Towards a Scalable Hierarchical High-order CFD Solver

Zan Xu*, Léopold Cambier†, Juan J. Alonso‡ and Eric Darve§
Stanford University, Stanford, CA, 94305

Development of highly scalable and robust algorithms for large-scale CFD simulations has been identified as one of the key ingredients to achieve NASA's CFD Vision 2030 goals. In order to improve simulation capability and to effectively leverage new high-performance computing hardware, the most computationally intensive parts of CFD solution algorithms —namely, linear solvers and preconditioners— need to achieve asymptotic behavior on massively parallel and heterogeneous architectures and preserve convergence rates as the meshes are refined further. In this work, we present a scalable high-order implicit Discontinuous Galerkin solver from the SU2 framework using a promising preconditioning technique based on algebraic sparsified nested dissection algorithm with low-rank approximations, and communication-avoiding Krylov subspace methods to enable scalability with very large processor counts. The overall approach is tested on a canonical 2D NACA0012 test case of increasing size to demonstrate its scalability on multiple processing cores. Both the preconditioner and the linear solver are shown to exhibit near-linear weak scaling up to 2,048 cores with no significant degradation of the convergence rate.

I. Nomenclature

A	=	Matrix of linear system	R^n	=	Spatial residual evaluated at timestep n
D_k	=	Discrete element k	$\frac{\partial R}{\partial u}$	=	Spatial Jacobian
F	=	Convective flux	s	=	Step size in s-step GMRES
F^*	=	Numerical flux	t	=	Temporal coordinates
H	=	Upper Hessenberg matrix	U	=	Upper triangular matrix
I	=	Identity matrix	u	=	Conserved variables
K	=	Number of discrete elements	$\hat{u}$	=	Modal coefficients
L	=	Lower triangular matrix	$\bar{u}$	=	Nodal coefficients
ℓ	=	level in Nested Dissection ordering	$\vec{x}$	=	Cartesian coordinates
l_i	=	Lagrange polynomial	α	=	Angle of attack
M	=	Mass matrix	$\Delta\tau$	=	Time step
M_∞	=	Free-stream Mach number	ε	=	tolerance in spaND
N	=	Dimension of Jacobian matrix	λ	=	Spectral radius of the convective operator
N_p	=	Number of degrees of freedom	ρ	=	Density
n_i	=	Normal vector at interfaces	ψ_i	=	Legendre polynomial
p	=	Polynomial order	Ω	=	Physical domain
Q	=	Orthogonal matrix	$\partial\Omega$	=	Boundary of physical domain

II. Introduction

Advancements in computational capabilities have enabled many large-scale simulations of complex fluid flow problems. With the expected arrival of next-generation exascale supercomputers in 2021 [1], highly efficient, robust, and scalable CFD solvers are needed to fully harness the power of High Performance Computing (HPC). For various HPC codes solving the Navier-Stokes equations and their various approximations, the most computationally-intensive parts are often contained in numerical algorithms such as linear solvers and preconditioners. As processor count and

*Ph.D. Candidate, Department of Aeronautics and Astronautics, AIAA Student Member.
†Ph.D. Candidate, Institute for Computational and Mathematical Engineering.
‡Vance D. and Arlene C. Coffman Professor, Department of Aeronautics and Astronautics, AIAA Associate Fellow.
§Professor, Department of Mechanical Engineering.

problem size increase, the time spent in these numerical algorithms, relative to the rest of the application, grows and quickly dominates the total execution time [2]. This is especially true for high-order methods such as Discontinuous Galerkin (DG) methods that are computationally expensive. Hence, developing linear solvers and preconditioners suited to massive CFD problems and very large processor counts plays a crucial part in building scalable CFD solvers.

Unfortunately, despite decades of work in the area, the development of preconditioners for linear systems are still considered an art and the details depend on the specific problem being solved. Many options are available but none work well in all cases. Current options for large-scale implicit discretizations of the Navier-Stokes equations are limited to multigrid approaches, which are difficult to implement and yield limited benefits [3], and incomplete LU factorizations (ILU) which work for small problems but deteriorate rapidly for larger grids [4]. In addition, many numerical algorithms are found to incur overwhelming communication costs when moving data between levels of the memory hierarchy or between processors over a network on massively-parallel computers [5], especially when synchronization points across the entire parallel computer are an integral part of the algorithm. Achieving scalable linear solvers thus requires not only robust and reliable preconditioning techniques, but also a dramatic shift in algorithm design with a focus on reducing communication.

In this paper, we present our recent efforts towards constructing a scalable flow solver using an implicit high-order DG discretization. The work described in this paper focuses on the development of ideas for the core preconditioned linear solver step and does not go into much detail in the description of the DG discretization in the SU2 solver. For the linear system, we apply a promising algebraic preconditioner that is based on hierarchical matrices and low-rank approximations, and has shown scalability for many partial differential equation (PDE) problems [6]. To overcome communication overhead, we implement variants of a Krylov subspace method that avoids/hides communication to solve the preconditioned linear system. A number of scalability studies are performed to demonstrate the scalability of the overall approach. The paper is organized as follows: Section III gives technical details of the different aspects of the proposed scalable CFD solver, Section IV outlines the numerical results for the integrated approach, and Section V concludes and describes the intended future work.

III. Methodology

In this section, we present various components of the overall scalable CFD solver framework: a high-order DG flow solver, a novel preconditioning technique, and a communication-avoiding iterative linear solver. In combination, these methods can achieve truly scalable solution methods for complex CFD problems.

A. Flow Solver

1. Governing equations

In this work, we are interested in compressible form of the inviscid Euler equations of gas dynamics,

$$\frac{\partial u}{\partial t} + \nabla \cdot F = 0. \tag{1}$$

The governing equations are solved in a physical domain Ω with boundary $\partial\Omega$.

2. Implicit Discontinuous Galerkin discretization

DG methods form a class of numerical methods for solving PDE problems with high order of accuracy [7]. In a Discontinuous Galerkin Finite Element Discretization (DG-FEM), we divide the computational domain Ω into K discrete elements D_k,

$$\Omega = \bigcup_{k=1}^{K} D_k. \tag{2}$$

The numerical solution within each element D_k is constructed using nodal or modal representations as

$$u(\vec{x}, t) = \sum_{i=1}^{N_p} \hat{u}_i(t)\psi_i(\vec{x}) = \sum_{j=1}^{N_p} \bar{u}(x_j, t) l_j(\vec{x}). \tag{3}$$

The number of degrees of freedom N_p is defined by the order of the polynomial basis p. By allowing discontinuity at element boundaries, the weak form of the governing equations over each spatial element D_i can be expressed using a nodal representation as

$$\int_{D_k} \frac{\partial \bar{u}}{\partial t} l_j dV - \int_{D_k} F_i \frac{\partial l_j}{\partial x_i} dV + \oint_{\partial D_k} F_i^* n_i l_j dA = 0, \qquad j = 1, ..., N_p. \tag{4}$$

