with matrix \(E_l\) requires an MPI\_Allreduce operation. Note that if we were to replace vertex separators with edge separators (this option is included in parGeMSLR) then the latter multiplication would also be communication-free.

Finally, applying the preconditioner requires embarrassingly parallel triangular substitutions with the ILUT factorizations of the block-diagonal matrices \(B_l\) as well as dense matrix-vector multiplications with matrices \(W_{l,k}, W_{l,k}^H\), and \((I - R_{l,k})^{-1}\). A matrix-vector multiplication with the matrix \(W_{l,k}\) requires no communication among the MPI processes, while a matrix-vector multiplication with the matrix \(W_{l,k}^H\) requires an MPI\_Allreduce operation at level \(l\). Finally, the matrix-vector multiplication with the \(k \times k\) matrix \((I - R_{l,k})^{-1}\) is performed redundantly in each MPI process since \(k\) is typically pretty small.

4.2.1. Communication overhead analysis

In this section we focus on the communication overhead associated with setting up and applying the preconditioner implemented in parGeMSLR. For simplicity, we assume that the number of MPI processes \(n_p\) is equal to the number of partitions \(p\) at each level. The main parameters of the preconditioner are the number of levels \(l_e\) and the value of rank \(k\).

Let us first consider the application of \(m\) Arnoldi iterations to compute the matrices \(W_{l,k}\) and \((I - R_{l,k})^{-1}\) for some \(0 \leq l \leq l_e - 1\). As was discussed in the previous section, computing matrix-vector products with the matrix \(G_l\) requires communication only during the application of the matrices \(E_l\) and \(C_l^{-1}\). In turn, the latter requires computations with the distributed matrices \(C_i^{-1}, W_{l+1,k}, C_i^{-1}, W_{l+1,k}^H\), and so on, until we reach level \(l_e - 1\) where an ILUT of the matrix \(C_{l_e-1}\) is computed explicitly. Thus, a matrix-vector multiplication with the matrix \(G_l\) requires \(l_e - (l + 1)\) (low-rank correction term) and \(l_e - l\) \((C_l^{-1}\) recursion) MPI\_Allreduce operations. In summary, an \(m\)-length Arnoldi cycle with standard Gram-Schmidt orthonormalization requires \((2l_e - 2l + 1)m\) MPI\_Allreduce operations, where we also accounted for the two MPI\_Allreduce operations stemming from Gram-Schmidt and vector normalization at each iteration. This communication overhead is inversely proportional to the level index \(l\). Accounting for all \(l_e - 1\) levels, the total communication overhead associated with the setup phase of the preconditioner amounts is bounded by \(\delta(k) \prod_{l=0}^{l_e-1}(2l_e - 2l + 1)m\) MPI\_Allreduce operations, where \(\delta(k) \in \mathbb{N}\) denotes the maximum number of cycles performed by Arnoldi at any level. In parGeMSLR, the default cycle length is \(m = 2k\) iterations. Finally, after the set up phase, one full application of the preconditioner implemented in the parGeMSLR library requires \(2(l_e - l) + 1\) MPI\_Allreduce operations.

The analysis presented in this section demonstrates that the communication overhead associated with the construction of the GeMSLR preconditioner is directly proportional to an
increase in the value of \( l_v \). On the other hand, increasing the value of \( l_v \) can reduce the computational complexity associated with setting up the GeMSLR preconditioner in lower levels. Nonetheless, the value of \( l_v \) can not be too large, especially when the value of \( p \) is large, since the size of the vertex separator reduces dramatically between successive levels (as is demonstrated in Figure 1).

4.3. Applying \( C_{l_v-1}^{-1} \)

Due to partitioning with a multilevel vertex separator, the matrix \( C_{l_v-1} \) forms a separate partition which is replicated among all MPI processes. Therefore, the simplest approach to apply \( C_{l_v-1}^{-1} \) is to do so approximately, through computing an ILUT redundantly in each MPI process. However, for large problems, this approach can quickly become impractical, even if a shared-memory variant of ILUT is considered. On the other hand, applying a distributed-memory approach that requires communication among the MPI processes can lead to high communication overhead since the application of \( C_{l_v-1}^{-1} \) is the most common operation during the setup phase of the preconditioner.

parGeMSLR includes several options to apply an approximation of \( C_{l_v-1}^{-1} \). The default option considered throughout our experiments is to apply \( C_{l_v-1}^{-1} \) approximately through a block-Jacobi approach where \( C_{l_v-1} \) is first permuted by reverse RCM and then replaced by its on-diagonal block submatrices while the rest of the entries are discarded. Generally speaking, dropping these entries of \( C_{l_v-1}^{-1} \) has minor effects since \( C_{l_v-1}^{-1} \) is already close to being block-diagonal for modest values of \( l_v \) (e.g., three or four) as was already demonstrated in Figure 1. By default, the number of retained on-diagonal blocks of matrix \( C_{l_v-1} \) is set equal to \( p \). The approximate application of \( C_{l_v-1}^{-1} \) is then trivially parallel among the MPI processes, and each one of the retained on-diagonal blocks is applied through ILUT.

