SAND2019-15001C

Low-synchronization orthogonalization schemes for s-step and pipelined Krylov solvers in Trilinos

Ichitaro Yamazaki* Stephen Thomas† Mark Hoemmen* Erik G. Boman*
Katarzyna Świrydowicz† James J. Elliott*

Abstract

We investigate two single-reduce orthogonalization schemes for both s-step and pipelined GMRES. The first is based on classical Gram Schmidt with reorthogonalization (CGS2), and the second on modified Gram Schmidt (MGS). Standard iterated CGS2 requires three global reductions. In standard MGS, the number of global reductions is proportional to the number of vectors against which we are orthogonalizing. In both cases, we can reduce this to a single global reduction, including reorthogonalization for accuracy.

Our implementation is based on Trilinos software components, and therefore, is portable to different machine architectures with a single code base. We first demonstrate solver performance on the Intel Haswell nodes of the NERSC Cori Supercomputer. For these experiments, we integrated our solvers into `Nalu-wind`, a computational fluid dynamics application. At each time step, Nalu uses GMRES with a smoothed aggregation algebraic multigrid (SA-AMG) preconditioner to solve a pressure Poisson linear system. In this experiment, s-step GMRES reduced Nalu's total GMRES solve time by a factor of 1.4×.

We then benchmarked the single-reduce orthogonalization schemes on the ORNL Summit supercomputer. In these experiments, our low-synchronization CGS2 and MGS improved the s-step GMRES performance by a factor of 2.4× and 10.1× on 384 NVIDIA V100 GPUs, respectively, while on the IBM Power9 CPUs, they improved the stability of the pipelined GMRES without increasing the iteration time.

1 Introduction

Scientific and engineering simulations can spend much of their time solving large sparse linear systems of equations. Krylov subspace projection methods [17] are popular iterations for solving such systems. This is due to their lower storage costs, compared with direct factorization, and their ability to work with a variety of application-specific preconditioners for reducing iteration count.

On modern computers, communication often dominates simulation's run time. "Communication" includes both data movement through the local memory hierarchy, and data movement or synchronization between parallel processes or threads. Krylov solvers especially have communication-bound performance, because their computational kernels (e.g., dot products and matrix-vector multiplies) have little local data reuse and require global communication and/or synchronization among the processes.

To improve Krylov solvers' performance, communication-avoiding (CA) variants have been proposed [13, and the references within]. CA methods are based on the s-step Krylov algorithm that generates a block of s orthonormal basis vectors for the Krylov projection subspace at a time. Hence, they have the potential to reduce the communication costs by a factor of s. Another effort to improve solver performance led to the development of pipelined variants of Krylov solvers [7, 8]. These algorithms take inspiration from the "software pipelining" technique that compilers use to optimize code. They redesign the iteration algorithm to use nonblocking global reductions, and overlap the reductions with local computation or other communication.

As a part of the Exascale Computing Project (ECP) funded by the U.S. Department of Energy, we implemented CA and pipelined Krylov solvers in the Trilinos software framework [11]. In this paper, we summarize our efforts to develop portable high-performance versions of these solvers for upcoming exascale computer architectures. Here, we focus on variants of the Generalized Minimal Residual (GMRES) method [18] for solving nonsymmetric linear systems, although we have also implemented the pipelined variant [8] of the Conjugate Gradient method [12] exist for symmetric positive definite systems.

Here are the key contributions of our paper:

*Sandia National Laboratories, New Mexico, USA
†National Renewable Energy Laboratory, Colorado, USA

1. We extend the low-synchronization orthogonalization schemes [20], which orthogonalize one vector at time, for s-step GMRES to orthogonalize s vectors at once. These are based on either classical Gram Schmidt with reorthogonalization (CGS2), or modified Gram Schmidt (MGS), and require only one global reduction to orthogonalize a set of s basis vectors against the previous basis vectors and among themselves. In comparison, standard CGS2 requires three global reductions, while in standard MGS, the number of global reductions would be proportional to the number of vectors to be orthogonalized. Hence, with MGS, the total number of reductions grows quadratically with the number of GMRES iterations.

2. We deployed these new orthogonalizations in the pipelined GMRES and s-step GMRES implementations in the Trilinos software library [11]. This makes them available to many different applications. Our implementations use a combination of MPI and shared-memory parallelism. The latter uses the Kokkos [5] parallel programming model for portable performance on many different node architectures.

3. We measured performance of the new solvers on the distributed Haswell CPU nodes of the Cori supercomputer at NERSC, as integrated into `Nalu-wind` [4], a computational fluid dynamics application. Our s-step solver reduced Nalu's total solution time by a factor of 1.4×.

4. We demonstrate performance portability of the solvers with the NVIDIA Volta GPU nodes on the Summit supercomputer at ORNL. The new implementations of CGS2 and MGS improve GMRES performance by a factor of 2.4× and 10.1× on 384 GPUs on Summit, respectively.

5. We measured performance of pipelined GMRES on the IBM Power 9 CPUs on the Summit supercomputer. Single-reduce Gram Schmidt orthogonalization can improve the solver's numerical stability without increasing the iteration time.

2 Generalized Minimal Residual Method

The Generalized Minimal Residual (GMRES) [18] is a Krylov subspace projection method for solving a nonsymmetric linear system $A\mathbf{x} = \mathbf{b}$. The solution computed by GMRES minimizes the residual norm over the generated projection subspace. The j–th right-preconditioned GMRES iteration first generates a new basis vector $\mathbf{v}_{j+1}$ for the projection subspace by applying the preconditioner M^{-1} and the sparse-matrix vector product (`SpMV`) with the matrix A to the previously orthonormalized basis vector $\mathbf{q}_j$ (i.e., $\mathbf{v}_{j+1} := AM^{-1}\mathbf{q}_j$). The new vector $\mathbf{q}_{j+1}$ is then generated by orthonormalizing $\mathbf{v}_{j+1}$ against all the previous basis vectors $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_j$. This orthogonalization requires global reductions across all the MPI processes, and local computations are based on either BLAS-2 or BLAS-1 primitives. The resulting global synchronizations and lack of data reuse makes it difficult to achieve high performance from the orthogonalization kernels.

2.1 Communication-avoiding GMRES. In order to improve GMRES performance, the s-step method first generates a set of s new Krylov vectors by applying the preconditioner and `SpMV` without orthogonalization, $\mathbf{v}_{j+k} := AM^{-1}\mathbf{v}_{j+k-1}$ for $k = 1, 2, \ldots, s$, using the starting vector $\mathbf{v}_j := \mathbf{q}_j$. The new basis vectors are then orthonormalized against the previous basis vectors all at once, giving the potential to reduce the communication cost by a factor of s.