A common choice of the numerical flux F^* is the Roe flux [8]. The implicit discretization with a backwards Euler temporal discretization leads to a linear system at every timestep n

$$\left(\frac{1}{\Delta\tau^n} M + \left. \frac{\partial R}{\partial u} \right|^n \right) \Delta u^{n+1} = -R^n, \tag{5}$$

where

$$M = \int_{D_k} l_i l_j dV, \tag{6}$$

$$R^n = -\int_{D_k} F_i(u^n) \frac{\partial l_j}{\partial x_i} dV + \oint_{\partial D_k} F_i^*(u^n) n_i l_j dA. \tag{7}$$

For stability, the timestep $\Delta\tau$ is chosen based on the CFL number

$$\Delta\tau = \min(\Delta\tau_k) = \min\left(\frac{\text{CFL}}{\lambda_k} \right). \tag{8}$$

For steady-state problems, pseudo time-stepping is used to accelerate convergence where the CFL number at the nth timestep is computed by

$$\text{CFL}^n = \min\left(\text{CFL}^0 \frac{\|R^0\|}{\|R^{n-1}\|}, \text{CFL}^\infty \right). \tag{9}$$

DG methods are local methods in the sense that, by allowing discontinuity at element boundaries, the high-order solution within each discrete element depends only on its immediate neighbors. This has significant advantages in a distributed-memory architecture as each partition of the computational domain relies only on its interior elements and a single layer of halo elements at the boundaries of the partition to construct high-order solutions. Such partitions typically have a low surface-to-volume ratio that makes the solver highly parallelizable. In an implicit discretization, the locality of DG methods often gives rise to Jacobians with a sparse, block matrix structure. The low surface-to-volume ratio translates to minimal point-to-point communication costs in matrix kernels that scale well on increasing number of processors.

3. Implementation

The SU2 software suite [9–11] is an open-source collection of software tools written in C++ and Python for performing multi-physics simulation and design. It is built specifically for the analysis of PDEs and PDE-constrained optimization problems on general unstructured meshes with state-of-the-art numerical methods. The DG-FEM solver is one of the solvers in SU2. It supports all standard elements in two and three dimensions up to arbitrary polynomial orders with the option of local p-refinement. Other features of the DG-FEM solver include treatment of curved elements, treatment of viscous terms, shock-capturing capability, ADER-DG discretization, etc [12]. For the implicit DG discretization in SU2, the spatial Jacobian $\frac{\partial R}{\partial u}$ in Eq. (5) is evaluated exactly using the automatic differentiation (AD) tool CodiPack [13] which allows the implicit formulation to leverage all features available in SU2.

B. Sparsified Nested Dissection and TaskTorrent

1. Sparsified Nested Dissection algorithm

Sparsified Nested Dissection (spaND) is a fast multilevel algorithm for solving large sparse linear systems [6]. Let $Ax = b$ be the linear system from Eq. (5). The algorithm first computes a Nested Dissection (ND) ordering of the matrix A. This defines interiors, separators, and interfaces at each dissection level $0 \leq \ell < \ell_{\max}$. Level 0 is the leaf level and $\ell_{\max}$ is the top level. At each level, interiors are separated by the ND separators and eliminating an interior does not create fill-in beyond its adjacent separator. Interfaces are defined as subsets of separators adjacent to a given pair of interiors on each side of the separator.

The algorithm then proceeds level by level, from the leaf level to the top level. At each level ℓ:

- Interiors at level ℓ are eliminated using row-pivoted block LU factorization. This is the same algorithm as any sparse direct method. With $A_{ss} = PLU$ we find

$$\begin{bmatrix} L^{-1}P^\top & \\ -A_{ns}A_{ss}^{-1} & I \end{bmatrix} \begin{bmatrix} A_{ss} & A_{sn} \\ A_{ns} & A_{nn} \end{bmatrix} \begin{bmatrix} U^{-1} & -A_{ss}^{-1}A_{sn} \\ & I \end{bmatrix} = \begin{bmatrix} I & \\ & A_{nn} - A_{ns}A_{ss}^{-1}A_{sn} \end{bmatrix}. \quad (10)$$

  This step introduces fill-in on A_{nn} because of the $A_{ns}A_{ss}^{-1}A_{sn}$ term. Notice that many rows (respective columns) in A_{ns} (resp. A_{sn}) are zero and, as such, do not have to be updated.
- Interfaces at level ℓ are scaled using a row-pivoted block LU factorization. If $A_{ss} = PLU$, we have

$$\begin{bmatrix} L^{-1}P^\top & \\ & I \end{bmatrix} \begin{bmatrix} A_{ss} & A_{sn} \\ A_{ns} & A_{nn} \end{bmatrix} \begin{bmatrix} U^{-1} & \\ & I \end{bmatrix} = \begin{bmatrix} I & L^{-1}P^\top A_{sn} \\ A_{ns}U^{-1} & A_{nn} \end{bmatrix}. \quad (11)$$

  Scaling is required for accuracy as scaling the diagonal blocks leads to a much lower number of iterative method steps [6]. Note that this is not an approximation and does not create any fill-ins. It does, however, balance the matrix since all diagonal blocks now have the same unit norm. Note that in this step, it is preferable to balance L and U so that $\|U^{-1}\| \approx \|L^{-1}\|$. We do so by splitting the diagonal of the upper-triangular matrix from the LU factorization evenly between L and U.
- Interfaces at level ℓ are sparsified using a low-rank approximation (in practice, rank-revealing QR). Let s be an interface and n all the neighbors of s in the trailing matrix A. We first compute $Q_s = \begin{bmatrix} Q_{sc} & Q_{sf} \end{bmatrix}$ such that

$$\begin{bmatrix} A_{sn} & A_{ns}^\top \end{bmatrix} = Q_{sc} \begin{bmatrix} W_{cn} & W_{nc}^\top \end{bmatrix} + Q_{sf} \begin{bmatrix} W_{fn} & W_{nf}^\top \end{bmatrix}, \quad (12)$$

  with $\left\| \begin{bmatrix} W_{fn} & W_{nf}^\top \end{bmatrix} \right\|_2 = O(\varepsilon)$ where ε is a user-prescribed tolerance. Let w be the remaining degrees of freedom disconnected from s. Given this low-rank approximation, the trailing matrix can be factorized as

$$\begin{bmatrix} Q^\top & & \\ & I & \\ & & I \end{bmatrix} \begin{bmatrix} I & A_{sn} & \\ A_{ns} & A_{nn} & A_{nw} \\ & A_{wn} & A_{ww} \end{bmatrix} \begin{bmatrix} Q & & \\ & I & \\ & & I \end{bmatrix} = \begin{bmatrix} I & & W_{cn} & \\ & I & \varepsilon & \\ W_{nc} & \varepsilon & A_{nn} & A_{nw} \\ & & A_{wn} & A_{ww} \end{bmatrix}. \quad (13)$$

