Numerical Algorithms (2023) 93:731–764
https://doi.org/10.1007/s11075-022-01437-1

ORIGINAL PAPER


# Adaptively restarted block Krylov subspace methods with low-synchronization skeletons

**Kathryn Lund**[1]

Received: 19 August 2022 / Accepted: 8 October 2022 / Published online: 28 December 2022


## Abstract

With the recent realization of exascale performance by Oak Ridge National Laboratory's Frontier supercomputer, reducing communication in kernels like QR factorization has become even more imperative. Low-synchronization Gram-Schmidt methods, first introduced in Świrydowicz et al. (*Numer. Lin. Alg. Appl.* 28(2):e2343, 2020), have been shown to improve the scalability of the Arnoldi method in high-performance distributed computing. Block versions of low-synchronization Gram-Schmidt show further potential for speeding up algorithms, as column-batching allows for maximizing cache usage with matrix-matrix operations. In this work, low-synchronization block Gram-Schmidt variants from Carson et al. (*Linear Algebra Appl.* 638:150–195, 2022) are transformed into block Arnoldi variants for use in block full orthogonalization methods (BFOM) and block generalized minimal residual methods (BGMRES). An adaptive restarting heuristic is developed to handle instabilities that arise with the increasing condition number of the Krylov basis. The performance, accuracy, and stability of these methods are assessed via a flexible benchmarking tool written in MATLAB. The modularity of the tool additionally permits generalized block inner products, like the global inner product.

**Keywords** Gram-Schmidt · Krylov subspace methods · Arnoldi method · Block methods · Stability · Loss of orthogonality · Low-synchronization methods · High-performance computing · Communication-avoiding methods

**Mathematics Subject Classification (2010)** 65F10 · 65F25 · 65F50 · 15-04

✉ Kathryn Lund
lund@mpi-magdeburg.mpg.de

[1] Computational Methods in Systems and Control Theory, Max Planck Institute for Dynamics of Complex Technical Systems, Sandtorstr. 1, Magdeburg, 39106, Germany

## 1 Introduction and motivation

Oak Ridge National Laboratory reported in May 2022 that its Frontier supercomputer is the first machine to have achieved true exascale performance.[1] That is, for the first time ever, a supercomputer performed more than 1 exaflop (i.e., $10^{18}$ double-precision floating-point operations) in a single second. This astounding development is clear motivation for our work. Exascale computing is no longer a next-generation dream; it is reality, and the need for highly parallelized algorithms that take full advantage of exaflop computational potential while reducing global communication between nodes is urgent.

To this end we build on the low-synchronization ("low-sync") Gram-Schmidt methods of Barlow [1], Świrydowicz et al. [2], Yamazaki et al. [3], Thomas et al. [4], and Bielich et al. [5], as well as our own earlier work with block versions of these methods [6, 7]. Gram-Schmidt methods are an essential backbone in orthogonalization routines like QR factorization and in iterative methods like Krylov subspace methods for linear systems, matrix functions, and matrix equations [8–10]. Block Krylov subspace methods in particular make better use of L3 cache via matrix-matrix operations and feature often in communication-avoiding Krylov subspaces, such as $s$-step [11, 12], enlarged methods [13], and randomized methods [14].

As in most realms of life, there is no such thing as a free lunch here. While low-sync variations have the potential to speed up highly parallelized implementations of Gram-Schmidt [3], they introduce new floating-point errors and thus potential instability, due to the reformulation of inner products and normalizations. Instability surfaces in the loss of orthogonality between basis vectors and can lead to breakdowns or wildly inaccurate approximations in downstream applications [15, 16]. Stability bounds for some low-sync variants have been established, but it often takes much longer to carry out a rigorous stability analysis than to derive and deploy new methods [1, 4, 6, 7]. It can also happen that a backward error bound is established and later challenged by an obscure edge case [17, 18]. With this tension in mind, we have not only extended low-sync variants of block Gram-Schmidt to block Arnoldi but also developed a benchmarking tool for the community to explore the efficiency, stability, and accuracy of these new algorithms, in a similar vein as the `BlockStab`[2] comparison tool developed in tandem with a recent block Gram-Schmidt survey [7]. We refer to this new tool as LowSyncBlockArnoldi[3] and encourage the reader to explore the tool in parallel with the text.

Established in this earlier work is the fact that block variants of low-sync Gram-Schmidt are less stable than their column-wise counterparts. However, when these skeletons are transferred to block Arnoldi and used to solve linear systems, we gain the option to restart the process. Restarting can be effective at mitigating stability issues in communication-avoiding algorithms [19, 20]. As long as each node redundantly computes residual or error estimates and checks the stability via

[1] http://www.top500.org/news/ornls-frontier-first-to-break-the-exaflop-ceiling/. Accessed 8 August 2022.
[2] https://github.com/katlund/BlockStab
[3] https://gitlab.mpi-magdeburg.mpg.de/lund/low-sync-block-arnoldi

local quantities, restarting does not introduce additional synchronization points. Furthermore, adaptive restarting allows for robustness, as we can use basic look-ahead heuristics to foresee a breakdown and salvage progress without giving up completely at the first sign of trouble.

Given the modularity of our framework, we are also able to treat generalized block inner products, as described in [21, 22]. We focus in particular on the classical and global inner products.

The paper is organized as follows. In Section 2 we summarize terms, definitions, and concepts from high-performance (HPC) computing, generalized block inner products, block Gram-Schmidt algorithms, and block Krylov subspace methods with static restarting. We present new low-synchronization block Arnoldi skeletons in Section 3, and derive an adaptive restarting heuristic in Section 4. Section 5 features a more in-depth discussion of the `LowSyncBlockArnoldi` benchmarking tool as well as examples demonstrating how to compare different block Arnoldi variants. We summarize our findings in Section 6.

## 2 Background

This work is a combination of the generalized inner product framework of Frommer, Lund, and Szyld [21, 22] and the skeleton-muscle framework for block Gram-Schmidt (BGS) by Carson, Lund, Rozložník, and Thomas [6, 7]. Throughout the text, we focus on solving a linear system with multiple right-hand sides

$$AX = B, \tag{1}$$

where $A \in \mathbb{C}^{n\times n}$ is large and sparse (i.e., with $\mathcal{O}(n)$ nonzero entries) and $B \in \mathbb{C}^{n\times s}$ is a tall-skinny (i.e., $s \ll n$) matrix.

We employ standard numerical linear algebra notation throughout. In particular, $A^*$ denotes the Hermitian transpose of $A$, $\|\cdot\|$ refers to the Euclidean 2-norm, unless otherwise specified, and $\widehat{e}_k$ denotes the $k$th standard unit vector with the $k$th entry equal to 1 and all others 0.

In the following subsections, we define key concepts in HPC, block Gram-Schmidt methods, and block Krylov subspace methods.

### 2.1 Communication in high-performance computing

As floating-point operations have become faster and less energy-intensive, *communication*—the memory operations between levels of cache on a node or between parallelized processors on a network—has become a bottleneck in distributed computing. How expensive a memory operation is depends on the physical aspects of a specific system, specifically the *latency*, or the amount of time needed to pack and transmit a message, and the *bandwidth*, or how much information can be transmitted at a time. To improve algorithm performance in bandwidth-limited algorithms like Krylov subspace methods, it is therefore advantageous to increase the *computational intensity*, or the ratio between floating-point and memory operations [23]. We pay particular attention to *synchronization points* ("sync points"), i.e., the steps in an

algorithm that initiate a broadcast or reduce pattern to synchronize a quantity on all processors. Reducing calls to kernels with sync points is a straightforward way to improve computational intensity [24].

Sync points in Krylov subspace methods arise primarily in the orthonormalization procedure, such as Arnoldi or Lanczos, both of which are reformulations of the Gram-Schmidt method, a standard method for orthonormalizing a basis one (block) vector at a time. For large $n$, vectors are typically partitioned row-wise and distributed among processors, meaning that any time an operation like an inner product or normalization is performed—which is at least once per (block) vector in Gram-Schmidt—a sync point is inevitable.

Other possibly communication-intensive kernels include applications of the operator $A$[4] and applications of $\boldsymbol{\mathcal{V}}_m$, an $n \times ms$ Krylov basis matrix. We count each operation separately from sync points (block inner products and vector norms) in `LowSyncBlockArnoldi`; see Section 5.

### 2.2 Generalized block inner products

A *block vector* is a tall-skinny matrix $\boldsymbol{X} \in \mathbb{C}^{n\times s}$, and a *block matrix* is a matrix of $s \times s$ matrices, e.g.,

$$\mathcal{H} = \begin{bmatrix} H_{1,1} & H_{1,2} & \cdots & H_{1,p} \\ H_{2,1} & H_{2,2} & \cdots & H_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ H_{q,1} & H_{q,2} & \cdots & H_{q,p} \end{bmatrix} \in \mathbb{C}^{qs\times ps}.$$

We use a mixture of MATLAB- and block-indexing notation to handle block objects. In particular, we write $\boldsymbol{\mathcal{V}}_k$ to denote the first $k$ block vectors of the block-partitioned matrix $\boldsymbol{\mathcal{V}} = \begin{bmatrix} \boldsymbol{V}_1 & \boldsymbol{V}_2 & \cdots & \boldsymbol{V}_m \end{bmatrix}$ instead of $\boldsymbol{\mathcal{V}}_{:,1:\,ks}$ (i.e., the first $ks$ columns). In a similar vein, $s \times s$ block entries of $\mathcal{H}$ are denoted as $H_{j,k}$ instead of as $H_{(j-1)s+1:js,(k-1)s+1:ks}$. We denote block generalizations of the standard unit vectors $\widehat{e}_k$ as $\widehat{\boldsymbol{E}}_k := \widehat{e}_k \otimes I_s$, where $\otimes$ is the Kronecker product and $I_s$ the identity matrix of size $s$.

Blocking is a *batching* technique that can reduce the number of calls to the operator $A$ applied to individual column vectors, maximize computational intensity by filling up the local cache with BLAS3 operations, and reduce the total number of sync points by performing inner products and normalization en masse [25, 26]. In the context of Krylov subspaces, blocking can also lead to enriched subspaces by sharing information across column vectors instead of treating each right-hand side as an isolated problem. How much information is shared across columns depends on the choice of block inner product.

Let $\mathbb{S}$ be a *-subalgebra of $\mathbb{C}^{s\times s}$ with identity; i.e., $I \in \mathbb{S}$ and when $S, T \in \mathbb{S}$, $\alpha \in \mathbb{C}$, then $\alpha S + T$, $ST$, $S^* \in \mathbb{S}$.

[4]The term `matvec` is often used to refer to the multiplication of $A$ with a vector. Because we will be focusing on block vectors, we refrain from this term to avoid confusion.

**Definition 1** A mapping $\langle\!\langle \cdot, \cdot \rangle\!\rangle$ from $\mathbb{C}^{n\times s} \times \mathbb{C}^{n\times s}$ to $\mathbb{S}$ is called a *block inner product onto* $\mathbb{S}$ if it satisfies the following conditions for all $\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z} \in \mathbb{C}^{n\times s}$ and $C \in \mathbb{S}$:

(i) $\mathbb{S}$-*linearity*: $\langle\!\langle \boldsymbol{X} + \boldsymbol{Y}, \boldsymbol{Z}C \rangle\!\rangle_{\mathbb{S}} = \langle\!\langle \boldsymbol{X}, \boldsymbol{Z} \rangle\!\rangle_{\mathbb{S}} C + \langle\!\langle \boldsymbol{Y}, \boldsymbol{Z} \rangle\!\rangle_{\mathbb{S}} C$;
(ii) *symmetry*: $\langle\!\langle \boldsymbol{X}, \boldsymbol{Y} \rangle\!\rangle_{\mathbb{S}} = \langle\!\langle \boldsymbol{Y}, \boldsymbol{X} \rangle\!\rangle_{\mathbb{S}}^{*}$;
(iii) *definiteness*: $\langle\!\langle \boldsymbol{X}, \boldsymbol{X} \rangle\!\rangle_{\mathbb{S}}$ is positive definite if $\boldsymbol{X}$ has full rank, and $\langle\!\langle \boldsymbol{X}, \boldsymbol{X} \rangle\!\rangle_{\mathbb{S}} = 0$ if and only if $\boldsymbol{X} = 0$.

**Definition 2** A mapping $N$ which maps all $\boldsymbol{X} \in \mathbb{C}^{n\times s}$ with full rank on a matrix $N(\boldsymbol{X}) \in \mathbb{S}$ is called a *scaling quotient* if for all such $\boldsymbol{X}$, there exists $\boldsymbol{Y} \in \mathbb{C}^{n\times s}$ such that $\boldsymbol{X} = \boldsymbol{Y}N(\boldsymbol{X})$ and $\langle\!\langle \boldsymbol{Y}, \boldsymbol{Y} \rangle\!\rangle_{\mathbb{S}} = I_s$.

The scaling quotient is closely related to the intraorthogonalization routine discussed in Section 2.3. Block notions of orthogonality and normalization arise organically from Definitions 1 and 2.

**Definition 3** Let $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{C}^{n\times s}$ and $\{\boldsymbol{X}_j\}_{j=1}^{m} \subset \mathbb{C}^{n\times s}$.

(i) $\boldsymbol{X}, \boldsymbol{Y}$ are *block orthogonal*, if $\langle\!\langle \boldsymbol{X}, \boldsymbol{Y} \rangle\!\rangle_{\mathbb{S}} = 0_s$.
(ii) $\boldsymbol{X}$ is *block normalized* if $N(\boldsymbol{X}) = I_s$.
(iii) $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_m$ are *block orthonormal* if $\langle\!\langle \boldsymbol{X}_i, \boldsymbol{X}_j \rangle\!\rangle_{\mathbb{S}} = \delta_{ij} I_s$.

A set of vectors $\{\boldsymbol{X}_j\}_{j=1}^{m} \subset \mathbb{C}^{n\times s}$ *block spans* a space $\mathscr{K} \subseteq \mathbb{C}^{n\times s}$, and we write $\mathscr{K} = \operatorname{span}^{\mathbb{S}}\{\boldsymbol{X}_j\}_{j=1}^{m}$ if

$$\mathscr{K} = \left\{ \sum_{j=1}^{m} \boldsymbol{X}_j \Gamma_j : \Gamma_j \in \mathbb{S} \text{ for } j = 1, \ldots, m \right\}.$$

The set $\{\boldsymbol{X}_j\}_{j=1}^{m}$ constitutes a *block orthonormal basis* for $\mathscr{K} = \operatorname{span}^{\mathbb{S}}\{\boldsymbol{X}_j\}_{j=1}^{m}$ if it is block orthonormal.

In this work, we consider only the *classical* and *global* block paradigms, described in Table 1. These paradigms represent the two extremes of information-sharing, with the classical approach maximizing information shared among columns and the global approach minimizing it; see, e.g., [22, Theorem 3.3]. Moreover, the global paradigm leads to a lower complexity per iteration in Krylov subspace methods, because what are matrix-matrix products in the classical paradigm get reduced to scaling operations in the global one. Many other paradigms are also possible; see, e.g., [27, 28].