5. Numerical Experiments

In this section we demonstrate the parallel performance of parGeMSLR. We run our experiments on the QUARTZ cluster of Lawrence Livermore National Laboratory. Each node of QUARTZ has 128 GB memory and consists of 2 Intel Xeon E5-2695 CPUs with 36 cores in total. We use MVAPICH2 2.2.3, to compile parGeMSLR is compiled with MVAPICH2 2.2.3, following rank-to-core binding. By default, all of the experiments presented below are executed in double-precision. On top of distributed-memory parallelism, parGeMSLR can take advantage of shared memory parallelism using either OpenMP or CUDA. The current version of parGeMSLR uses LAPACK for sequential matrix decompositions and ParMETIS for distributed graph partitioning [59]. A detailed documentation of parGeMSLR can be found in the DOCS directory of https://github.com/Hitenze/pargemslr. This documentation provides detailed information on how to compile and run parGeMSLR, and includes a detailed description of all command-line parameters as well as visualization of the source code hierarchy. Several test drivers, and a sample input file, are also included.

Throughout the rest of this section, we choose Flexible GMRES (FGMRES) with a fixed restart size of fifty as the outer iterative solver. The motivation for using FGMRES instead of GMRES is that the application of the preconditioner is subject to variations due to the application of the inner solver in step 9 of Algorithm 2. The stopping tolerance for the relative residual norm in FGMRES is set equal to \( 1.0e^{-6} \). Unless mentioned otherwise, the solution of the linear system \( Ax = b \) will be equal to the vector of all ones with an initial approximation equal to zero. The low-rank correction term at each level consists of approximate Schur vectors such that the corresponding approximate eigenvalues are accurate to two digits of accuracy, and the restart cycle of thick-restart Arnoldi is equal to 2k.

Our distributed-memory experiments focus on the parallel efficiency of parGeMSLR both when the problem size remains fixed and \( n_p \) increases (strong scaling) and the problem size increases at the same rate with \( n_p \). In the case of weak scaling,
the parallel efficiency is equal to \( \frac{T_1}{T_{np}} \), where \( T_1 \) and \( T_{np} \) denote the wall-clock time achieved by the sequential and distributed-memory version (using \( np \) MPI processes) of parGeMSLR, respectively. Likewise, in the case of strong scaling, the parallel efficiency is equal to \( \frac{T_1}{T_{np}^k} \). In addition, we also compare parGeMSLR against: \( a) \) the BoomerAMG parallel implementation of the algebraic multigrid method in hypre, and \( b) \) the two-level SchurILU approach in [22]. The latter preconditioner uses partial ILU to form an approximation of the Schur complement matrix. The preconditioning step is then performed by applying GMRES with block-Jacobi preconditioning to solve the linear system associated with the sparsified Schur complement. The block-Jacobi preconditioner is applied through one step of ILUT, and our implementation of SchurILU is based on the parallel ILU(T) in hypre.

Throughout the rest of this section, we adopt the following notation:

- \( np \in \mathbb{N} \): total number of MPI processes.
- \( \text{fill} \in \mathbb{R} \): ratio between the number of non-zero entries of the preconditioner and that of matrix \( A \).
- \( \text{p-t} \in \mathbb{R} \): preconditioner setup time. This includes the time required to compute the ILUT factorizations and low-rank correction terms in parGeMSLR.
- \( \text{i-t} \in \mathbb{R} \): iteration time of FGMRES.
- \( \text{its} \in \mathbb{N} \): total number of FGMRES iterations.
- \( k \in \mathbb{N} \): number of low-rank correction terms at each level.
- \( F \): flag signaling that FGMRES failed to converge within 1000 iterations.

5.1. A Model Problem

This section considers a Finite Difference discretization of the model problem

\[
-\Delta u - cu = f \quad \text{in} \quad \Omega,
\]

\[
u = 0 \quad \text{on} \quad \partial \Omega.
\]

(10)

We consider a 7-pt stencil and set \( \Omega = (0, 1)^3 \).

5.1.1. Weak scaling

Our first set of experiments studies the weak scaling efficiency of parGeMSLR. Since varying the values of \( l_p \) and \( k \) lead to different convergence rates, we first consider the case where the number of FGMRES iterations is set equal to thirty, regardless of whether convergence was achieved or not. The problem size on each MPI process is fixed to 50, while the number of subdomains at each level is set equal to 8 \( \times \) \( np \). Moreover, the number of levels is varied as \( l_v \in \{2, 3\} \) while the rank of the low-rank correction terms is varied as \( k \in \{0, 100, 200\} \).