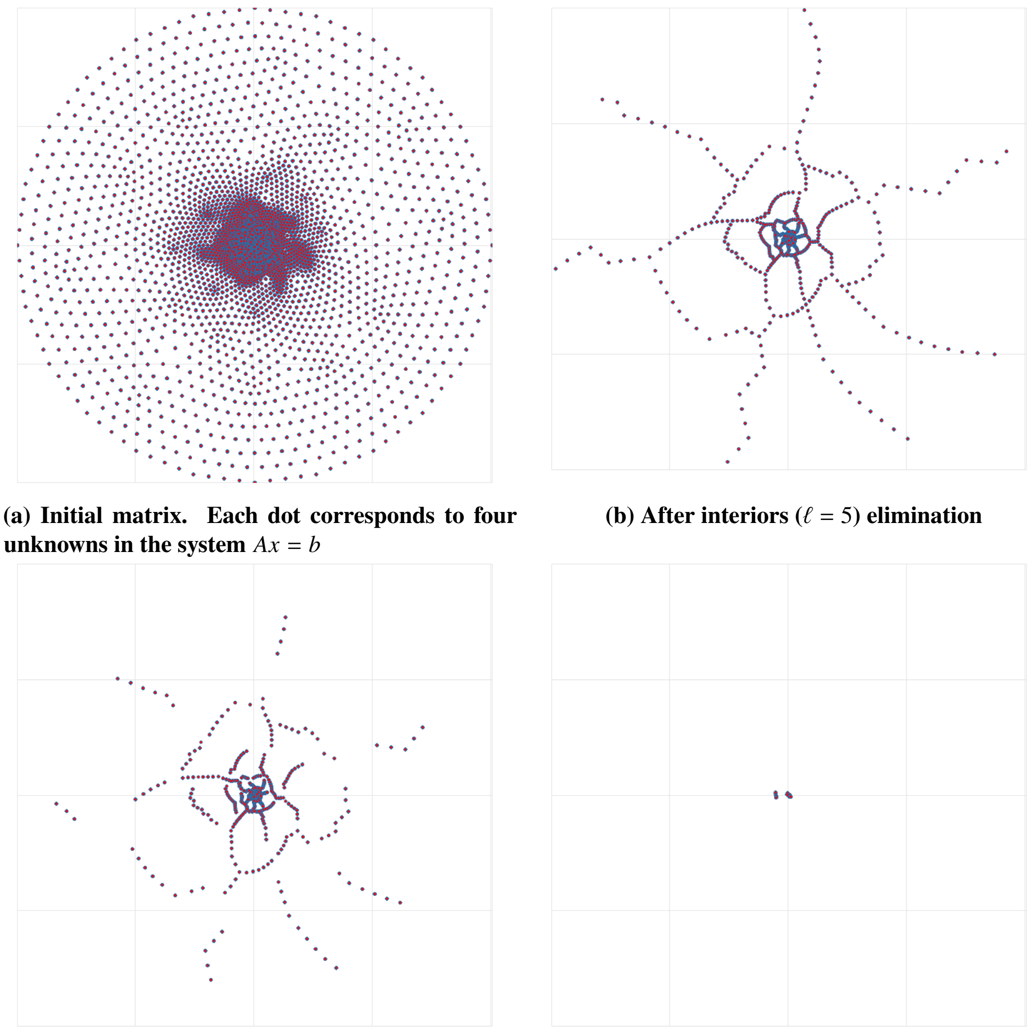
  Assuming $\varepsilon \approx 0$, the rows and columns corresponding to the variable f are effectively 0, so f is approximately eliminated. In addition, this procedure did not introduce any fill-in on the neighbors n of s (A_{nn} is unchanged). So this (approximately) eliminated some of the non-interior unknowns without introducing any fill-in in the trailing matrix.

  In practice, we first compute all Q_i without updating the trailing matrix. Only then do we compress every A_{ij} block and replace it with $A_{ij}^+ = Q_{ic}^\top A_{ij} Q_{jc}$. This leads to the same approximation as described above, but is more concurrent since every rank-revealing QR can be done simultaneously.
- We then merge all the clusters and proceed to the next level.

At the end, the algorithm produces an approximate factorization of A, $A \approx \prod_i F_i$ where F_i is a sparse triangular matrix (from the elimination or block scaling) or a sparse orthogonal matrix (from the sparsification). This is then used as a preconditioner for Krylov iterative methods, such as Conjugate Gradient [14], Generalized Minimum RESidual (GMRES) [15], etc.

Figure 1 illustrates the algorithm applied to a linear system generated from implicit discretization of a typical 2D airfoil mesh. Figure 1a shows all degrees of freedom in the system. Figure 1b shows the remaining degrees of freedom after 5 levels of ND elimination. We clearly see the eliminated interiors (the large white areas) and the remaining separators and interfaces. Figure 1c shows the effect of interface sparsification. All interfaces (i.e., subsets of a separator separating a pair of interiors) are sparsified, which reduces their size without introducing fill-in. This reduces the sizes of all separators in the systems. Figure 1d shows the top separator before its final elimination. Instead of a usual ND separator cutting across the domain, only a few points are left.

By repeating this process at every level, this algorithm keeps the separator size small. In 2D problems, separators typically have size $O(1)$ (instead of $O(N^{1/2})$ without sparsification), while in 3D, separators now have size $O(N^{1/3})$ (instead of $O(N^{2/3})$). This leads to a great reduction in computational cost. A direct method using an ND ordering usually has complexity $O(N^{3/2})$ for 2D problems. In contrast, assuming the separators shrink to size $O(1)$, spaND has complexity $O(N)$ in 2D, for both factorization and solve time. It is then typically used as a preconditioner coupled with CG or GMRES. Assuming a small and slowly growing iteration count, this leads to a linear or near-linear time algorithm.


(a) Initial matrix. Each dot corresponds to four unknowns in the system $Ax = b$

(b) After interiors ($\ell = 5$) elimination

(c) After interface ($\ell = 5$) sparsification. All interfaces are reduced in size.

(d) Last top separator. A typical ND separator would be a line cutting through the entire domain.

Fig. 1 Illustration of the spaND algorithm. At every level, interiors are eliminated and the remaining interfaces are sparsified. Interfaces are then merged and the algorithm proceeds to the next level. This leads to separators of size $O(1)$ instead of lines of size $O(N^{1/2})$.

2. Parallel implementation using TaskTorrent

We finally consider a parallel version of the spaND algorithm. To do so, we use a task-based runtime system in C++, TaskTorrent [16]. TaskTorrent is a lightweight and distributed task-based runtime system where computations are expressed as a directed acyclic graph (DAG) of tasks using a parametrized task graph formulation. Tasks in the DAG

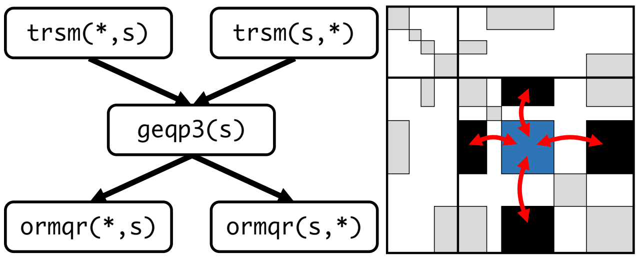


Fig. 2 Task-based sparsification of an interface s. The left shows the local DAG, with the in- and out-dependencies of task `geqp3(s)`. The right figure shows the trailing matrix, with the block (s, s) in blue. The first four diagonal blocks are interiors (at this point, all eliminated) and other diagonal blocks are interfaces. Red arrows show incoming and outgoing dependencies of task `geqp3(s)`.

are then run as soon as ready by the runtime. Active messages (a form of one-sided asynchronous communications combining a function and a payload) let tasks trigger other tasks on remote nodes. TaskTorrent uses C++ threads for intra-node parallelism and MPI for inter-nodes communications.

