Task-parallel in situ temporal compression of large-scale computational fluid dynamics data

Heather Pacella¹, Alec Dunton², Alireza Doostan³ and Gianluca Iaccarino¹

Abstract
Present day computational fluid dynamics (CFD) simulations generate considerable amounts of data, sometimes on the order of TB/s. Often, a significant fraction of this data is discarded because current storage systems are unable to keep pace. To address this, data compression algorithms can be applied to data arrays containing flow quantities of interest (QoIs) to reduce the overall required storage. The matrix column interpolative decomposition (ID) can be implemented as a type of lossy compression for data matrices that factor the original data matrix into a product of two smaller factor matrices. One of these matrices consists of a subset of the columns of the original data matrix, while the other is a coefficient matrix which approximates the original data matrix columns as linear combinations of the selected columns. Motivating this work is the observation that the structure of ID algorithms makes them well suited for the asynchronous nature of task-based parallelism: they can operate independently on subdomains of the system of interest and, as a result, provide varied levels of compression. Using the task-based Legion programming model, a single-pass ID algorithm (SPID) for CFD applications is implemented. Performance studies, scalability, and the accuracy of the compression algorithm are presented for a benchmark analytical Taylor-Green vortex problem, as well as large-scale implementations of both low and high Reynolds number (Re) compressible Taylor-Green vortices using a high-order Navier-Stokes solver. In the case of the analytical solution, the resulting compressed solution was rank-one, with error on the order of machine precision. For the low-Re vortex, compression factors between 1000 and 10,000 were achieved for errors in the range $10^{-2}$–$10^{-3}$. Similar error values were seen for the high-Re vortex, this time with compression factors between 100 and 1000. Moreover, strong and weak scaling results demonstrate that introducing SPID to solvers leads to negligible increases in runtime.

Keywords
lossy data compression, high-performance computing, interpolative decomposition, low-rank approximation

Introduction and motivation
Advances in supercomputing over the past few decades have introduced a memory bottleneck. Soon-to-be-deployed Exascale computers are expected to offer a 1,000–10,000-fold increase in floating point performance, but provide a relatively mere 10–100-fold increase in available disk memory, working memory, and access speed (Ang et al. 2012). This asymmetric technological progress gives rise to the following issue: floating point operations (FLOPs) are cheap, while memory, communication, and input/output (I/O) are not. This general trend has been emphasized in other reports such as (Amarasinghe et al. 2009; Ashby et al. 2010; Sprague et al. 2017; Asch et al. 2018; Gerber et al. 2018) as well as in the design of next generation Exascale systems (Kunkel et al. 2014).

The amount of data generated on these computers can easily exceed 1 TB/s in large-scale simulations of complex systems, for example, flow control for wing design (Rasquin et al. 2014). If this is not reduced, storage systems will easily become overloaded and users will not be able to use the

¹Department of Mechanical Engineering, Stanford University, Stanford, CA, USA
²Department of Applied Mathematics, University of Colorado Boulder, Boulder, CO, USA
³Smead Aerospace Engineering Sciences, University of Colorado Boulder, Boulder, CO, USA

Corresponding author:
Heather Pacella, Department of Mechanical Engineering, Stanford University, Building 500 488 Escondido Mall Stanford, CA 94305, USA.
Email: hpacella@stanford.edu
simulation for visualization, analysis, or other post-processing operations. This issue has inspired work in data compression, in which a memory-reduced version of an original dataset is stored, with the hope that the compressed format maintains an accurate representation of the original dataset.

Data compression algorithms may be categorized as lossless or lossy. Lossless compression methods enable exact reconstruction of datasets, but with a small compression factor, defined as the ratio of the size of the original dataset to that of the compressed dataset. Due to the exorbitant size of data being generated on modern supercomputers, particularly for the category of physical simulations addressed in this work, we do not explore lossless compression methods. For the interested reader, an extensive review of lossless compression techniques can be found in Li et al. (2018).

Lossy compression methods do not enable exact reconstruction of datasets. They do, however, allow for much larger compression factors. Algorithms for lossy data compression rely on a diverse set of tools to construct memory-efficient representations of datasets. Such methods include bit truncation (Gong et al. 2012) and predictive coding techniques. Notable examples of predictive coders include Compvox (Fowler and Yagel 1994), SZ (Di and Cappello 2016; Tao et al. 2017; Liang et al. 2018), FPZIP (which supports both lossless and lossy compression (Lindstrom and Isenburg 2006)), Lorenzo (Ibarria et al. 2003), and Isabela (Lakshminarasimhan et al. 2011; Lehmann and Jung 2014). Transform-based methods entail the truncation of a set of coefficients obtained via, for example, the discrete Legendre (Otero et al. 2018; Marin et al. 2016), discrete cosine (Yeo and Liu 1995), and wavelet transforms (Cohen et al. 1992; Farge 1992; Strang and Nguyen 1996). Other examples of transform-based algorithms for compression include ZFP (Lindstrom 2014), the Karhunen-Loeve transform (Loeve 1977; Therrien 1992), and tensor approximations (Hitchcock 1927; Tucker 1966; Kroonenberg and De Leeuw 1980; De Lathauwer et al. 2000; Vannieuwenhoven et al. 2012; Austin et al. 2016).

Recent work has also explored the potential for deep learning to enable in situ compression of turbulent flows (Glaws et al. 2020). In this present work, low-rank matrix approximations—a transform-based method—are investigated for lossy compression.

**Contribution of this work**

Low-rank matrix methods have been used for the compression of large-scale simulation data in works such as Azizie et al. (2019). Computing low-rank matrix approximations in which simulation data arrives into working memory online has been addressed in Brand (2006), Zimmermann et al. (2018), and Tropp et al. (2019). The present effort is focused on using low-rank matrix approximation to build in situ compression methods which exploit task-based parallelism to achieve high compression and concurrency on heterogeneous modern computing architectures.

To this end, we use the matrix interpolative decomposition (ID) for low-rank approximation (Cheng et al. 2005; Halko et al. 2011). Our specific approaches follow directly from ID-based CFD compression algorithms presented in Dunton et al. (2020). The matrix ID is inserted into and applied to data from computational fluid dynamics (CFD) applications using Legion (Bauer et al. 2012). Legion is a data-centric parallel programming and runtime system designed for high-performance computing applications. It allows for heterogeneous CFD-ID applications to be computationally efficient via asynchronous, task-based parallel execution. We also detail how additional parallelism can be extracted through Legion’s custom mapping interface, which allows the user to execute a single application code on multiple processor types. Although other algorithms, such as the randomized SVD (Halko et al. 2011; Yu et al. 2017), are also available for compression in simulation, we focus on deterministic ID methods for reasons discussed in the Interpolative Decomposition section.

The rest of the paper is organized as follows. In the Interpolative Decomposition section, we provide background and derivations of the matrix ID algorithms used for lossy data compression. In the ID Methods and Task-Based Parallelism section, we outline how ID methods may be incorporated into a task-based parallel environment. In the Legion Programming Model section, we outline the Legion programming model, which enables task-based parallelism. In the Numerical Experiments section, we present numerical experiments from applying the ID to (1) an analytical Taylor-Green vortex solution of the incompressible Navier-Stokes equations and (2) large-scale implementations of both low-Re and high-Re compressible Taylor-Green vortices using a high-order Navier-Stokes solver. In the Conclusions and Future Work section, we draw conclusions and propose future avenues of research. Theoretical results relevant to the Interpolative Decomposition section are provided in the Appendix 1.

**Interpolative decomposition**

We begin our background on the matrix ID by introducing conventions used throughout this work. We assume that a data matrix $A \in \mathbb{R}^{m \times n}$ is arranged such that each of its columns corresponds to a physical quantity of interest (QoI), such as pressure or velocity measured on a grid of size $m$. The QoI is obtained on a discrete set of locations in the physical domain, possibly without specific ordering, for example, a flattened representation of an unstructured grid.
The index of a column is assumed to correspond to a specific time-instance in the simulation from which the data is generated, as shown in Figure 1. Corresponding notation conventions used in this work are also provided in Table 1.

Importantly, for large-scale applications we never form the matrix $A$ explicitly, as we assume that $A$ is too large to be stored in working memory; however, we will continue to refer to $A$ throughout the paper for clarity.

If the spatial or temporal variation of the QoI is smooth, the matrix $A$ may admit a low-rank structure; its columns may contain redundant information. An assumption crucial to this work is that $A$ is *numerically low rank*, that is, its singular value decay is sharp and $A$ may therefore be approximated with small reconstruction error. Equivalently, the QoI may be accurately reconstructed on a low-dimensional linear subspace.

We assume that there are $k$ columns of $A$ comprising this low-dimensional linear subspace and that $k$ is much smaller than the original column dimension $n$. This assumption may be rephrased as follows. For each column of $A$, which we denote $a_i = A(:,i-1), k \ll n$, with $v_i$ such that $\|v_i\|_2 = 1$, and $\epsilon$ such that $0 < \epsilon/\|a_i\|_2 \ll 1$, we may write

$$ a_i = \sum_{j=0}^{k-1} c_{ij} a_I(j) + \epsilon v_i $$

(1)

where $I \subseteq \{0, \ldots, n-1\}$ and $|I| = k$. In words, any column of $A$ may be approximated by a linear combination of a fixed subset of $k$ of its columns, indexed by $I$. This constitutes a $k$-rank approximation to the matrix $A$, where $k$ is the target rank of the approximation.

The coefficients $c_{ij}$, comprising the vector $c$, are obtained via the least-squares problem

$$ c_i = \arg\min_{\hat{c}} \left\| a_i - \sum_{j=0}^{k-1} \hat{c}_{I(j)} a_I(j) \right\|_2 $$

(2)

The approximations of each column $a_i$ may be concatenated to form an approximation to the original data matrix $A$ via the matrix-matrix product

$$ A \approx A(:,I)C $$

(3)

where $C \in \mathbb{R}^{k \times n}$ is the *coefficient matrix* whose entries are comprised of the least-squares coefficients $c_{ij}$ corresponding to each column $a_i$,

$$ C = [c_0 \ c_1 \ \cdots \ c_{k-2} \ c_{k-1}] $$

(4)

The matrix $A(:,I) \in \mathbb{R}^{m \times k}$, which is a subset of the columns of $A$ indexed by $I$, is called the *column skeleton* of $A$. The decomposition (3) is the so-called column interpolative decomposition (column ID) of a matrix $A$ (Cheng et al. 2005). For the interested reader, properties of the column ID are enumerated in Lemma A.1 in the Appendix.

The column ID possesses favorable properties relative to orthogonal decompositions such as the SVD, among them (Martinsson 2018):

- It yields factor matrices which are sparsity and entry-wise non-negativity preserving; if $A$ is sparse or entry-wise non-negative, so is $A(:,I)$.

---

**Table 1. Notation conventions used in this work.**

| Symbol | Meaning |
|--------|---------|
| $A$    | Data matrix |
| $B$    | Data matrix sketch |
| $C$    | Column ID coefficient matrix |
| $M$    | Interpolation operator |
| $a_i$  | $i^{th}$ column of $A$, i.e., $A(:,i-1)$ |
| $I$    | Column index vector |
| $m$    | Size of fine grid mesh |
| $m_c$  | Size of coarse grid mesh |
| $n$    | Number of snapshots |
| $t$    | Number of points in stencil |
| $A(:,I)$ | Column skeleton of data matrix |
| $A(J,:)$ | Row skeleton of data matrix |
| $A(:,k-1)$ | First $k$ columns of $A$ |
| $A(:,k)$ | Last $n-k$ columns of $A$ |
| $+$    | Moore-Penrose pseudo-inverse |
| $\|\cdot\|_2$ | Matrix spectral norm |
| $\|\cdot\|_F$ | Matrix Frobenius norm |
Pacella et al.

391

The column ID is self-expressive; the matrix is decomposed in terms of one of its own columns. For domain scientists, this may make the factor matrices in the ID more interpretable than, for example, the singular value decomposition (SVD).

The column skeleton extracted by column ID preserves physical constraints from the original data matrix.

There are numerous schemes to produce a column ID; in this work, we deploy the column pivoted QR based on a modified Gram-Schmidt scheme (Golub and Van Loan 2012). Approaches such as the rank-revealing QR (Gu and Eisenstat 1996) provide stronger theoretical guarantees than a basic pivoted QR, though these guarantees come at the cost of increased computational expense. Other methods available for computing decompositions comprised of a column subset and corresponding coefficient matrix may be found in, for example, Mahoney and Drineas (2009), Elhamifar and Vidal (2009), Dyer et al. (2015), and Perry et al. (2019).