To orthogonalize the new set of basis vectors against the previous vectors, the two traditional block orthogonalization schemes are modified Gram-Schmidt (MGS) and classical Gram-Schmidt (CGS). In MGS, the new basis vectors $V_{j:j+s}$ are orthogonalized against the previous basis vectors one at a time, as follows:

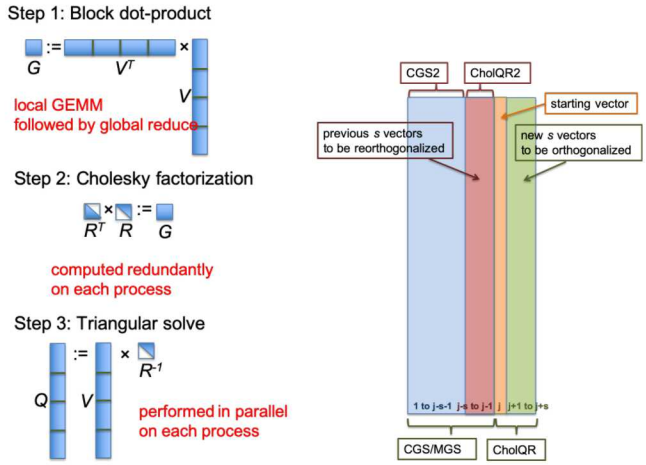
$$\begin{aligned} V_{j:j+s} &:= V_{j:j+s} - \mathbf{q}_k(\mathbf{q}_k^T V_{j:j+s}) \\ &\quad \text{for } k = 1, 2, \ldots, j-1 \\ &= \prod_{k=1}^{j-1} (I - \mathbf{q}_k \mathbf{q}_k^T) V_{j:j+s}, \end{aligned}$$

while in CGS, they are orthogonalized against all the previous vectors at once, as follows:

$$\begin{aligned} V_{j:j+s} &:= V_{j:j+s} - Q_{1:j-1}(Q_{1:j-1}^T V_{j:j+s}) \\ &= (I - Q_{1:j-1} Q_{1:j-1}^T) V_{j:j+s}. \end{aligned} \tag{2.1}$$

Therefore, MGS performs $(j-1)$ dot products, while CGS performs only one dot product. Each dot product requires a global reduction. In addition, MGS performs most of its local computation using BLAS-2 matrix-vector operations, whereas CGS is based on BLAS-3 matrix-matrix primitives. Therefore, in comparison to MGS, CGS often achieves higher performance. However, when computing in finite-precision floating-point arithmetic, CGS often results in a faster $\mathcal{O}(\varepsilon\kappa(V)^2)$ loss of orthogonality, compared with $\mathcal{O}(\varepsilon\kappa(V))$ of MGS. This may call for reorthogonalization in order to maintain the basis vectors' orthogonality and to avoid the early stagnation of the solution convergence.

After one pass of CGS or MGS, the new basis vectors need to be orthogonalized among themselves. In



(a) CholQR algorithm.

(b) Vectors being orthogonalized.

Figure 1: Single-reduce orthogonalization.

this paper, the focus is on Cholesky QR (`CholQR`) [22]. As illustrated in Figure 1(a), `CholQR` requires only one global reduce, and performs most of the local computations using BLAS-3 primitives. However, to maintain orthogonality, the vectors may need to be reorthogonalized by applying `CholQR` again.

In this implementation, the first column $\mathbf{v}_j$ of the block columns $V_{j:j+s}$ being orthonormalized is the last column $\mathbf{q}_j$ from the previous orthogonalization step. Thus, $\mathbf{q}_j$ has already been orthonormalized against all the previous vectors once. However, this starting vector is included in the orthogonalization process, and reorthonormalized against all the previous vectors $Q_{1:j-1}$. This improves the numerical stability of the solver, and is critical for implementing our single-reduce orthogonalization schemes in Section 3.

It is possible to apply `SpMV` in a CA fashion [14]. To reduce communication latency, each process communicates all the vector elements with neighboring processes, which are needed to apply an `SpMV` s times without further communication. Unfortunately, it is still a challenge to apply the preconditioner in a CA fashion [23, 10]. Researchers have only figured out how to do so for a few preconditioners. In particular, the large class of effective and scalable preconditioners that the terms "multilevel" or "multigrid" encompass do not fit naturally into the CA framework. In order to support a wide range of applications, which use different preconditioners, our current implementation does not use the CA variant of `SpMV` and communicates for each application of the `SpMV` or a preconditioner (both of which are used as user-provided black-box subroutines). In many application, the preconditioner scales better than the orthogonalization scheme, and the solver performance may be improved merely by avoiding communication during the orthogonalization process. Previous work shows that focusing only on orthogonalization can improve performance significantly [14].

2.2 Pipelined GMRES. Rather than avoiding global reductions, pipelined GMRES [7] was designed to hide communication by pipelining the Krylov iterations and overlapping the global reduction with one or more subsequent iterations. In this paper, the focus is on a pipeline depth of one. With the pipeline depth of one, `SpMV` is applied to the previous basis vector $\mathbf{v}_j$, which has not yet been orthogonalized, such that the global reduction required for the orthogonalization can be overlapped with the `SpMV`.

Because `SpMV` is applied to the vector before orthogonalization, pipelined GMRES must update the vector resulting from the `SpMV`,

$$\begin{aligned} \mathbf{v}_{j+1} &:= A(\mathbf{v}_j - Q_{1:j-1}\mathbf{r}_{1:j-1,j}) \\ &= \mathbf{w}_j - W_{1:j-1}\mathbf{r}_{1:j-1,j} \end{aligned}$$

before orthogonalizing $\mathbf{v}_{j+1}$. This additional update requires about the half of the computation cost of CGS1, and an additional set of basis vectors $W_{1:j} = AQ_{1:j}$ must be stored. However, with this additional storage and the local BLAS-2 operation, pipelined GMRES attempts to hide the global reductions. With a deeper pipeline depth ℓ, the global reduction may be overlapped with the `SpMV` and orthogonalization in the subsequent ℓ iterations.

Merely overlapping global communication may achieve a maximum possible speedup of two. When the iterations can be effectively pipelined, a greater speedup may be possible. Pipelined Krylov methods have also been shown to mitigate hardware performance variations ("noise" or "jitter") [15, 6].

3 Single-reduce Block Gram Schmidt

To implement the single-reduce orthogonalization schemes for s-step GMRES, we first merged block CGS' global reduction with CholQR's global reduction, as shown in Figure 2.