The key step in parallelizing spaND using TaskTorrent is the definitions of the DAG of tasks. We do so by transforming every block operation into a corresponding task. This is done at the granularity of interfaces. Furthermore, we parallelize each level independently. This means there is one distinct DAG of task per level in the algorithm. In TaskTorrent, this requires identifying the number of incoming dependencies for each task, the computational routine of the task itself, and the outgoing dependencies of each task.

Consider for instance the rank-revealing QR factorization related to the sparsification of interface s, denoted `geqp3(s)`. Let $n_1, \dots, n_k$ denote all the neighboring interfaces of s.

- The task requires $A_{sn_1}, \dots, A_{sn_k}, A_{n_1 s}, \dots, A_{n_k s}$. As such, `geqp3(s)` has $2k$ incoming dependencies.
- Given $A_{sn_1}, \dots, A_{sn_k}, A_{n_1 s}, \dots, A_{n_k s}$, the task computes $\begin{bmatrix} Q_c & Q_f \end{bmatrix} = \text{geqp3}\left(\begin{bmatrix} A_{sn} & A_{ns}^\top \end{bmatrix}\right)$. This is done using LAPACK's `geqp3` function.
- Q_c is used during the compression steps on blocks $A_{sn_1}, \dots, A_{sn_k}, A_{n_1 s}, \dots, A_{n_k s}$. The compression of block A_{ij} is denoted `ormqr(i,j)` and performs $A_{ij} \leftarrow Q_{ic}^\top A_{ij} Q_{jc}$. As such, `geqp3(s)` has to fulfill the dependencies of tasks `ormqr(s,n1)`, ..., `ormqr(s,nk)`, `ormqr(n1,s)`, ..., `ormqr(nk,s)` and send the basis Q_c to the corresponding MPI ranks.

Fig. 2 illustrates the related DAG and the dependencies in the trailing matrix. All tasks in spaND are parallelized using TaskTorrent in a similar way to miminize load imbalance. The overall framework of spaND and TaskTorrent are incorporated in the SU2 codebase.

C. Communication-avoiding (CA) Krylov Subspace methods

Krylov subspace methods are a class of iterative methods commonly applied to large, sparse linear systems in CFD codes. However, on large machines, traditional Krylov solvers do not scale well due to their reliance on expensive communication steps in various kernels, particularly those involving global communication constructs. To alleviate performance bottlenecks due to communication overhead, we implement CA Krylov subspace methods to achieve asymptotic performance improvements when solving large-scale linear systems.

In this work, we are interested in the GMRES solver and its CA variants. The classical GMRES algorithm shown in Algorithm 1 will incur Point-to-Point communication costs during Sparse Matrix-Vector (SpMV) multiplication (line 3) in a distributed memory setting. In addition, dot product computations (lines 5, 8) in the Modified Gram-Schmidt (MGS) algorithm require global reductions which are very costly synchronization steps, especially in the presence of system noise and load imbalances. The number of global reductions in MGS also grows quadratically with the iteration count. These kernels make the performance of the GMRES solver communication-bound.

To improve the performance of GMRES, CA variants of GMRES have been proposed [5, 17, 18]. This work employs the s-step CAGMRES algorithm in [5], which attempts to avoid communication by generating s Krylov basis vectors at once and orthogonalize them using a block version of the Classical Gram-Schmidt (CGS) algorithm followed by a Cholesky QR factorization (CholQR). This reduces the number of reductions required to one per s vectors, achieving a reduction in the communication latency by a factor of s at the expense of additional arithmetic operations. However, at large values of s, consecutive SpMVs produce vectors that converge to the principal eigenvector of the matrix, which then introduce numerical instability in subsequent orthogonalization steps. To overcome this, a different basis function is often incorporated to condition the s block. Re-orthogonalization schemes, such as two CGSs and two CholQRs, that do not require extra communication are also introduced to minimize the orthogonality error [19, 20]. A skeleton of the s-step CAGMRES algorithm with single-reduce re-orthogonalization is shown in Algorithm 2. Only one single reduction is required for each s vector (line 6). This is achieved by lagging the re-orthogonalization and re-normalization of previous s vectors (line 7, 8) and combining them with the orthogonalization and normalization of s vectors in the next iteration (line 12, 13). A detailed algorithm can be found in [19]. It is worth pointing out that s times SpMV in line 4 can also be applied in a CA manner, see [5]. However, it is challenging to apply preconditioners in a similar CA fashion. Therefore, the CA variant of SpMV kernel is left for future work. Similar to spaND, CAGMRES algorithm in this work is implemented in the SU2 codebase.

Algorithm 1 Classical GMRES

```
    Input: $n \times n$ matrix $A$, right hand side vector $b$, initial guess vector $x_0$
    Output: $x$, solution to the linear system $Ax = b$.
 1: $r := b - Ax_0, q_0 := r/||r||_2$
 2: for $j = 0, 1, \ldots$ do
 3:     $v = Aq_j$                                   ▷ Sparse Matrix-Vector Multiplication (SpMV)
 4:     for $i = 0, \ldots, j$ do                      ▷ Modified Gram-Schmidt (MGS)
 5:         $h_{ij} = v^T q_i$                         ▷ global reduction
 6:         $v = v - h_{ij} q_i$
 7:     end for
 8:     $h_{j+1,j} = ||v||_2$                          ▷ global reduction
 9:     $q_{j+1} = v/h_{j+1,j}$
10:     Apply Givens rotation to update matrix $H_j$
11:     Check for convergence
12: end for
13: $y = \text{argmin}||(H_j y - ||r||_2 e_0)||_2$
14: $x = x_0 + Q_j y$
```

Algorithm 2 s-step CAGMRES with single-reduce re-orthogonalization

```
    Input: $n \times n$ matrix $A$, right hand side vector $b$, initial guess vector $x_0$, step size $s$.
    Output: $x$, solution to the linear system $Ax = b$.
 1: $r := b - Ax_0, q_0 := r/||r||_2$
 2: for $j = 0, s \ldots$ do
 3:     for $k = j \ldots j + s - 1$ do
 4:         $Q_{:,k+1} = AQ_{:,k}$                      ▷ $s$ times SpMV
 5:     end for
 6:     $[R_{:,j-s:j-1}, R_{:,j:j+s}] = Q^T_{:,0:j+s}[Q_{:,j-s:j-1}, Q_{:,j:j+s}]$      ▷ single reduction
 7:     CGS($Q_{:,0:j-s-1}, Q_{:,j-s:j-1}, R_{:,j-s:j-1}, R_{:,j:j+s}$)      ▷ Re-orthogonalize $Q_{:,j-s:j-1}$
 8:     CholQR($Q_{:,j-s:j-1}, R_{:,j-s:j-1}, R_{:,j:j+s}$)      ▷ Re-normalize $Q_{:,j-s:j-1}$
 9:     Assemble upper Hessenberg matrix $H_{j-1}$
10:     Apply Givens rotation to update matrix $H_{j-1}$
11:     Check for convergence
12:     CGS($Q_{:,0:j-1}, Q_{:,j:j+s}, R_{:,j:j+s}$)      ▷ Orthogonalize $Q_{:,j:j+s}$
13:     CholQR($Q_{:,j:j+s}, R_{:,j:j+s}$)      ▷ Normalize $Q_{:,j:j+s}$
14: end for
15: $y = \text{argmin}||(H_{j-1} y - ||r||_2 e_0)||_2$
16: $x = x_0 + Q_{:,0:j-1} y$
```