In order to form the ID, we first form a column pivoted QR of \( A \)

\[
AZ = QR
\]  

where \( Z \) is a permutation encoding the pivots, \( Q \in \mathbb{R}^{m \times n} \) is a unitary matrix, and \( R \in \mathbb{R}^{n \times n} \) is an upper triangular matrix. We then partition \( Q \) and \( R \) into blocks such that

\[
Q = \begin{bmatrix} Q_{11} & Q_{12} \end{bmatrix}
\]  

where \( Q_{11} \in \mathbb{R}^{m \times k} \) and \( Q_{12} \in \mathbb{R}^{m \times (n-k)} \), and

\[
R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}
\]  

where \( R_{11} \in \mathbb{R}^{k \times k} \), \( R_{12} \in \mathbb{R}^{k \times (n-k)} \), and \( R_{22} \in \mathbb{R}^{(n-k) \times (n-k)} \).

We then form a low-rank approximation of \( A \)

\[
AZ \approx Q_{11} [R_{11} \ R_{12}]
\]  

\[
= Q_{11} R_{11} [I \ R_{11}^T R_{12}]
\]

As \( Q_{11} R_{11} = AZ(\cdot, I) \), it follows that

\[
A \approx A(\cdot, I) [I \ R_{11}^T R_{11}] Z^T
\]  

Setting

\[
C = [I \ R_{11}^T R_{11}] Z^T
\]

we obtain a column ID of \( A \)

\[
A \approx A(\cdot, I) C
\]

We present the Column ID as Algorithm 1. Our approach takes as input an approximation rank in Step 2 and then computes a rank-\( k \) pivoted QR decomposition of the input matrix in Step 3 based on a modified Gram-Schmidt procedure—denoted \( mgsq \)r within Algorithm 1—see Golub and Van Loan (2012) for more details. Steps 4 through 6 demonstrate the formation of the coefficient matrix \( C \). This, combined with the index vector obtained in Step 3, enables construction of the final approximation of our matrix.

Algorithm 1 Column ID \( A \approx A(\cdot, I) C \) (Cheng et al. 2005)

1: procedure id \( (A \in \mathbb{R}^{m \times n}) \)
2: \( k \leftarrow \) approximation rank
3: \( Q, R I \leftarrow mgsqr(A, k) \)
4: \( R_{11} \leftarrow R(:, k-1,:k-1) \)
5: \( R_{12} \leftarrow R(:, k-1,k:) \)
6: \( Z \leftarrow I_{m}([I, I^T]) \)
7: \( C \leftarrow [I_{k} R_{11}^T R_{12}] Z^T \)
8: return \( A(\cdot, I), C \)
9: end procedure

Computing the ID using a pivoted QR-based approach costs at most \( O(mn \min(m, n)) \) FLOPs; the cost is typically closer to \( O(mnk) \). Several strategies are available to reduce the computational cost of computing the column ID of \( A \). Some schemes are based on the observation that the column indices \( I \) and coefficient matrix \( C \) may be computed on a lower dimensional matrix sketch of \( A \), herein denoted \( B \in \mathbb{R}^{\ell \times n} \) with \( k \ll \ell \ll m \). Using the sketch to form the ID works if \( B \) is constructed from \( A \) such that it encodes a sufficient amount of geometric information from the range of the original matrix \( A \). More precisely, let \( B^*B \) be the orthogonal projection onto the range of \( B \). Then, we may seek a sketch \( B \) which satisfies

\[
\|A - A(B^*B)\|_F \leq \epsilon \|A\|_F, \quad \epsilon = 2, \quad F
\]

In words, if \( A \) projected onto the range of \( B \) is close in proximity to \( A \) with respect to, for example, the spectral or Frobenius norm, then \( B \) will perform well as a matrix sketch in a column ID. To form a column ID using a sketch \( B \), we first compute its column ID

\[
B \approx B(\cdot, I_B) C_B
\]

\( I_B \) and \( C_B \) may then be used to construct a column ID for \( A \)

\[
A \approx A(\cdot, I_B) C_B
\]

The cost of computing the index vector \( I_B \) and coefficient matrix \( C_B \) is reduced significantly relative to computing \( I \) and \( C \) from the full matrix as in (12). Using \( B \) in place of \( A \) also increases parallel efficiency in ID algorithms, as we demonstrate in the Numerical Experiments section.

Forming the matrix \( B \) has been an active area of research for the past few decades. Many approaches use random linear embeddings to map data into lower dimensions,
referred to as *random projections*. In one example of random projection, the matrix \( A \) is left-multiplied by a matrix \( \Omega \in \mathbb{R}^{m \times n} \) whose entries are independent and identically distributed Gaussian random variables. This generates the sketch matrix \( B = \Omega A \) (Liberty et al. 2007). For a thorough review on the subject, we refer the interested reader to (Woodruff 2014).

Following Dunton et al. (2020), this work takes a different approach, focusing on *deterministically* projecting \( A \) to a lower dimension by constructing a *coarse grid representation*. This provides a sketch of the fine grid data by subsampling the spatial domain with, for example, direct subsampling or weighted averaging. The resulting coarse grid exploits the smoothness of the QoI in the spatial domain, as opposed to the data-agnostic nature of randomized sketches. One deterministic strategy entails subsampling the rows of \( A \), and therefore the domain of the corresponding QoI, to obtain the sketch matrix \( B \). Computing the column ID

\[
B \approx \tilde{A}(\mathcal{I}_B)C_B = A(\mathcal{J}, \mathcal{I}_B)C_B
\]  

yields the components required to generate a viable column ID of \( A \)

\[
A \approx \tilde{A}(\mathcal{I}_B)C_B \tag{17}
\]

Forming a column ID of \( A \) from \( B = \tilde{A}(\mathcal{J}, :) \) in this way constitutes the subsampled ID (SubID), herein presented as Algorithm 2. The procedure is nearly identical that of the standard Column ID, except we subsample the input matrix \( A \) in Step 3 prior to the pivoted QR computation. The index vector and coefficient matrix computed from the subsampled matrix are used to form the final approximation of our matrix. Accuracy guarantees for SubID are provided in the appendix; the interested reader is referred to Dunton et al. (2020) for more details.

**Algorithm 2 SubID** \( A \approx \tilde{A}(\mathcal{I}_B)C_B \) (Dunton et al. 2020)

1: **procedure** SubID \( (A \in \mathbb{R}^{m \times n}) \)
2: \( k \leftarrow \) approximation rank
3: \( B \leftarrow \text{subsample}(A) \)
4: \( Q_B, R_B, \mathcal{I}_B \leftarrow \text{mgsvr}(B, k) \)
5: \( R_{11} \leftarrow R_B(:, k - 1, : - k + 1) \)
6: \( R_{12} \leftarrow R_B(:, k - 1, : ) \)
7: \( Z_B \leftarrow I_{\mathcal{I}_B}(\mathcal{I}_B, \mathcal{I}_B') \)
8: \( C_B \leftarrow [I_k | R_{11}' R_{12} Z_B'] \)
9: return \( \tilde{A}(\mathcal{I}_B)C_B \)
10: **end procedure**

Computing a column ID using SubID requires two passes over the data matrix: one pass to compute the indices \( \mathcal{I} \) and coefficient matrix \( C \), then a second pass to obtain the column skeleton \( \tilde{A}(\mathcal{I}, :) \). When the amount of data generated in the solver step (which generates the data matrix) exceeds the storage available in RAM or disk, the user may be required to run a second simulation in order to construct \( A(\mathcal{I}, :) \). In order to avoid this second pass over \( A \)—and potentially the need to run a second simulation—we also present a single-pass variation of SubID: the so-called single-pass ID (SPID) (Dunton et al. 2020). In this variation of column ID, instead of generating the approximation \( A \approx \tilde{A}(\mathcal{I}_B)C_B \) as in SubID, we approximate \( A \) as

\[
A \approx MB(\mathcal{I}_B)C_B \tag{18}
\]

where \( M \) is an interpolation operator which interpolates the coarse grid QoI matrix \( B \) onto the fine grid. Accuracy guarantees for SPID are provided in the appendix; the interested reader is also referred to Dunton et al. (2020) for more details.

We present SPID as Algorithm 3. This algorithm only differs from SubID in that in the final step, instead of returning \( A(\mathcal{I}_B)C_B \), we return the column skeleton of the coarsened data matrix \( B(\mathcal{I}_B) \) and the interpolation operator \( M \). Both the column indices \( \mathcal{I}_B \) and coefficient matrix \( C_B \) are computed from the coarse grid data matrix \( B \).

**Algorithm 3 SPID** \( A \approx MB(\mathcal{I}_B)C_B \) (Dunton et al. 2020)

1: **procedure** SPID \( (A \in \mathbb{R}^{m \times n}) \)
2: \( k \leftarrow \) approximation rank
3: \( B \leftarrow \text{subsample}(A) \)
4: \( Q_B, R_B \)
5: \( R_{11} \leftarrow R_B(:, k - 1, : - k + 1) \)
6: \( R_{12} \leftarrow R_B(:, k - 1, : ) \)
7: \( Z_B \leftarrow I_{\mathcal{I}_B}(\mathcal{I}_B, \mathcal{I}_B') \)
8: \( C_B \leftarrow [I_k | R_{11}' R_{12} Z_B'] \)
9: return \( B(\mathcal{I}_B) \), \( C_B \), \( M \)
10: **end procedure**

The column ID enables data compression due to the fact that the factor matrices are comprised of a total of \( k(m+n) \) entries—in the case of SPID this is even smaller—while the original matrix \( A \) contains \( mn \) elements. Under the assumption that \( A \) is numerically low rank, we may achieve an accurate column ID low-rank approximation with \( k \ll m \) and \( k \ll n \), and thus \( k(m+n) \ll mn \). Therefore, if \( A \) is numerically low-rank, storing \( A(\mathcal{I}, :) \) and \( C \) in lieu of \( A \) will yield significant memory savings. We measure these savings using a metric called the compression factor (CF), which we define

\[
CF = \frac{mn}{k(m+n)} \tag{19}
\]

We provide results for computational complexity, disk storage, and RAM usage for the three algorithms in Table 2. We observe that SubID and SPID both offer improvements over ID in terms of complexity, reducing the complexity from \( O(mnk) \) to \( O(mnk) \) with \( m_1 \ll m \). SPID also decreases the amount of memory used to store the compressed data to disk. In terms of RAM usage, SubID is slightly more efficient than SPID due to the interpolation step required in SPID.
Remark 2.1. Although we have used the convention that the rank of the ID approximation is fixed to be $k$ for simplicity, the mgsqr subroutine may be augmented to adaptively determine the rank of the approximation. Once the approximation reaches a given tolerance, the method terminates and returns a target rank as output, along with a corresponding column skeleton and coefficient matrix.

ID methods and task-based parallelism

In this section, we outline the implementation of SubID and SPID as compressors of simulation data in a task-based parallel environment. We begin with an overview of parallelism in CFD codes, followed by parallelism in ID codes, then provide a description of how an ID routine may be embedded in a CFD code, and finally a detailed description of the implementation.

Parallelism of CFD codes

Computational fluid dynamics (CFD) simulations are concerned with the numerical solution of a set of partial differential equations (PDEs) on a discretized grid representation of the physical domain of interest. For large simulations, this discretized domain is partitioned into subdomains (blocks) of data, which are then distributed across the available computing resources in a process called domain decomposition. This strategy of extracting concurrency is referred to as data-parallelism. Each “step” of the simulation consists of two coupled processes. First, identical sets of instructions are executed on each block (independent of the other blocks). Second, a communication stage, where relevant data is exchanged between the blocks of the domain (e.g., information from stencil points for a finite difference method), is performed. This “back-and-forth” repeats until the simulation is completed. Often, these simulations are implemented using a synchronous or bulk-synchronous programming model; that is, a barrier is inserted after each step of the simulation to prevent the simulation from proceeding until all current processes are complete and have exchanged the information required by neighboring domains to continue the computations (see Figure 2).