**Table 1** Choices of $\mathbb{S}$,$\langle\langle \cdot, \cdot \rangle\rangle_{\mathbb{S}}$, and $N$ in the classical and global block paradigms

| | $\mathbb{S}$ | $\langle\langle \boldsymbol{X}, \boldsymbol{Y} \rangle\rangle_{\mathbb{S}}$ | $N(\boldsymbol{X})$ |
|---|---|---|---|
| classical (`cl`) | $\mathbb{C}^{s\times s}$ | $\boldsymbol{X}^*\boldsymbol{Y}$ | $R$, where $\boldsymbol{X} = \boldsymbol{Q}R$, and $\boldsymbol{Q} \in \mathbb{C}^{n\times s}$, $\boldsymbol{Q}^*\boldsymbol{Q} = I_s$ |
| global (`gl`) | $\mathbb{C}I_s$ | $\frac{1}{s} trace(\boldsymbol{X}^*\boldsymbol{Y}) I_s$ | $\frac{1}{\sqrt{s}} \|\boldsymbol{X}\|_{\mathrm{F}} I_s$ |

### 2.3 Block Gram-Schmidt

Block Gram-Schmidt (BGS) is a routine for orthonormalizing a set of block vectors $\{\boldsymbol{X}_j\}_{j=1}^m \subset \mathbb{C}^{n\times s}$. Writing

$$\boldsymbol{\mathcal{X}} := \begin{bmatrix}\boldsymbol{X}_1 & \boldsymbol{X}_2 & \cdots & \boldsymbol{X}_m\end{bmatrix} \in \mathbb{C}^{n\times ms},$$

we define a BGS method as one that returns a block orthonormal $\boldsymbol{\mathcal{Q}} \in \mathbb{C}^{n\times ms}$ and a block upper triangular $\mathcal{R} \in \mathbb{C}^{ms\times ms}$ such that $\mathcal{X} = \mathcal{Q}\mathcal{R}$. Important measures in the analysis of BGS methods are the condition number of $\boldsymbol{\mathcal{X}}$,

$$\kappa(\boldsymbol{\mathcal{X}}) := \frac{\sigma_{\max}(\boldsymbol{\mathcal{X}})}{\sigma_{\min}(\boldsymbol{\mathcal{X}})}, \tag{2}$$

i.e., the ratio between the largest and smallest singular values of $\boldsymbol{\mathcal{X}}$, and the *loss of orthogonality (LOO)*,

$$\left\| I - \langle\!\langle \boldsymbol{\mathcal{Q}}, \boldsymbol{\mathcal{Q}} \rangle\!\rangle_{\mathbb{S}} \right\|, \tag{3}$$

where $\langle\!\langle \cdot, \cdot \rangle\!\rangle_{\mathbb{S}}$ is a generalized inner product as described in Section 2.4.

When we discuss the stability of BGS methods, we refer to bounds on the loss of orthogonality in terms of machine precision, $\varepsilon$. We assume IEEE double precision here, so $\varepsilon = \mathcal{O}\left(10^{-16}\right)$.

For categorizing BGS variants, we recycle the skeleton-muscle notation from [7, 12], where *skeleton* refers to the *inter*orthogonalization routine between block vectors, and the *muscle* refers to the *intra*orthogonalization routine between the columns of a single block vector. As a prototype, consider the Block Modified Gram-Schmidt (`BMGS`) skeleton, given by Algorithm 1. Here, `IntraOrtho` denotes a generic muscle that takes $\boldsymbol{X} \in \mathbb{C}^{n\times s}$ and returns $\boldsymbol{Q} \in \mathbb{C}^{n\times s}$ and $R \in \mathbb{C}^{s\times s}$ such that $\langle\!\langle \boldsymbol{Q}, \boldsymbol{Q} \rangle\!\rangle_{\mathbb{S}} = I_s$ and $\boldsymbol{X} = \boldsymbol{Q}R$. For the classical paradigm, this could be any implementation of a QR factorization: a column-wise Gram-Schmidt routine, Householder QR (`HouseQR`), Cholesky QR (`CholQR`), etc. As for the global paradigm, there is only one possible muscle, given by the global scaling quotient, which effectively reduces to normalizing block vectors with a scaled Frobenius norm. Consequently, intraorthogonalization does not actually occur in the global paradigm, as the columns of block vectors are not orthogonalized with respect to one another at all.

We regard a single call to either $\langle\!\langle \cdot, \cdot \rangle\!\rangle_{\mathbb{S}}$ or `IntraOrtho` as one sync point, which is only possible in practice if single-reduce algorithms like `CholQR` [29] or `TSQR`/`AllReduceQR` [30, 31] are employed for `IntraOrtho`.

### 2.4 Block Krylov subspace methods

The $m$th *block Krylov subspace for* $A$ *and* $\boldsymbol{B}$ (*with respect to* $\mathbb{S}$) is defined as

$$\mathscr{K}_m^{\mathbb{S}}(A, \boldsymbol{B}) := \mathbb{S}\{\boldsymbol{B}, A\boldsymbol{B}, \ldots, A^{m-1}\boldsymbol{B}\}. \tag{4}$$

Block Arnoldi is often used to compute a basis for $\mathscr{K}_m^{\mathbb{S}}(A, \boldsymbol{B})$, and it is typically implemented with `BMGS` as the skeleton; see Algorithm 2. `BMGS-Arnoldi` accrues a high number of sync points due to the inner `for`-loop, where an increasing number of inner products is performed per block column.

$[\boldsymbol{Q}_1, R_{11}] = \texttt{IntraOrtho}(\boldsymbol{X}_1)$
**for** $k = 1, \ldots, p-1$ **do**
  $\boldsymbol{W} = \boldsymbol{X}_{k+1}$
  **for** $j = 1, \ldots, k$ **do**
    $R_{j,k+1} = \langle\!\langle \boldsymbol{Q}_j, \boldsymbol{W} \rangle\!\rangle_{\mathbb{S}}$
    $\boldsymbol{W} = \boldsymbol{W} - \boldsymbol{Q}_j R_{j,k+1}$
  **end for**
  $[\boldsymbol{Q}_{k+1}, R_{k+1,k+1}] = \texttt{IntraOrtho}(\boldsymbol{W})$
**end for**
**return** $\mathcal{Q} = [\boldsymbol{Q}_1, \ldots, \boldsymbol{Q}_p], \mathcal{R} = (R_{jk})$

**Algorithm 1** $[\mathcal{Q}, \mathcal{R}] = \texttt{BMGS}(\boldsymbol{\mathcal{X}})$.

Performing $m$ steps of a block Arnoldi routine returns the *block Arnoldi relation*

$$A\boldsymbol{\mathcal{V}}_m = \boldsymbol{\mathcal{V}}_m \mathcal{H}_m + \boldsymbol{V}_{m+1} H_{m+1,m}, \tag{5}$$

where $\boldsymbol{\mathcal{V}}_m$ $\mathbb{S}$-spans $\mathscr{K}_m^{\mathbb{S}}(A, \boldsymbol{B})$ and $\mathcal{H}_m$ denotes the $ms \times ms$ principal submatrix of $\mathcal{H}_{m+1,m}$.

### 2.4.1 Block full orthogonalization methods with low-rank modifications

We define

$$\boldsymbol{X}_m := \boldsymbol{\mathcal{V}}_m \big(\mathcal{H}_m + \mathcal{M}\big)^{-1} \widehat{\boldsymbol{E}}_1 B, \tag{6}$$

where $\widehat{\boldsymbol{E}}_1 = \widehat{\boldsymbol{e}}_1 \otimes I_s$ is a standard block unit vector, as the *(modified) block full orthogonalization method (BFOM)* for approximating (1). When $\mathcal{M} = 0$, we recover BFOM, which minimizes the error in the $A$-weighted $\mathbb{S}$-norm for $A$ hermitian positive definite [21]. There are infinitely many choices for $\mathcal{M}$, but perhaps only a few useful ones, some of which are discussed in [22]. We will concern ourselves here with just $\mathcal{M} = \mathcal{H}_m^{-*}\big(\widehat{\boldsymbol{E}}_m H_{m+1,m}^* H_{m+1,m}\big)\widehat{\boldsymbol{E}}_m^*$, which gives rise to a block generalized minimal residual method (BGMRES) [32–34]. As in [22], we implement BGMRES as a modified BFOM here, with an eye towards downstream applications like $f(A)\boldsymbol{B}$ where the BFOM form is explicitly needed. In practice, there may be computational savings with a less modular implementation; see, e.g., [35–37].

### 2.4.2 Static restarting and cospatial factors

Restarting is a well-established technique for reconciling a growing basis with memory limitations. Define the residual of (6) as

$$\boldsymbol{R}_m := \boldsymbol{B} - A\boldsymbol{X}_m. \tag{7}$$

The basic idea of restarts is to use $\boldsymbol{R}_m$ to build a new Krylov subspace, which we then use to approximate the error $\boldsymbol{E}_m := A^{-1}\boldsymbol{B} - \boldsymbol{X}_m$, which solves $A\boldsymbol{E} = \boldsymbol{R}_m$ in exact arithmetic. Building a new Krylov subspace from $\boldsymbol{R}_m$ directly is not a great idea, because it would require an extra computation with $A$. Furthermore, we need a cheap, accurate, and ideally locally computable way to approximate $\|\boldsymbol{R}_m\|$ from

1: $[\boldsymbol{V}_1, \beta] = \texttt{IntraOrtho}(B)$
2: **for** $k = 1, \ldots, m$ **do**
3: $\quad \boldsymbol{W} = A\boldsymbol{V}_k$
4: $\quad$ **for** $j = 1, \ldots, k$ **do**
5: $\quad\quad H_{j,k} = \langle\!\langle \boldsymbol{V}_j, \boldsymbol{W} \rangle\!\rangle_{\mathbb{S}}$
6: $\quad\quad \boldsymbol{W} = \boldsymbol{W} - \boldsymbol{V}_j H_{j,k}$
7: $\quad$ **end for**
8: $\quad [\boldsymbol{V}_{k+1}, H_{k+1,k}] = \texttt{IntraOrtho}\,(\boldsymbol{W})$
9: **end for**
10: **return** $\mathcal{V}_{m+1} = [\boldsymbol{V}_1, \ldots, \boldsymbol{V}_{m+1}]$, $\mathcal{H}_{m+1,m} = (H_{jk})$, $\beta$

**Algorithm 2** $[\boldsymbol{\mathcal{V}}_{m+1}, \mathcal{H}_{m+1,m}, B] = \texttt{BMGS-Arnoldi}(A, \boldsymbol{B}, m)$.

one cycle to the next in order to monitor convergence. In [22] a static restarting method for low-rank modified BFOM is introduced that satisfies these requirements. By "static," we mean the basis size $m$ is fixed from one restart cycle to the next, in contrast to adaptive or dynamic restart cycle lengths. We restate [22, Theorem 4.1], which enables an efficient residual approximation and restarting procedure.

**Theorem 2.1** *Suppose* $\mathcal{M} = \boldsymbol{M}\widehat{\boldsymbol{E}}_m^*$, *where* $\boldsymbol{M} \in \mathbb{C}^{ms\times s}$ *and* $\widehat{\boldsymbol{E}}_m = \widehat{\boldsymbol{e}}_m \otimes I_s$. *Define* $\boldsymbol{U}_m := \boldsymbol{\mathcal{V}}_{m+1}\begin{bmatrix}\boldsymbol{M}\\ -H_{m+1,m}\end{bmatrix}$ *and let* $\boldsymbol{\Xi}_m := (\mathcal{H}_m + \mathcal{M})^{-1}\widehat{\boldsymbol{E}}_1 B$ *be the block coefficient vector for the approximation* $\boldsymbol{X}_m = \boldsymbol{\mathcal{V}}_m \boldsymbol{\Xi}_m$ (6) *of the system* (1). *With* $\boldsymbol{R}_m$ *as in* (7) *it then holds that*

$$\boldsymbol{R}_m = \boldsymbol{U}_m\big(\widehat{\boldsymbol{E}}_m^*\boldsymbol{\Xi}_m\big). \tag{8}$$

We refer to the $s \times s$ matrix $\widehat{\boldsymbol{E}}_m^*\boldsymbol{\Xi}_m$ as a *cospatial factor*, and (8) as the *cospatial residual relation*. The term *cospatial* refers to the fact that the columns of $\boldsymbol{R}_m$ and those of $\boldsymbol{U}_m$ span the same space. Moreover, in exact arithmetic, it is not hard to see that

$$\|\boldsymbol{R}_m\|_{\mathrm{F}} = \left\|\begin{bmatrix}\boldsymbol{M}\\ -H_{m+1,m}\end{bmatrix}\big(\widehat{\boldsymbol{E}}_m^*\boldsymbol{\Xi}_m\big)\right\|_{\mathrm{F}}, \tag{9}$$

and the right-hand term can be computed locally (and possibly redundantly on each processor) for $m \ll n$.

If the approximate residual norm does not meet the desired tolerance, then we can compute the Arnoldi relation for $\mathscr{K}_m(A, \boldsymbol{U}_m)$ to obtain $\boldsymbol{\mathcal{V}}_{m+1}^{(2)}$, $\mathcal{H}_m^{(2)}$, $H_{m+1,m}^{(2)}$, and $B^{(2)}$, where the superscript here and later denotes association to the restarted Krylov subspace. We then approximate $\boldsymbol{E}_m$ as

$$\boldsymbol{D}_m := \boldsymbol{\mathcal{V}}_m^{(2)}(\mathcal{H}_m^{(2)} + \mathcal{M}^{(2)})^{-1}\widehat{\boldsymbol{E}}_1 B^{(2)}\big(\widehat{\boldsymbol{E}}_m^*\boldsymbol{\Xi}_m\big),$$

and update $\boldsymbol{X}_m$ as

$$\boldsymbol{X}_m^{(2)} := \boldsymbol{X}_m + \boldsymbol{D}_m.$$

The process is repeated, applying Theorem 2.1 iteratively, until the desired residual tolerance is reached.

*Remark 1* The analysis in [21, 22] is carried out in exact arithmetic. Therefore, when we replace Algorithm 2 with low-sync versions in Section 3, all the results summarized in this section still hold, because all block Gram-Schmidt variants generate the same QR factorization in exact arithmetic.

## 3 Low-synchronization variants of block Arnoldi

To distinguish between block Arnoldi variants, we default to the name of the underlying block Gram-Schmidt skeleton. We specify a *configuration* as `ip-skelo(musc)`: inner product, skeleton, and muscle, respectively. This naturally leads to bit of an "alphabet soup," for which we ask the reader's patience, as it is crucial to precisely define algorithmic configurations for benchmarking. Please refer often to Table 2, which summarizes acronyms for all the Gram-Schmidt skeletons we consider in this text. Note that the coefficient in front of the number of sync points per cycle is often used to describe low-sync methods; e.g., `BCGS-PIP` is a "one-sync" method, while `BMGS-SVL` is a "three-sync" method.

*Remark 2* The methods presented here are closely related to but not quite the same as the block methods used by Yamazaki et al. in [3], where `BMGS`, `BCGS-PIP`, and `BCGSI+LS` are employed as Gram-Schmidt skeletons in $s$-step Arnoldi (also known as communication-avoiding Arnoldi) [11, 12, 23], which is used to solve a linear system with a single right-hand side. Recall that we are solving (1), i.e., multiple right-hand sides simultaneously.