Then, in order to maintain the orthogonality of the new basis vectors against the previous vectors, reorthogonalization is performed by applying CGS twice (CGS2). It is possible to apply this reorthogonalization without additional global reduce by taking advantage of

1. *// compute dot-product*
   $R_{1:j+s,j:j+s} := [Q_{1:j-1}, V_{j:j+s}]^T V_{j:j+s}$
2. *// CGS to orthogonalize against previous vectors*
   $V_{j:j+s} := V_{j:j+s} - Q_{1:j-1} R_{1:j-1,j:j+s}$
3. *// update the Gram matrix*
   $R_{j:j+s,j:j+s} := R_{j:j+s,j:j+s} - R^T_{1:j-1,j:j+s} R_{1:j-1,j:j+s}$
4. *// compute Cholesky factorization*
   $R_{j:j+s,j:j+s} := \texttt{chol}(R_{j:j+s,j:j+s})$
5. *// triangular-solve to generate orthonormal vectors*
   $Q_{j:j+s} := V_{j:j+s} R^{-1}_{j:j+s,j:j+s}$

Figure 2: Single-reduce CGS+CholQR algorithm to orthonormalize the $s+1$ vectors $V_{j:j+s}$ against $Q_{1:j-1}$.

the following equalities:

$$\begin{aligned} V_{j:j+s} &= (I - Q_{1:j-1}Q^T_{1:j-1})(I - Q_{1:j-1}Q^T_{1:j-1})V_{j:j+s} \\ &= (I - Q_{1:j-1}(2I - T)Q^T_{1:j-1})V_{j:j+s} \\ &= V_{j:j+s} - Q_{1:j-1}(2I - T)R_{1:j-1,j:j+s} \\ &= V_{j:j+s} - Q_{1:j-1}\widehat{R}_{1:j-1,j:j+s}, \end{aligned} \tag{3.2}$$

where $R_{1:j-1,j:j+s} = Q^T_{1:j-1}V_{j:j+s}$ and $T = Q^T_{1:j-1}Q_{1:j-1}$. Hence, for the single-reduce CGS2, the j–th through the $(j+s)$–th columns $R_{1:j-1,j:j+s}$ of the upper-triangular matrix $R_{1:j+s,j:j+s}$ is replaced with $\widehat{R}_{1:j-1,j:j+s} = (2I - T)R_{1:j-1,j:j+s}$. At every s–th step, the last s columns of T and R are computed with a single global reduce, i.e., $[T_{1:j-1,j-s:j-1}, R_{1:j-1,j:j+s}] := Q^T_{1:j-1}[Q_{j-s:j-1}, V_{j:j+s}]$.

An alternative approach to improve orthogonality is to replace CGS with MGS. It is possible to apply MGS with a single reduction, based on the following equality:

$$\begin{aligned} V_{j:j+s} &= \prod_{k=1}^{j-1}(I - \mathbf{q}_k\mathbf{q}_k^T)V_{j:j+s} \\ &= V_{j:j+s} - Q_{1:j-1}(I - L)^{-1}R_{1:j-1,j:j+s} \\ &= V_{j:j+s} - Q_{1:j-1}\widetilde{R}_{1:j-1,j:j+s}, \end{aligned} \tag{3.3}$$

where L is the strictly lower-triangular part of T. Hence, for single-reduce MGS, $R_{1:j-1,j:j+s}$ is replaced with $\widetilde{R}_{1:j-1,j:j+s} := (I - L)^{-1}R_{1:j-1,j:j+s}$. In exact arithmetic, $T = I$ and $L = 0$, and hence, $\widehat{R}_{1:j-1,j:j+s}$ in (3.2) and $\widetilde{R}_{1:j-1,j:j+s}$ in (3.3) both equal $R_{1:j-1,j:j+s}$.

Due to the rounding error, we may also loose the orthogonality among the new s basis vectors. Since the single-reduce CGS2 and MGS compute $T_{1:j-1,j-s:j-1} := Q^T_{1:j-1}Q_{j-s:j-1}$, CholQR can use the last s rows of the resulting matrix, $T_{j-s:j-1,j-s:j-1}$, for reorthogonalizing $Q_{j-s:j-1}$ among themselves. Hence, we combine this reorthogonalization with CGS2 or MGS to orthogonalize the next s basis vectors $V_{j+s,j+2s}$ against $Q_{1:j+s-1}$ without needing an additional global reduction. Figure 1(b) illustrates the vectors being orthogonalized.

The last vector $\mathbf{q}_{j+s}$ was not reorthogonalizaed using CholQR. Instead, the next s-step orthogonalization process reorthogonalizes $\mathbf{q}_{j+s}$, which is the starting vector for generating the next s basis vectors $V_{j+s+1:j+2s}$ (as discussed in Section 2). If the starting vector $\mathbf{q}_{j+s}$ is reorthogonalized with CholQR, then the vectors $V_{j+s+1:j+2s}$, which are generated by applying an `SpMV` to $\mathbf{q}_{j+s}$, must be updated accordingly. This requires additional computation and storage (similar to what pipelined GMRES does, as explained in Section 2.2).

Furthermore, at Step 3 in Figure 2, single-reduce CGS+CholQR updates the Gram matrix, assuming the orthogonality of $Q_{1:j-1}$. Using the auxiliary matrix T, it is possible to update the Gram matrix without this assumption:

$$\begin{aligned} &V^T_{j:j+s}V_{j:j+s} \\ &\quad= (V_{j:j+s} - Q_{1:j-1}R_{1:j-1,j:j+s})^T(V_{j:j+s} - Q_{1:j-1}R_{1:j-1,j:j+s}) \\ &\quad= V^T_{j:j+s}V_{j:j+s} + R^T_{1:j-1,j:j+s}(Q^T_{1:j-1}Q_{1:j-1})R_{1:j-1,j:j+s} \\ &\qquad -R^T_{1:j-1,j:j+s}Q^T_{1:j-1}V_{j:j+s} - V^T_{j:j+s}Q_{1:j-1}R_{1:j-1,j:j+s} \\ &\quad= R_{j:j+s,j:j+s} + R^T_{1:j-1,j:j+s}(T_{1:j-1,1:j-1})R_{1:j-1,j:j+s} \\ &\qquad -R^T_{1:j-1,j:j+s}\bar{R}_{1:j-1,j:j+s} - \bar{R}^T_{1:j-1,j:j+s}R_{1:j-1,j:j+s}, \end{aligned}$$

where $\bar{R}_{1:j-1,j:j+s} = \widehat{R}_{1:j-1,j:j+s}$ or $\bar{R}_{1:j-1,j:j+s} = \widetilde{R}_{1:j-1,j:j+s}$ using single-reduce CGS2 or MGS from (3.2) or (3.3), respectively. In the previous scheme in Figure 2, it was assumed $T = I$ and hence, $\bar{R}_{1:j-1,j:j+s} = R_{1:j-1,j:j+s}$.

Figure 3 shows the resulting single-reduce orthogonalization schemes, and Figure 4 lists the cost of each. Compared with the standard orthogonalizations, their single-reduce counterparts require about the same computational cost, but perform fewer global reductions.

Previously, single-reduce orthogonalization schemes were used to orthogonalize one basis vector at a time for the standard GMRES iteration, by merging two global reductions [21, 7]. The low-synchronization CGS2 and MGS algorithms, with $\mathcal{O}(\varepsilon)$ and $\mathcal{O}(\varepsilon\kappa(V)$ orthogonality errors, respectively, were presented in [16, 20]. We extend these algorithms to orthogonalize a set of s basis vectors in s-step GMRES.