IV. Numerical Results

A. Simulation Setup

To demonstrate our scalable hierarchical CFD solver, we perform numerical experiments on a 2D NACA0012 airfoil in an inviscid compressible flow at an angle of attack $\alpha = 2°$ and a free-stream Mach number of $M_\infty = 0.3$. The governing equations are discretized using the implicit DG formulation described in Section III.A. The computational mesh is a structured O-mesh with $p = 1$ quadrilateral elements. The flow field is initialized uniformly with free-stream values and the simulations are converged until the density residual norm decreases by at least 8 orders of magnitude. As spaND is capable of factorizing ill-conditioned systems [6], the initial CFL number is boosted to $\mathrm{CFL}^0 = 1000$ to accelerate convergence.

B. Scalability study of spaND

For scalability studies, we perform weak scaling analysis by varying the total number of elements in the mesh from 16k to 4M. As each first-order quadrilateral element has 4 nodal degrees of freedom and each node is represented by 4 conserved variables in 2D, the dimension of the resulting Jacobian matrix N varies from 260k to 67M. The number of processing cores[*] are scaled proportionally with respect to N. In this set of studies, a standard GMRES linear solver is used to solve the preconditioned linear system with a tolerance of 10^{-3} and maximum iteration count of 20. The statistics of all tests as well as tunable parameters used in the overall solver setup are tabulated in Table 1. The convergence histories are shown in Fig. 3 and all tests converge within 35 nonlinear iterations.

Table 1 Summary of statistics and tunable parameters of tests

Test	No. of elements	Jacobian dimension, N	No. of cores	SpaND tolerance, ε	$\ell_{\max}$	GMRES tolerance	GMRES max. steps
1	16,384	262,144	8	10^{-3}	10	10^{-3}	20
2	65,536	1,048,576	32	10^{-3}	12	10^{-3}	20
3	262,144	4,194,304	128	10^{-3}	14	10^{-3}	20
4	1,048,576	16,777,216	512	10^{-3}	16	10^{-3}	20
5	4,194,304	67,108,864	2,048	10^{-3}	18	10^{-3}	20

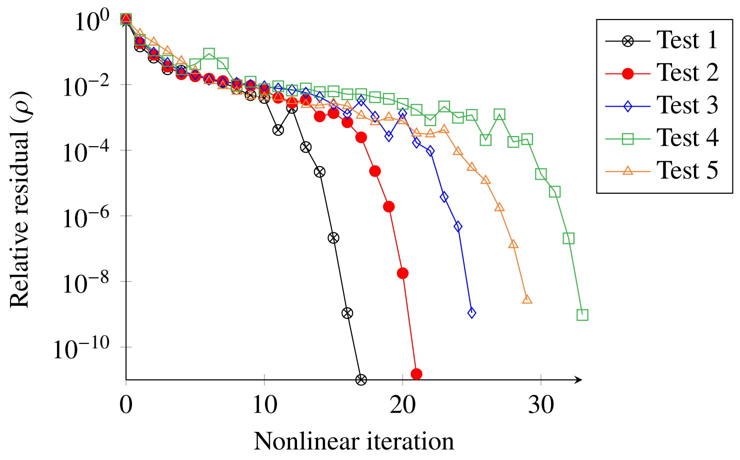


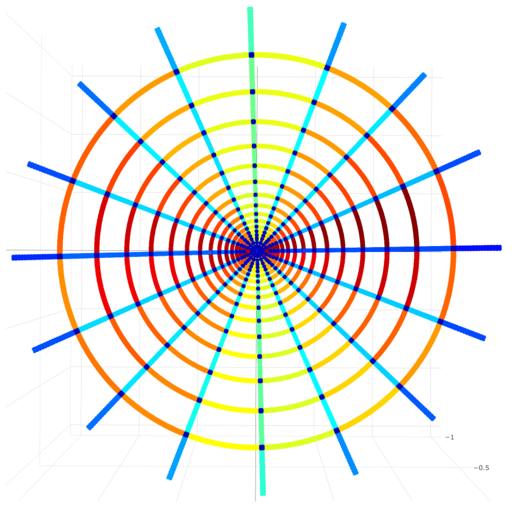
Fig. 3 Relative convergence versus number of nonlinear iterations of Tests 1-5.

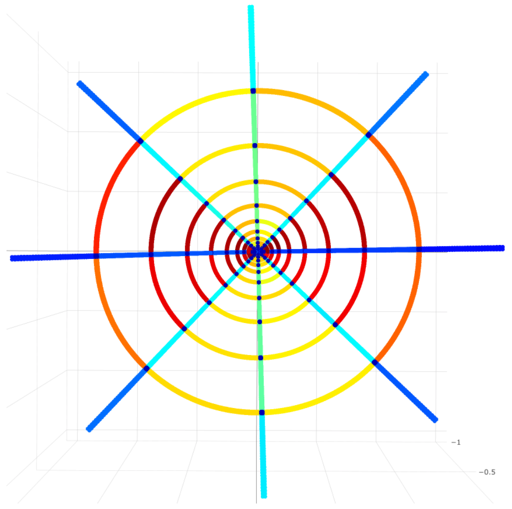
At each nonlinear iteration, spaND takes in the Jacobian matrix and generates an approximate factorization that is used as a preconditioner for linear solvers. Due to the nature of the discretization scheme, the Jacobian matrix is unsymmetric but with a symmetric sparsity pattern and a natural block structure. The mesh has a regular, structured

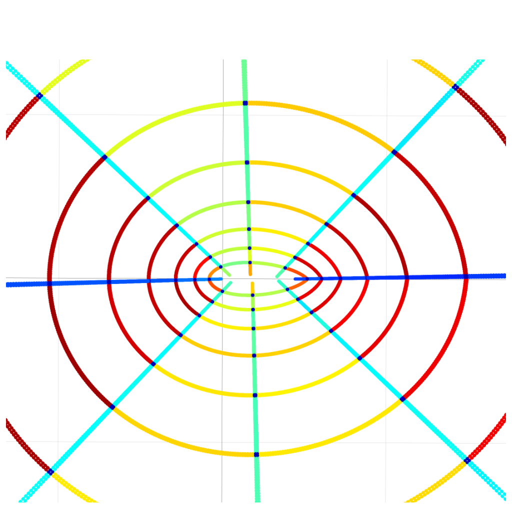
[*]Tests performed on a cluster equipped with dual-sockets and 16 cores Intel(R) Xeon(R) CPU E5-2670 0 @2.60GHz with 32GB of RAM per node

topology such that each interior element is connected to four adjacent neighboring elements. However, due to the need to refine near-wall flow, the mesh is highly non-uniform with a fine mesh around the airfoil and coarse mesh near far-field boundaries. During the pre-processing stage, we map the non-uniform mesh to a regular grid by assigning to each element a tuple (i, j) of integers with i increasing monotonically with the radius and j with the angle, respectively. The matrix is then partitioned using those (i, j) coordinates with a standard recursive bisection algorithm.