*Remark 3* In the pseudocode for each algorithm, intermediate quantities like $\boldsymbol{W}$ and $\boldsymbol{U}$ are defined explicitly each iteration for readability. In general, we purposefully avoid redefining quantities in a given iteration and instead only set an output (i.e., entries in $B$, $\boldsymbol{\mathcal{V}}_m$, or $\mathcal{H}_{m+1,m}$) once all computations pertaining to that value are complete. This approach simplifies mathematical analysis. Exceptions include Algorithms 1 and 2, where $\boldsymbol{W}$ is redefined inside the `for`-loop as projected components are subtracted away from it. In practice, it is preferable to save storage by overwriting block vectors of $\boldsymbol{\mathcal{V}}_m$ instead of allocating separate memory for $\boldsymbol{W}$ and $\boldsymbol{U}$, for which there anyway may not be space.

### 3.1 `BCGS-PIP` and `BCGS-PIO`

A simple idea for reducing the number of sync points in `BMGS` is to condense the `for`-loop in lines 4–7 of Algorithm 2 into a single inner product and subtraction,

$$\begin{aligned} \mathcal{H}_{1:k,k} &= \langle\!\langle \boldsymbol{\mathcal{V}}_k, \boldsymbol{W} \rangle\!\rangle_{\mathbb{S}} \\ \boldsymbol{W} &= \boldsymbol{W} - \boldsymbol{W}\mathcal{H}_{1:k,k} \end{aligned}$$

This exchange gives rise to what is commonly referred to as the block classical Gram-Schmidt (`BCGS`) method. It is, however, rather unstable, with a loss of orthogonality worse than $\mathcal{O}(\varepsilon)\kappa^2([\boldsymbol{B}\ A\boldsymbol{\mathcal{V}}_m])$ [6]. However, by making a correction

**Table 2** Acronyms for BGS skeletons. Here "$m$-cycle" refers to a restart cycle, or the construction of $\mathcal{V}_{m+1}$. Loss of orthogonality is defined in (3), and here $\kappa$ is shorthand for $\kappa([\boldsymbol{B}\ A\mathcal{V}_m])$. The loss of orthogonality bound for `BMGS-LTS` is conjecture and for `BMGS-CWY`, `BMGS-ICWY`, and `BCGSI+LS`, unknown

| Underlying Gram-Schmidt skeleton | Meaning behind abbreviations | Section | number of sync points per $m$-cycle | bound on loss of orthogonality, assumption on $\kappa$ |
|---|---|---|---|---|
| `BMGS` | Block Modified Gram-Schmidt | 2.4 | $\frac{m(m+1)}{2}$ | $\mathcal{O}(\varepsilon)\kappa$, $\mathcal{O}(\varepsilon)\kappa < 1$ |
| `BCGS-PIP` | Block Classical GS, Pythagorean with Inner Product | 3.1 | $m+1$ | $\mathcal{O}(\varepsilon)\kappa^2$, $\mathcal{O}(\varepsilon)\kappa^2 < 1$ |
| `BCGS-PIO` | Block Classical GS, Pythagorean with Intraorthogonalization | 3.1 | $2m+1$ | $\mathcal{O}(\varepsilon)\kappa^2$, $\mathcal{O}(\varepsilon)\kappa^2 < 1$ |
| `BMGS-SVL/BMGS-LTS` | Schreiber & Van Loan/Lower Triangular Solve | 3.2 | $3m$ | $\mathcal{O}(\varepsilon)\kappa$, $\mathcal{O}(\varepsilon)\kappa < 1$ |
| `BMGS-CWY/BMGS-ICWY` | Compact WY/Inverse Compact WY | 3.3 | $m+2$ | – |
| `BCGSI+LS` | Inner Reorthogonalization (+), Low-Sync | 3.4 | $m+2$ | – |

based on the block Pythagorean theorem (as derived in, e.g., Section 2.1 [6]), we can guarantee a loss of orthogonality bounded by $\mathcal{O}(\varepsilon)\,\kappa^2([\boldsymbol{B}\ A\boldsymbol{\mathcal{V}}_m])$, as long as $\mathcal{O}\left(\sqrt{\varepsilon}\right)\kappa([\boldsymbol{B}\ A\boldsymbol{\mathcal{V}}_m]) \leq 1$.

One version of the corrected algorithm is given as Algorithm 3. The acronym "PIP" stands for "Pythagorean (variant) with Inner Product," due to how the factor $H_{k+1,k}$ is computed. An alternative formulation based off `BCGS-PIO` (where "PIO" stands for "Pythagoren with IntraOrthogonalization") is also possible and is given as Algorithm 4. Note that in line 5, we use $\sim$ to denote that a full block vector need not be computed or stored here, just the $2s \times 2s$ scaling quotient $\Omega$. For subtle reasons, `BCGS-PIO` appears to be less reliable in practice (see Section 4).

1: $[\boldsymbol{V}_1, B] = \texttt{IntraOrtho}(\boldsymbol{B})$
2: **for** $k = 1, \ldots, m$ **do**
3: $\quad \boldsymbol{W} = A\boldsymbol{V}_k$
4: $\quad \begin{bmatrix} \mathcal{H}_{1:k,k} \\ \Omega \end{bmatrix} = \langle\!\langle \begin{bmatrix} \boldsymbol{\mathcal{V}}_k & \boldsymbol{W} \end{bmatrix}, \boldsymbol{W} \rangle\!\rangle_{\mathbb{S}}$
5: $\quad H_{k+1,k} = \texttt{chol}(\Omega - \mathcal{H}_{1:k,k}^{*}\mathcal{H}_{1:k,k})$
6: $\quad \boldsymbol{V}_{k+1} = \left(\boldsymbol{W} - \boldsymbol{\mathcal{V}}_k\mathcal{H}_{1:k,k}\right)H_{k+1,k}^{-1}$
7: **end for**
8: **return** $\boldsymbol{\mathcal{V}}_{m+1} = [\boldsymbol{V}_1, \ldots, \boldsymbol{V}_{m+1}]$, $\mathcal{H}_{m+1,m} = (H_{jk})$, $B$

**Algorithm 3** $[\boldsymbol{\mathcal{V}}_{m+1}, \mathcal{H}_{m+1,m}, B] = \texttt{BCGS-PIP-Arnoldi}(A, \boldsymbol{B}, m)$.

1: $[\boldsymbol{V}_1, B] = \texttt{IntraOrtho}(\boldsymbol{B})$
2: **for** $k = 1, \ldots, m$ **do**
3: $\quad \boldsymbol{W} = A\boldsymbol{V}_k$
4: $\quad \mathcal{H}_{1:k,k} = \langle\!\langle \boldsymbol{\mathcal{V}}_k, \boldsymbol{W} \rangle\!\rangle_{\mathbb{S}}$
5: $\quad \left[\sim, \begin{bmatrix} W & 0 \\ 0 & H \end{bmatrix}\right] = \texttt{IntraOrtho}\left(\begin{bmatrix} \boldsymbol{W} & 0 \\ 0 & \mathcal{H}_{1:k,k} \end{bmatrix}\right)$
6: $\quad H_{k+1,k} = \texttt{chol}(W^*W - H^*H)$
7: $\quad \boldsymbol{V}_{k+1} = \left(\boldsymbol{W} - \boldsymbol{\mathcal{V}}_k\mathcal{H}_{1:k,k}\right)H_{k+1,k}^{-1}$
8: **end for**
9: **return** $\boldsymbol{\mathcal{V}}_{m+1} = [\boldsymbol{V}_1, \ldots, \boldsymbol{V}_{m+1}]$, $\mathcal{H}_{m+1,m} = (H_{jk})$, $B$

**Algorithm 4** $[\boldsymbol{\mathcal{V}}_{m+1}, \mathcal{H}_{m+1,m}, B] = \texttt{BCGS-PIO-Arnoldi}(A, \boldsymbol{B}, m)$.

### 3.2 `BMGS-SVL` and `BMGS-LTS`

Barlow developed and analyzed one of the first stabilized low-sync Gram-Schmidt methods by using the Schreiber-Van Loan representation of products of Householder transformations [1, 38]. Under modest conditions, this method—which we denote

here as BMGS-SVL—has loss of orthogonality like BMGS. Its success depends on tracking the loss of orthogonality via an auxiliary matrix $\mathcal{T}$ (as defined in lines 1, 2, 6, and 9 of Algorithm 5) and using this matrix to make corrections each iteration. A closely related method is BMGS-LTS, which is identical to BMGS-SVL except that the $\mathcal{T}$ matrix is formed via lower-triangular solves instead of matrix products. A column version of BMGS-LTS was first developed by Świrydowicz et al. [2] and generalized to blocks by Carson et al. [7]. Although BMGS-LTS appears to behave identically to BMGS-SVL in practice, a formal analysis for the former remains open. We present Arnoldi versions of BMGS-SVL and BMGS-LTS as, with different colors highlighting the small differences between the methods. In both methods, the main inner product in line 4 is performed as in BCGS. Meanwhile $\mathcal{T}$ acts as a kind of buffer, storing the loss of orthogonality per iteration, which is used in successive iterations to make small corrections to the computation in line 4. Balabanov and Grigori use a similar technique to stabilize randomized sketches of inner products, where instead of explicitly computing and storing $\mathcal{T}$, they solve least squares problems to compute $\mathcal{H}_{1:k,k}$ [14, 39].

1: $\mathcal{T} = I_{ms}$
2: $[V_1, B, T_{11}] = \texttt{IntraOrtho}\,(\boldsymbol{B})$
3: **for** $k = 1, \ldots, m$ **do**
4: $\quad \boldsymbol{W} = A\boldsymbol{V}_k$
5: $\quad \boldsymbol{Y} = \langle\!\langle \boldsymbol{\mathcal{V}}_k, \boldsymbol{W} \rangle\!\rangle_{\mathbb{S}}$
6: $\quad \mathcal{H}_{1:k,k} = \mathcal{T}^*_{1:k,1:k}\boldsymbol{Y}$ OR $\mathcal{T}^{-*}_{1:k,1:k}\boldsymbol{Y}$
7: $\quad [V_{k+1}, H_{k+1,k}, T_{k+1,k+1}] = \texttt{IntraOrtho}\,(\boldsymbol{W} - \boldsymbol{\mathcal{V}}_k\mathcal{H}_{1:k,k})$
8: $\quad \boldsymbol{Z} = \langle\!\langle \boldsymbol{\mathcal{V}}_k, \boldsymbol{V}_{k+1} \rangle\!\rangle_{\mathbb{S}}$
9: $\quad \mathcal{T}_{1:k,k+1} = -\mathcal{T}_{1:k,1:k}\boldsymbol{Z}\,T_{k+1,k+1}$ OR $\boldsymbol{Z}\,T_{k+1,k+1}$
10: **end for**
11: **return** $\boldsymbol{\mathcal{V}}_{m+1} = [V_1, \ldots, V_{m+1}]$, $\mathcal{H}_{m+1,m} = (H_{jk})$, $B$

**Algorithm 5** $[\boldsymbol{\mathcal{V}}_{m+1}, \mathcal{H}_{m+1,m}, B] =$ BMGS-SVL/BMGS-LTS-Arnoldi$(A, \boldsymbol{B}, m)$.

### 3.3 BMGS-CWY/BMGS-ICWY

A column-wise version of this algorithm was first presented by Świrydowicz et al. as [2, Algorithm 8]. To the best of our knowledge, we are the first to develop a block-wise formulation, which we refer to here as BMGS-CWY-Arnoldi, where CWY stands for "compact WY," an alternative way to represent Householder transformations used to originally derive this algorithm. A related Arnoldi algorithm, not treated in either [2] or [4], is based on the inverse CWY (ICWY) form, and is given simultaneously with BMGS-CWY in Algorithm 6.

It is important to note that BMGS-CWY-Arnoldi would not reduce to [2, Algorithm 8] or [4, Algorithm 6] for $s = 1$, as we have one total sync point, due to the lack

of a reorthonormalization step for $\boldsymbol{V}_k$. Algorithm 6 was largely derived by transforming `BMGS-CWY` and `BMGS-ICWY` from [7] into a block Arnoldi routine. The most challenging part is tracking how the $\mathcal{R}$ factor in the Gram-Schmidt formulation maps to $\mathcal{H}_{m+1,m}$ and determining where to scale by the off-diagonal entry $H_{k,k-1}$ each iteration. It is also possible to compute only with $\mathcal{R}$ and reconstruct $\mathcal{H}_{m+1,m}$ after $\boldsymbol{\mathcal{V}}_{m+1}$ is finished; this approach proved to be much less stable in practice, however, due to the growing condition number of $\mathcal{R}$.

$\mathcal{T} = I_{(m+1)s}$
$[V_1, B] = \texttt{IntraOrtho}(B)$
$U = V_1$
**for** $k = 1, \ldots, m+1$ **do**
  $W = AU$
  **if** $k = 1$ **then**
    $H_{1,1} = \langle\!\langle U, W \rangle\!\rangle_{\mathbb{S}}$
    $U = W - V_1 H_{1,1}$
  **else**
    $\begin{bmatrix} Y & Z \\ \Omega & \tilde{P} \end{bmatrix} = \langle\!\langle \left[\boldsymbol{\mathcal{V}}_{k-1}\ U\right], \left[U\ W\right] \rangle\!\rangle_{\mathbb{S}}$
    $H_{k,k-1} = \texttt{chol}(\Omega)$
    $P = H_{k,k-1}^{-*} \tilde{P}$
    $\mathcal{T}_{1:k-1,k} = -\mathcal{T}_{1:k-1,1:k-1}\left(Y H_{k,k-1}^{-1}\right)$ OR $Y H_{k,k-1}^{-1}$
    $H_{1:k,k} = \mathcal{T}_{1:k,1:k}^{*}\left(\begin{bmatrix} Z \\ P \end{bmatrix} H_{k,k-1}^{-1}\right)$ OR $\mathcal{T}_{1:k,1:k}^{-*}\left(\begin{bmatrix} Z \\ P \end{bmatrix} H_{k,k-1}^{-1}\right)$
  **end if**
  $V_k = U H_{k,k-1}^{-1}$
  $U = W H_{k,k-1}^{-1} - \boldsymbol{\mathcal{V}}_{k-1} H_{1:k,k}$
**end for**
**return** $\boldsymbol{\mathcal{V}}_{m+1} = [V_1, \ldots, V_{m+1}]$, $\mathcal{H}_{m+1,m} = (H_{jk})$, $B$

**Algorithm 6** $[\boldsymbol{\mathcal{V}}_{m+1}, \mathcal{H}_{m+1,m}, B] =$ `BMGS-CWY/BMGS-ICWY-Arnoldi`$(A, \boldsymbol{B}, m)$.

### 3.4 `BCGSI+LS`

One of the most intriguing of all the low-sync algorithms is DCGS2 [5], referred to as `CGSI+LS` in [7]. This algorithm is a reformulation of reorthogonalized CGS with a single sync point derived by "delaying" normalization to the next iteration, where operations are batched in a kind of $s$-step approach (where $s = 2$). The column-wise version exhibits $\mathcal{O}(\varepsilon)$ loss of orthogonality; a rigorous proof of the backward stability bounds remains open, however. The block version, `BCGSI+LS`, does not exhibit perfect $\mathcal{O}(\varepsilon)$ LOO; see numerical results in [7].