These single-reduce orthogonalization techniques can be also integrated into pipelined GMRES. While standard CGS1 computes the non-blocking dot products $Q^T_{1:j-1}\mathbf{v}_j$ for generating the coefficients $\mathbf{r}_{1:j-1,j}$ in (2.1), single-reduce CGS2 and MGS compute $Q^T_{1:j-1}[\mathbf{q}_{j-1}, \mathbf{v}_j]$. Though this does not increase the communication latency, it doubles the computational cost and communication volume for the non-blocking operation. However, the main benefit is to improve numerical stability of the pipelined solver.

// Global reduce
$[L_{1:j+s,j-s:j-1}, R_{1:j+s,j:j+s}] := [Q_{1:j-1}, V_{j:j+s}]^T [Q_{j-s:j-1}, V_{j:j+s}]$

// Lagged re-normalization by CholQR
// Cholesky factorization
$\widehat{R}_{j-s:j-1,j-s:j-1} := \text{chol}(L_{j-s:j-1,j-s:j-1})$
// re-normalize the previous vectors $Q_{j-s:j-1}$
$Q_{j-s:j-1} := Q_{j-s:j-1} \widehat{R}^{-1}_{j-s:j-1,j-s:j-1}$
// update the previous normalization coefficients
$R_{j-s:j-1,j-s:j-1} := \widehat{R}_{j-s:j-1,j-s:j-1} R_{j-s:j-1,j-s:j-1}$

// update the coefficients $Q^T_{1:j-1} Q_{j-s:j-1}$
$L_{1:j-1,j-s:j-1} := L_{1:j-1,j-s:j-1} \widehat{R}^{-1}_{j-s:j-1,j-s:j-1}$
// update the coefficients $Q^T_{j-s:j-1} Q_{j-s,j-1}$
$L_{j-s:j-1,j-s:j-1} := \widehat{R}^{-T}_{j-s:j-1,j-s:j-1} L_{j-s:j-1,j-s:j-1}$

// convergence check for GMRES

// Orthogonalize new vectors against previous vectors
$\widetilde{R}_{1:j-1,j:j+s} := R_{1:j-1,j:j+s}$ // save original coefficient
// update coefficient for block MGS
if MGS **then**
  $R_{1:j-1,j:j+s} := (I - L_{1:j-1,1:j-1})^{-1} R_{1:j-1,j:j+s}$
end if
// form $T := L + L^T - I$
$T_{1:j-1,k:j} := L_{1:j-1,k:j}$
$T_{k:j,1:j-1} := L^T_{1:j-1,k:j}$
// update coefficient for block CGS2
if CGS2 **then**
  $R_{1:j-1,j:j+s} := R_{1:j-1,j:j+s}$
    $-T_{1:j-1,1:j-1} \widetilde{R}_{1:j-1,j+1:j+s}$
end if
// Orthogonalize $V_{j:j+s}$ against $Q_{1:j-1}$
$V_{j:j+s} := V_{j:j+s} - Q_{1:j-1} R_{1:j-1,j:j+s}$

// Normalization by single-reduce CholQR
// update coefficient
$R_{j:j+s,j:j+s} := R_{j:j+s,j:j+s} + \widetilde{R}^T_{1:j-1,j:j+s} T_{1:j-1,1:j-1} \widetilde{R}_{1:j-1,j:j+s}$
$\quad - \widetilde{R}^T_{1:j-1,j:j+s} R_{1:j-1,j:j+s} - R^T_{1:j-1,j:j+s} \widetilde{R}_{1:j-1,j:j+s}$
// Cholesky factorization
$R_{j:j+s,j:j+s} := \text{chol}(R_{j:j+s,j:j+s})$
// normalize new vectors
$Q_{j:j+s} := V_{j:j+s} R^{-1}_{j:j+s,j:j+s}$

Figure 3: Pseudocode of single-reduce orthogonalization schemes to orthogonalize a set of $\hat{s}$ vectors $V_{\hat{s}}$ against a set of $j+1$ vectors $Q_{0:j}$ and among $V_{\hat{s}}$. It also performs lagged normalization of the previous s vectors. L is stored as T in the actual code.

	# of reduces	flop count
standard CGS + CholQR	2	$(4sj + 2s^2)n$
single-reduce CGS + CholQR	1	$(4sj + 2s^2)n$
standard MGS + CholQR	j	$(4sj + 2s^2)n$
single-reduce MGS + CholQR	1	$(6sj + 2s^2)n$
standard CGS2 + CholQR2	4	$(8sj + 4s^2)n$
single-reduce CGS2 + CholQR2	1	$(6sj + 2s^2)n$

Figure 4: Cost of orthonormalizing $s+1$ vectors $Q_{j:j+s}$ against the previous $j-1$ vectors $Q_{1:j-1}$, where n is the number of rows in the basis vectors.

4 Numerical Results with Single-reduce GS

4.1 Pipelined GMRES. We first give numerical results using standard and pipelined GMRES combined with three different orthogonalization schemes: CGS1, CGS2, and MGS. Figure 5 shows the decreasing relative residual norm $\|A\mathbf{x}_j - \mathbf{b}\|/\|A\mathbf{x}_0 - \mathbf{b}\|$ and increasing orthogonality error $\|T - I\|$ at each iteration when solving the linear system with diagonal coefficient matrix of dimension 100, $A = \text{diag}(0.001, 1, 2, \ldots, 99)$ [19]. All of our experiments were conducted in double precision.

In Figure 5, the pink, blue, and green lines with the $\times$, $+$, or * markers show convergence of GMRES with different orthogonalizations. CGS2- and MGS-based GMRES have the same convergence rates (the blue and green lines overlap) and both reach the minimum achievable residual norm $\epsilon \cdot (\|\mathbf{b}\|_2 + \|A\|_2\|\mathbf{x}\|_2)$ with the machine epsilon ϵ. However, when using CGS1 (pink line with *), GMRES stagnates before reaching the minimal residual norm, because the built-up loss of orthogonality eventually affects the basis quality. Similar to CGS1, the orthogonality error with MGS increases, but at a slower rate $\mathcal{O}(\varepsilon\kappa)$ (compared with $\mathcal{O}(\varepsilon\kappa^2)$ with CGS1), where κ is the condition number of the Arnoldi matrix $[r_0, AV_m]$. In fact, MGS-based GMRES can always obtain the minimal residual norm before complete loss of orthogonality [9]. Finally, CGS2 maintains orthogonality, and GMRES converges without any stagnation.

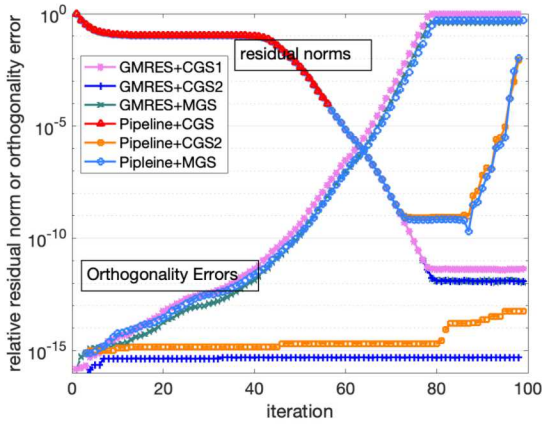


Figure 5: Orthogonality error and relative residual norm using standard or pipelined GMRES with different orthogonalization schemes, for solving the linear system with the diagonal matrix $\text{diag}(0.001, 1, 2, \ldots, 99)$ [19].