We then compute the factorization of the linear systems using spaND with TaskTorrent at every nonlinear step. We use partial pivoted LU as the block scaling algorithm and a tolerance of $\varepsilon = 10^{-3}$ for low-rank approximation. Since the distribution of matrix ranks is not known beforehand, spaND distributes an equal number of columns of the matrix to each MPI rank at the beginning of the algorithm, using a 1D mapping of columns to MPI ranks. This indicates that if matrix ranks generated from low-rank approximation are higher in some parts of the domain than in others, load balancing may be sub-optimal as different amounts of computation is required in different processing cores.


(a) $\ell = 4$. Ranks from 40 (blue) to 296 (brown), average of 162


(b) $\ell = 6$. Ranks from 47 (blue) to 353 (brown), average of 198


(c) $\ell = 6$, zoom on the airfoil.

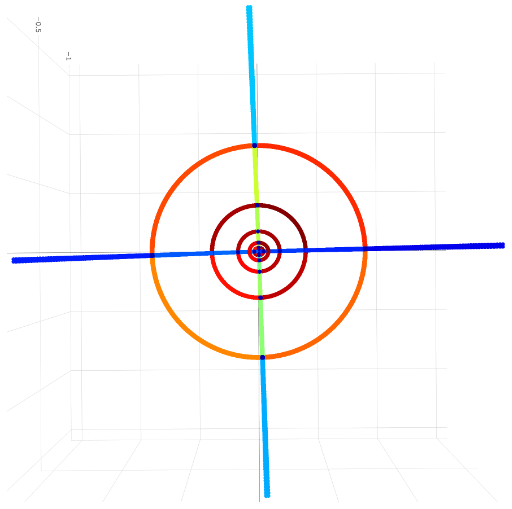
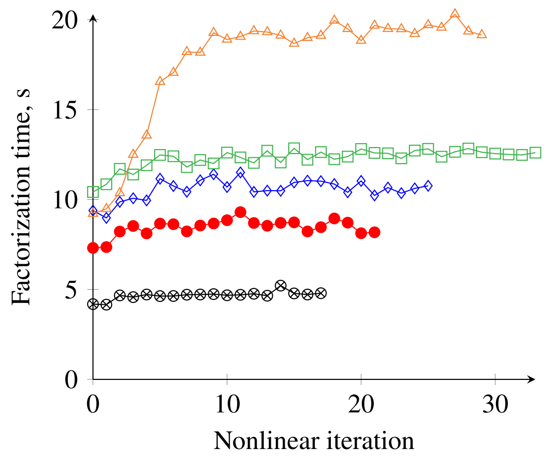

(d) $\ell = 8$. Ranks from 40 (blue) to 385 (brown), average of 231

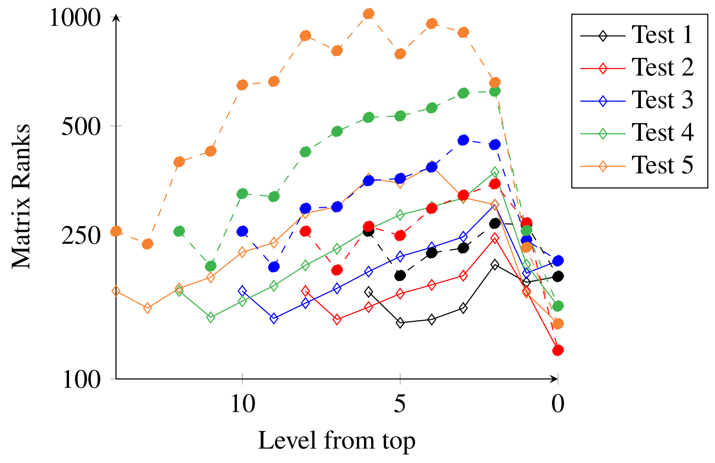
Fig. 4 Matrix rank distribution from rank-revealing QR at various level ℓ with $\ell_{\max} = 14$ in Test 3 at the last nonlinear iteration (iteration 25).

To investigate this, we visualize the matrix rank distribution throughout the whole computational domain. Fig. 4 shows a typical matrix rank distribution generated by spaND. We note that there is a strong directionality, with matrix ranks much higher in the flow direction, but smaller in the orthogonal direction. This presents some challenges since, as a consequence of wide range of matrix ranks, the load balancing is much less favorable.

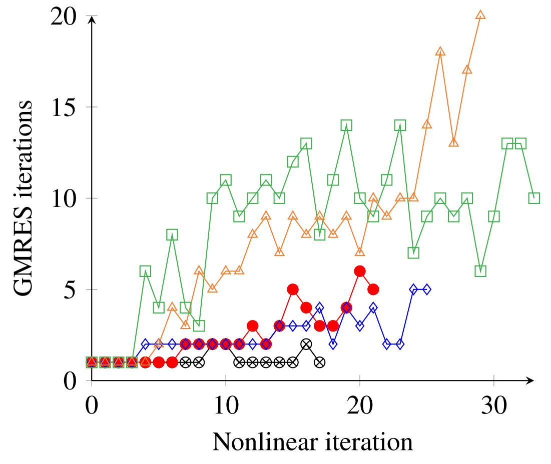
Fig. 5 presents all the results throughout all nonlinear iterations. Here, we make a few remarks on the scalability of spaND. The factorization time of computing the preconditioner, spaND, scales well with the problem size, albeit with an increase in the larger test case (Fig. 5a). The average matrix ranks grow slowly with the problem size whereas the maximum matrix ranks grow at a higher rate and are not localized to the low-level leaves (Fig. 5b). This can slow down spaND significantly since high ranks lead to longer sparsification time and high ranks towards the top of the hierarchical tree lead to poorer concurrency. We conjecture that the relatively high maximum ranks in larger test cases are degrading performance and causing the increase in factorization time. The number of GMRES steps remains very low at all problem sizes and does not vary much during the nonlinear convergence (Fig. 5c). The solve time also scales with the number of GMRES steps, indicating that applying the preconditioner scales well with the problem size (Fig. 5d).