Bielich et al. present a column-wise Arnoldi based on DCGS2 as Algorithm 4 in [5]. Our Algorithm 7 is a direct block generalization of this algorithm with slight

reformulations to match the aesthetics of Algorithm 6 and principles of Remark 3. Note that, as in Algorithm 6, we are able to compute $\mathcal{H}_m$ directly, but we must track an auxiliary matrix $\boldsymbol{J}$ and scale several quantities by $H_{k-1,k-2}$. An alternative version of Algorithm 7 based more directly on `BCGSI+LS` from [7, Algorithm 7] is included in the code but not described here.

1: $[\boldsymbol{V}_1, B] = \texttt{IntraOrtho}(\boldsymbol{B})$
2: $\boldsymbol{U} = \boldsymbol{V}_1$
3: **for** $k = 1, \ldots, m+1$ **do**
4: $\quad \boldsymbol{W} = A\boldsymbol{U}$
5: $\quad$ **if** $k = 1$ **then**
6: $\quad\quad \boldsymbol{J} = \langle\!\langle \boldsymbol{U}, \boldsymbol{W} \rangle\!\rangle_{\mathbb{S}}$
7: $\quad\quad H_{1,1} = \boldsymbol{J}$
8: $\quad\quad \boldsymbol{U} = \boldsymbol{W} - \boldsymbol{V}_1 \boldsymbol{J}$
9: $\quad$ **else**
10: $\quad\quad \begin{bmatrix} \boldsymbol{Y} & \boldsymbol{Z} \\ \widetilde{\Omega} & \widetilde{P} \end{bmatrix} = \langle\!\langle \begin{bmatrix} \boldsymbol{\mathcal{V}}_{k-1} & \boldsymbol{U} \end{bmatrix}, \begin{bmatrix} \boldsymbol{U}, \boldsymbol{W} \end{bmatrix} \rangle\!\rangle_{\mathbb{S}}$
11: $\quad\quad \Omega = \widetilde{\Omega} - \boldsymbol{Y}^* \boldsymbol{Y}$
12: $\quad\quad H_{k,k-1} = \texttt{chol}(\Omega)$
13: $\quad\quad \mathcal{H}_{1:k-1,k-1} = \boldsymbol{J} + \boldsymbol{Y}$
14: $\quad\quad P = H_{k,k-1}^{-*}(\widetilde{P} - \boldsymbol{Y}^* \boldsymbol{Z})$
15: $\quad\quad \boldsymbol{J} = \left( \begin{bmatrix} \boldsymbol{Z} \\ P \end{bmatrix} - \mathcal{H}_{1:k,1:k-1} \boldsymbol{Y} \right) H_{k,k-1}^{-1}$
16: $\quad\quad \boldsymbol{V}_k = (\boldsymbol{U} - \boldsymbol{\mathcal{V}}_{k-1} \boldsymbol{Y}) H_{k,k-1}^{-1}$
17: $\quad\quad \boldsymbol{U} = \left( \boldsymbol{W} - \boldsymbol{\mathcal{V}}_k \begin{bmatrix} \boldsymbol{Z} \\ P \end{bmatrix} \right) H_{k,k-1}^{-1}$
18: $\quad$ **end if**
19: **end for**
20: **return** $\boldsymbol{\mathcal{V}}_{m+1} = [\boldsymbol{V}_1, \ldots, \boldsymbol{V}_{m+1}]$, $\mathcal{H}_{m+1,m} = (H_{jk})$, $B$

**Algorithm 7** $[\boldsymbol{\mathcal{V}}_{m+1}, \mathcal{H}_{m+1,m}, B] = \texttt{BCGSI+LS-Arnoldi}(A, \boldsymbol{B}, m)$.

## 4 Adaptive restarting

Reproducibility and stability are not mutually exclusive. This realization is precisely the motivation for an adaptive restarting routine and can be demonstrated by a simple example.

Consider the `tridiag` test case from Section 5.1 with $n = 100$. Notably, both $A$ and $\boldsymbol{B}$ are deterministic quantities; neither is defined with random elements. In MATLAB, it is possible to specify the number of threads on which a script is executed via

the built-in `maxNumCompThreads` function.[5] We solve $AX = B$ with Algorithms 3 and 4 while varying the multithreading setting from 1 to 16 on a standard node of the Mechthild cluster; see the beginning of Section 5 for more details about the cluster. For both algorithms, we employ a variant of MATLAB 's Cholesky routine `chol`, which stores a flag when `chol` determines a matrix is too ill-conditioned to be factorized. This flag is fed to the linear solver driver of `LowSyncBlockArnoldi` (`bfom`), which halts the process when the flag is true. Through the following discussion, we refer to this flag as the "`NaN`-flag," because ignoring it leads to computations with ill-defined quantities.

Figure 1 displays the loss of orthogonality (3) and $\kappa([B \;\; A\mathcal{V}_k])$ for different thread counts. The condition numbers for all thread counts and both methods are hardly affected, except for some slight deviation for `BCGS-PIP` and 16 threads. The LOO plots are more telling: for both methods, changing the thread count directly affects the LOO and how many iterations the method can compute before encountering a `NaN`-flag. We allowed for a maximum basis size of $m = 50$, but no method can compute that far. `BCGS-PIO` with 8 threads gives up first at 16 iterations; `BCGS-PIP` with 1 and 4 threads makes it all the way to 35 iterations. Among the `BCGS-PIO` methods, there are orders of magnitude differences between the attained LOO.

This situation is perplexing on the surface: the problem is static, and the same code has been run every time. The only variable is the thread count.

There are two subtle issues that affect reproducibility in this case: 1) the configuration of math kernel libraries according to the parameters of the operating system and hardware,[6] and 2) guaranteed stability bounds. As for stability bounds, it is important to note that both `BCGS-PIO` and `BCGS-PIP` have a complete backward stability analysis [6]. Both methods have $\mathcal{O}(\varepsilon)\,\kappa^2([B \;\; A\mathcal{V}_k])$ loss of orthogonality, as long as $\kappa([B \;\; A\mathcal{V}_k]) \leq \mathcal{O}\left(\frac{1}{\sqrt{\varepsilon}}\right) = \mathcal{O}\left(10^8\right)$ and as long as the `IntraOrtho` for `BCGS-PIO` behaves no worse than `CholQR`. (For this test, we used `HouseQR`, MATLAB 's built-in `qr` routine, which is unconditionally stable and therefore behaves better than `CholQR` [15].) For both methods, $\kappa([B \;\; A\mathcal{V}_k])$ exceeds $\mathcal{O}\left(10^8\right)$ around iteration 15. At that point, the assumptions for the LOO bounds are no longer satisfied. The fact that either algorithm continues to compute something useful after that point is a lucky accident.

Computing $\kappa([B \;\; A\mathcal{V}_k])$ every iteration to check whether the LOO bounds are satisfied is not practical. We therefore propose a simple adaptive restarting regime based on whether `chol` raises a `NaN`-flag, which happens whenever `chol` is fed a numerically non-positive definite matrix. When a `NaN`-flag is raised, we give up computing a new basis vector and go back to the last safely computed basis vector, which is then used to restart. Simultaneously, the maximum basis size $m$ is also reduced. It is possible that an algorithm exhausts its maximum allowed restarts and basis size before converging; indeed, we have observed this often for `BCGS-PIP` in examples not reported here. At the same time, there are many scenarios in which restarting is

[5] https://mathworks.com/help/matlab/ref/maxnumcompthreads.html. Accessed 8 August 2022.

[6] https://www.intel.com/content/www/us/en/develop/documentation/onemkl-linux-developer-guide/top/obtaining-numerically-reproducible-results/reproducibility-conditions.html. Accessed 8 August 2022.

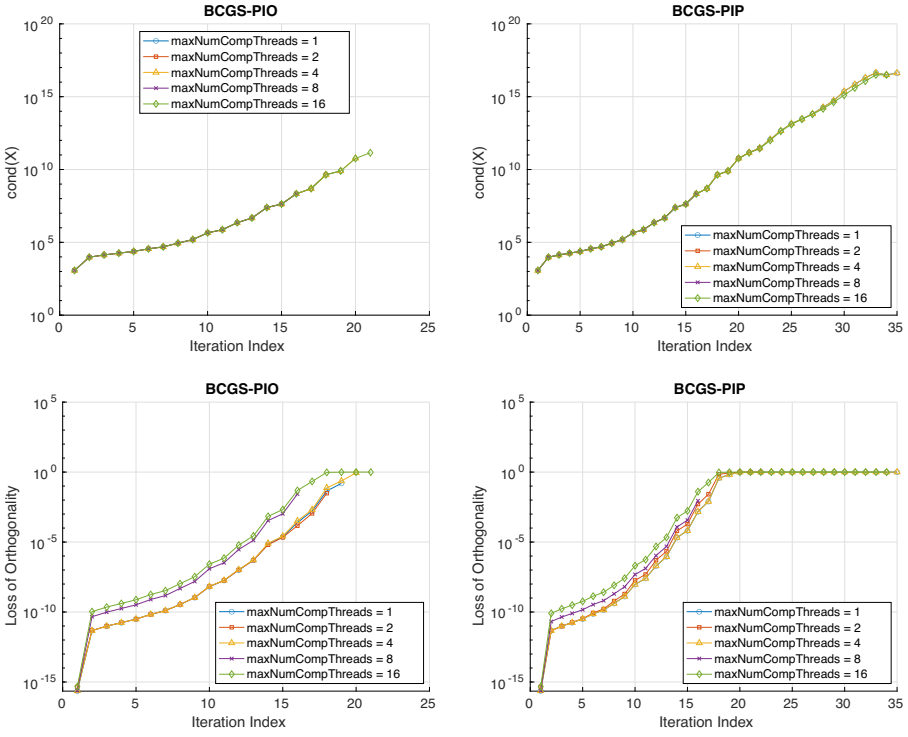


**Fig. 1** Multithreading example for `tridiag` problem with $n = 100$, no restarts, and maximum basis size $m = 50$

an adequate band-aid, thus allowing computationally cheap, one-sync algorithms line `BCGS-PIP` to salvage progress and converge, oftentimes faster than competitors. See Section 5 for demonstrations.

*Remark 4* The restarted framework outlined in Section 2.4.2 does not change fundamentally with adaptive cycle lengths; only the notation becomes more complicated. We omit the details here.

## 5 Numerical benchmarks

Our treatment of BGS and block Krylov methods is hardly exhaustive. It is not our goal to determine the optimal block Arnoldi configuration at this stage, but rather to demonstrate the functionality of a benchmarking tool for the fair comparison of possible configurations on different problems. To this end, we restrict ourselves to the options below:

- inner products: `cl` (classical), `gl` (global)
- skeletons: Table 2

- muscles: `CholQR`, which has $\mathcal{O}(\varepsilon)\kappa^2$ loss of orthogonality guaranteed only for $\mathcal{O}(\varepsilon)\kappa^2 < 1$, but is a simple, single-reduce algorithm. In practice, we would recommend `TSQR`/`AllReduceQR` [30, 31], which has $\mathcal{O}(\varepsilon)$ loss of orthogonality and the same number of sync points, but is difficult to program in MATLAB due to limited parallelization and message-passing features. Other low-sync muscles are programmed in `LowSyncBlockArnoldi` as well, and the user can easily integrate their own. Note that `BCGS-PIP` does not require a muscle, and `BMGS-CWY`, `BMGS-ICWY`, and `BCGSI+LS` only call a muscle once, in the first iteration of a new basis. `BMGS-SVL` and `BMGS-LTS` are forced to use their column-wise counterparts `MGS-SVL` and `MGS-LTS` (both 3-sync), respectively, and global methods are forced to use the global muscle (i.e., normalization without intraorthogonalization via the scaled Frobenius norm).
- modification: `none` (FOM), `harmonic` (GMRES)

All results are generated by the `LowSyncBlockArnoldi` MATLAB package. A single script (`paper_script.m`) comprises all the calls for generating the results in this manuscript. `LowSyncBlockArnoldi` is written as modularly as possible, to facilitate the exchange of inner products, skeletons, muscles, and modifications. While the timings reported certainly do not reflect the optimal performance for any of the methods, they do reflect a fair comparison across implementations and provide insights for possible speed-ups when these methods are ported to more complex architectures. The code is also written so that sync points (`inner_prod` and `intra_ortho`) and other potentially communication-intensive operations (`matvec` and `basis_eval`) are separate functions that can be tuned individually.

Every test script (including the example from Section 4) has been executed in MATLAB R2019b on 16 threads of a single, standard node of Linux Cluster Mechthild at the Max Planck Institute for Dynamics of Complex Technical Systems in Magdeburg, Germany.[7] A standard node comprises 2 Intel Xeon Silver 4110 (Skylake) CPUs with 8 Cores each (64KB L1 cache, 1024KB L2 cache), a clockrate of 2.1 GHz (3.0 GHz max), and 12MB shared L3 cache each. We further focus on small problems that easily fit in the L3 Cache, which is easy to guarantee with sparse $A$, $n \leq 10^4$, and $s \leq 10$. Given that the latency between CPUs on a single node is small relative to exascale machines, we expect small improvements observed in these test cases to translate to bigger gains in a more complex setting.

For the timings, we measure the total time spent to reach a specified error tolerance. We run each test 5 times and average over the timings. We also calculate several intermediate measures, namely counts for $A$-calls, applications of $\boldsymbol{\mathcal{V}}_k$, and sync points. In addition, we plot the convergence history in terms of the following quantities per iteration: relative residual, relative error, $\kappa([\boldsymbol{B}\ A\boldsymbol{\mathcal{V}}_k])$, and loss of orthogonality (LOO) (3). When a ground truth solution $\boldsymbol{X}_*$ is provided, the error is calculated as

$$\|\boldsymbol{X}_k - \boldsymbol{X}_*\|_{\mathrm{F}} / \|\boldsymbol{X}_*\|_{\mathrm{F}},$$

[7] https://www.mpi-magdeburg.mpg.de/cluster/mechthild. Accessed 8 August 2022.

For all our examples, $\boldsymbol{X}_*$ is computed by MATLAB 's built-in `backslash` operator. The residual is approximated by (9) and is scaled by $\|\boldsymbol{B}\|_F$. A summary of the parameters for all benchmarks can be found in Table 3. Except for `tridiag` and `lapl_2d`, all examples are taken from the SuiteSparse Matrix Collection [40]. Via the `suite_sparse.m` script, it is possible to run tests on any benchmark from this collection.

### 5.1 `tridiag`

The operator $A$ is defined as a sparse, tridiagonal matrix with 1 on the off-diagonals and $-1, -2, \ldots, -n$ on the diagonal, where $n$ is also the size of $A$. Clearly $A$ is symmetric. The right-hand side $\boldsymbol{B}$ has two columns, where the first has identical elements $\frac{1}{\sqrt{n}}$ and the second is $1, 2, \ldots, n$. This example is actually procedural, in the sense that a user can choose a desired $n$. At the same time, a larger $n$ necessarily leads to a worse condition number.