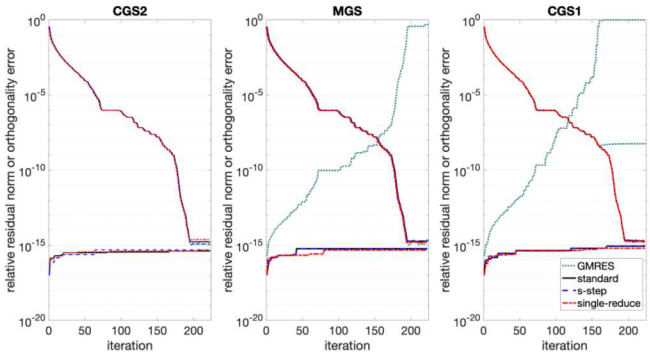
The red, orange, and blue lines with the square, circle, and triangle markers show convergence results for pipelined GMRES with the single-reduce variants of the same three orthogonalization schemes: "CGS1" (red line) based on the previous scheme [7], and "CGS2" (orange line) and "MGS" (blue line) introduced in this paper. Our experiment was designed to stress the numerical stability of GMRES (e.g., generating the projection space of dimension n without restart). Pipelined GMRES, which implicitly computes two sets of the vectors Q and W, faced numerical difficulties, even when the orthogonality among the basis vectors was maintained. Consequently, the pipelined GMRES with CGS1 failed

at the 56-th iteration due to a non-positive normalization factor. Although the new orthogonalization scheme could not achieve the solution accuracy obtained by the standard GMRES, it was able to obtain a higher solution accuracy than that obtained by the previous orthogonalization scheme.

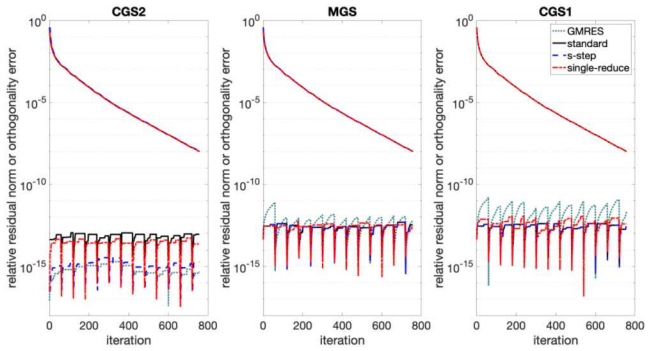
4.2 CA GMRES. Figure 6 displays the relative residual norm and the orthogonality error for s-step GMRES when solving two different problems: one numerically difficult problem with `steam1` in Figure 6(a), and the standard 3D Laplace problem in Figure 6(b)). Figure 6(a) shows the same convergence behavior of the standard GMRES (i.e., green lines), as in Figure 5, where MGS- and CGS2-based GMRES converge, but CGS1-based GMRES stagnates.

Figure 6(a) also demonstrates that when $s = 1$, our implementation of s-step GMRES is more stable than the standard GMRES, because it reorthogonalizes the starting vector every s-th step. In fact, with $s = 1$, the algorithm behaves similarly to GMRES+CGS2.

Figure 6(b) then compares the loss of orthogonality and residual norms of the standard GMRES, with those obtained from the s-step GMRES with $s = 5$, when solving a 3D Laplace problem. For the standard Laplace problem, the single-reduce schemes obtained similar orthogonality errors when compared to the standard scheme.



(a) `steam1` matrix, with $n = 240$ and $s = 1$.



(b) `Laplace 3D` matrix, with $n = 100000$ and $s = 1$.

Figure 6: Orthogonality error and relative residual norm using different orthogonalization schemes, "GMRES" displays results based upon the standard GMRES, while the remainder employ s-step GMRES with different orthogonalization algorithms, e.g., for CGS2, "standard" uses CGS2 for both block QR and TSQR, and "s-step" applies CGS2 for block QR and then CholQR2 for TSQR, while "single-reduce" relies on the single-reduce CGS2+CholQR2.

5 Trilinos Software Framework

Trilinos [11] is a collection of C++ software packages for large-scale, complex, multiphysics engineering and scientific applications. Many of these packages work together to solve large sparse linear systems via Krylov subspace methods and preconditioners. Several applications depend on Trilinos for linear solvers. Furthermore, Trilinos provides and uses software components that make it easier to write high-performance solvers that are portable to different computer architectures with a single code base, including architectures that do not exist yet. Thus, making our new solvers available in Trilinos could have a large impact on several applications on different current and future architectures.

5.1 Tpetra. We used Trilinos' `Tpetra` software package [2] to implement our solvers. Tpetra provides parallel data structures and computational kernels for constructing and solving large sparse linear systems. "Parallel" here includes both distributed memory (via the Message Passing Interface (MPI)) and shared memory. Tpetra uses Kokkos [5] for shared-memory parallelism and memory management. Kokkos presents a single programming model and software interface for many different underlying programming models (e.g., OpenMP and CUDA) and architectures (e.g., CPUs and GPU). It promises "performance portability" – that is, one code base can perform reasonably well on many different node architectures, even if they don't exist yet. Some shared-memory parallel computational kernels need to be tuned specifically for each architecture. For this, Tpetra uses the `kokkos-kernels` package for a portable interface to these kernels.

In order to implement pipelined solvers, we have introduced a new function `idot` to Tpetra to perform non-blocking dot-products. `idot` uses the nonblocking

collectives in version 3 of MPI. Nevertheless, Tpetra must support older versions of MPI as well, and in fact, must build even if MPI is not available. For correctness' sake, `idot` must always be nonblocking. We solved this problem using C++ abstractions as follows.

- `idot` returns `std::shared_ptr<CommRequest>`.
- `CommRequest` is an abstract base class (in C++ terms) representing the pending communication request. Its `wait` method blocks until the pending request completes, and its destructor cancels the request. The latter avoids memory leaks if users let the request fall out of scope.
- Use of `std::shared_ptr` avoids "double cancellation" if users make extra copies of the request.
- If Tpetra was built with MPI 3, `idot` calls `MPI_Iallreduce` to sum the results of the local dot products. The returned `CommRequest` manages the resulting `MPI_Request`. Its `wait` method calls `MPI_Wait`, and its destructor calls `MPI_Cancel`.
- If Tpetra was built without MPI or with a version of MPI older than 3, then the returned `CommRequest` hides a closure (a C++ "lambda") that captures the input and output to the `idot`, and calls `MPI_Allreduce` (if MPI is available). The closure "delays" the blocking all-reduce until the user calls `wait`.
- With all implementations, `idot` can capture the input and output using reference-counting types. This prevents users from accidentally deallocating space for the input and output before the all-reduce completes.