Table 2. Computational complexity of ID, subsampled ID and single-pass ID with input matrix of size $m \times n$ and target rank $k$. Variable $m_c \ll m$ represents the row dimension of the input matrix after subsampling. $t$ is the size of the stencil for the interpolation scheme. Disk storage is given as the total number of matrix entries stored following execution of the algorithm.

| Method       | Computational complexity | Disk storage | RAM usage |
|--------------|--------------------------|--------------|-----------|
| ID (Algorithm 1) | $O(mnk)$                | $k(m+n)$     | $k(m+n) + mn$ |
| SubID (Algorithm 2) | $O(m_cnk)$             | $k(m+n)$     | $k(m_c+n) + m(cn)$ |
| SPID (Algorithm 3) | $O(m_cnk)$             | $k(m_c+n)$   | $k(m_c+n) + m(cn) + tm$ |

Figure 2. Flowchart for a general CFD process; processors $P_0, P_1, \ldots, P_N$ at the $i^{th}$ time step perform required calculations, exchange data, then synchronize before continuing on to the next time step.

Parallelism of ID methods

From a computational perspective, ID differs greatly from CFD simulations. ID compression algorithms can be applied independently to blocks of arbitrary sizes; no communication is required between these blocks to perform the data compression. This means that the user does not need to devise a separate domain decomposition strategy for the ID; the existing flow solver decomposition is suitable. Additionally, because there is no communication involved between the subdomains, the ID compression can begin as soon as all required data has been computed, without the need for a synchronization stage. So, if the ID compression is implemented in such a way that it begins while the flow solver is still running, the computations required to perform the ID analysis on the most recent solution time step can be performed while waiting for the data synchronization.
required to compute the flow solver update for the next simulation step. Thus, the overhead cost of performing the ID approximation can be reduced or hidden if the computing system has a large amount of communication latency (as is often the case for extremely large simulations).

In addition to increasing computational efficiency, applying ID individually to subdomains of the data can help to increase the overall compression. This is due to the fact that the data corresponding to some of the subdomains may be locally low-rank, that is, on individual subblocks, but globally high-rank. As an example, consider Figure 3, whose color map displays the values of the entries of a simple, low-rank test matrix. This matrix is deliberately constructed to highlight the compressive benefits of domain partitioning prior to ID computation.

This test matrix, which has a rank of 48, was partitioned into various submatrices along the rows (representing typical partitions in the spatial domain), and the ID algorithm was applied to each individual partition with a target rank of 50. The overall compression factor was then computed. Table 3 shows the rank of the resulting ID approximations for each partition, and Figure 4 shows their corresponding compression factors. As the figures demonstrate, partitioning this matrix and independently applying ID to these submatrices increased the overall compression factor; when the matrix is partitioned into 5 submatrices, the overall ID compression factor is almost 70% larger than that of the single partition. Though these results are for a specific test matrix, this behavior can be generally observed in CFD applications due to the structure of the relevant data matrices. In these matrices, adjacent row entries correspond to adjacent points in the spatial domain, and adjacent column entries correspond to adjacent points in the temporal domain. Because of this, variation of physical QoIs (whose values make up the entries of the matrix) is less drastic over local regions of the matrix. Therefore, partitioning the data matrix is a favorable approach because it allows the column ID to extract low rank structure more efficiently.

Additionally, if the QoI across the domain is highly variable, the number of calculations required to find the ID low-rank approximation for each block can vary by a large amount when using a fixed error tolerance setting instead of a fixed rank. For both of these reasons, conventional synchronous or barrier-synchronous parallelization models (such as OpenMP, MPI, and CUDA) are not efficient for partitioned ID because of the difficulty in ensuring balanced load across computing units. This problem is avoided with an asynchronous parallelization model because of the inherent flexibility in handling variable workloads common in complex and/or large simulations. Alternative strategies like over-partitioning or randomized work stealing as detailed by Gu et al. (2021) might also be attractive but have not been investigated here.

**Task-based parallelization of ID methods in CFD codes**

Task-based (or functional) parallelism is one such model that does not depend on a synchronous parallelization strategy, and is easily adaptable to heterogeneous

---

**Table 3.** Rank of submatrices as a function of number of partitions of the original test matrix (Figure 3). The rank of the submatrices decreases as the number of partitions is increased.

| No. of partitions | Ranks of submatrices |
|-------------------|----------------------|
| 1                 | 48                   |
| 2                 | 24, 30               |
| 4                 | 11, 18, 18, 17       |
| 5                 | 6, 12, 6, 12, 12     |

**Figure 3.** Low-rank test matrix.

**Figure 4.** Matrix compression factor (equation (19)) plotted against the number of partitions of the 100 × 100 test matrix.
computing environments. As the name implies, the parallelism is based on tasks, which are units of work made up of statements, blocks, or loops within the program. These tasks can operate on independent or dependent subsets of data; the data dependencies between the tasks of a program can be represented as a task graph, as shown in Figure 5. Each node of the graph represents an individual task and each edge represents a dependency. Parallel execution of the program is opportunistic; it is only dependent upon information contained in the task graph, with the launch-time of a task determined by any dependencies on previously launched tasks. There are no barriers in the program aside from these task dependencies, making program execution asynchronous (Rauber and Rünger 2010).

There are several advantages to using task-based parallelism, among other parallel programming models, for extremely large and complex simulations. Scientific computing is increasingly trending toward multiphysics, multi-scale applications. A task-based approach is well-suited for complex codes, where subdomains of the data might not necessarily require the same operations. Additionally, the current trend in computer architectures is the adoption of accelerator and, in general, heterogeneous computing frameworks with complex memory hierarchies. Task-based parallelism allows the user to take advantage of hybrid computer architectures because it opportunistically assigns tasks to the available “worker pool” of compute resources, removing the need for user-specified load balancing for each type of processor (Rauber and Rünger 2010).

Task-based parallelism also reduces synchronization bottlenecks experienced by data-based parallelism on large machines, which is crucial because the costs of data movement within these architectures dominates overall compute time as the cost of floating point operations exponentially decreases. Thus, task-based parallelism provides an ideal environment for the implementation of ID algorithms within CFD codes. One example of a programming system that uses this model is Legion (Bauer et al. 2012), which will be discussed in more detail in the Legion Programming Model section 4.

**Implementation**

A task-parallel CFD code will have a layout like that shown in Figure 6. Each processor contains local data about a subsection of the physical domain. At the \( t \)th time step of the simulation, the \( j \)th processor \( P_j \) receives data from the other compute processors, \( P_{k\neq j} \), which is used in addition to its local data to perform computations and advance the fluid solver in time. \( P_j \) sends information to other processors as well. (However, this send message is non-blocking for \( P_j \).) This process repeats until the final simulation time is reached.

As shown in Algorithm 4, this can be represented as a for-loop nested within a while-loop. The task update\_fluid\_step is launched on all \( N + 1 \) processors at each time step of the simulation. This task requires both the local processor’s block of fluid domain data, as well as the data from the stencil regions of the block. As expected, there are no synchronization barriers in this implementation; dependencies in execution time depend only upon the communication of the stencil data between processors. This data exchange is to set the boundary conditions in each subdomain; the stencil determines how these subdomains are connected. The incorporation of ID data compression into flow solvers of this nature will be discussed in the following sections.

**Algorithm 4 Task-Parallel CFD Solver**

1: while \( t < t_{\text{final}} \) do
2: for \( id = 0, N \) do
3: update\_fluid\_step(block[id], stencil[id])
4: end for
5: end while
Naive SPID

The SPID algorithm detailed in the Interpolative Decomposition section is performed after the completion of the flow solver simulation. Figure 7 shows this application execution. In this implementation, the entirety of the flow simulation data (that is, the values of the QoI in the fluid domain at each time step) must be stored. These are then used for SPID.

Algorithm 5 Task-Parallel CFD Solver with Naive SPID

```plaintext
1: while \( t < t_{\text{final}} \) do
2: for \( id = 0, N \) do
3: update\_fluid\_step(block\[id\], stencil\[id\])
4: store\_subsampled\_data(block\[id\])
5: end for
6: end while
7: for \( id = 0, N \) do
8: SPID(subsampled\_block\[id\])
9: end for
```

Figure 7. Application of naive SPID (Algorithm 3) to a CFD simulation.

Figure 8. Formation of column vector of sketch matrix with every 5th entry subsampled on a 2D domain.

Modified SPID

This two-stage algorithm, referred to as Modified SPID, reduces the size of the data matrix being operated on and allows the data compression algorithm to begin while the flow solver is still running.

Stage 1. The first stage applies the ID algorithm with a target rank of \( k \) to \( N \) individual blocks of time steps of the domain, so that the following matrix, which we denote \( A^0 \), is formed:

\[
A^0 = [A_0(:,I_0)\ldots A_{N-1}(:,I_{N-1})C_{N-1}] \tag{20}
\]

where the subscript 0 corresponds to the first \( n/N \) time steps, index 1 corresponds to the next \( n/N \) time steps, and so on until index \( N - 1 \).

We may then decompose \( A^0 \) to be the product

\[
A^0 = A^0C^\omega \tag{21}
\]

\[
= [A_0(:,I_0)\ldots A_{N-1}(:,I_{N-1})]C^\omega \tag{22}
\]
where $C^0 \in \mathbb{R}^{k(N-1) \times N}$ is a sparse matrix comprised of rectangular blocks given coefficient matrices $C_i$ constructed as follows

$$
C^0 = \begin{bmatrix}
C_0 & 0 & 0 & \cdots & 0 \\
0 & C_1 & 0 & \cdots & 0 \\
0 & 0 & C_2 & \cdots & 0 \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & C_{N-1}
\end{bmatrix}
$$

(23)

**Stage 2.** To eliminate any redundant columns in the horizontally concatenated column skeleton matrix $A^0$ and achieve further compression, we apply the ID algorithm, this time with a tolerance instead of a target rank, to $A^0$ to obtain the following approximation

$$
A^0 \approx A^1 C^1
$$

(24)

where

$$
A^0 = A^0(:, \mathcal{I}_U)
$$

(25)

$$
\mathcal{I}_U = \bigcup_{j=0}^{n-1} \left( I_j + jn \right)
$$

(26)

$$
A^0 \approx A^1 C^1
$$

(27)

$\mathcal{I}_U$ indexes the columns of the original matrix $A^0$ extracted prior to second application of column ID, which yields $A^1$ and $C^1$. The subscript $U$ suggests that these indices come from the union of the index vectors $I_j$ in $A^0$ (with an added term to adjust for the indexing in the original data stream). A similar approach to this is employed in Pi et al. (2013), though their approach involves a truncated SVD as a pre-processing step, and does not emphasize pass-efficiency.

The final ID approximation of the entire data is then

$$
A = A^0 C^0 \approx A^1 C^1 C^0 = A^1 C^{11}
$$

(28)

where $A^{11} \in \mathbb{R}^{m \times n}$ and $C^{11} \in \mathbb{R}^{k \times n}$.

**Remark 3.1.** The same procedure may be applied to the case in which we block along the spatial domain; we would simply just replace $A$ with $A^T$ in the above analysis.

The structure of a CFD solver with this ID implementation is shown in Figure 9. The first stage of the SPID implementation is performed during the fluid solve, with only the second stage taking place after the completion of the fluid solve.

This is detailed in Algorithm 6. Similar to Algorithm 5, the update_fluid_step and save_subsampled_data tasks are launched for each processor at each time step. However, there is now an additional task, first_SPID, which is launched if the solver time is at some critical time $t_{crit}$. This critical time, which occurs each time a block of time step data has been subsampled and saved, is defined as

$$
t_{crit} = \Delta t_{crit}
$$

(29)

where $i_{crit}$ is an iteration index that satisfies

$$
i_{crit} \text{ mod} \left( \frac{n}{N} \right) = 0
$$

(30)

This implementation reduces both memory overhead and computation time; since the ID algorithm is applied to each processor independently of the others, the computations for the first stage of the SPID can be performed while the flow solver is waiting for the communication of stencil data from neighboring regions for the next time step. Though the second_SPID task is not launched until the completion of the fluid solve, it is operating on a much smaller matrix than that of Algorithm 5.