Figure 2 presents the total run time per configuration as well as operator counts as a bar chart; see Table 4 in the Appendix for more details. The fastest methods are the stabilized low-sync variants. Despite being the computationally cheapest classical method per iteration, `cl-BCGS-PIP` is notably slower than `cl-BMGS`, because its inherent instability requires restarting 3 times (and therefore additional applications of $A$ and $\boldsymbol{\mathcal{V}}_k$) before converging. The method with the fewest $\boldsymbol{\mathcal{V}}_k$ evaluations is `cl-BMGS`, which is to be expected, since the basis is split up and applied one block column at a time in the inner-most loop; see Algorithm 1.

The fastest global method, `gl-BCGS-PIP`, is significantly slower even than the slowest classical method. In fact, all global methods require over 6 times as many total iterations as the fastest classical method to converge; this is in line with the theory of Section 2.4. In this particular case, the floating-point savings per iteration do not outweigh the sheer amount of time needed for all the extra $A$-calls. Nevertheless, the one-sync global methods (`gl-BCGS-PIP`, `gl-BMGS-CWY`, `gl-BMGS-ICWY`, and `gl-BCGSI+LSs`) have relatively low sync counts, compared even to `cl-BMGS`.

Figures 3 and 4 display convergence histories for a subset of the methods in Table 4 in the Appendix. The convergence histories for all global `BMGS` variants are very similar; we omit `BMGS-SVL` and `BMGS-CWY`, as they are visually identical to

**Table 3** Test properties and parameter choices

| test name | $\kappa(A)$ | $n$ | $s$ | $m$ | modification | tol |
|---|---|---|---|---|---|---|
| `tridiag` | $\mathcal{O}(10^3)$ | 1000 | 2 | 70 | FOM | $10^{-10}$ |
| `1138_bus` | $\mathcal{O}(10^6)$ | 1138 | 5 | 30 | GMRES | $10^{-6}$ |
| `circuit_2` | $\mathcal{O}(10^5)$ | 4510 | 5 | 10 | GMRES | $10^{-6}$ |
| `rajat03` | $\mathcal{O}(10^7)$ | 7602 | 5 | 10 | GMRES | $10^{-6}$ |
| `Kaufhold` | $\mathcal{O}(10^{14})$ | 8765 | 5 | 10 | GMRES | $10^{-6}$ |
| `t2d_q9` | $\mathcal{O}(10^3)$ | 9801 | 5 | 10 | GMRES | $10^{-6}$ |
| `lapl_2d` | $\mathcal{O}(10^3)$ | 10000 | 10 | 25 | FOM | $10^{-6}$ |

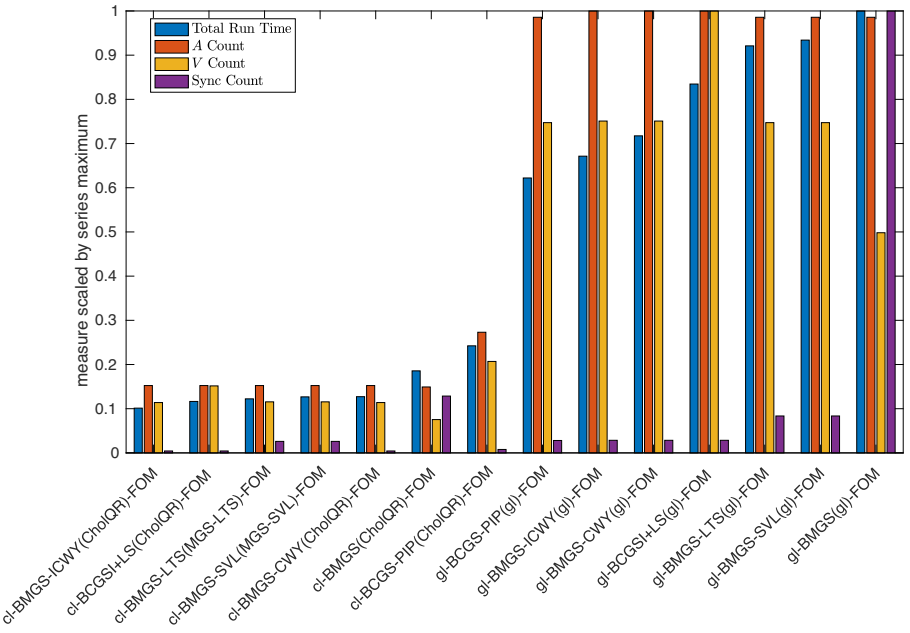


**Fig. 2** Results from `tridiag` example

`BMGS-LTS` and `BMGS-ICWY`, respectively. `BMGS` is identical to `BMGS-SVL` and `BMGS-LTS` and is therefore also omitted.

Both the classical and global variants of `BCGS-PIP` show the robustness of the adaptive restarting procedure in action. In the global case, the LOO exceeds $\mathcal{O}\left(10^{-10}\right)$ and reaches $\mathcal{O}(1)$ in `cl-BCGS-PIP`. Despite the loss of orthogonality, restarting allows the methods to recover and eventually converge. All other low-sync variants remain stable, only restarting once the basis size limit of $m = 70$ has been reached. Although hardly perceptible, `BMGS-ICWY` does have a slightly worse LOO than that of `BMGS-LTS`, which can be seen by zooming in on the last few iterations of the global plots in Fig. 3 or of the classical plots in Fig. 4.

We also note that the residual estimate (7) for all methods follows the same qualitative trend as that of the error. In the worst case, `cl-BCGS-PIP`, the residual is nearly 3 orders of magnitude lower than the error in some places, which could lead to premature convergence. For all other methods, the difference is between 1 and 2 orders of magnitude. We would thus recommend setting the residual tolerance a couple orders of magnitude lower in practice, to ensure that the true error is accurate enough.

### 5.2 `1138_bus`

Now we turn to a slightly more complicated matrix. The matrix $A$ comes from a power network problem and is real and symmetric positive definite, while entries

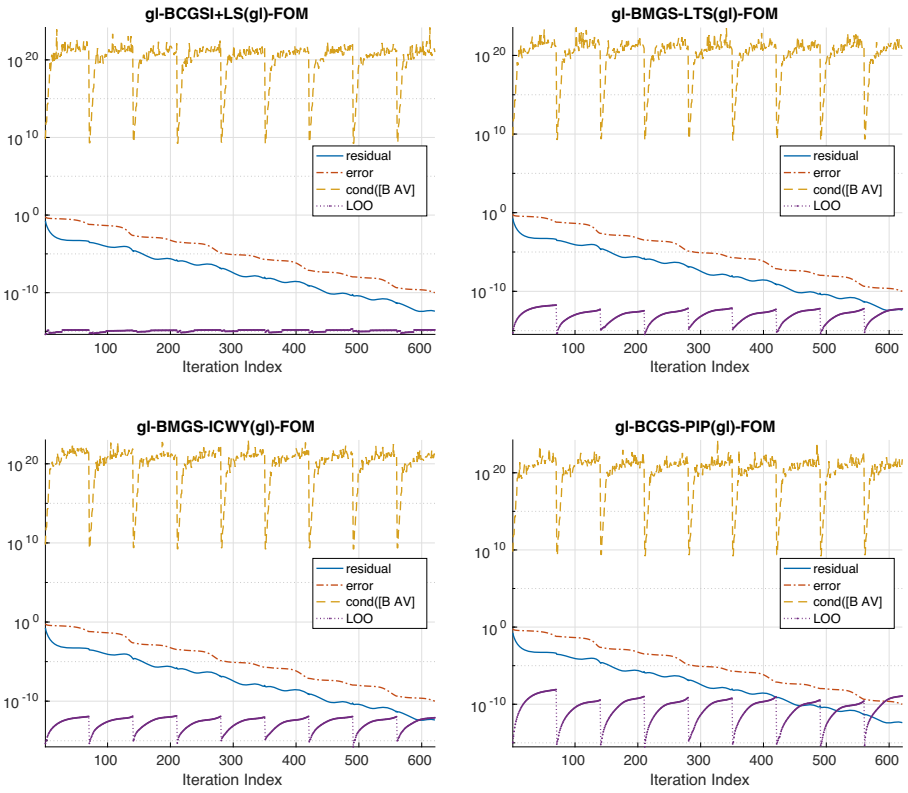


**Fig. 3** Convergence histories of some global variants for `tridiag` example

of $\boldsymbol{B}$ are drawn randomly from the uniform distribution. Moreover we apply an incomplete LU (ILU) preconditioner with no fill, using MATLAB 's built-in `ilu`.

Even with the preconditioner, none of the global methods converges. We adjusted the thread count to see if it would aid convergence, to no avail. This is perhaps an extreme case of [22, Theorem 3.3], wherein the global method is much less accurate than the classical method in the first cycle and cannot manage to catch up even after restarting. A preconditioner better attuned to the structure of the problem may alleviate stagnation for global methods, but we do not explore this here.

In Fig. 5 we see the performance results for the convergent classical methods; more details can be found in Table 5 in the Appendix. Most notably, the one-sync methods `BMGS-CWY`, `BMGS-ICWY`, and `BCGSI+LS` improve over `BMGS` only slightly in terms of timings. `BCGS-PIP` is much slower, due to a quick loss of orthogonality and need to restart more often. However, it is clear that sync counts for all one-sync methods are drastically reduced compared to that of `BMGS`.

We examine the convergence histories of `cl-BCGS-PIP` and `cl-BMGS-ICWY` more closely in Fig. 6. Although not discernible on the graph, we found that `cl-BCGS-PIP` actually restarts every 28 iterations, meaning in the first cycle it

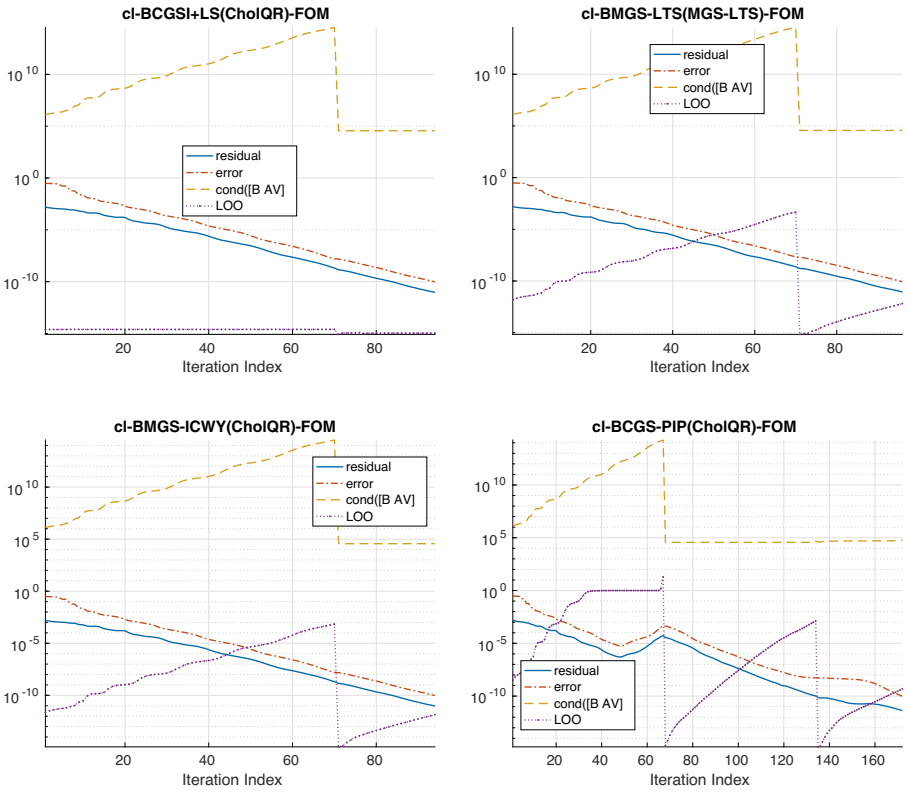


**Fig. 4** Convergence histories of some classical variants for `tridiag` example

encountered a `NaN`-flag and reduced the maximum basis size to $m = 28$ for all subsequent cycles. Instability in the first cycle thus hinders `cl-BCGS-PIP` greatly. On the other hand, `BMGS-ICWY` (as well as the other variants) is stable enough to exhaust the entire basis size allowance, which allows for further error reduction in the first cycle.

### 5.3 `circuit_2`

The next example comes from a circuit simulation problem. The matrix $A$ is real but not symmetric or positive definite. We again apply an ILU preconditioner with no fill.

All the one-sync classical and global methods converge, and their performance data is presented in Fig. 7 with further details in Table 6 in the Appendix. In fact, some global methods, like `gl-BCGS-PIP`, are even faster than some classical methods, due to the fact that they require the same number of iterations to converge, and therefore fewer floating-point operations.

Figure 8 demonstrates how close in accuracy the global and classical `BCGS-PIP` variants are for this problem. The global method even has a slightly better LOO, but

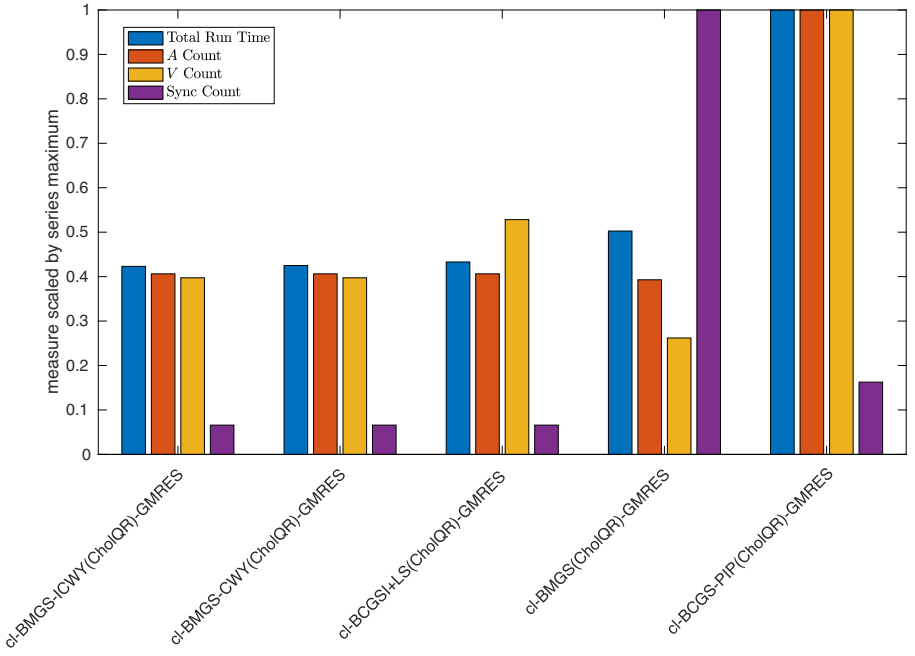


**Fig. 5** Performance results for `1138_bus` example

it should be noted that global LOO is measured according to a different inner product than classical LOO; see Section 2.2 and (3).

### 5.4 `rajat03`

Another circuit simulation problem highlights slightly different behavior. In this case, $A$ is again real but neither symmetric nor positive definite, and we again use an ILU preconditioner with no fill.