Figure 4 plots the weak scaling efficiency of parGeMSLR on up to \( np = 1,024 \) MPI processes. The achieved efficiency is similar for both options of \( l_v \) with a slightly higher efficiency observed for the case \( l_v = 3 \). As expected, the highest efficiency achieved during the preconditioner setup phase was for the case \( k = 0 \), since there is no communication overhead stemming from the low-rank correction terms. Nonetheless, even in this case there is some loss in efficiency due to load imbalanc ing introduced by the ILUT factorizations at different levels. Regardless of the value of \( k \), the efficiency of parGeMSLR drops the most when the number of MPI processes is small, regardless of the value of \( l_v \). This reduction is owed to the relatively large increase on the size of the local Schur complement versus when a larger number of MPI processes is utilized. Note though, although not reported in our experiments, that the weak scaling efficiency is typically much higher when each MPI process handles exactly one subdomain. Finally, the efficiency of the reordering phase is rather limited, since the wall-clock time requires to partition the graph associated with the matrix \(|A| + |A^T|\) and permute the distributed matrix \( A \) increases as the problem size grows.

Figure 5 plots the weak scalability of parGeMSLR and two-level SchurILU, where this time we allow enough iterations in FGMRES until convergence. As previously, we use eight sub-

![Figure 4: Weak scaling of parGeMSLR for the Poisson problem when the number of iterations performed by FGMRES is fixed to thirty, and the number of levels is set to \( l_v = 2 \) and \( l_v = 3 \). The number of unknowns on each MPI process is 125, 000, for a maximum problem size \( n = 800 \times 400 \times 400 \).](image-url)
domains per MPI process, but this time we fix \( l_v = 3 \) and \( k = 10 \). In summary, \textit{parGeMSLR} is both faster and more scalable than SchurILU during the solve phase. Moreover, \textit{parGeMSLR} also converges much faster than SchurILU, and the number of total FGMRES iterations increases only marginally with the problem size. On the other hand, the weak scaling of the preconditioner setup phase of \textit{parGeMSLR} is impacted negatively as the problem size increases due to the need to perform more Arnoldi iterations to compute the low-rank correction terms.

### 5.1.2. Strong scaling

We now present strong scaling results obtained by solving \( \text{(10)} \) with \textit{parGeMSLR} on a regular mesh of fixed size as the numbers of MPI processes varies. More specifically, the size of the problem is fixed to \( n = 320^3 \) while the number of MPI processes varies up to \( n_p = 1,024 \). The values of \( l_v \) and \( k \) are varied as previously.

Figure 5 plots the strong scaling of \textit{parGeMSLR}. In contrast to the weak scaling case, setting \( l_v = 2 \) leads to higher efficiency during both the setup and application phases of the preconditioner. The reason for this behavior is twofold. First, increasing the value of \( l_v \) generally deteriorates the effectiveness of the preconditioner unless \( k \) is large and the threshold used in the local ILUT factorizations is small. Second, decreasing the value of \( l_v \) enhances strong scalability since it leads to smaller communication overheads (i.e., recall the discussion in Section 4). As a general remark, we note that the setup phase of \textit{parGeMSLR} generally becomes more expensive in terms of floating-point arithmetic operations as \( l_v \) decreases, thus although scalability deteriorates as \( l_v \) increases, the actual wall-clock time might actually decrease if the number of MPI processes used is small.

### 5.2. General Problems

This section discusses the performance of \textit{parGeMSLR} on a variety of problems in engineering.

#### 5.2.1. Unstructured Poisson problem on a crooked pipe

We consider the numerical solution of \( \text{(10)} \) where \( f = 1 \) and \( c = 0 \) on a 3D crooked pipe mesh. The problem is discretized by second-order Finite Elements using the MFEM library \cite{MFEM} with local uniform and parallel mesh refinement. The initial approximation of the solution is set equal to zero. We visualize the (inhomogeneous) mesh using the package GLVis \cite{GLVis} in Figure 7. Our experiments consider different refinement levels to generate problems of different sizes. Moreover, the maximum number of inner iterations in step 9 of Algorithm 2 is varied between three and five. We compare \textit{parGeMSLR} against BoomerAMG with Hybrid Modified Independent Set (HMIS) coarsening, where we consider both Gauss-Seidel and \( l_1 \) Jacobi smoother \cite{BoomerAMG}, and report the corresponding results in Table 1. \textit{parGeMSLR} is able to outperform Schur ILU, especially for larger problems. Moreover, the iteration time of \textit{parGeMSLR}}
5.2.2. Linear elasticity equation

In the section we consider the solution of the following linear elasticity equation:

\[ \mu \Delta u + (\lambda + \mu) \nabla (\nabla \cdot u) = f \quad \text{in} \quad \Omega, \]

where \( \Omega \) is a 3D cantilever beam as shown in Figure 8. The left end of the beam is fixed, while a constant force (represented by \( f \)) pulls down the beam from the right end. Herein, \( u \) is the displacement, while \( \lambda \) and \( \mu \) are the material’s Lamé constants.