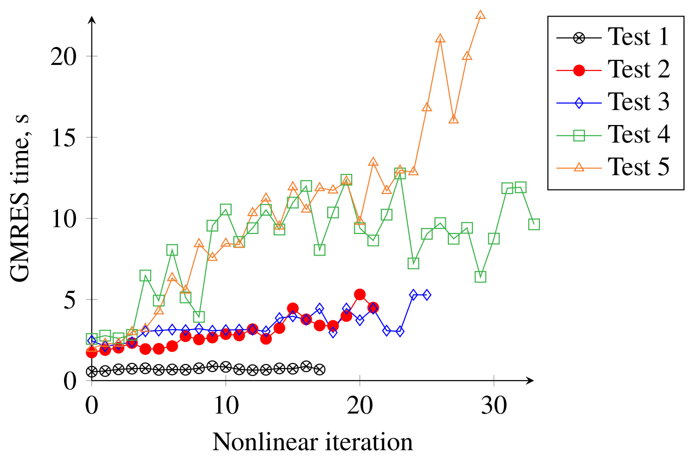
(a) Factorization time



(b) Ranks at the last nonlinear iteration. Average (solid line, diamond) and maximums (dashed line, circle).



(c) GMRES iterations



(d) Solve time

Fig. 5 Weak scaling results from 8 cores for Test 1 to 2048 cores for Test 5.

C. Scalability study of CAGMRES

As seen in the previous section, linear systems preconditioned by spaND require very few GMRES steps to reach convergence. The communication cost is therefore minimal, which justifies the use of a standard GMRES implementation. However, one could also adjust the tunable tolerance ε in spaND to achieve better performance at the expense of more GMRES iterations. On large machines, the communication overhead becomes significant and it requires CA techniques to achieve scalability.

For this purpose, we conduct an additional set of numerical experiments designed to stress the Krylov linear solver in order to demonstrate the scalability of CAGMRES. In this section, we focus on the scalability of the CAGMRES linear solver instead of the overall non-linear convergence. The tolerance parameter in spaND is increased to 10^{-1} and the tolerance of the CAGMRES solver is lowered to 10^{-10}. Maximum number of linear iterations is increased to 100 as well to collect meaningful performance data. Scalability tests are again in the form of weak scaling tests where the number of processing cores scale linearly with the problem size[†]. The statistics and tunable parameters of the tests used in this section are summarized in Table 2.

Table 2 Summary of statistics and tunable paramters of additional tests used in Sec. IV.C

Test	No. of elements	Jacobian dimension, N	No. of cores	SpaND tolerance, ε	$\ell_{\max}$	CAGMRES tolerance	CAGMRES max. steps
6	65,536	1,048,576	48	10^{-1}	12	10^{-10}	100
7	262,144	4,194,304	192	10^{-1}	14	10^{-10}	100
8	1,048,576	16,777,216	768	10^{-1}	16	10^{-10}	100

At each nonlinear iteration, the preconditioned linear system is solved using s-step CAGMRES with single-reduce re-orthogonalization scheme presented in Algorithm 2. A step size $s = 4$ is used to reduce the latency cost as only one global reduction is required for every s basis vectors. The effective linear iteration count is reduced by a factor of s correspondingly. Single-reduce re-orthogonalization scheme using two CGS steps and two CholQR steps is incorporated to tackle the numerical instability introduced by finite-precision implementation of s-step CAGMRES. This introduces additional computational cost but is found necessary as traditional block CGS in the original s-step CAGMRES implementation [5] has an orthogonality error that is proportional to the square of the condition number of the s block [19, 20]. A Newton basis is included for the same stability concern [21]. The shifts used for the Newton basis are computed using the Ritz value from s iterations of standard GMRES and are arranged in a Leja ordering for real arithmetic [21, 22]. The effect of different stability measures on orthogonality error are illustrated in Fig. 6.

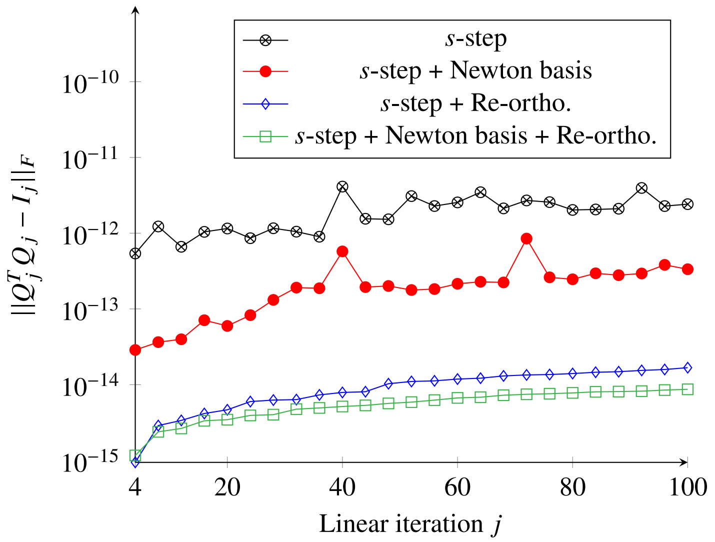


Fig. 6 Orthogonality error ($||Q_j^T Q_j - I_j||_F$) of CAGMRES with different stability measures for Jacobian matrix generated in Test 6 at nonlinear iteration 5. A step size of $s = 4$ is used such that the linear iteration j increases in increment of s.

Using s-step CAGMRES with single-reduce re-orthogonalization schemes and Newton basis, the total computational time for 100 linear iterations of CAGMRES at each nonlinear iteration in each test is collected. Fig. 7 shows the weak

[†]Tests performed on a cluster equipped with dual-sockets and 24 cores Intel(R) Xeon(R) CPU E5-2680v3 @2.50GHz with 128GB of RAM per node.

scaling results along with standard GMRES timings as baseline for comparison. We first highlight the near-constant timings of all components of CAGMRES (dashed line) except the solve step which corresponds to applying the preconditioner, spaND. Comparing to its GMRES counterpart, CAGMRES achieves scalability mainly due to its reduced number of global reductions which become more and more expensive on larger computing architecture. The orthogonalization step CGS takes less computational time than MGS as CGS can leverage on BLAS-3 matrix-matrix primitives that are more optimized than BLAS-2 matrix-vector kernels in MGS. The re-orthogonalization scheme introduces additional computational cost for stability but does not introduce extra communication cost. Applying the preconditioner takes about 80% of total time in all cases but grow slowly with respect to the problem size.

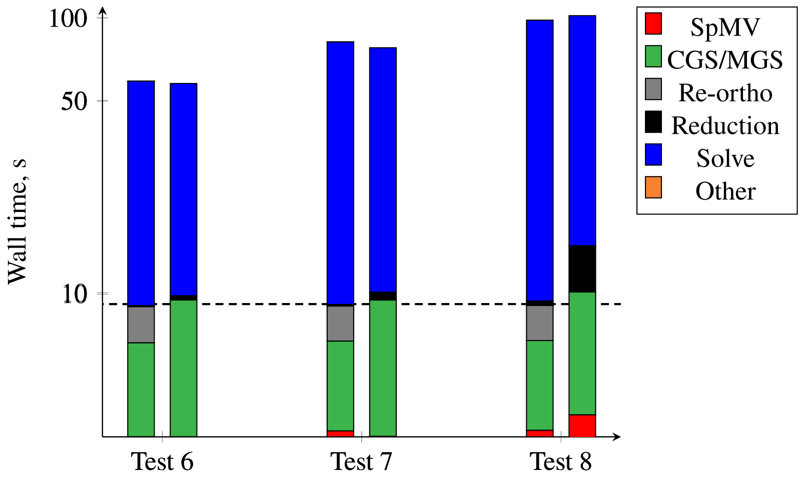


Fig. 7 Timing of various components of CAGMRES (left) and GMRES (right) for Tests 6-8 at nonlinear iteration 5. Step-size $s = 4$. (CGS/MGS: timing for arithmetic computation of CGS or MGS. Reduction: timing for MPI-based global reduction. Solve: timing for applying the preconditioner, spaND.)