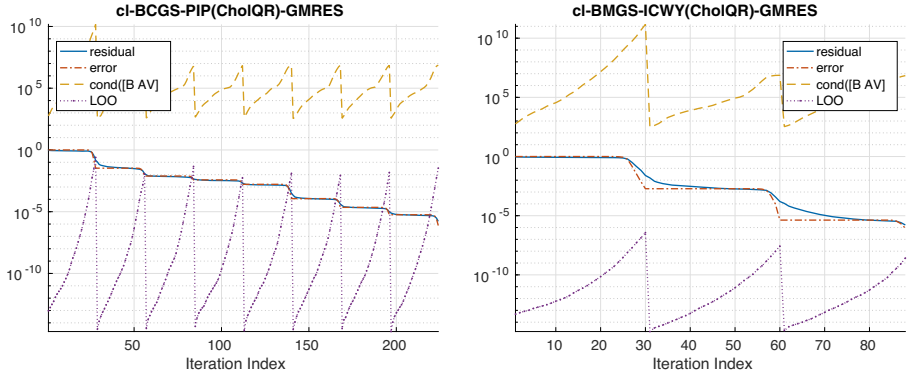


**Fig. 6** Subset of convergence histories for `1138_bus` example

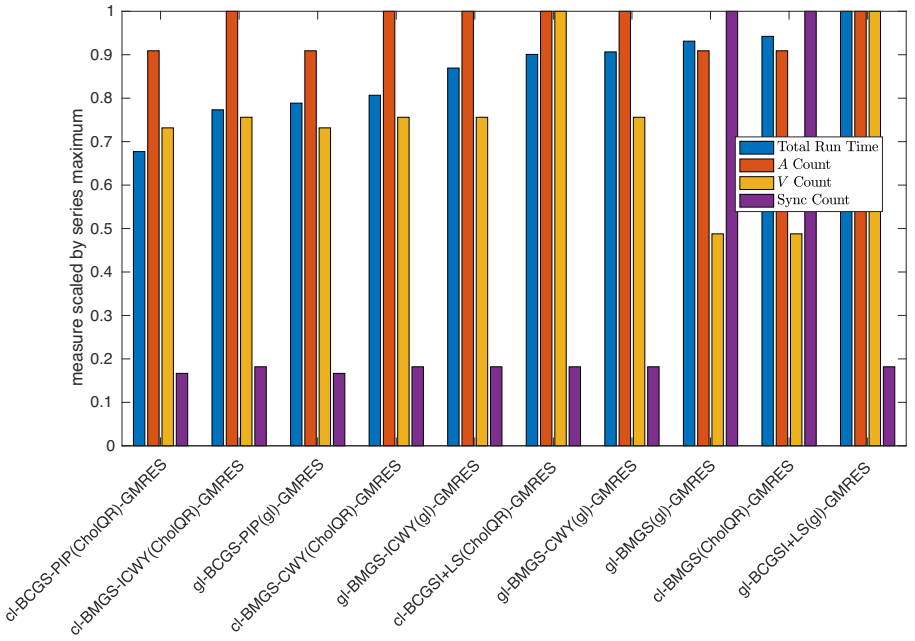


**Fig. 7** Performance results for `circuit_2` example

Figure 9 summarizes the performance results, with details given in Table 7 in the Appendix. It should be noted right away that `cl-BCGS-PIP` fails to converge for this problem, while `gl-BCGS-PIP` does not, and takes second place in terms of the timings. More specifically, `cl-BCGS-PIP` encounters a `NaN`-flag it cannot resolve, which means that every time it reduces the basis size, it cannot avoid a `NaN`-flag. However, because global methods do not use Cholesky at all, non-positive definite factors do not pose a problem, unless their trace is numerically zero, which occurs with very low probability. Otherwise, `cl-BMGS-CWY` shows a small improvement over `cl-BMGS`.

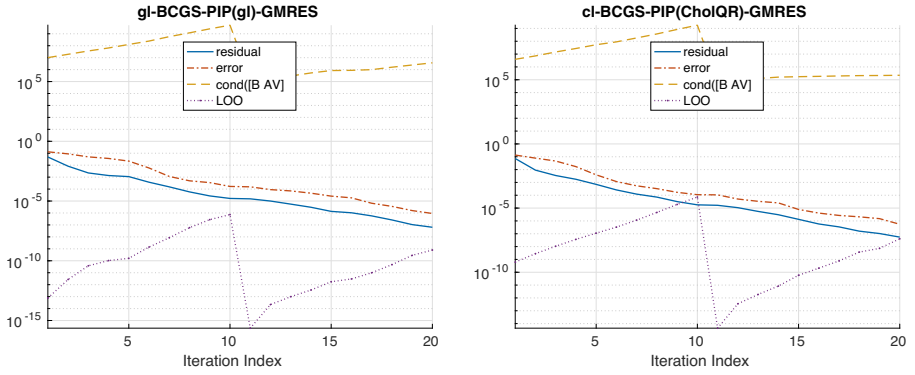


**Fig. 8** Convergence histories of the `BCGS-PIP` variants for the `circuit_2` example

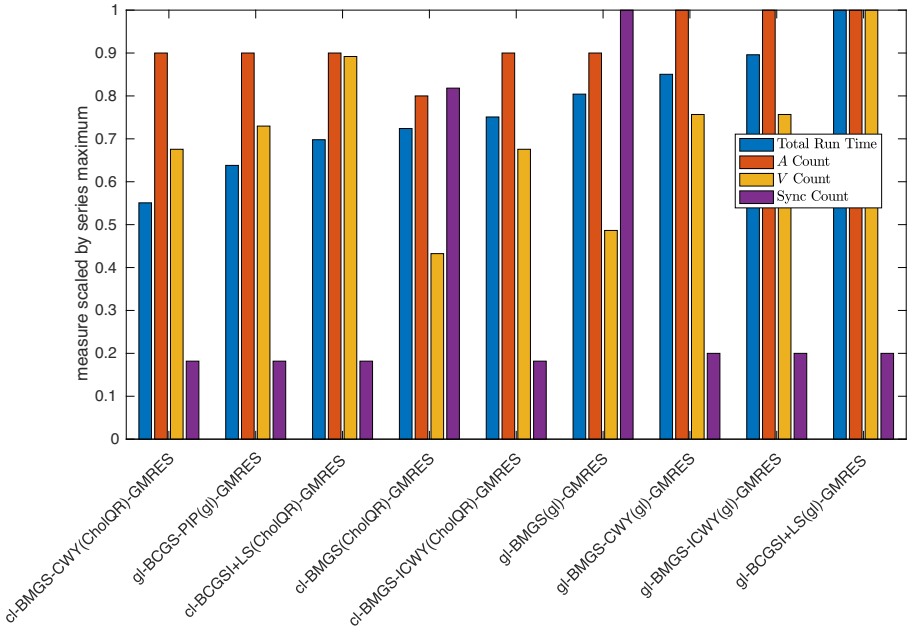


**Fig. 9** Performance results for `rajat03` example

Table 7 in the Appendix confirms that none of the methods requires restarting despite how high the condition number becomes in later iterations; see also Fig. 10. It is again interesting to see how close the error and residual plots are between the global and classical methods. In fact, the residual for the global method underestimates convergence by a couple orders of magnitude.

### 5.5 `Kaufhold`

This example treats a nearly numerically singular matrix with an extremely high condition number. Also notable, the norm of $A$ is nearly $\mathcal{O}\left(10^{15}\right)$. The matrix is

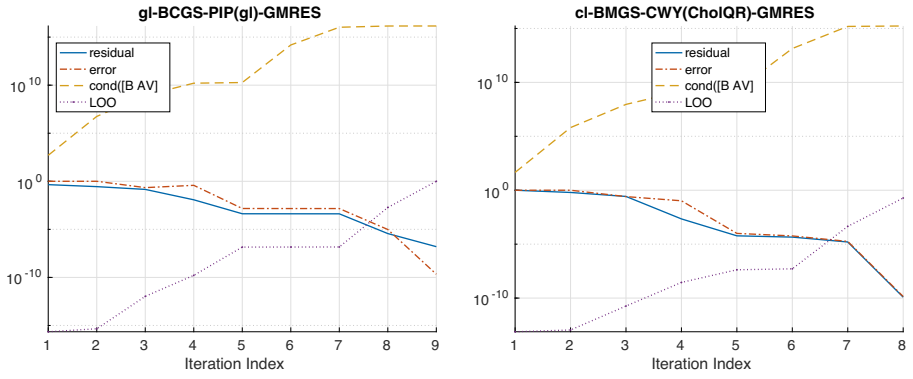


**Fig. 10** Convergence histories of the two fastest variants for the `rajat03` example

real, but neither symmetric nor positive definite, and it was designed to trigger a bug in Gaussian elimination in a 2002 version of MATLAB. We again apply an ILU preconditioner with no fill.

Figure 11 shows `cl-BCGS-PIP` to be the fastest of the classical one-sync methods, but the improvement over `cl-BMGS` is small. The global methods are all much slower. A look at the convergence histories in Fig. 12 shows a stubborn error curve despite significant progress in the initial iterations. For both `BCGS-PIP` methods the LOO is moderately high in the first cycle, matching the high condition numbers, but the situation is not bad enough to trigger a `NaN`-flag, and the LOO drops after restarting.

### 5.6 `t2d_q9`

We now examine a nonlinear diffusion problem, specifically a biquadratic mesh of a temperature field. The matrix $A$ is real but not symmetric or positive definite, and we again use an ILU preconditioner with no fill.

Figure 13 shows that both `BCGS-PIP` are the fastest overall, with `cl-BMGS` in second-to-last place; see Table 9 in the Appendix for more details. Interestingly, even `gl-BMGS` is faster than `cl-BMGS` in this scenario.

Both `BCGSI+LS` variants are rather slow in this example. Despite having just one sync per iteration, `BCGSI+LS` does generally have a higher complexity than its one-sync counterparts, which manifests here as a disadvantage.

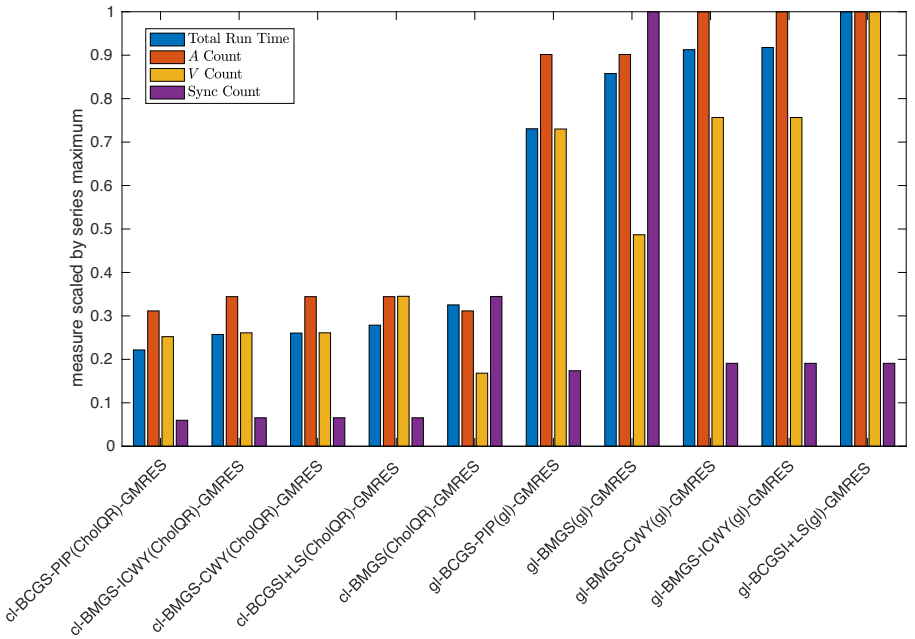


**Fig. 11** Performance results for `Kaufhold` example

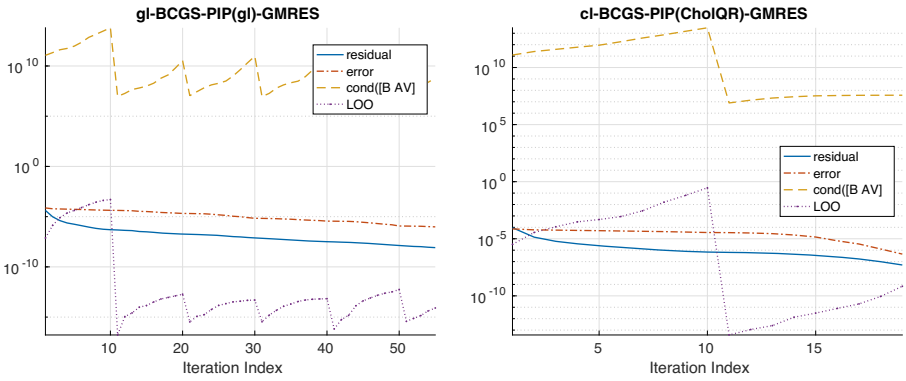


**Fig. 12** Convergence histories of the `BCGS-PIP` variants for the `Kaufhold` example

The convergence behavior for the `BCGS-PIP` variants is given in Fig. 14. Here we see that despite the global condition number having a high variation relative to the classical method, the global LOO is overall much less. This phenomenon is not unique to this example, however, it just happens to be more noticeable.

### 5.7 `lapl_2d`

Our last problem is taken directly from [21, Section 5.4], a discretized two-dimensional Laplacian matrix. $A$ is thus banded, real, and symmetric positive

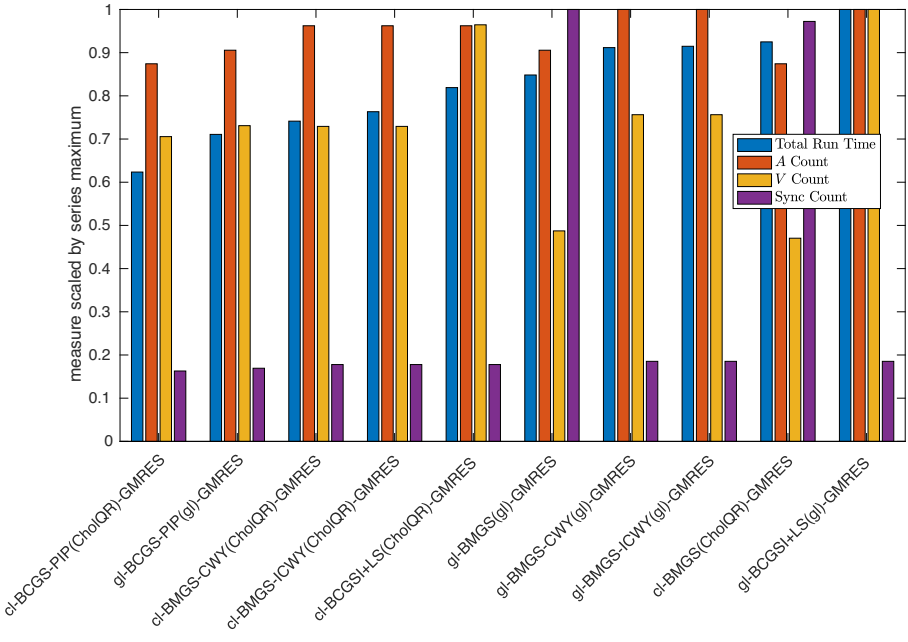


**Fig. 13** Performance results for `t2d_q9` example

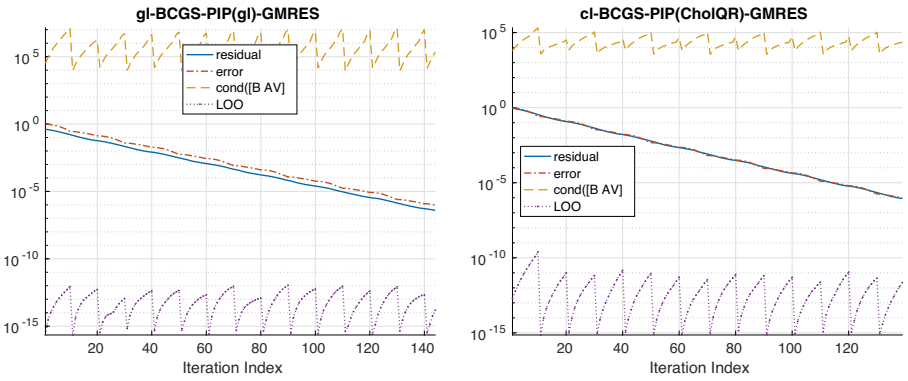


**Fig. 14** Convergence histories of the `BCGS-PIP` variants for the `t2d_q9` example

definite. We do not apply a preconditioner and look at all skeletons considered in the text.