Figure 8: Linear elasticity problem on a 3D beam.

Table 1: Solving (10) on a crooked pipe mesh.

| prec      | size     | \( n_p \) | k fill | p-t | i-t | its |
|-----------|----------|-----------|--------|-----|-----|-----|
| Boomer AMG GS | 126,805 16 | -1.71 0.17 | 0.69 106 |
|           | 966,609 32 | -1.79 0.79 | 5.7 198 |
| Boomer AMG Jacobi | 7,544,257 64 | -1.81 3.36 | 45.12 751 |
| Schur ILU | 126,805 16 | -1.53 0.22 | 0.51 65 |
|           | 966,609 32 | -1.79 0.8 | 10.95 431 |
|           | 7,544,257 64 | -1.81 3.39 | 72.1 568 |
| par GeMSLR | 126,805 16 | 10 1.05 0.54 | 0.46 25 |
|           | 966,609 32 | 10 1.18 3.59 | 4.70 53 |
|           | 7,544,257 64 | 10 1.32 11.76 | 48.35 128 |

5.2.3. Helmholtz equation

In this section we consider the complex version of parGeMSLR and apply it to solve the Helmholtz problem

\[-(\Delta + \omega^2)u = f \quad \text{in} \quad \Omega = [0, 1]^3, \]

where we use the Perfectly Matched Layer (PML) boundary condition and set the number of points per wavelength equal to eight. We used random initial guesses.

Our first set of experiments focuses on the performance of parGeMSLR where the number of low-rank terms is varied as \( k = \{10, 20, \ldots, 100\} \), and the number of levels is set equal to \( l_{\text{ev}} = 3 \). The size of the Helmholtz problem is set equal to \( n = 50^3 \). The maximum fill-in attributed to the low-rank correction term was roughly equal to three. Figure 2 plots the parallel wall-clock time as a function of the number of low-rank terms \( k \) while the number of MPI processes is fixed equal to
sixteen. Overall, larger values of $k$ lead to lower total and iteration times up to the point where the time increase associated with constructing the parGeMSLR preconditioner outweighs the gains from improving the convergence rate during the iterative solution by FGMRES. Next, we consider the same problem but achieved during the solution phase if GPUs are enabled. We set the number of levels equal to $l_v = 2$ and $l_e = 3$, and vary the low-rank correction terms as $k \in \{0, 100, 200, 300, 400, 500\}$.

At each level, we apply a 4-way partition and assign each partition to a separate MPI process binded to a V100 NVIDIA GPU. Figure 10 plots the speedups achieved by the hybrid CPU+GPU version of parGeMSLR during its solve phase. As expected, the peak speedup is obtained for the case $k = 500$, since the cost to apply the low-rank correction term increases linearly with the value of $k$.

Figure 10: Speedup of the solution phase of parGeMSLR if GPU acceleration is enabled when $l_v = \{2, 3\}$, and $k \in \{0, 100, 200, 300, 400, 500\}$. The problem size is equal to $n = 128^3$.

6. Concluding remarks and future work

In this paper we presented parGeMSLR, a C++ parallel software library for the iterative solution of general sparse systems distributed among several processor groups communicating via MPI environments [24]. parGeMSLR is based on the GeMSLR preconditioner and can be applied to both real and complex systems of linear algebraic equations. The performance of parGeMSLR on distributed-memory computing environments was demonstrated on both model and real-world problems, verifying the efficiency of the library as a general-purpose solver.

As future work we plan to replace standard Arnoldi by either its block variant or randomized subspace iteration. This should improve performance by reducing latency during the preconditioner setup phase. Moreover, the cost of the setup phase can be amortized over the solution of linear systems with multiple right-hand sides, e.g., see [73, 74, 75, 76], and we plan to apply parGeMSLR to this type of problems. In this context, we also plan to apply parGeMSLR to the solution of sparse linear systems appearing in eigenvalue solvers based on rational filtering [77, 78], and domain decomposition [79, 80].

References

1. Y. Saad, Iterative Methods for Sparse Linear Systems, Other Titles in Applied Mathematics, Society for Industrial and Applied Mathematics, 2003. doi:10.1137/1.9780898718003

2. H. A. Van der Vorst, Iterative Krylov methods for large linear systems, no. 13, Cambridge University Press, 2003.