Algorithm 6 Task-Parallel CFD Solver with modified SPID

1: while $t < t_{final}$ do
2: for id = 0, $N$ do
3: update_fluid_step(block[id], stencil[id])
4: save_subsampled_data(block[id])
5: if $t \equiv t_{crit}$ then
6: first_SPID(subsampled_block[id])
7: end if
8: end for
9: end while
10: for id = 0, $N$ do
11: second_SPID(subsampled_block[id])
12: end for
The Legion programming model

Legion, an open-source, collaborative effort between Stanford University, Los Alamos National Laboratory (LANL), NVIDIA Research, and Stanford Linear Accelerator Center (SLAC), is a parallel programming model that uses task-based parallelism to create highly efficient and portable code for high-performance computing applications (Bauer et al. 2012). The main unit of abstraction in Legion is the logical region. Logical regions allow a precise definition of how data is being used by the various tasks of an application, and can be further partitioned into various sub-regions. Legion builds a task graph that analyzes the relationship between these subregions and the tasks that access them, and the Legion runtime dynamically assigns eligible tasks to available computing units (CPUs, GPUs, etc.). Legion has been specifically designed to target heterogeneous distributed computer architectures with deep memory hierarchies (Bauer et al. 2012). We highlight several areas that demonstrate how these considerations have been taken into account; each provides users targeting these architectures with distinct advantages over traditional parallel programming methodologies.

Portability with legion

The user is not responsible for explicitly declaring processors on which tasks are run, or where logical regions are placed within memory hierarchies available to their application. Instead, a mapping interface is provided that makes machine-specific execution decisions. This mapper is responsible for determining things like

- which processor and node a task will run on,
- which variant (such as CPU or GPU) of a task will be used, and
- the exact location in the machine’s memory hierarchy all data associated with a task will be laid out.

This mapping capability allows a user to have fine-tuned performance control of their application without accessing or modifying their main code; two distinct, valid mapping sets (such as for two different architectures) will produce an identical application solution. Because of this, Legion applications can be easily ported to new architectures, with performance orthogonal to accuracy. Additionally, if a user does not want to develop a custom mapper, there is a default mapper option available. This default mapper makes generic decisions, can be used to run simulations on all types of platforms (including heterogeneous, multinode execution), and allows applications to be developed in a time-efficient manner.

Portability is greatly increased, as the user is only required to write a single Legion application. This application can be run on CPUs, GPUs, a mix of CPUs and GPUs, shared memory systems, and distributed memory systems without modification. This is unlike MPI+X applications, where MPI is used for inter-process communication and “X” refers to a programming model used for finer-grain parallelism (CUDA, OpenMP, Pthreads, and so forth) (Morris et al. 2020). These applications must be modified and/or rewritten for each individual type of processor and architecture. Because these features are directly embedded within the application, there is also the possibility that solution inaccuracies are introduced to the application each time it is modified. This is not an issue with Legion applications, as the mapper is a standalone component.

Simplicity with legion

Since parallelism is implicitly extracted, Legion applications are written in sequential, easy-to-read code. This also means that, in general, a Legion application requires fewer lines of code than an application that uses MPI+X. These savings can be substantial; in large multiphysics codes, up to 20% of the total code can be dedicated to explicit communication or synchronization instructions. Legion is a C++ library; users also have the option of writing their applications in Regent, which is a high-level programming language built on top of Legion (Slaughter et al. 2015).

Regent contains several task annotations that allow the user to easily control how the application is run on a machine, even if they are using the default mapping feature. For example, users can generate a GPU-enabled variant of a task by simply adding __demand(__cuda) to the top of a task. Annotations can also be applied to for-loops; users can create OpenMP variants, or generate vectorization. These features, as well as the high-level, sequential nature of Regent itself, allow for easier understanding and troubleshooting of existing applications than traditional MPI+X applications.

Performance with legion

Parallelism is implicitly extracted from the task graph that is maintained by the Legion runtime. This task graph is constructed via explicit user declarations of the data a task will require access to, as well as privileges that describe how this data is used (e.g., read, write) within the task. A simple example of these declarations is shown in Figure 10. The Legion runtime uses this task graph to determine the ordering of the tasks during program execution; its overhead is minimized by performing runtime analysis in parallel with the application (Aiken et al. 2014). Because the runtime dynamically assigns tasks, it is guaranteed to fully adapt to the inherent unpredictability of machine architectures. This is in direct contrast to an application that uses MPI+X. In these applications, the task and data placements, as well as
synchronization, are the responsibility of the user. For complex applications, these placements can become difficult, and it may be hard to program optimal configurations. There are several tools that allow users to monitor the execution of their Legion application on a machine and assess parallel performance. The first, LegionSpy (Figure 11), visualizes events (such as tasks or data copies) and the dependencies between them. This can be used to confirm that the task graph is indeed correct; an unexpected result may indicate that one or more dependencies between tasks or privileges on data may have been declared improperly.

The second tool is LegionProf, which visualizes the execution of an application across a set of provided compute resources. An example profile is shown in Figure 12. Each colored block represents a completed task, and white space indicates idle time. The tool also provides total resource and runtime utilization.

CFD and multiphysics solvers using legion

A quantitative assessment of these features has been performed for the baseline NS Legion flow solver used in this work. Previous research (Pacella 2021) compared this solver with an existing FORTRAN+MPI implementation, and found that the Legion solver out-performed the existing FORTRAN+MPI version in all three areas. The Legion solver, which was written using Regent, was over 50% fewer source lines of code than the original implementation and is easily ported to GPUs, which is not possible with the FORTRAN+MPI code. Additionally, the Legion CPU variant was 1.5× faster and the Legion GPU variant was more than 10× faster than the existing FORTRAN+MPI implementation.

There are other computational fluid dynamics and multiphysics codes that use and benefit from the Legion programming model, some of which are listed here. The first large-scale multiphysics application to use Legion, S3D-Legion, is a direct numerical simulation solver for turbulent combustion with complex chemistry Treichler et al. (2017). Another large-scale multiphysics application, Soleil-X (Torres 2021), is a Regent solver that simulates particle-laden turbulent flow exposed to a radiation environment. The Hypersonic Task-based Research (DiRenzo et al. 2020) solver, also written in Regent, performs direct numerical simulations of canonical hypersonic flows at high Reynolds numbers. We direct the interested reader to these resources for further analysis and discussion of the benefits in portability, simplicity, and performance provided by Legion.

Numerical experiments

The test cases below demonstrate the successful application of ID for data compression into several canonical Taylor-Green vortex problems. The first two examples, a two-dimensional incompressible Taylor-Green vortex and a three-dimensional, compressible low-Re Taylor-Green vortex, were selected because their solution is low rank; application of ID to these data matrices should result in ideal compression values. Further, the first problem presented (the incompressible Taylor-Green vortex) has an analytic solution. Therefore, failure to compress data generated from simulations of these systems would be due to the algorithmic implementation itself. The final example, a three-dimensional, compressible high-Re Taylor-Green vortex, demonstrates the application of the ID algorithm to a problem whose solution is more complex (and therefore higher rank) than the previous two examples.

Analytical Taylor-Green vortex

The two-dimensional incompressible Taylor-Green vortex, a decaying vortex with unsteady flow, is an exact closed form solution of the incompressible Navier-Stokes equations in a Cartesian coordinate system.

Let $u = (u_1, u_2)$ be the velocity components of the flow, $p$ the pressure field of the flow, $\rho$ the density, and $\nu$ the kinematic viscosity of the fluid. The exact Taylor-Green vortex
Figure 11. An example of the LegionSpy debugging tool (LegionSpy); this tool visualizes the task graph of an application. Blocks represent events and arrows represent dependencies between them, with dependency flowing from left to right.

Figure 12. An example of the LegionProf profiling tool. This tool provides detailed information about the specific execution of an application on a machine, including the task distribution across compute processors, overall resource utilization, and runtime activity.
solution of this system on the doubly periodic domain $0 \leq x_1, x_2 < 2\pi$ is

$$u_1(x_1, x_2, t) = \sin(x_1)\cos(x_2)\exp(-2vt) \tag{31}$$

$$u_2(x_1, x_2, t) = -\cos(x_1)\sin(x_2)\exp(-2vt) \tag{32}$$

$$p(x_1, x_2, t) = \frac{\rho}{4} (\cos(2x_1) + \sin(2x_2))\exp(-4vt) \tag{33}$$

The ID algorithm described in the Implementation section was applied to a Legion implementation of the Taylor-Green solution defined on both a structured grid and an unstructured, randomly generated grid. The reconstructed (compressed) solution $A_{\text{approx}}$ was then compared to the exact solution $A_{\text{exact}}$ at the grid points, and the normalized Frobenius norm (matrix entry-wise 2-norm) of the error

$$\text{Relative Frobenius Error} = \frac{\|A_{\text{exact}} - A_{\text{approx}}\|}{\|A_{\text{exact}}\|_F} \tag{34}$$

was computed. Error results for the data compression of the $u_1$ component of velocity as a function of domain partitions for $20^2$ domain points for both the structured and unstructured grids are shown in Figures 13–15. For a total simulated time of 10 s (100 total “time steps” of the analytical solution), the resulting rank of the ID approximation is one for both grids (which corresponds to a compression factor $\approx 80$). The normalized Frobenius norm of the error is on the order of machine precision for both grids, as well. Similar accuracy and compression factor results are observed for the $u_2$ component of velocity, as well as the pressure.

This demonstrates that the SPID algorithm can achieve a large compression factor for low-rank solutions even when there is no “sorting” to the underlying points of the spatial domain. It is also important to note that the accuracy of the approximation improves as the number of spatial partitions grows; partitioning the spatial domain can increase both the computational efficiency and the compression factor without sacrificing accuracy.

The exact and reconstructed velocity magnitude at $t = 10$ s for both the $20^2$ structured and unstructured grid domains with 10 partitions in both the $x_1$- and $x_2$-directions can be seen in Figure 16. The normalized Frobenius norm of the error for this velocity magnitude is $O(10^{-16})$. Even though there are a large number of partitions in the spatial domain, each with their own individual ID approximation, there is no “disjointedness” apparent in this reconstructed solution.

### High-order Navier-Stokes solver

Our remaining test cases employ a high-order Cartesian solver for the Navier-Stokes equations that is used as an off-body solver for rotorcraft wake simulation (Wissink et al. 2010), as well as a test bed for the development of novel numerical methods (Leffell et al. 2016). It solves the conservation form of the 3D Navier-Stokes equations

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_1} (\rho u_1) + \frac{\partial}{\partial x_2} (\rho u_2) + \frac{\partial}{\partial x_3} (\rho u_3) + \rho e = 0 \tag{36}$$

where $q$ is the vector of conserved variables, $f_i$ is the convective flux vector in the $i^{th}$ direction, and $g_i$ is the viscous flux vector in the $i^{th}$ direction

$$f_i = \begin{bmatrix} \rho \left( u_i - \hat{x}_i \right) \\ \rho u_1 \left( u_i - \hat{x}_1 \right) + p \\ \rho u_2 \left( u_i - \hat{x}_2 \right) \\ \rho u_3 \left( u_i - \hat{x}_3 \right) \\ \rho e \left( u_i - \hat{x}_1 \right) + pu_i \end{bmatrix} \tag{37}$$

$$g_i = \begin{bmatrix} 0 & \tau_{11} & \tau_{12} & \tau_{13} & \tau_{14} - k_i \end{bmatrix}^T \tag{38}$$

$u = (u_1, u_2, u_3)$ is the fluid velocity, $\hat{x} = (\hat{x}_1, \hat{x}_2, \hat{x}_3)$ is the reference frame velocity, $\tau$ is the viscous stress tensor, $k$ is the heat flux, $\rho$ and $\rho e$ are the fluid density and pressure, respectively, and $e$ is the total energy.

A Newtonian fluid assumption is used, so the terms of the viscous stress tensor are defined as

$$\tau_{ij} = \rho \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} - \frac{2}{3} \frac{\partial \rho}{\partial x_i} \frac{\partial u_i}{\partial x_j} \right) \tag{39}$$
The dynamic viscosity $\mu$ is considered to be a function of temperature according to Sutherland’s Law

$$
\mu(T) = \mu_{ref} \left( \frac{T}{T_{ref}} \right)^{3/2} \left( T_{ref} + S \right) \quad (40)
$$

where $S$ is the Sutherland temperature, $T_{ref}$ is a reference temperature, and $\mu_{ref}$ is the dynamic viscosity at the reference temperature. The entries of the heat flux vector are

$$
k_i = -k \frac{\partial T}{\partial x_i} \quad (41)
$$

with the thermal conductivity $k$ found from the definition of the Prandtl number $Pr$

$$
k = \frac{\mu c_p}{Pr} \quad (42)
$$

where $c_p$ is the constant pressure specific heat. The system is closed with the equation of state for an ideal gas

$$
p = (\gamma - 1) \left( \rho e - \frac{1}{2} \rho (\mu_1^2 + \mu_2^2 + \mu_3^2) \right) \quad (43)
$$

and the speed of sound is defined as

$$
a = \sqrt{\frac{\gamma}{\rho} p} \quad (44)
$$

where $\gamma = c_p/c_v$ is the ratio of the constant pressure and constant volume specific heats.