Figure 15 shows the performance results; more details can be found in Table 4 in the Appendix. All one-sync classical methods except for `cl-BCGSI+LS` beat `cl-BMGS`, along with a number of global methods. The slowest classical methods are the three-sync ones, and some one-sync global methods follow behind. The fastest method, `cl-BCGS-PIP` also happens to have the highest $A$ count and applications of $\mathcal{V}_k$, due to its high number of restarts. Both `cl-BMGS-CWY` and `cl-BMGS-ICWY`, however, have fewer sync counts, as well as $A$ counts and $\mathcal{V}_k$ counts, and are very close in terms of timings.

The methods with the highest sync counts are `cl-BMGS-SVL` and `cl-BMGS-LTS`. The reason is that they cannot use `CholQR` as a muscle,[8] and this problem requires many iterations to converge. `LowSyncBlockArnoldi` is written to count sync points within the muscles as well, and with `MGS-SVL` and `MGS-LTS` each contributing $1 + 3s$ per call, the total number of sync points eventually passes that of `cl-BMGS`, which can use a communication-light muscle like `CholQR`.

## 6 Conclusions and outlook

Stability bounds and floating-point analysis are challenging to work out rigorously, and it is therefore simultaneously important to search for counterexamples and edge cases while trying to prove conjectured bounds. In general, rigorous loss of orthogonality and backward error bounds for all these methods could lead to new insights and improvements in the quest for a reliable, scalable Krylov subspace solver. Our flexible benchmarking tool can aid in that process, and it can easily be extended to accommodate new algorithm configurations, test cases, and measures.

[8] Strictly speaking, they can use whatever muscle they are programmed to use, but `BMGS-SVL` requires `MGS-SVL` to be stable; see [1, 7].

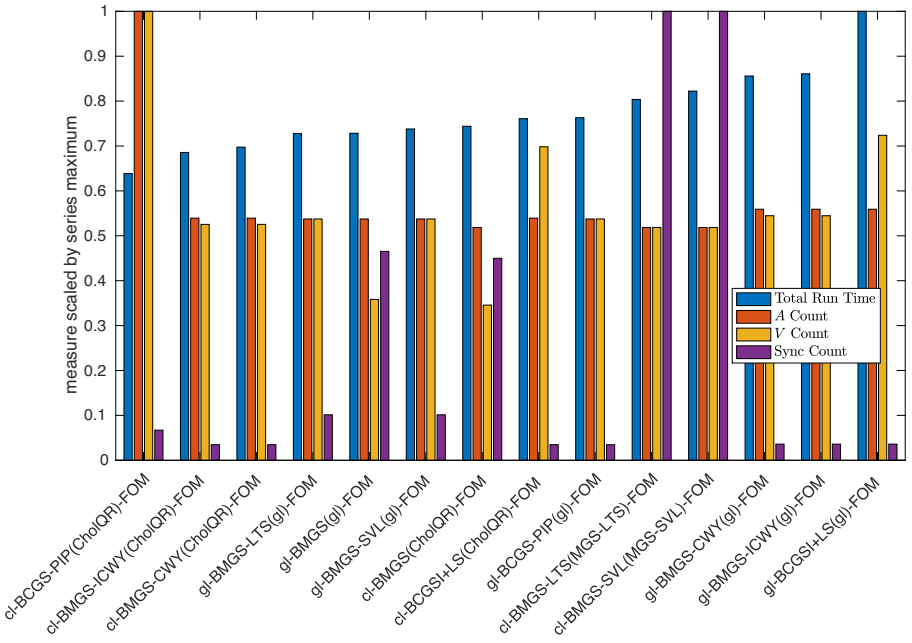


**Fig. 15** Performance results for `lapl_2d` example

At the same time, low-sync block Arnoldi algorithms with adaptive restarting are clearly already useful and robust enough for a wide variety of problems, especially where $A$ is reasonably conditioned and memory limitations cap basis sizes. In every benchmark, we have observed that at least one low-sync method outperformed both the classical and global BMGS-based Arnoldi methods. More research is needed to determine which low-sync skeletons are best for which problems and architectures, particularly computational models that account not only for operation counts but also for performance variations relative to block size [26, 41, 42]. Most likely the best configuration allows for switching between skeletons and muscles depending on convergence behavior.

For scenarios where the basic adaptive restarting procedure is not sufficient to rescue convergence, it might be possible to improve the heuristics with a cheap estimate of the loss of orthogonality computed, e.g., a randomized sketched inner product [39]. With such a cheap estimate, we could not only decrease the basis size when there are problems, but increase it again in later cycles. Randomized algorithms themselves are known to reduce communication, and a thorough comparison and combination of the methods proposed here and in [39] could lead to powerful Krylov subspace method well suited for exascale architectures.

Global methods are unfortunately less promising. They are almost always slower than even the slowest classical method, due to requiring more cycles, and thus operator calls and sync points, to converge. However, the benchmarks do suggest that, in cases with a good preconditioner known to guarantee convergence in a few iterations,

global methods may become competitive again, especially in single-node or "laptop" applications, where their reduced computational intensity per iteration is favorable.

## Appendix. Raw data from tests

A subset of raw data corresponding to the performance plots in Section 5 is provided below. Many headers are abbreviated for space reasons: "Accel." refers to "acceleration" or "speed-up"; "Ct." refers to "Count"; and "Iter." refers to "Iteration".

**Table 4** Results from `tridiag` example

| Configuration | Time (s) | % Accel. | Cycle Ct. | Iter. Ct. | *A* Ct. | *V* Ct. | Sync Ct. |
|---|---|---|---|---|---|---|---|
| `gl-BMGS ∘ gl-FOM` | 2.20e+00 | 0.00 | 9 | 621 | 621 | 1242 | 22401 |
| `gl-BMGS-SVL ∘ gl-FOM` | 2.05e+00 | 6.59 | 9 | 621 | 621 | 1863 | 1872 |
| `gl-BMGS-LTS ∘ gl-FOM` | 2.03e+00 | 7.90 | 9 | 621 | 621 | 1863 | 1872 |
| `gl-BCGSI+LS ∘ gl-FOM` | 1.84e+00 | 16.53 | 9 | 621 | 630 | 2493 | 639 |
| `gl-BMGS-CWY ∘ gl-FOM` | 1.58e+00 | 28.26 | 9 | 621 | 630 | 1872 | 639 |
| `gl-BMGS-ICWY ∘ gl-FOM` | 1.48e+00 | 32.86 | 9 | 621 | 630 | 1872 | 639 |
| `gl-BCGS-PIP ∘ gl-FOM` | 1.37e+00 | 37.79 | 9 | 621 | 621 | 1863 | 630 |
| `cl-BCGS-PIP ∘ CholQR-FOM` | 5.33e-01 | 75.77 | 3 | 172 | 172 | 516 | 175 |
| `cl-BMGS ∘ CholQR-FOM` | 4.08e-01 | 81.44 | 2 | 94 | 94 | 188 | 2881 |
| `cl-BMGS-CWY ∘ CholQR-FOM` | 2.80e-01 | 87.28 | 2 | 94 | 96 | 284 | 98 |
| `cl-BMGS-SVL ∘ MGS-SVL-FOM` | 2.79e-01 | 87.32 | 2 | 96 | 96 | 288 | 584 |
| `cl-BMGS-LTS ∘ MGS-LTS-FOM` | 2.69e-01 | 87.77 | 2 | 96 | 96 | 288 | 584 |
| `cl-BCGSI+LS ∘ CholQR-FOM` | 2.56e-01 | 88.35 | 2 | 94 | 96 | 378 | 98 |
| `cl-BMGS-ICWY ∘ CholQR-FOM` | 2.23e-01 | 89.87 | 2 | 94 | 96 | 284 | 98 |

**Table 5** Results from `1138_bus` example

| Configuration | Time (s) | % Accel. | Cycle Ct. | Iter. Ct. | *A* Ct. | *V* Ct. | Sync Ct. |
|---|---|---|---|---|---|---|---|
| `cl-BCGS-PIP ∘ CholQR-GMRES` | 1.84e+00 | 0.00 | 8 | 224 | 224 | 672 | 232 |
| `cl-BMGS ∘ CholQR-GMRES` | 9.25e-01 | 49.75 | 3 | 88 | 88 | 176 | 1427 |
| `cl-BCGSI+LS ∘ CholQR-GMRES` | 7.97e-01 | 56.70 | 3 | 88 | 91 | 355 | 94 |
| `cl-BMGS-CWY ∘ CholQR-GMRES` | 7.82e-01 | 57.50 | 3 | 88 | 91 | 267 | 94 |
| `cl-BMGS-ICWY ∘ CholQR-GMRES` | 7.78e-01 | 57.70 | 3 | 88 | 91 | 267 | 94 |

**Table 6** Results for `circuit_2` example

| Configuration | Time (s) | % Accel. | Cycle Ct. | Iter. Ct. | $A$ Ct. | $V$ Ct. | Sync Ct. |
|---|---|---|---|---|---|---|---|
| `gl-BCGSI+LS ∘ gl-GMRES` | 1.26e-01 | 0.00 | 2 | 20 | 22 | 82 | 24 |
| `cl-BMGS ∘ CholQR-GMRES` | 1.19e-01 | 5.76 | 2 | 20 | 20 | 40 | 132 |
| `gl-BMGS ∘ gl-GMRES` | 1.18e-01 | 6.88 | 2 | 20 | 20 | 40 | 132 |
| `gl-BMGS-CWY ∘ gl-GMRES` | 1.15e-01 | 9.35 | 2 | 20 | 22 | 62 | 24 |
| `cl-BCGSI+LS ∘ CholQR-GMRES` | 1.14e-01 | 9.92 | 2 | 20 | 22 | 82 | 24 |
| `gl-BMGS-ICWY ∘ gl-GMRES` | 1.10e-01 | 13.06 | 2 | 20 | 22 | 62 | 24 |
| `cl-BMGS-CWY ∘ CholQR-GMRES` | 1.02e-01 | 19.32 | 2 | 20 | 22 | 62 | 24 |
| `gl-BCGS-PIP ∘ gl-GMRES` | 9.96e-02 | 21.14 | 2 | 20 | 20 | 60 | 22 |
| `cl-BMGS-ICWY ∘ CholQR-GMRES` | 9.77e-02 | 22.67 | 2 | 20 | 22 | 62 | 24 |
| `cl-BCGS-PIP ∘ CholQR-GMRES` | 8.55e-02 | 32.29 | 2 | 20 | 20 | 60 | 22 |

**Table 7** Results for `rajat03` example

| Configuration | Time (s) | % Accel. | Cycle Ct. | Iter. Ct. | $A$ Ct. | $V$ Ct. | Sync Ct. |
|---|---|---|---|---|---|---|---|
| `gl-BCGSI+LS ∘ gl-GMRES` | 9.54e-02 | 0.00 | 1 | 9 | 10 | 37 | 11 |
| `gl-BMGS-ICWY ∘ gl-GMRES` | 8.55e-02 | 10.41 | 1 | 9 | 10 | 28 | 11 |
| `gl-BMGS-CWY ∘ gl-GMRES` | 8.11e-02 | 14.96 | 1 | 9 | 10 | 28 | 11 |
| `gl-BMGS ∘ gl-GMRES` | 7.67e-02 | 19.58 | 1 | 9 | 9 | 18 | 55 |
| `cl-BMGS-ICWY ∘ CholQR-GMRES` | 7.16e-02 | 24.91 | 1 | 8 | 9 | 25 | 10 |
| `cl-BMGS ∘ CholQR-GMRES` | 6.91e-02 | 27.60 | 1 | 8 | 8 | 16 | 45 |
| `cl-BCGSI+LS ∘ CholQR-GMRES` | 6.66e-02 | 30.21 | 1 | 8 | 9 | 33 | 10 |
| `gl-BCGS-PIP ∘ gl-GMRES` | 6.09e-02 | 36.19 | 1 | 9 | 9 | 27 | 10 |
| `cl-BMGS-CWY ∘ CholQR-GMRES` | 5.26e-02 | 44.92 | 1 | 8 | 9 | 25 | 10 |

**Table 8** Results for `Kaufhold` example

| Configuration | Time (s) | % Accel. | Cycle Ct. | Iter. Ct. | $A$ Ct. | $V$ Ct. | Sync Ct. |
|---|---|---|---|---|---|---|---|
| `gl-BCGSI+LS ∘ gl-GMRES` | 7.24e-01 | 0.00 | 6 | 55 | 61 | 226 | 67 |
| `gl-BMGS-ICWY ∘ gl-GMRES` | 6.65e-01 | 8.22 | 6 | 55 | 61 | 171 | 67 |
| `gl-BMGS-CWY ∘ gl-GMRES` | 6.61e-01 | 8.73 | 6 | 55 | 61 | 171 | 67 |
| `gl-BMGS ∘ gl-GMRES` | 6.21e-01 | 14.21 | 6 | 55 | 55 | 110 | 351 |
| `gl-BCGS-PIP ∘ gl-GMRES` | 5.29e-01 | 26.93 | 6 | 55 | 55 | 165 | 61 |
| `cl-BMGS ∘ CholQR-GMRES` | 2.36e-01 | 67.47 | 2 | 19 | 19 | 38 | 121 |
| `cl-BCGSI+LS ∘ CholQR-GMRES` | 2.02e-01 | 72.13 | 2 | 19 | 21 | 78 | 23 |
| `cl-BMGS-CWY ∘ CholQR-GMRES` | 1.89e-01 | 73.95 | 2 | 19 | 21 | 59 | 23 |
| `cl-BMGS-ICWY ∘ CholQR-GMRES` | 1.86e-01 | 74.27 | 2 | 19 | 21 | 59 | 23 |
| `cl-BCGS-PIP ∘ CholQR-GMRES` | 1.60e-01 | 77.84 | 2 | 19 | 19 | 57 | 21 |