`idot` can handle both a single dot product of two vectors, and multiple dot products of one vector with each of a collection of vectors in turn. The latter operation is useful for pipelined Krylov methods. Tpetra optimizes it by only doing a single local computational kernel launch and a single all-reduce for all the vectors.

5.2 Belos. Trilinos' Belos package implements Krylov subspace methods [3]. All solvers implement the `Belos::SolverManager` interface. Users can get a solver instance by passing the solver name to `Belos::SolverFactory`. They may then set parameters (e.g., stopping criteria and orthogonalization scheme) and the problem (linear operator, preconditioner, and right-hand-side vector). Finally, the user can compute the solution by calling the solve function. Figure 7 illustrates the interface.

```
// create solver
Belos::SolverFactory<SC, MV, OP> factory;
RCP<Belos::SolverManager<SC, MV, OP> > solver;
solver = factory.create (solverName, Teuchos::null);

// set solver parameters
params->set ("Convergence Tolerance",
             commandLineOptions.tol );
params->set ("Maximum Iterations",
             commandLineOptions.maxNumIters);
params->set ("Step Size",
             commandLineOptions.stepSize);
params->set ("Compute Ritz Values",
             commandLineOptions.computeRitzValues);
params->setRightPrec(M);

solver->setParameters (params);

// set linear system to solve
auto problem
       = rcp (new Belos::LinearProblem<SC, MV, OP>
                        (A, rcpFromRef (X), B));
problem->setProblem ();
solver->setProblem (problem);

// solve the linear system
solver->solve ();
```

Figure 7: Illustration of Belos linear solver interface.

Belos uses so-called "traits" classes to define a fixed interface to vector, matrix, and preconditioner operations that the solvers can use. Users can specialize the traits classes for their own vector, matrix, and preconditioner types. Belos provides specializations of the traits classes for Trilinos' native linear algebra classes, including `Tpetra`.

Belos' use of traits classes for polymorphism makes it difficult to expand the set of linear algebra operations that "generic" solvers can use. This is because the traits classes use compile-time polymorphism through template specialization. If we add methods to the generic traits class and use them in a generic Belos solver, then users who have specialized the traits for their own linear algebra classes won't be able to compile Belos with their classes any more. This would break backwards compatibility – an important consideration for production software.

This is a problem for CA and pipelined solvers that require new types of linear algebra operations, such as nonblocking dot products and the matrix powers kernel. However, Belos has an option for developers to implement solvers that are specific to a set of linear algebra types, and make them available through `SolverFactory`, just like Belos' generic solvers. We use this facility to develop our new solvers using native Tpetra functionality, like `idot`. Applications who use `SolverFactory` can, in turn, access our solvers without changing their C++ code, just by using a different solver name in their input decks.

6 Performance with Nalu-wind on Cori

To study the performance of our solvers in a real application, we integrated our solvers into `Nalu-wind`, a computational fluid dynamics application. Nalu can simulate air flow over tens or hundreds of wind turbines, and is one of the ECP applications. At each time step of the simulation, it solves a nonlinear problem by self-consistent iteration over several linear systems. It represents the linear systems with Tpetra sparse matrices and dense vectors, and solves them with Krylov methods from Trilinos' Belos package and preconditioners from Trilinos' Ifpack2 and MueLu packages. We focus in this paper on the "pressure Poisson" linear system, which is usually the most difficult of the linear systems. Nalu normally solves this with Belos' GMRES and an algebraic multigrid preconditioner from the MueLu package.

We deployed our new Krylov solvers by making them available through Belos' solver factory (Section 5.2). This means that we did not need to change any Nalu C++ code. We merely had to change one line, specifying the solver name, in Nalu's "input deck," a text file that users write and that Nalu reads on startup.

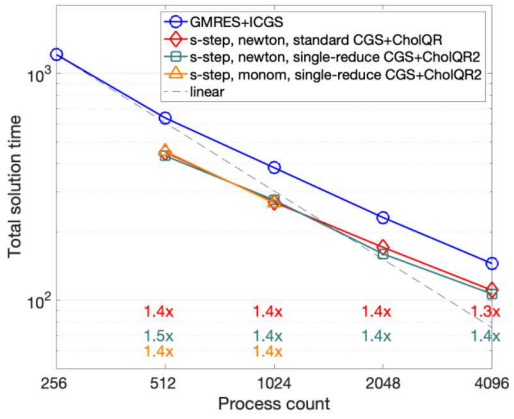


Figure 8: GMRES performance on Cori Intel Haswell nodes for solving the pressure Poisson systems from `Nalu-wind` ($n = 95$M). With the monomial basis, s-step failed on 2048 processes.

Figure 8 shows the total time needed to solve the pressure Poisson systems during a 20 time step simulation on the Intel Haswell nodes of the Cori Supercomputer at NERSC. We complied our code using the Cray compiler wrapper `CC` for the Intel `C++` compiler and linked to the Cray Scientific Libraries package, LibSci. We used the default `Nalu-wind` setup: GMRES with a restart length of 100 and the smoothed aggregation algebraic multigrid (SA-AMG) preconditioner in `MueLu`. (Our solvers can be used with any `SpMV` and preconditioner kernels.) The solution was considered to have converged when the relative residual norm became less than 10^{-5}. With this setup, GMRES needed 62–193 iterations for solving each system.

For s-step GMRES, we used $s = 5$, with a Newton basis [1] for numerical stability. We computed shifts for the Newton basis with Ritz values from five iterations of standard GMRES. By reducing the communication needed for orthogonalization, the s-step method was able to reduce GMRES iteration time (red diamond markers compared with blue circle markers). The single-reduce schemes further reduce orthogonalization time (green square markers), although in this application, the iteration time was not significantly reduced. This is because the s-step method reduced orthogonalization time such that iteration time is now dominated by the multigrid preconditioner.

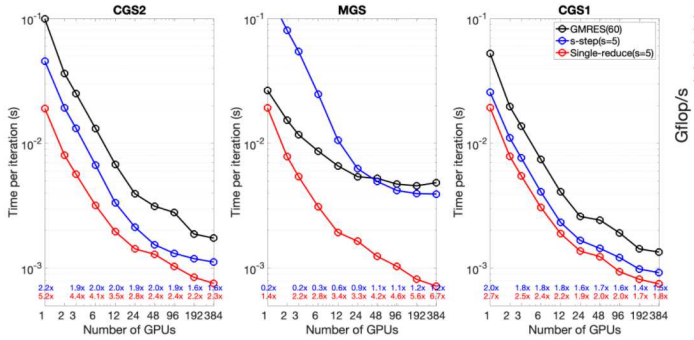
Overall, on a small number of CPUs, single-reduce CGS+CholQR2 was slower than standard CGS+CholQR due to the additional computation for applying CholQR twice, but the single-reduce algorithm provides more stability. When the CPU count increases, the communication cost for the orthogonalization becomes more dominant, and the single-reduce scheme becomes faster than the standard scheme (single-reduce with one global reduce compared with the standard CGS+CholQR requiring two global reduces).