V. Conclusions

This paper has described a new methodology for preconditioned scalable solvers for CFD workflows. The methodology consists of a hierarchical preconditioning strategy combined with communication-avoiding GMRES to enable scaling to very large numbers of processors in a weak-scaling sense. Both the preconditioning technique, spaND (with TaskTorrent), and the CAGMRES linear solver demonstrate near-linear weak scaling up to 2,048 cores in the context of a high-order DG solver within the SU2 framework. spaND shows scalable computational cost in the factorization of the Jacobian matrix while approximating the inverse with high accuracy, leading to constantly-small number of subsequent GMRES iterations. The CAGMRES solver developed minimizes communication overhead by employing the s-step technique to reduce latency and single-reduce re-orthogonalization scheme for stability.

The set of numerical experiments presented in this work focuses on h-refinement while keeping the Jacobian block structure constant. Future efforts will extend the scope of the scalability analysis to include p-refinement as well. The final objective aims at demonstrating scalable, asymptotic performance for large-scale turbulent flows with complex geometries.

Acknowledgements

Part of the computing resources for this runs executed in this paper were performed on the Stanford Research Computing Center cluster. We would like to thank Stanford University and the Stanford Research Computing Center for providing computational resources and support that contributed to these research results. Léopold Cambier was supported by a fellowship from Total SE. Zan Xu was supported by NASA Grant 80NSSC18M0152 from the NASA Transformational Tools and Technologies program.

References

[1] Stevens, R., Ramprakash, J., Messina, P., Papka, M., and Riley, K., "Aurora: Argonne's Next-Generation Exascale Supercomputer," 2019.

[2] Ang, J., Evans, K., Geist, A., Heroux, M., Hovland, P., Marques, O., McInnes, L., Ng, E., and Wild, S., "Report on the workshop on extreme-scale solvers: Transitions to future architectures," *Office of Advanced Scientific Computing Research, US Department of Energy*, 2012, pp. 8–9.

[3] Baker, A. H., Falgout, R. D., Kolev, T. V., and Yang, U. M., "Scaling hypre's multigrid solvers to 100,000 cores," *High-Performance Scientific Computing*, Springer, 2012, pp. 261–279.

[4] Saad, Y., "ILUT: A dual threshold incomplete LU factorization," *Numerical linear algebra with applications*, Vol. 1, No. 4, 1994, pp. 387–402.

[5] Hoemmen, M. F., "Communication-avoiding Krylov subspace methods," 2010.

[6] Cambier, L., Chen, C., Boman, E. G., Rajamanickam, S., Tuminaro, R. S., and Darve, E., "An algebraic sparsified nested dissection algorithm using low-rank approximations," *SIAM Journal on Matrix Analysis and Applications*, Vol. 41, No. 2, 2020, pp. 715–746.

[7] Hesthaven, J. S., and Warburton, T., *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*, Springer Science & Business Media, 2007.

[8] Roe, P. L., "Approximate Riemann solvers, parameter vectors, and difference schemes," *Journal of computational physics*, Vol. 43, No. 2, 1981, pp. 357–372.

[9] Palacios, F., Alonso, J., Duraisamy, K., Colonno, M., Hicken, J., Aranake, A., Campos, A., Copeland, S., Economon, T., Lonkar, A., et al., "Stanford university unstructured (su 2): an open-source integrated computational environment for multi-physics simulation and design," *51st AIAA aerospace sciences meeting including the new horizons forum and aerospace exposition*, 2013, p. 287.

[10] Palacios, F., Economon, T. D., Aranake, A., Copeland, S. R., Lonkar, A. K., Lukaczyk, T. W., Manosalvas, D. E., Naik, K. R., Padron, S., Tracey, B., et al., "Stanford university unstructured (SU2): Analysis and design technology for turbulent flows," *52nd Aerospace Sciences Meeting*, 2014, p. 0243.

[11] Economon, T. D., Palacios, F., Copeland, S. R., Lukaczyk, T. W., and Alonso, J. J., "SU2: An Open-Source Suite for Multiphysics Simulation and Design," *AIAA Journal*, Vol. 54, No. 3, 2015, pp. 828–846. https://doi.org/10.2514/1.J053813, URL http://dx.doi.org/10.2514/1.J053813.

[12] Choi, J. H., Alonso, J. J., and van der Weide, E., "A simple and robust shock-capturing approach for discontinuous Galerkin discretizations," *Energies*, Vol. 12, No. 14, 2019, p. 2651.

[13] Sagebaum, M., Albring, T., and Gauger, N. R., "High-Performance Derivative Computations using CoDiPack," *arXiv preprint arXiv:1709.07229*, 2017. URL https://arxiv.org/abs/1709.07229.

[14] Hestenes, M. R., Stiefel, E., et al., "Methods of conjugate gradients for solving linear systems," *Journal of research of the National Bureau of Standards*, Vol. 49, No. 6, 1952, pp. 409–436.

[15] Saad, Y., and Schultz, M. H., "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on scientific and statistical computing*, Vol. 7, No. 3, 1986, pp. 856–869.

[16] Cambier, L., Qian, Y., and Darve, E., "TaskTorrent: a Lightweight Distributed Task-Based Runtime System in C++," *arXiv preprint arXiv:2009.10697*, 2020.

[17] Ghysels, P., Ashby, T. J., Meerbergen, K., and Vanroose, W., "Hiding global communication latency in the GMRES algorithm on massively parallel machines," *SIAM journal on scientific computing*, Vol. 35, No. 1, 2013, pp. C48–C71.

[18] Yamazaki, I., Hoemmen, M., Luszczek, P., and Dongarra, J., "Improving performance of GMRES by reducing communication and pipelining global collectives," *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2017, pp. 1118–1127.

[19] Yamazaki, I., Thomas, S., Hoemmen, M., Boman, E. G., Świrydowicz, K., and Elliott, J. J., "Low-synchronization orthogonalization schemes for s-step and pipelined Krylov solvers in Trilinos," *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, 2020, pp. 118–128.

[20] Swirydowicz, K., Langou, J., Ananthan, S., Yang, U., and Thomas, S., "Low synchronization Gram–Schmidt and generalized minimal residual algorithms," *Numerical Linear Algebra with Applications*, 2020.

[21] Bai, Z., Hu, D., and Reichel, L., "A Newton basis GMRES implementation," *IMA Journal of Numerical Analysis*, Vol. 14, No. 4, 1994, pp. 563–581.

[22] Reichel, L., "Newton interpolation at Leja points," *BIT*, Vol. 30, No. 2, 1990, pp. 332–346.