### Spatial discretization

The Navier-Stokes equations are transformed into ordinary differential equations by discretizing in space on the Cartesian grid with cell spacing $(\Delta x_1, \Delta x_2, \Delta x_3)$, so that

$$
\frac{dq_j}{dt} = \sum_{i=1}^{3} \left( \frac{\partial f_{ij}}{\partial x_i} - \frac{\partial f_{ij}}{\partial x_i} \right) \quad (45)
$$

The solver is based on sixth-order accurate inviscid flux discretizations and fourth-order accurate viscous flux discretizations. There is also a fifth-order accurate artificial dissipative flux term introduced into the inviscid flux reconstructions for stability purposes; this term depends on a user-specified tolerance, $\epsilon$. These terms all use a seven-point stencil in the spatial domain (Figure 17); for example, the inviscid flux gradients at the cell centers are approximated as

$$
\frac{\partial f_i}{\partial x_i} \approx \frac{\hat{f}_{ij}^{+1/2} - \hat{f}_{ij}^{-1/2}}{\Delta x_i} \quad (46)
$$

where $\hat{f}_{ij}^{+1/2}$ and $\hat{f}_{ij}^{-1/2}$ are the fluxes at the cell faces in the $i^{th}$ direction. The fluxes at the cell faces are defined as

$$
\hat{f}_{ij} = \frac{1}{60} \left( f_{i+2} - 8f_{i+1} + 37f_{i} + 37f_{i-1} - 8f_{i-2} + f_{i-3} \right) \quad (47)
$$

These flux reconstructions result in three fringe points at each boundary in the partitioned numerical domain.

### Temporal discretization

The Navier-Stokes equations are explicitly advanced in time using a low-storage third-order Runge-Kutta method (Kennedy et al. 2000). For the following ODE

$$
\frac{dq}{dt} = -R(q) \quad (48)
$$

the solution at the next time step, $q^{(n+1)}$, is found from

$$
q^* = q^{(n)} - a_1 \Delta t R(q^{(n)}) \quad (49)
$$

$$
q^{**} = q^{(n)} - a_2 \Delta t R(q^{(n)}) \quad (50)
$$

$$
q^{***} = q^* - a_3 \Delta t R(q^{**}) \quad (51)
$$

$$
q^{(n+1)} = q^* - a_4 \Delta t R(q^{***}) \quad (52)
$$

where $a_1 = 1/4$, $a_2 = 8/15$, $a_3 = 5/12$, and $a_4 = 3/4$.

Overall, there is a high amount of communication required for this solver; this three-stage process for the time-advancement scheme amplifies the already large amount of data that must be communicated between neighboring partitions due to the seven-point stencil in the spatial domain.

### Low-Re compressible Taylor-Green vortex

This solver was re-engineered in the Legion framework, and data compression via the Modified SPID algorithm was added. To test how well this Navier-Stokes-Interpolative Decomposition (NSID) implementation performs, a compressible Taylor-Green vortex with Reynolds number $Re = 100$ and initial conditions

$$
u_1(t_0) = V_0 \sin \left( \frac{x}{L} \right) \cos \left( \frac{y}{L} \right) \cos \left( \frac{z}{L} \right) \quad (53)
$$

$$
u_2(t_0) = -V_0 \cos \left( \frac{x}{L} \right) \sin \left( \frac{y}{L} \right) \cos \left( \frac{z}{L} \right) \quad (54)
$$

$$
u_3(t_0) = 0 \quad (55)
$$
was simulated on the triply-periodic domain \( \pi L \leq x_1, x_2, x_3 < \pi L \) with \( L = 1 \) m, \( V_0 = 1 \) m/s, \( \rho_0 = 1 \) kg/m\(^3\), and \( M_0 = 0.1 \). Table 4 provides the problem parameters for the results discussed in the following sections, which quantify how the relative tolerance and the subsampling interval of the ID affect the overall compression of the vortex. A target rank of 10\% of the original matrix column dimension was used, as well as 100 time step intervals for Stage 1 of the Modified SPID algorithm.

### Table 4. Problem setup for a low-Re compressible Taylor-Green vortex.

| Variable                  | Value |
|---------------------------|-------|
| Domain size \((128, 128, 128)\) |       |
| \(x_1, x_2, x_3\) \((0, 0, 0)\)   |       |
| Re                        | 100   |
| Pr                        | 0.71  |
| \(\gamma\)                | 1.4   |
| Time steps \(10,000\)      |       |
| \(dt\)                    | 0.01  |
| \(\epsilon\)              | 0.1   |
| Target rank \(1000\)       |       |
| SPID stage 1 intervals \(100\) |     |

### Compression as a function of relative tolerance

Table 5 shows results for how the relative tolerance (used as the stopping criterion for Stage 2 of the Modified SPID algorithm) affects the final accuracy and compression factor of the approximate solution. These results can be seen graphically in Figure 19. Figure 18 also shows the mean kinetic energy across the domain as a function of the relative tolerance.

As expected, the compressed approximation of the vortex became more accurate, with a higher overall rank and compression ratio, as the relative tolerance became more restrictive (shown in Figure 19(a)–(c)). A relative tolerance of \(10^{-3}\) resulted in a rank-two approximation with a compression factor of almost 5,000, but was more than 10\% different from the exact solution. However, a relative tolerance of \(10^{-5}\), which resulted in a rank-seven approximation, was only 0.5\% different from the exact solution, while still providing a compression factor of over 1400.

Figure 19(d) demonstrates that the indices of the time steps used to generate the skeleton column of the ID approximation overlap for each tolerance value—as the relative tolerance decreases, the ID approximation retains the columns of the previous result, and appends supplemental columns. It is worth noting that the distribution of the skeleton columns for these results is fairly uniform across the temporal domain. Additionally, though the different tolerances provided varying accuracy and compression factor results, the reconstruction of the mean kinetic energy in the domain at each time step was similar to that of the exact solution.

For this test case, it was beneficial to use the two-stage Modified SPID algorithm—if only a single stage, fixed-rank ID algorithm (like the SubID) was used, it would have resulted in a compressed solution with a rank of 1000 (since the target rank of the compression was 10\% that of the original solution). However, the second stage of the modified algorithm allowed for the removal of redundant information, resulting in a compression factor more than 100 times larger. This is very useful for simulations where the time step and spatial resolution are limited by stability and

### Table 5. Effect of relative tolerance on the compression of a low-Re compressible Taylor-Green vortex for the problem given in Table 4.

| Re   | Sub. Int | Rel. Tol | Final rank | Skeleton columns | Frob. Norm | CF  |
|------|----------|----------|------------|------------------|------------|-----|
| 100  | 1        | \(10^{-3}\) | 2          | \([1, 5130]\)    | 0.101791   | 4976|
| 100  | 1        | \(10^{-4}\) | 4          | \([1, 5130, 2460, 9096]\) | 0.030128   | 2488|
| 100  | 1        | \(10^{-5}\) | 7          | \([1, 5130, 2460, 9096, 1091, 3733, 6944]\) | 0.004740   | 1421|

### Table 6. Effect of the subsampling interval on the compression of a low-Re compressible Taylor-Green vortex for the problem given in Table 4.

| Re   | Sub. Int | Rel. Tol | Final rank | Skeleton columns | Frob. Norm | CF  |
|------|----------|----------|------------|------------------|------------|-----|
| 100  | 1        | \(10^{-5}\) | 7          | \([1, 5130, 2460, 9096, 1091, 3733, 6944]\) | 0.004740   | 1421|
| 100  | 3        | \(10^{-5}\) | 4          | \([1, 4991, 2425, 8916]\) | 0.03546    | 55081|
| 100  | 7        | \(10^{-5}\) | 2          | \([1, 4892]\)     | 0.115936   | 582542|
| 100  | 9        | \(10^{-5}\) | 2          | \([1, 4610]\)     | 0.127435   | 743881|
accuracy requirements for the numerical methods being used; even if the flow simulation itself must be restrictive in these capacities, the Modified SPID algorithm can be used to greatly reduce the overall amount data that must be saved.

**Compression as a function of subsampling interval**

The previous results discussed the effect of the relative tolerance on the compressed solution for a subsampling interval of 1. However, it is also useful to study how the subsampling interval affects the compressed solution, since many complex multiphysics simulations require very fine meshes. Table 6 and Figure 20 show results for how the subsampling interval of the spatial domain affects the final accuracy and compression factor (this time with a fixed relative tolerance of $10^{-3}$). Figure 21 also shows the mean kinetic energy across the domain as a function of the subsampling interval used.

The subsampling interval provides similar accuracy results to that of the relative tolerance results, but at substantially higher compression ratios since the matrix operator that interpolates from the subsampled grid back to the original grid does not have to be stored. A subsampling interval of 9 resulted in a rank-two approximation with a compression factor of over 700,000, though this was at the cost of accuracy. However, a subsampling interval of 3, which resulted in a compressed result with rank 4, was only 3.5% different from the exact solution but still provided a very large compression factor of 55,000. As with the relative tolerance results, the skeleton columns saved for the ID approximation are uniformly distributed across the domain (Figure 20(d)). However, the subsampling interval did not capture the mean kinetic energy across the domain as well as when the relative tolerance was varied. Though the general behavior was the same, each subsampling interval tended to under-compute the mean kinetic energy. The accuracy of this result, and SubID results in general, may be affected by the interpolation operator used to transform the subsampled grid to the full grid. In this case, trilinear interpolation was used; another method, such as tricubic interpolation, may yield improved results.

**Table 7.** Problem setup for a high-Re compressible Taylor-Green vortex.

| Variable                  | Value                  |
|---------------------------|------------------------|
| Domain size               | (128, 128, 128)       |
| $x_1$, $x_2$, $x_3$       | (0, 0, 0)              |
| $M_0$                     | 0.1                    |
| Re                        | 1600                   |
| Pr                        | 0.71                   |
| $\gamma$                  | 1.4                    |
| Time steps                | 10,000                 |
| $dt$                      | 0.01                   |
| $\epsilon$               | 0.1                    |
| Target rank               | 1000                   |
| SPID stage 1 intervals    | 100                    |

**Table 8.** Effect of relative tolerance on the compression of a high-Re compressible Taylor-Green vortex given in Table 7.

| Re  | Sub. Int. | Rel. Tol. | Final rank | Skeleton columns | Frob. Norm CF |
|-----|-----------|-----------|------------|------------------|--------------|
| 1600| 1         | $10^{-3}$ | 9          | [1, 7036, 4750, 9900, 8367, 3001, 6069, 9122, 7751] | 0.065 585 1105 |
| 1600| 1         | $10^{-4}$ | 17         | [1, 7036, 4750, 9900, 8367, 3001, 6069, 9122, 7751, 8733, 6583, 9550, 7424, 5451, 8061, 1718, 4053] | 0.021 325 585 |
| 1600| 1         | $10^{-5}$ | 32         | [1, 7036, 4750, 9900, 8367, 3001, 6069, 9122, 7751, 8733, 6583, 9550, 7424, 5451, 8061, 1718, 4053, 8918, 9782, 8533, 6336, 5108, 7234, 700, 9320, 5769, 7908, 6818, 4422, 3578, 7583, 8218] | 0.005 549 311 |

**Table 9.** Effect of the subsampling interval on the compression of a high-Re compressible Taylor-Green vortex for the problem given in Table 7.

| Re  | Sub. Int. | Rel. Tol. | Final rank | Skeleton columns | Frob. Norm CF |
|-----|-----------|-----------|------------|------------------|--------------|
| 1600| 1         | $10^{-3}$ | 32         | [1, 7036, 4750, 9900, 8367, 3001, 6069, 9122, 7751, 8733, 6583, 9550, 7424, 5451, 8061, 1718, 4053, 8918, 9782, 8533, 6336, 5108, 7234, 700, 9320, 5769, 7908, 6818, 4422, 3578, 7583, 8218] | 0.005 549 311 |
| 1600| 3         | $10^{-5}$ | 15         | [3150, 1, 7378, 5192, 9822, 8566, 6451, 7959, 9101, 4321, 6937, 5883, 1761, 8264, 9458] | 0.070 291 14 688 |
| 1600| 7         | $10^{-5}$ | 5          | [3532, 1, 7292, 9900, 5192] | 0.179 342 233 016 |
| 1600| 9         | $10^{-5}$ | 3          | [3404, 1, 7427] | 0.223 979 495 921 |
High-Re compressible Taylor-Green vortex

The same problem in the Low-Re Compressible Taylor-Green Vortex section was simulated, this time with \( Re = 1600 \) to guarantee the development of turbulence in the domain; problem parameters are listed in Table 7. The impact of both the relative tolerance and the subsampling interval will again be discussed, using the same target rank and number of Stage 1 time step intervals as before.