7 Performance with 3D Laplace on Summit

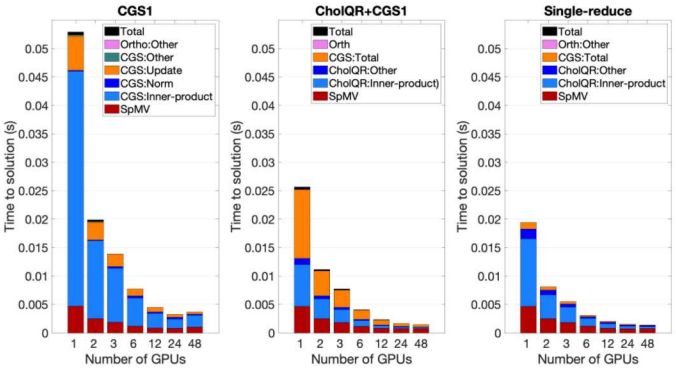
To demonstrate our solver implementation's portability, we show solver performance on the Summit supercomputer at Oak Ridge National Laboratory. Each Summit node has two 21-core IBM Power 9 CPUs and six NVIDIA Volta V100 GPUs. The code was compiled using `g++` compiler version 6.4 or NVIDIA Cuda 9.2 with the optimization flag `-O3`, and linked to the IBM Engineering and Scientific Subroutine Library (ESSL) version 6.1 and Spectrum Cuda-aware MPI version 10.3.

7.1 CA GMRES on Volta GPUs. Figure 9(a) displays the average iteration time when applying the different orthogonalization schemes for solving the 3D Laplace problem, without a preconditioner. On a fixed number of GPUs, GMRES converges with the same iteration count when using any of the orthogonalization schemes (except that the s-step method can result in a maximum of $s-1$ additional iterations because it checks for convergence every s steps). Thus, the speedup obtained in terms of the time to solution is the same as the speedup obtained with respect to the time per iteration.

The standard GMRES iteration time (i.e., the black line) in Figure (9(a)) shows that MGS-based GMRES



(a) Time per iteration in seconds.



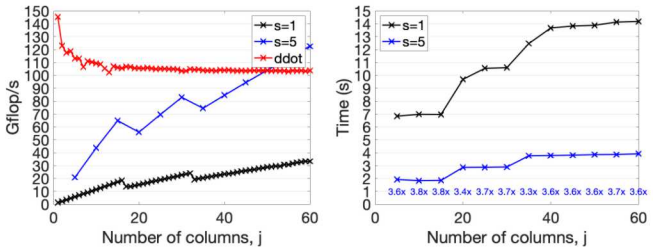
(b) Breakdown of iteration time with CGS.

Figure 9: GMRES performance on Summit NVIDIA V100 nodes for solving 3D Laplace ($n = 8$M).

is faster than GMRES with CGS1 on a small number of GPUs. Also, when using MGS, s-step GMRES is slower than standard GMRES. These unexpected results are because both orthogonalization and solver performance depend strongly upon local computational kernel performance, especially on a small number of GPUs.

For computing dot products with multiple vectors at a time, we use NVIDIA's `cublasDgemm` function, wrapped in kokkos-kernels' portable `gemm` interface. To compute single-vector dot products, we use the portable kokkos-kernels function `dot`. We have found that `cublasDgemm` is not optimized for many cases of "tall and skinny" matrices. Figure 10 shows that `cublasDgemm` does not perform as well as `dot` unless there are a large number of basis vectors (e.g., $s = 50$).

Developing a high-performance dense matrix-matrix multiply kernel for the GPU is not only a lot of work, but it calls for architecture-specific (even



(a) Gflop/s. (b) Time (s).

Figure 10: `DGEMM` and `DDOT` performance of computing the dot products $Q_{1:j}^T Q_{j+1:j+s}$ on NVIDIA Volta 100 GPU ($n = 8$M).

GPU model-specific) details that would hinder our goal of performance portability. This is why we rely on NVIDIA's native cuBLAS library. Some of the authors have experience implementing custom CUDA kernels for tall and skinny matrix-matrix multiplies, but an important goal for Trilinos development is avoiding architecture-specific code. Our next step is to use Kokkos to implement these kernels in a platform-independent way. The performance of the solvers using CGS, and its s-step and single-reduce variants, is expected to improve further when a matrix-matrix multiply kernel optimized for tall and skinny matrices becomes available.

Nevertheless, s-step GMRES still outperforms standard GMRES on a large enough number of GPUs, where the inter-GPU communication becomes much more significant. Then, by further reducing communication cost, the single-reduce variant of GMRES outperforms the s-step variant (respective maximum speedups of 2.4×, 10.1×, and 1.4× over s-step, using CGS2, MGS, and CGS1). In fact, the single-reduce variant of CGS2 or MGS takes about the same iteration time as the CGS1 variant, suggesting that using the CGS2 or MGS variant, the stability of the GMRES iteration can be greatly improved without slowing down the iteration time.

Figure 9(b) shows the breakdown of the iteration time. On a large enough number of GPUs, the iteration is dominated by `SpMV` and the benefit of using the single-reduce orthogonalization kernel starts to diminish.[1]

[1] With an increase in the GPU count, the standard GMRES iteration time will start to grow due to the quadratic time inter-GPU communication cost. For these extreme strong-scaling tests, the s-step or single-reduce iteration will delay the increase in the iteration time to a larger number of GPUs. These extreme setup cases are not shown as they are of no interest for the practical cases.

#Size	Overall(us)	Compute(us)	Pure Comm.(us)	Overlap(%)
8	10.43	6.47	5.66	30.00
16	10.43	6.47	5.68	30.27
32	10.70	6.59	5.73	28.30
64	10.49	6.47	5.75	30.03
128	11.61	7.24	5.91	25.87
256	11.75	7.29	5.98	25.48
512	13.04	8.01	6.79	26.00
1024	13.64	8.72	7.11	30.73
2048	15.22	9.41	8.23	29.34

Figure 11: Non-blocking allreduce latency test between four processes, with two process on one node of Summit.

7.2 Pipelined GMRES on Power9 CPUs. We now show performance of the single-reduce orthogonalization schemes, in combination with pipelined GMRES, on the ORNL Summit Power 9 CPUs.

The performance of pipelined GMRES depends strongly on the performance of nonblocking MPI collective communication operations, in particular, how well they make asynchronous progress. MPI does not promise that nonblocking collectives do anything at all until users wait on them. In practice, MPI implementations use an extra thread per MPI process to drive asynchronous progress. This "progress threads" option may impose a cost on every MPI send and receive, and it may not be enabled by default. For our experiments, we did not use the progress threads (it improved the overlapping of the communication and computation, but slowed down the iteration, i.e., `SpMV`).