**Table 9** Results for `t2d_q9` example

| Configuration | Time (s) | % Accel. | Cycle Ct. | Iter. Ct. | $A$ Ct. | $V$ Ct. | Sync Ct. |
|---|---|---|---|---|---|---|---|
| `gl-BCGSI+LS ∘ gl-GMRES` | 2.02e+00 | 0.00 | 15 | 144 | 159 | 591 | 174 |
| `cl-BMGS ∘ CholQR-GMRES` | 1.86e+00 | 7.51 | 14 | 139 | 139 | 278 | 913 |
| `gl-BMGS-ICWY ∘ gl-GMRES` | 1.84e+00 | 8.53 | 15 | 144 | 159 | 447 | 174 |
| `gl-BMGS-CWY ∘ gl-GMRES` | 1.84e+00 | 8.83 | 15 | 144 | 159 | 447 | 174 |
| `gl-BMGS ∘ gl-GMRES` | 1.71e+00 | 15.17 | 15 | 144 | 144 | 288 | 939 |
| `cl-BCGSI+LS ∘ CholQR-GMRES` | 1.65e+00 | 18.09 | 14 | 139 | 153 | 570 | 167 |
| `cl-BMGS-ICWY ∘ CholQR-GMRES` | 1.54e+00 | 23.67 | 14 | 139 | 153 | 431 | 167 |
| `cl-BMGS-CWY ∘ CholQR-GMRES` | 1.49e+00 | 25.86 | 14 | 139 | 153 | 431 | 167 |
| `gl-BCGS-PIP ∘ gl-GMRES` | 1.43e+00 | 28.92 | 15 | 144 | 144 | 432 | 159 |
| `cl-BCGS-PIP ∘ CholQR-GMRES` | 1.26e+00 | 37.64 | 14 | 139 | 139 | 417 | 153 |

**Table 10** Results for `lapl_2d` example

| Configuration | Time (s) | % Accel. | Cycle Ct. | Iter. Ct. | $A$ Ct. | $V$ Ct. | Sync Ct. |
|---|---|---|---|---|---|---|---|
| `gl-BCGSI+LS ∘ gl-FOM` | 5.34e+01 | 0.00 | 47 | 1162 | 1209 | 4695 | 1256 |
| `gl-BMGS-ICWY ∘ gl-FOM` | 4.60e+01 | 13.92 | 47 | 1162 | 1209 | 3533 | 1256 |
| `gl-BMGS-CWY ∘ gl-FOM` | 4.57e+01 | 14.41 | 47 | 1162 | 1209 | 3533 | 1256 |
| `cl-BMGS-SVL ∘ MGS-SVL-FOM` | 4.39e+01 | 17.77 | 45 | 1121 | 1121 | 3363 | 34890 |
| `cl-BMGS-LTS ∘ MGS-LTS-FOM` | 4.29e+01 | 19.63 | 45 | 1121 | 1121 | 3363 | 34890 |
| `gl-BCGS-PIP ∘ gl-FOM` | 4.07e+01 | 23.70 | 47 | 1162 | 1162 | 3486 | 1209 |
| `cl-BCGSI+LS ∘ CholQR-FOM` | 4.06e+01 | 23.91 | 45 | 1121 | 1166 | 4529 | 1211 |
| `cl-BMGS ∘ CholQR-FOM` | 3.97e+01 | 25.60 | 45 | 1121 | 1121 | 2242 | 15697 |
| `gl-BMGS-SVL ∘ gl-FOM` | 3.94e+01 | 26.21 | 47 | 1162 | 1162 | 3486 | 3533 |
| `gl-BMGS ∘ gl-FOM` | 3.89e+01 | 27.15 | 47 | 1162 | 1162 | 2324 | 16237 |
| `gl-BMGS-LTS ∘ gl-FOM` | 3.88e+01 | 27.24 | 47 | 1162 | 1162 | 3486 | 3533 |
| `cl-BMGS-CWY ∘ CholQR-FOM` | 3.72e+01 | 30.25 | 45 | 1121 | 1166 | 3408 | 1211 |
| `cl-BMGS-ICWY ∘ CholQR-FOM` | 3.66e+01 | 31.44 | 45 | 1121 | 1166 | 3408 | 1211 |
| `cl-BCGS-PIP ∘ CholQR-FOM` | 3.41e+01 | 36.12 | 181 | 2162 | 2162 | 6486 | 2343 |

**Acknowledgements** The author is indebted to Stéphane Gaudreault, Teodor Nikolov, and Erin Carson for stimulating discussions that inspired this work. The author is also grateful to Jens Saak and Martin Köhler for answering questions about the Mechthild cluster and multithreading in MATLAB and to two anonymous reviewers for their constructive feedback.

**Author contribution** K. Lund is the sole author of the manuscript and associated code.

**Funding** Open Access funding enabled and organized by Projekt DEAL. K. Lund is a contracted employee of Max Planck Institute for Dynamics of Complex Technical Systems and did not receive any additional funding to support this project.

**Availability of supporting data** All code and scripts to reproduce plots can be found at https://gitlab.mpi-magdeburg.mpg.de/lund/low-sync-block-arnoldi.

## Declarations

**Ethics approval and consent to participate** The author certifies that this manuscript has been submitted to only one journal at this time, that the work is original, and that the results are not fabricated or skewed. The work is entirely the author's own, and to the best of the author's ability, the work is complete in its own right and without error or misappropriation.

**Consent for publication** As the sole author, K. Lund provides consent for publication.

**Human and animal ethics** Not applicable

**Competing interests** The author declares no competing interests.



## References

1. Barlow, J.L.: Block modified Gram-Schmidt algorithms and their analysis. SIAM J. Matrix Anal. Appl **40**(4), 1257–1290 (2019). https://doi.org/10.1137/18M1197400
2. Świrydowicz, K., Langou, J., Ananthan, S., Yang, U., Thomas, S.: Low synchronization Gram-Schmidt and generalized minimum residual algorithms. Numer. Lin. Alg. Appl., 28(2), https://doi.org/10.1002/nla.2343 (2020)
3. Yamazaki, I., Thomas, S., Hoemmen, M., Boman, E.G., Świrydowicz, K., Eilliot, J.J.: Low-synchronization orthogonalization schemes for s-step and pipelined Krylov solvers in Trilinos. In: Proceedings of the 2020 SIAM conference on parallel processing for scientific computing (PP), pp. 118–128, https://doi.org/10.1137/1.9781611976137.11 (2020)
4. Thomas, S., Carson, E., Rozložník, M., Carr, A., Świrydowicz, K.: Iterated-gauss-seidel GMRES. arXiv:2205.07805v2 (2022)
5. Bielich, D., Langou, J., Thomas, S., Świrydowicz, K., Yamazaki, I., Boman, E.G.: Low-synch gram–schmidt with delayed reorthogonalization for krylov solvers. Parallel Comput. **112**, 102940 (2022). https://doi.org/10.1016/j.parco.2022.102940
6. Carson, E., Lund, K., RozloCzník, M.: The stability of block variants of classical Gram-Schmidt. SIAM J. Matrix Anal. Appl. **42**(3), 1365–1380 (2021). https://doi.org/10.1137/21M1394424
7. Carson, E., Lund, K., Rozložník, M., Thomas, S.: Block Gram-Schmidt algorithms and their stability properties. Linear Algebra Appl. **638**(20), 150–195 (2022). https://doi.org/10.1016/j.laa.2021.12.017
8. Saad, Y.: Iterative methods for sparse linear systems, 2nd edn., p. 528. SIAM. https://doi.org/10.1137/1.9780898718003 (2003)
9. Güttel, S.: Rational Krylov approximation of matrix functions: numerical methods and optimal pole selection. GAMM-Mitteilungen **36**(1), 8–31 (2013). https://doi.org/10.1002/gamm.201310002

10. Simoncini, V.: Analysis of the rational Krylov subspace projection method for large-scale algebraic Riccati equations. SIAM J. Matrix Anal. Appl. **37**(4), 1655–1674 (2016). https://doi.org/10.1137/16M1059382
11. Carson, E.: Communication-Avoiding Krylov Subspace Methods in Theory And Practice. Ph.D. Thesis, Department of Computer Science. University of California, Berkeley (2015). http://escholarship.org/uc/item/6r91c407
12. Hoemmen, M.: Communication-avoiding Krylov subspace methods. Ph.D. Thesis, department of computer science university of california at berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-37.pdf (2010)
13. Grigori, L., Moufawad, S., Nataf, F.: Enlarged Krylov subspace conjugate gradient methods for reducing communicaiton. SIAM J. Matrix Anal. Appl. **37**(2), 744–773 (2016). https://doi.org/10.1137/140989492
14. Balabanov, O., Grigori, L.: Randomized block Gram-Schmidt process for solution of linear systems and eigenvalue problems. arXiv:2111.14641 (2021)
15. Higham, N.J.: Accuracy and stability of numerical algorithms, 2nd edn. Appl. Math., p. 663. SIAM Publications, https://doi.org/10.1137/1.9780898718027 (2002)
16. Huckle, T., Neckel, T.: Bits and Bugs: a scientific and historical review of software failures in computational science. Softw. Environ. Tools, vol. 29. SIAM Publications, https://doi.org/10.1137/1.9781611975567 (2019)
17. Giraud, L., Langou, J., Rozložník, M., Van Den Eshof, J.: Rounding error analysis of the classical Gram-Schmidt orthogonalization process. Numer. Math. **101**, 87–100 (2005). https://doi.org/10.1007/005-0615-4
18. Smoktunowicz, A., Barlow, J.L., Langou, J.: A note on the error analysis of classical Gram-Schmidt. Numer. Math. **105**(2), 299–313 (2006). https://doi.org/10.1007/s00211-006-0042-1
19. Carson, E.: The adaptive s-step conjugate gradient method. SIAM J. Matrix Anal. Appl. **39**(3), 1318–1338 (2018). https://doi.org/10.1137/16M1107942
20. Carson, E.C.: An adaptive s-step conjugate gradient algorithm with dynamic basis updating. Appl. Math. **65**, 123–151 (2020). https://doi.org/10.21136/AM.2020.0136–19
21. Frommer, A., Lund, K., Szyld, D.B.: Block Krylov subspace methods for functions of matrices. Electron. Trans. Numer. Anal. **47**, 100–126 (2017)
22. Frommer, A., Lund, K., Szyld, D.B.: Block Krylov subspace methods for functions of matrices II: modified block FOM. SIAM J. Matrix Anal. Appl. **41**(2), 804–837 (2020). https://doi.org/10.1137/1255847
23. Ballard, G., Carson, E., Demmel, J.W., Hoemmen, M., Knight, N., Schwartz, O.: Communication lower bounds and optimal algorithms for numerical linear algebra. Acta Numer. **23**(2014), 1–155 (2014). https://doi.org/10.1017/S0962492914000038
24. Anzt, H., Boman, E.G., Falgout, R., Ghysels, P., Heroux, M., Li, X., Curfman McInnes, L., Mills, R.T., Rajamanickam, S., Rupp, K., Smith, B., Yamazaki, I., Yang, U.M.: Preparing sparse solvers for exascale computing. Philos. Trans. Royal Soc. A **378**(2166), 20190053 (2020). https://doi.org/10.1098/rsta.2019.0053
25. Baker, A.H., Dennis, J.M., Jessup, E.R.: On improving linear solver performance: a block variant of GMRES. SIAM J. Sci. Comput. **27**(5), 1608–1626 (2006). https://doi.org/10.1137/040608088
26. Birk, S.: Deflated Shifted block Krylov subspace methods for hermitian positive definite matrices. Ph.d. Thesis, Fakultät für Mathematik und Naturwissenschaften, Bergische Universität Wuppertal. http://elpub.bib.uni-wuppertal.de/servlets/DocumentServlet?id=4880 (2015)
27. Dreier, N.-A.: Hardware-oriented Krylov methods for high-performance computing. Ph.D. thesis, Fachbereich Mathematik und Informatik der Mathematisch-Naturwissenschaftlichen Fakultät der Westfälische Wilhelms-Universität Münster. https://www.proquest.com/docview/2607316034/abstract/A334B3B058D24AF2PQ/1 (2020)
28. Dreier, N.-A., Engwer, C.: Strategies for the vectorized block conjugate gradients method. In: Vermolen, F.J., Vuik, C. (eds.) Numerical mathematics and advanced applications ENUMATH 2019. Lecture notes in computational science and engineering, vol. 139, pp. 381–388. Springer, https://doi.org/10.1007/978-3-030-55874-1_37 (2020)
29. Yamamoto, Y., Nakatsukasa, Y., Yanagisawa, Y., Fukaya, T.: Roundoff error analysis of the Cholesky QR2 algorithm. Electron. Trans. Numer. Anal. **44**, 306–326 (2015)

30. Demmel, J., Grigori, L., Hoemmen, M., Langou, J.: Communication-optimal parallel and sequential QR and LU factorizations. SIAM J. Sci. Comput. **34**(1), 206–239 (2012). https://doi.org/10.1137/080731992
31. Mori, D., Yamamoto, Y., Zhang, S.L.: Backward error analysis of the AllReduce algorithm for householder QR decomposition. Jpn. J. Ind. Appl. Math. **29**(1), 111–130 (2012). https://doi.org/10.1007/s13160-011-0053-x
32. Simoncini, V.: Ritz and Pseudo-Ritz values using matrix polynomials. Linear Algebra Appl. **241-243**, 787–801 (1996). https://doi.org/10.1016/0024-3795(95)00682-6
33. Simoncini, V., Gallopoulos, E.: Convergence properties of block GMRES and matrix polynomials. Linear Algebra Appl. **247**, 97–119 (1996). https://doi.org/10.1016/0024-3795(95)00093-3
34. Simoncini, V., Gallopoulos, E.: A hybrid block GMRES method for nonsymmetric systems with multiple right-hand sides. J. Comput. Appl. Math. **66**, 457–469 (1996). https://doi.org/10.1016/0377-0427(95)00198-0
35. Gutknecht, M.H.: Block Krylov space methods for linear systems with multiple right-hand sides: an introduction. In: Siddiqi, A.H., Duff, I.S., Christensen, O. (eds.) Mod. math. model. methods algorithms real world syst, pp. 420-447. Anamaya New Delhi (2007)
36. Gutknecht, M.H., Schmelzer, T.: Updating the QR decomposition of block tridiagonal and block Hessenberg matrices. Appl. Numer. Math. **58**(6), 871–883 (2008). https://doi.org/10.1016/j.apnum.2007.04.010
37. Gutknecht, M.H., Schmelzer, T.: The block grade of a block Krylov space. Linear Algebra Appl. **430**, 174–185 (2009). https://doi.org/10.1016/j.laa.2008.07.008
38. Schreiber, R., Van Loan, C.: A storage-efficient $WY$ representation for products of householder transformations. SIAM J. Sci. Statist. Comput. **10**(1), 53–57 (1989). https://doi.org/10.1137/0910005
39. Balabanov, O., Grigori, L.: Randomized Gram–Schmidt process with application to GMRES. SIAM J. Sci. Comput. **44**(3), 1450–1474 (2022). https://doi.org/10.1137/20M138870X
40. Davis, T.A., Hu, Y.: The University of Florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1–25 (2011). https://doi.org/10.1145/2049662.2049663
41. Boman, E.G., Higgins, A.J., Szyld, D.B.: Optimal size of the block in block GMRES on GPUs: computational model and experiments. e-print 22-04-30, department of mathematics, Temple University, Philadelphia, PA. https://www.math.temple.edu/szyld/reports/BGMRES_GPU_rev.report.pdf (2022)
42. Parks, M.L., Soodhalter, K.M., Szyld, D.B.: A block recycled GMRES method with investigations into aspects of solver performance. arXiv:1604.01713v1 (2016)