Compression as a function of relative tolerance

Table 8 shows results for how the relative tolerance for Stage 2 of the Modified SPID algorithm (with a subsampling interval of one) affects the final accuracy and compression factor, and Figure 22 shows these results graphically. Figure 23 also shows the mean kinetic energy across the domain as a function of the relative tolerance used.

The compression for the high-Re Taylor-Green vortex was less than that of the low-Re vortex due to the formation of turbulent structures, which increase the complexity of the domain. However, a high level of compression was still achieved. A relative tolerance of \( 10^{-3} \) resulted in a rank-nine approximation with a compression factor of 1100, but was almost 7% different from the exact solution. Using a relative tolerance of \( 10^{-5} \) was much more accurate—it resulted in a rank-32 approximation with a compression factor of over 300, and was less than 1% different than the exact solution. Again, it was beneficial to use the two-stage Modified SPID implementation, since the rank of the compressed result for a very accurate solution was still much smaller than that of the original target rank of 1000.

Like with the low-Re vortex, as the relative tolerance became successively more restrictive, the columns of the skeleton matrix built upon the previous tolerance level, though this time most of the columns are from the end of the temporal domain (as the turbulent structures form). All relative tolerance levels captured the magnitude of the mean kinetic energy in the domain, but there is an almost “oscillatory” behavior to the values at times between the time steps contained in the skeleton matrix of the ID approximation.

Compression as a function of subsampling interval

Table 9 and Figure 24 detail results for how the subsampling interval of the spatial domain (with a relative tolerance of \( 10^{-5} \)) affects the final accuracy and compression factor. Figure 25 also shows the mean kinetic energy across the domain as a function of the subsampling interval.

As with the low-Re vortex, increasing the subsampling interval resulted in extremely large compression factors, but at the expense of accuracy. A subsampling interval of 3 (with a relative tolerance of \( 10^{-5} \)) provided similar accuracy (approximately 7% different than the exact solution) as

| Case         | Normalized runtime |
|--------------|--------------------|
| NS, 0 QoI   | 1.00               |
| NS, 1 QoI   | 1.22               |
| NS, 2 QoI   | 1.22               |
| NS, 3 QoI   | 1.23               |

| Case          | Normalized runtime |
|---------------|--------------------|
| NS, 0 QoI     | 1.00               |
| NSID, 1 QoI   | 1.22               |
| NSID, 2 QoI   | 1.22               |
| NSID, 3 QoI   | 1.23               |
when the relative tolerance was $10^{-3}$ and no subsampling was used, though the compression factor increased from $O(1000)$ to $O(10,000)$, demonstrating the advantages of physically subsampling the domain to reduce storage overhead.

Again, most of the time steps used in the skeleton column are from later time steps of the simulation. The same oscillatory behavior in the approximated mean kinetic energy is seen as with the relative tolerance study, though on a much larger scale; subsampling the physical domain leads to non-physical behavior for this parameter. Because of this, it may be better in practice to be more restrictive with the selected subsampling interval while simultaneously relaxing the relative tolerance, instead of only using extreme values for either parameter. Investigations of this nature are continuing work; however, this test case has showed that it is possible to achieve high compression ratios for turbulent CFD-ID solvers in legion

Though the compression and accuracy were ideal for the test cases presented in the Numerical Experiments section, we were also interested in the computational suitability of the

Figure 16. Exact and reconstructed solutions for the velocity magnitude of a $20^2$ domain on a structured grid ((a) and (b)) and an unstructured grid ((c) and (d)); total simulated time is 10 s.

Figure 17. Seven-point stencil for one direction of the spatial domain; this requires three fringe points that must be exchanged between boundaries for each spatial direction.

Figure 18. Mean kinetic energy as a function of relative tolerance for the problem given in Table 4.
Figure 19. Rank (a), compression factor (b), and relative Frobenius norm error (c) as a function of relative tolerance for the problem given in Table 4. The columns comprising the skeleton matrix are shown in (d).

Figure 20. Rank (a), compression factor (b), and relative Frobenius norm error (c) as a function of subsampling interval for the problem given in Table 4. The columns comprising the skeleton matrix are shown in (d).
Legion system for a flow solver using ID for data compression, especially since the Legion flow solver used here compared favorably to a more traditional implementation (Pacella 2021). We will assess this on the basis of the three Legion features described in the Legion Programming Model section: portability, simplicity, and performance. These assessments draw on results from two different variants: a CPU-only variant where both the flow solver and data compression are performed on the same set of processors (a homogeneous system), and a CPU-GPU variant, where the flow solver was run on GPUs and the data compression on CPUs (a heterogeneous system). A simple

\begin{figure}[h]
\centering
\begin{subfigure}{0.45\textwidth}
\centering
\includegraphics[width=\textwidth]{mean_kinetic_energy.png}
\caption{Mean kinetic energy as a function of subsampling interval for the problem given in Table 4.}
\end{subfigure}
\begin{subfigure}{0.45\textwidth}
\centering
\includegraphics[width=\textwidth]{mean_kinetic_energy_relative_tolerance.png}
\caption{Mean kinetic energy as a function of relative tolerance for the problem given in Table 7.}
\end{subfigure}
\caption*{Figure 21. Mean kinetic energy as a function of subsampling interval for the problem given in Table 4. Figure 23. Mean kinetic energy as a function of relative tolerance for the problem given in Table 7.}
\end{figure}

\begin{figure}[h]
\centering
\begin{subfigure}{0.45\textwidth}
\centering
\includegraphics[width=\textwidth]{rank_compression_factor_relative_error.png}
\caption{Rank (a), compression factor (b), and relative Frobenius norm error (c) as a function of relative tolerance for the problem given in Table 7. The columns used in the skeleton matrix are shown in (d).}
\end{subfigure}
\caption{Figure 22. Rank (a), compression factor (b), and relative Frobenius norm error (c) as a function of relative tolerance for the problem given in Table 7. The columns used in the skeleton matrix are shown in (d).}
\end{figure}
mapper was used to control the processor type used for task execution, but all other decisions were left to Legion’s default mapper. Examining both of these variants with limited execution-tuning allows for a broader baseline evaluation of how well ID data compression can be incorporated into CFD applications in a task-parallel environment.

Data were collected using the Yellowstone computing cluster (Stanford HPCC), which is comprised of CPU and heterogeneous CPU-GPU compute nodes. Multinode scaling performance results were run on CPU-only nodes, which have 2 8-core Intel Xeon E5-2670 processors and 32 GB of memory per node. Profiling results where the flow solver was run on GPUs and the ID was computed on CPUs were generated from runs on a heterogeneous node, which has one NVIDIA Kepler K20 GPU, 6 8-core Intel Xeon E5-2670 processors, and 256 GB memory per node. All nodes use a Mellanox InfiniBand for inter-node communication.

**CFD-ID portability**

Legion allows application engineers to port their code to multiple types of processors. The ability to take advantage of this is useful for the paired fluid dynamics-data compression algorithm being discussed here, because there are different characteristics between the two algorithms. The flow solver tasks have low memory requirements, are computation-intensive, and can be executed concurrently across each subdomain of our partition. These characteristics are well-suited to GPU architectures. However, data

![Figure 24](image_url)

Figure 24. Rank (a), compression factor (b), and relative Frobenius norm error (c) as a function of subsampling interval for the problem given in Table 7. The columns used in the skeleton matrix are shown in (d).

![Figure 25](image_url)

Figure 25. Mean kinetic energy as a function of subsampling interval for the problem given in Table 7.
Compression algorithms are the exact opposite; memory requirements can be high, especially if large numbers of time steps are being analyzed, and they consist of linear algebra computations that work in serial across the data matrix. Because of this, they are better suited to CPU-type architectures. To address this disparity, a custom mapper within the Legion framework was developed to direct the placement of the fluid solve and the data compression tasks on heterogeneous architectures, with the fluid solve taking place on GPUs and the data compression taking place on CPUs. Figure 26 shows an example profile for 25 time steps of the Navier-Stokes Interpolative Decomposition (NSID) solver mapped in this manner. As expected, the Stage 1 data compression process begins while the fluid solver is still running.

This increase in portability provided by Legion can also lead to savings in total runtime. When using a homogeneous architecture, for example, strictly CPUs, any reduction in the runtime of the NSID solver due to Stage 1 of the ID algorithm will be due to the distributed memory nature of the machine. The typical mapping for this scenario is to have one domain partition per processor; latency in the fluid solver occurs due to the exchange of fringe point data across the machine network. If there is enough latency in the system, the Stage 1 ID tasks can be computed between time step updates of the fluid solver, increasing efficiency.

Figure 26. Mapping of the NSID code onto a heterogeneous architecture for 25 time steps of a 64³ domain; the fluid solver is mapped to a GPU, and the ID is mapped to a CPU. Figure (a) shows the overall LegionProf profile while Figure (b) highlights Stage 1 and Stage 2 of the SPID algorithm.

Figure 27. Runtimes normalized against a 1 GPU N-S solve runtime of 5.26 s. The problem is solved on a 64³ domain for 1 block and 100 time steps. In the CPU-GPU setup, the GPU is used for the fluid solver and the CPU is used for data compression.
However, there is no way to improve the solver performance within a single (shared memory) node. This changes if a user can access and utilize the entire heterogeneous architecture. By using Legion’s mapping capabilities to port all flow solver tasks to a GPU that was previously not accessible to the user, Stage 1 of the SPID algorithm truly becomes parallel with the flow solver; essentially, this stage of the compression is “free.”

Figure 27 shows an example of this. For a single CPU, there are no savings within a single node for increasing the number of time step intervals analyzed for Stage 1 of the ID. However, if the flow solver is run on a GPU and the data compression is run on a CPU, there are significant runtime savings as the number of time step intervals is increased. Running the flow solver on a GPU instead of a CPU also decreases the overall runtime by more than a factor of 10. This result can be extended to the compression of multiple flow QoIs; since there are traditionally many more CPUs than GPUs on heterogeneous architectures, and each QoI can be independently compressed for each fluid subdomain, it is possible that many flow QoIs could be compressed in the same amount of solver time.

A demonstration of this is shown in Figure 28, which is a profile for the same problem as in Figure 26, this time with two QoIs being compressed. Table 10 also shows the normalized runtimes for the flow solver and (if applicable) both stages of the ID compression, as a function of total compressed QoIs. As the table shows, the total cost of the NSID solve is constant as the number of compressed QoIs increases, and takes only around 20% longer than the flow solver itself. It is important to note that this cost would likely be reduced for multinode runs, where communication is now required for the flow solver, as well as by using sub-sampling of the domain for the Modified SPID algorithm. Possible extensions of this work will be discussed in the next section.

Figure 28. Mapping of the NSID code, this time compressing two QoIs, onto a heterogeneous architecture; the flow solver is mapped to a GPU, and the ID is mapped to a CPU. Figure (a) shows the overall LegionProf profile while Figure (b) is the same profile, this time highlighting Stage 1 and Stage 2 of the SPID algorithm for both QoIs.
CFD-ID simplicity

The Regent programming language was used, and both CPU and GPU task variants were created for the flow update tasks in both the NS and NSID solvers. Figure 29 shows LegionSpy task graph analyses for both one time step of the Navier-Stokes (NS) solver and one time step of the NSID solver. Each block represents an event required for the time step, and each arrow corresponds to a dependency that requires data communication from the stencil points. As expected, parallelism was implicitly extracted for both codes, eliminating the need for the user to hard-code all

Figure 29. LegionSpy task graphs for a single time step of (a) the Navier-Stokes (NS) solver, and (b) the Navier-Stokes Interpolative Decomposition (NSID) solver; blocks represent events and arrows between blocks represent dependencies between events. There are two possible task executions at the end of the NSID solver time step; if $t \neq t_{\text{crit}}$, then the data is simply subsampled and flattened. However, if $t = t_{\text{crit}}$, a Stage 1 ID analysis is performed.
communication stages between the flow solver and the SPID stages. This, combined with the high-level nature of the Regent programming language (as well as the ability to generate GPU variants of tasks with a single line of code), leads to much simpler implementations of the solver than with an MPI+X approach.