To study this, we first ran the Ohio State University Micro-benchmarks. Figure 11 shows an overlap of around 30% between communication and local computation. Unfortunately, this may not reflect the actual solver performance. This is because the solver also performs point-to-point communication for `SpMV` between the nonblocking collective and the local `SpMV` computation. Thus, in order for collective communication to overlap with computation, the collective operations must also be pipelined with the point-to-point communication. Overall, Figure 12 suggests only a marginal benefit from pipelining on thousands of processors.

However, the focus of the experiments is to study the overhead of our single-reduce CGS2 or MGS orthogonalization schemes over the performance of the pipelined GMRES with the standard CGS. In Section 4, we have already demonstrated that these improve the stability for solving ill-conditioned linear systems. For the performance experiments in Figure 12, the 3D Laplace problem was examined (also in Figure 9). For this problem, the pipelined GMRES does not encounter the same numerical difficulties, and thus, all variations of pipelined GMRES performed the same number of iterations with the equivalent MPI process count. The figure shows that the extra stability can be obtained without significant overhead (in some case, it can be faster due to the performance of underlying kernels).

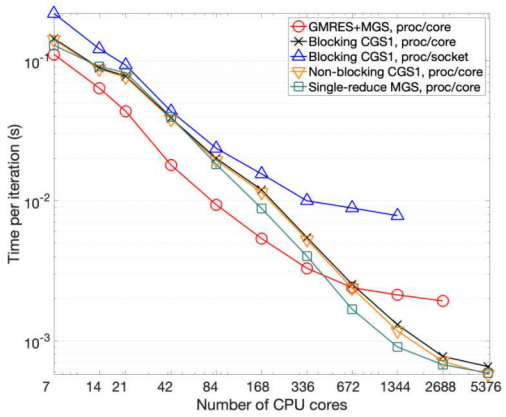


Figure 12: Time per iteration for solving 3D Laplace problem ($n = 8$M) using the pipelined GMRES on Summit Power 9 CPUs.

8 Conclusion

We studied the performance of single-reduce Gram Schmidt orthogonalization schemes for s-step and pipelined GMRES. These implementations reduce the communication cost of the corresponding orthogonalization schemes, or equivalently, improve the orthogonality of the generated basis vectors with a single global reduce. Our implementation is based on the Trilinos software framework, which allows us to develop solvers that perform well on many different computer architectures with a single code base. Our numerical and performance results on three different architectures demonstrated the potential of the new orthogonalization schemes.

We have also shown that the performance of these solvers depends strongly on the performance of the underlying communication and computational kernels. We expect better performance of the communication-avoiding or pipelined solver, and the benefit of the single-reduce orthogonalization, as the performance of these kernels mature on the new architectures (e.g., the BLAS-3 kernel for the tall and skinny dot products). Also, it is still a challenge to overlap the nonblocking all-reduce (followed by point-to-point communication) with the computation, especially with the GPUs. We are currently looking at supporting this feature more effectively in Trilinos.

We are studying the extension of the orthogonalization algorithms, to improve the orthogonality of the basis vector generated by single-reduce CGS2 with extra computation. The GPU implementations of different preconditioners are beyond the scope of this paper, but we would like to study the performance of our solvers in a real application on GPUs.

Acknowledgments

Funding was provided by the Exascale Computing Project (17-SC-20-SC). The National Renewable Energy Laboratory is operated by Alliance for Sustainable Energy, LLC, for the U.S. Department of Energy (DOE) under Contract No. DE-AC36-08GO28308. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy National Nuclear Security Administration under contract DE-NA0003525.

References

[1] Z. Bai, D. Hu, and L. Reichel. A Newton basis GMRES implementation. *IMA J. Numer. Anal.*, 14:563–581, 1994.

[2] C. Baker and M. Heroux. Tpetra, and the use of generic programming in scientific computing. *Scientific Programming*, 20(2):115–128, 2012.

[3] Eric Bavier, Mark Hoemmen, Sivasankaran Rajamanickam, and Heidi Thornquist. Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems. *Scientific Programming*, 31(3):241–255, 2012.

[4] S. Domino. Sierra low mach module: Nalu theory manual 1.0. Technical Report SAND2015-3107W, Sandia National Laboratories Unclassified Unlimited Release (UUR), 2015. `https://github.com/Exawind/nalu-wind`.

[5] H. Edwards, C. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[6] Paul R. Eller, Torsten Hoefler, and William Gropp. Using performance models to understand scalable Krylov solver performance at scale for structured grid problems. In *Proceedings of the ACM International Conference on Supercomputing*, pages 138–149, 2019.

[7] P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose. Hiding global communication latency in the GMRES algorithm on massively parallel machines. *SIAM Journal on Scientific Computing*, 35(1):C48–C71, 2013.

[8] P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. *Parallel Computing*, 40(7):224 – 238, 2014. 7th Workshop on Parallel Matrix Algorithms and Applications.

[9] A. Greenbaum, M. Rozložník, and Z. Strakoš. Numerical behaviour of the modified Gram-Schmidt GMRES implementation. *BIT Numerical Mathematics*, 37:706–719, 1997.

[10] L. Grigori and S. Moufawad. Communication Avoiding ILU0 Preconditioner. *SIAM Journal on Scientific Computing*, 37(2):C217–C246, 2015.

[11] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, Jonathan J Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, et al. An Overview of the Trilinos Project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, 2005.

[12] Magnus Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49:409–439, 1952.

[13] M. Hoemmen. *Communication-avoiding Krylov subspace methods*. PhD thesis, EECS Department, University of California, Berkeley, 2010.

[14] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 36:1–36:12, 2009.

[15] H. Morgan, M. Knepley, P. Sanan, and L. Scott. A stochastic performance model for pipelined Krylov methods. *Concurrency and Compututation: Practice and Experience*, 28:4532–4542, 2016.

[16] A. Ruhe. Numerical aspects of Gram-Schmidt orthogonalization of vectors. *Linear Algebra and its Applications*, 52:591–601, 1983.

[17] Y. Saad. *Iterative methods for sparse linear systems*. the Society for Industrial and Applied Mathematics, Philadelphia, PA, 3 edition, 2003.

[18] Y. Saad and M. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7:856–869, 1986.

[19] V. Simoncini and D. Szyld. Theory of inexact Krylov subspace methods and applications to scientific computing. *SIAM Journal on Scientific Computing*, 25:454–477, 2003.

[20] K. Świrydowicz, J. Langou, S. Ananthan, U. Yang, and S. Thomas. Low synchronization Gram-Schmidt and GMRES algorithms. *Numerical Linear Algebra with Applications*, 2019. accepted.

[21] H. Walker and L. Zhou. A simpler GMRES. *Numerical Linear Algebra with Applications*, 1:571–581, 1994.

[22] K. Wu and H. Simon. Thick-restart Lanczos method for large symmetric eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 22:602–616, 2000.

[23] I. Yamazaki, S. Rajamanickam, E. Boman, M. Hoemmen, M. Heroux, and S. Tomov. Domain Decomposition Preconditioners for Communication-avoiding Krylov Methods on a Hybrid CPU-GPU Cluster. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 933–944, 2014.