**CFD-ID performance**

From a performance perspective, data compression using ID can only be considered beneficial if it does not negatively impact the scaling of the original solver, and does not significantly increase the overall runtime. Both the NS and NSID solvers were mapped to CPUs to test the strong scaling of the solver for a larger number of nodes. A problem size of 10 time steps on a 128^3 point domain with 4^3 spatial partitions was selected; for each run, domain partitions were evenly distributed across the physical CPUs of the nodes, with task execution and memory placement determined by Legion’s default mapper. The results in Figure 30(a) show that both codes scaled well up to 8 nodes, but the NSID solver scaled slightly better than the Navier-Stokes solver on its own. This is expected, as the SPID algorithm introduces additional work into the NS solver without requiring any additional communication.

**Results**

An examination of the portability, simplicity, and performance of the NSID solver indicates that the Legion runtime was capable of seamlessly integrating ID data compression into a Navier-Stokes solver, even in a heterogeneous compute environment. This is important for users, as the speedups offered by GPUs for scientific applications become extremely important at large scales. Legion’s implicit parallelism, as well as the ability to port to multiple architectures with a single code, also allows users to utilize a range wide of machines without rewriting their code every time, and initial scaling and simulation time results do not indicate any negative impacts due to the overhead of the Legion runtime executing two very different algorithms. It is also important to note that the results presented here were a baseline assessment—though the results presented here are positive, additional increases in performance optimization would be realized using a more detailed Legion mapper.

**Conclusions and future work**

In this paper, we have presented a novel task-based parallel framework for in situ temporal compression of data generated from large-scale computational fluid dynamics simulations using the matrix interpolative decomposition. This framework, implemented using the Legion programming system, allows the user to achieve significant compression factors of simulation data with reduced disk memory requirements following simulation and a minimal effect on scaling and runtime. We verify this claim on several test problems: an analytical, incompressible Taylor-Green vortex and both high-Re and low-Re compressible Taylor-Green vortices simulated using a large-scale Navier-Stokes solver implemented in Region, a high-level programming language built on top of Legion.

The numerical accuracy of the modified SPID algorithm was confirmed with the analytical solution of the incompressible Taylor-Green vortex for both structured and unstructured grids; all compressed results were rank-one with error on the order of machine precision. For the low-Re compressible Taylor-Green vortex, compression factors on
the order of $1,000\text{–}10,000$ were achieved, with error values between 0.001 and 0.01. When the Reynolds number was increased to induce turbulence in the vortex, the compression factors, though still large, dropped to $O(100\text{–}1,000)$ for the same level of accuracy (0.001–0.01).

Preliminary comparisons between the Navier-Stokes flow solver and the same flow solver augmented with ID data compression were performed for both multinode, CPU runs and single-node heterogeneous CPU-GPU runs where the flow solver ran on the GPU and the data compression was performed on the CPU. For CPU-only runs for 100 time steps on a $128^3$ domain with $4^3$ spatial partitions, the flow solver with data compression scales almost linearly up to 8 nodes, slightly better than the original flow solver. Additionally, the runtime of the flow solver with data compression was at most 10% larger than that of the flow solver by itself. For the single-node CPU-GPU runs, it was demonstrated that, for 25 time steps of a $64^3$ domain, up to three QoIs could be simultaneously compressed with only an approximately 20% increase in runtime over the flow solver itself.

Since this work indicates that flow solvers with ID data compression are worth further investigation, focus will shift to the development of a more robust custom mapper to optimize performance. The custom mapper used to gather the results presented here was designed only to control which variant of a task was used (CPU or GPU), so that baseline behavior of the modified SPID algorithm could be studied with minimal interference. However, the new mapper will fully utilize the functionality offered by Legion, taking into account both inter- and intra-node tuning parameters. These parameters dictate the placement and execution of tasks at the processor level, as well as the location of region data within memory hierarchies, for example, where fluid data is placed in memory when both the CPU and GPU will require access to it. This mapper will also be used to further investigate performance for the compression of multiple flow QoIs, especially on heterogeneous machines. The behavior and performance of ID within ensemble simulations will also be studied. Future work also includes development of newer ID algorithms as open-source software built in Regent.

**Acknowledgements**

The computations in this paper were performed on the Yellowstone cluster at the Stanford HPC Center, supported through awards from the National Science Foundation, DOD HPCMP, and Office of Naval Research.

**Declaration of conflicting interests**

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

**Funding**

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was funded by the United States Department of Energy’s National Nuclear Security Administration under the Predictive Science Academic Alliance Program II at Stanford University, Grant 495 DE-NA-0002373. The work of AD was supported by the AFOSR grant FA9550-20-1-0138.

**References**

Treichler S, Bauer M and Aiken A (2014) Realm: event-based low-level runtime for distributed memory architectures. In: 2014 23rd international conference on parallel architecture and compilation techniques (PACT). Edmonton, AB, Canada, 23–27 August, 2014, pp. 263–275. DOI: 10.1145/2628071.2628084

Amarasinghe S, Campbell D, Carlson W, et al. (2009) Exascale Software Study: Software Challenges in Extreme Scale Systems. DARPA IPTO, Air Force Research Labs, Tech. Rep. pp. 1–153.

Ang J, Evans K, Geist A, et al. (2012) Report on the Workshop on Extreme-Scale Solvers: Transitions to Future Architectures. Office of Advanced Scientific Computing ResearchUS Department of Energy. 8–9.

Asch M, Moore T, Badia R, et al. (2018) Big data and extreme-scale computing. *The International Journal of High Performance Computing Applications* 32(4): 435–479.

Ashby S, Beckman P, Chen J, et al. (2010) *The Opportunities and Challenges of Exascale Computing*. Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, 1–77.

Austin W, Ballard G and Kolda TG (2016) Parallel tensor compression for large-scale scientific data. In: IEEE international parallel and distributed processing symposium (IPDPS), May 23–27, 2016, Chicago, IL IEEE, pp. 912–922.

Azáiez M, Lestandi L and Chacón Rebollo T (2019) Low rank approximation of multidimensional data. In: *High-Performance Computing of Big Data for Turbulence and Combustion*. Springer, pp. 187–250.

Bauer M, Treichler S, Slaughter E, et al. (2012) Legion: Expressing locality and independence with logical regions. In: SC ’12: proceedings of the international conference on high performance computing, networking, storage and analysis, Salt Lake City, UT, USA, 10–16 November, 2012, pp. 1–11. DOI: 10.1109/SC.2012.71

Brand M (2006) Fast low-rank modifications of the thin singular value decomposition. *Linear Algebra and Its Applications* 415(1): 20–30.

Cheng H, G nibutas Z, Martinsson PG, et al. (2005) On the compression of low rank matrices. *SIAM Journal on Scientific Computing* 26(4): 1389–1404.

Cohen A, Daubechies I and Feauveau J-C (1992) Biorthogonal bases of compactly supported wavelets. *Communications on Pure and Applied Mathematics* 45(5): 485–560.
De Lathauwer L, De Moor B and Vandewalle J (2000) On the best
rank-1 and rank-(R1,R2, . . . ,RN) approximation of higher-
order tensors, (m) approximation of higher-order tensors. SIAM
Journal on Matrix Analysis and Applications 21(4): 1324–1342.

Di S and Cappello F (2016) Fast error-bounded lossy hpc data
compression with sz. In: 2016 ieee international parallel and
distributed processing symposium (ipdps), Chicago, IL,
USA, 23–27 May 2016, IEEE, pp. 730–739.

DiRenzo M, Fu L and Urzay J (2020) Htr solver: An open-source
library, 22, pp. 438–458.

Dunton AM, Jofre L, Iaccarino G, et al. (2020) Pass-efficient
methods for compression of high-dimensional turbulent flow
data. Journal of Computational Physics 423: 109704.

Dyer EL, Goldstein TA, Patel R, et al. (2015) Self-expressive
decompositions for matrix approximation and clustering.
arXiv preprint arXiv:1505.00824.

Elhamifar E and Vidal R (2009) Sparse subspace clustering. In:
IEEE conference on computer vision and pattern recognition,
Miami, FL, USA, 20–25 June 2009, IEEE, pp. 2790–2797.

Farge M (1992) Wavelet transforms and their applications to
turbulence. Annual Review of Fluid Mechanics 24(1):
395–458.

Fowler JE and Yagel R (1994) Lossless compression of volume
data. In: Proceedings of the 1994 symposium on volume
visualization, Oct 17-18, 1994, Tysons Corner, VA, ACM
pp. 43–50.

Gerber R, Hack J, Riley K, et al. (2018) Crosscut Report: Exascale
Requirements Reviews, March 9–10, 2017–Tysons Corner,
Virginia. An Office of Science Review Sponsored by: Ad-
vanced Scientific Computing Research, Basic Energy Sci-
ences, Biological and Environmental Research, Fusion
Energy Sciences, High Energy Physics, Nuclear Physics.
Oak Ridge, TN (United States)Argonne: Oak Ridge National
Lab.(ORNL). Technical report.

Giaws A, King R and Sprague M (2020) Deep learning for in situ
data compression of large turbulent flow simulations.
Physical Review Fluids 5(11): 114602.

Golub GH and Van Loan CF (2012) Matrix Computations. JHU
press, Vol. 3.

Gong Z, Lakshminarasimhan S, Jenkins J, et al. (2012) Multi-level
layout optimization for efficient spatio-temporal queries on
isabela:compressed data. In: 2012 IEEE 26th international
parallel and distributed processing symposium, Shanghai,
China, 21–25 May, 2012. IEEE, pp. 873–884.

Gu M and Eisenstat SC (1996) Efficient algorithms for computing
a strong rank-revealing qr factorization. SIAM Journal on
Scientific Computing 17(4): 848–869.

Gu Y, Napier Z and Sun Y (2021) Analysis of Work-Stealing and
Parallel Cache Complexity. CoRR. abs/2111.04994 [Online].
Available at: https://arxiv.org/abs/2111.04994

Halko N, Martinsson PG and Tropp JA (2011) Finding structure
with randomness: Probabilistic algorithms for constructing
approximate matrix decompositions. SIAM Review 53(2):
217–288.

Hampton J, Fairbanks HR, Narayan A, et al. (2018) Practical error
bounds for a non-intrusive bi-fidelity approach to parametric/
stochastic model reduction. Journal of Computational
Physics 368: 315–332.

Hitchcock FL (1927) The expression of a tensor or a polyadic as a
sum of products. Journal of Mathematics and Physics 6(1–4):
164–189.

Ibarria L, Lindstrom P, Rossignac J, et al. (2003) Out-of-core
compression and decompression of large n-dimensional
scalar fields. In: Computer Graphics Forum. Wiley Online
Library, 22, pp. 343–348.

Kennedy CA, Carpenter MH and Lewis RM (2000) Low-storage,
explicit Runge-Kutta schemes for the compressible Naver-
Stokes equations. Applied Numerical Mathematics 35:
177–219.

Kroonenberg PM and De Leeuw J (1980) Principal component
analysis of three-mode data by means of alternating least
squares algorithms. Psychometrika 45(1): 69–97.

Kunkel JM, Kuhn M and Ludwig T (2014) Exascale storage
systems: an analytical study of expenses. Supercomputing
Frontiers and Innovations 1(1): 116–134.

Lakshminarasimhan S, Shah N, Ethier S, et al. (2011) Compressing
the incompressible with isabela: in-situ reduction of spatio-
temporal data. In: European Conference on Parallel Pro-
cessing. Berlin, Germany: Springer, pp. 366–379.

Leffell JI, Sitaraman J, Lakshminarayana VK, et al. (2016) Towards
efficient parallel-in-time simulation of periodic flows. In: 54th
AIAA aerospace sciences meeting, Jan 4-8 2016, San Diego,
CA, AIAA. pp. 1–34. URL: DOI: 10.2514/6.2016-0066
https://arc.aiaa.org/doi/abs/10.2514/6.2016-0066

LegionSpy (2020) Legion programming system: Debugging. URL
https://legion.stanford.edu/debugging/

Stanford High Performance Computing Center. http://hpcc. stanford.edu/

Lehmann H and Jung B (2014) In-situ multi-resolution and
temporal data compression for visual exploration of large-
scale scientific simulations. In: 2014 ieee 4th symposium on
large data analysis and visualization (ldav), Paris, France, 9–
10 November, 2014, IEEE, pp. 51–58.

Li S, Marsaglia N, Garth C, et al. (2018) Data reduction techniques
for simulation, visualization and data analysis. In: Computer
Graphics Forum. Wiley Online Library, 37, pp. 422–447.

Liang X, Di S, Tao D, et al. (2018) Error-controlled lossy com-
pression optimized for high compression ratios of scientific
datasets. In: IEEE International Conference on Big Data (Big
Data), Seattle, WA, USA, 10–13 December, 2018. IEEE,
pp. 438–447.

Liberty E, Woolfe F, Martinsson P-G, et al. (2007) Randomized
algorithms for the low-rank approximation of matrices.
Pacella H (2014) Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics* 20(12): 2674–2683.

Lindstrom P and Isenburg M (2006) Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics* 12(5): 1245–1250.

Loève M (1977) Elementary probability theory. In: *Probability Theory I*. Springer, pp. 1–52.

Mahoney MW and Drineas P (2009) Cur matrix decompositions for improved data analysis. *Proceedings of the National Academy of Sciences* 106(3): 697–702.

Marin O, Schanen M and Fischer P (2016) Large-scale Lossy Data Compression Based on an a Priori Error Estimator in a Spectral Element Code. Technical report ANL/MCS-P6024-0616.

Martinsson P-G (2018) Randomized methods for matrix computations. *IAS/Park City Mathematics Series* 25: 187–229.

Martinsson P-G, Rokhlin V and Tygert M (2011) A randomized algorithm for the decomposition of matrices. *Applied and Computational Harmonic Analysis* 30: 47–68.

Morris K, Badia RM and Chamberlain BL (2020) Message from the workshop chairs. In: 2020 IEEE/ACM 3rd annual parallel applications workshop: Alternatives to MPI+X (PAW-ATM), Nov 9-19, 2020, Virtual Event, IEEE.pp. 5. DOI: 10.1109/PAWATM51920.2020.00004

Otero E, Vinuesa R, Marin O, et al. (2018) Lossy data compression effects on wall-bounded turbulence: Bounds on data reduction. *Flow, Turbulence and Combustion* 101(2): 365–387.

Pacella H (2021) Modern Programming Models for GPU-Accelerated Heterogeneous Supercomputers: Computational Fluid Dynamics and In Situ Data Compression. PhD thesis. Stanford University.

Perry DJ, Kirby RM, Narayan A, et al. (2019) Allocation strategies for high fidelity models in the multifidelity regime. *SIAM Journal on Uncertainty Quantification* 7(1): 203–231.

Pi Y, Peng H, Zhou S, et al. (2013) A scalable approach to column-based low-rank matrix approximation. In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, August 3-9, 2013, Beijing, China, IJCAI. pp. 1600–1606.

Rasquin M, Smith C, Chitale K, et al. (2014) Scalable implicit flow solver for realistic wing simulations with flow control. *Computing in Science & Engineering* 16(6): 13–21.

Rauber T and Rünger G (2010) *Parallel Programming for Multicore and Cluster Systems*. Springer.

Slaughter E, Lee W, Treichler S, et al. (2015) Regent: high-productivity programming language for hpc with logical regions. In: *SC ’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12. DOI: 10.1145/2807591.2807629

Sprague MA, Boldyrev S, Fischer P, et al. (2017) *Turbulent Flow Simulation at the Exascale: Opportunities and Challenges Workshop: August 4-5, 2015, Washington, DC*. Golden, CO (United States): National Renewable Energy Lab.(NREL). Technical report.

Strang G and Nguyen T (1996) *Wavelets and Filter Banks*. SIAM.

Tao D, Di S, Chen Z, et al. (2017) Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In: *IEEE international parallel and distributed processing symposium (IPDPS)*, May 29-June 2, 2017, Orlando, FL IEEE, pp. 1129–1139.

Therrien CW (1992) *Discrete Random Signals and Statistical Signal Processing*. Prentice Hall PTR.

Torres H (2021) *Simulations of Particle Laden Turbulence in a Radiation Environment Using Heterogeneous High Performance Computing Systems*. PhD thesis. Stanford University.

Treichler S, Bauer M, Bhagatwala A, et al. (2017) *S3D-Legion*, Exascale Scientific Applications. In KB Straatsma, TP Antypas and TJ Williams, eds, *Scalability and Performance Portability*, pages 257–278. Chapman and Hall/CRC, New York, NY, USA.

Tropp JA, Yurtsever A, Udell M, et al. (2019) Streaming low-rank matrix approximation with an application to scientific simulation. *SIAM Journal on Scientific Computing* 41(4): A2430–A2463.

Tucker LR (1966) Some mathematical notes on three-mode factor analysis. *Psychometrika* 31(3): 279–311.

Vannieuwenhoven N, Vandebril R and Meerbergen K (2012) A new truncation strategy for the higher-order singular value decomposition. *SIAM Journal on Scientific Computing* 34(2): A1027–A1052.

Wissink A, Kamkar S, Pulliam T, et al. (2010) Cartesian adaptive mesh refinement for rotofcraft wake resolution. In: 28th AIAA applied aerodynamics conference, June 28-Jul 1, 2010, Chicago, IL, AIAA. pp. 1–18. DOI: 10.2514/6.2010-4554

Woodruff DP (2014) *Sketching as a Tool for Numerical Linear Algebra*. arXiv preprint arXiv:1411.4357.

Boon-Lock Yeo BL and Bede Liu B (1995) *Volume rendering of floating-point data*. *IEEE Transactions on Visualization and Computer Graphics* 1(1): 29–43.

Yu W, Gu Y, Li J, et al. (2017) *Single-pass Pca of Large High-Dimensional Data*. arXiv preprint arXiv:1704.07669.

Zimmermann R, Peherstorfer B and Willcox K (2018) Geometric subspace updates with applications to online adaptive nonlinear model reduction. *SIAM Journal on Matrix Analysis and Applications* 39(1): 234–261.
Author biographies

Heather Pacella is a Postdoctoral Appointee at Sandia National Laboratories. She earned her PhD in Mechanical Engineering from Stanford University (2021). Before that, Heather obtained her BS in Mechanical Engineering from the University of Pittsburgh (2010), her MS in Aerospace Engineering from North Carolina State University (2014), and her MS in Computational & Mathematical Engineering from Stanford (2017). She also spent five years as a mechanical engineer for the US Army’s Armament Research, Development, and Engineering Center. Heather’s research focus is on the use of task-based parallelism for in situ data compression of large-scale computational fluid dynamics simulations, with an emphasis on developing heterogeneous algorithms for heterogeneous computing architectures.

Alec Dunton is a machine learning postdoctoral researcher in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. Prior to his postdoctoral appointment at Livermore, he was a graduate researcher in the Uncertainty Quantification, Learning and Estimation from Data group at the University of Colorado Boulder, earning his PhD in Applied Mathematics (2021). He obtained his BS in Mathematics from Harvey Mudd College (2016) and his MS in Applied Mathematics from the University of Colorado Boulder (2018). Alec’s research focus is on linear and nonlinear approaches for dimensionality reduction of large-scale data.

Alireza Doostan is an H. Joseph Smead Faculty Fellow and Associate Professor of Aerospace Engineering Sciences Department at the University of Colorado Boulder. He is also the director of the Aerospace Mechanics Research Center (AMReC) and an affiliated faculty of the Applied Mathematics Department. Prior to his appointment at CU Boulder in 2010, he was an Engineering Research Associate in the Center for Turbulence Research at Stanford University. Alireza received his PhD in Structural Engineering and M.A. in Applied Mathematics and Statistics from the Johns Hopkins University both in 2007. He is a recipient of both DOE (ASCR) and NSF (Engineering Design) Early Career awards, as well as multiple teaching awards from CU Boulder and AIAA. His research interests include: Uncertainty quantification, reduced order and data-driven modeling, optimization under uncertainty, and computational stochastic mechanics.

Gianluca Iaccarino is professor in Mechanical Engineering and Director of the Institute for Computational Mathematical Engineering at Stanford University. He received his PhD from the Politecnico di Bari and worked for several years at the Center for Turbulence Research (NASA Ames & Stanford) before joining the faculty at Stanford in 2007. Since 2014 he has served as the Director of the PSAAP Center at Stanford, funded by the US Department of Energy: a $20M research Center focused on multiphysics simulations, uncertainty quantification and exascale computing. In 2010, Professor Iaccarino received the Presidential Early Career Award for Scientists and Engineers (PECASE) award. He received best paper awards from AIAA, ASME IMECE and Turbo Expo Conferences. Professor Iaccarino’s interests in research and teaching revolve around the use of computing and data to solve problems in energy, biomedicine, aerodynamics, propulsion, and design.

Appendix I

The following lemma enumerates some of the properties of the ID.

Lemma A.1. (Lemma 3.1 of Martinsson et al. (2011)). For a matrix $C \in \mathbb{R}^{m \times n}$ there exists a set $\mathcal{I} \subseteq \{0, \ldots, n-1\}$ with $|\mathcal{I}| = k$, a corresponding column skeleton $A(\cdot, \mathcal{I}) \in \mathbb{R}^{m \times k}$, and coefficient matrix $A \in \mathbb{R}^{m \times n}$ which satisfies the following properties.

1. $C(\cdot, \mathcal{I})$ is the $k \times k$ identity matrix.
2. no entry of $C$ has absolute value greater than 1.
3. $\|C\|_2 \leq \sqrt{k(n-k) + 1}$
4. the $k$th largest singular value of $C$, $\sigma_k$, is at least one.
5. $A = A(\cdot, \mathcal{I})C$ when $k = m$ or $k = n$.
6. $\|A - A(\cdot, \mathcal{I})C\|_2 \leq \sqrt{k(n-k) + 1}\sigma_{k+1}$

The following two theorems provide error estimates for SubID and SPID. Note that Theorem A.2 is essentially a restatement of Theorem 1 from Hampton et al. (2018).

Theorem A.2. (Theorem 1 of Dunton et al. (2020)). Let $A$ be a PDE data matrix, $B = A(\mathcal{J}, \cdot)$ the subsampled (coarsened) matrix, $B$ the rank $k$ ID approximation to $B$ as in (16), and $\hat{A}$ the subsampled ID approximation as in (17). For any $\tau \geq 0$, let

$$\epsilon(\tau) := \lambda_{\text{max}}(AA^T - \tau BB^T)$$

where $\lambda_{\text{max}}$ denotes the largest eigenvalue. Then

$$\|A - \hat{A}\|_2 \leq \min_{\tau, k \leq \text{rank}(B)} \rho_k(\tau)$$

where $\rho_k(\tau)$ is the $k$th largest eigenvalue of $\rho_k(\tau)(BB^T)^{-1}$. Theorem A.2 has been adapted from Theorem 1 in Dunton et al. (2020).
\[ \rho_k(\tau) := (1 + \|C\|_2)\sqrt{\tau \sigma^2_{k+1} + \epsilon(\tau)} \]

\[ + \left\| B - \tilde{B} \right\|_2 \sqrt{\tau + \epsilon(\tau)\sigma^{-2}} \]

(63)

where \( \sigma_k \) and \( \sigma_{k+1} \) are the \( k^{th} \) and \((k+1)^{th}\) largest singular values of \( B \), respectively.

**Theorem A.1.** (Theorem 2 of Dunton et al. (2020)). Let \( A \) be a PDE data matrix, \( B \) the subsampled (coarsened) matrix, \( M \) the interpolation operator as in (18) and with associated interpolation error \( E_i := A - MB \). Then, the error of the single-pass ID approximation \( \hat{A} \) in (18) is bounded as follows

\[ \left\| A - \hat{A} \right\|_2 \leq \|E_i\|_2 + \|M\|_2 \left\| B - \tilde{B} \right\|_2 \]

(65)