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

Heather Pacella
Dept. of Mech. Engineering
Stanford University
Stanford, USA
hpacella@stanford.edu

Alec Dunton
Dept. of Appl. Mathematics
University of Colorado Boulder
Boulder, USA
alec.dunton@colorado.edu

Alireza Doostan
Smead Aerospace Engr. Sciences
University of Colorado Boulder
Boulder, USA
alireza.doostan@colorado.edu

Gianluca Iaccarino
Dept. of Mech. Engineering
Stanford University
Stanford, USA
jops@stanford.edu

***Abstract*—Present day computational fluid dynamics simulations generate extremely large amounts of data; most of this data is discarded because current storage systems are unable to keep pace. Data compression algorithms can be applied to this data to reduce the overall amount of storage while either exactly retaining the original dataset (lossless compression) or an approximate representation of the original dataset (lossy compression). *Interpolative decomposition* (ID) is a type of lossy compression that factors the original data matrix as the product of two (smaller) matrices; one of these matrices consists of columns of the original data matrix, while the other is a coefficient matrix. The structure of ID algorithms make them a natural fit for task-based parallelism. Our presented work will specifically focus on using the task-based Legion programming model to implement a single-pass ID algorithm (SPID) in several fluid dynamics applications. Performance studies, scalability, and the accuracy of the compressed results will be discussed in detail during our presentation.**

***Index Terms*—Computational Fluid Dynamics, Data Compression, Regent, Legion, Task Based Parallelism**

I. Introduction and Motivation

Advances in supercomputing over the past several years have introduced a computational bottleneck. Soon-to-be-deployed exascale computers offer a $10^3 - 10^4$-fold increase in floating point performance, but provide a mere 10-fold increase in the amount of memory available and access speed [1]. This asymmetric technological progress leads to the following issue: floating point operations (FLOPs) are cheap, while memory, communication, and input/output (I/O) are not. For example, the amount of data generated on these computers will easily exceed 1 TB/s; if this is not reduced, storage systems will easily become overloaded and practitioners will not be able to use the data for visualization, analysis, or other post-processing operations. This has inspired work in the field of *data compression*, in which an accurate, memory-reduced version of an original dataset is stored.

All data compression algorithms may be categorized as either 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 the size of the compressed dataset. Due to the exorbitant size of data generated on modern supercomputers, in particular in the physical simulations with which we are interested [2], we are instead interested in lossy data compression. Lossy compression methods, unlike their lossless counterparts, do not enable exact reconstruction of datasets. They do, however, allow for much larger compression factors. In this work, low-rank matrix approximations are used to achieve lossy compression.

II. Interpolative Decomposition

The data matrices we are concerned with, which generally are computed from the simulation of partial differential equations, have the form $\boldsymbol{A} \in \mathbb{R}^{m \times n}$ such that each column corresponds to a specific time in the simulation, where each entry of the column contains data for a physical quantity of interest (QoI) (such as pressure, velocity, or flux) at a discrete location within the spatial domain. Our data matrix is therefore structured as shown in Figure 1.

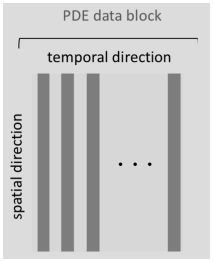

Fig. 1: Schematic of PDE data block

If the spatial or temporal variation of the QoI is smooth, the matrix $\boldsymbol{A}$ may admit a low-rank structure; its rows or columns may contain redundant information. An assumption crucial to this work is that $\boldsymbol{A}$ is *numerically low rank*: that its singular value decay is sharp and the matrix may therefore be approximated with minimal reconstruction error. Equivalently, this means that the the QoI may be accurately reconstructed from a linear subspace much smaller than that spanned by the original data columns.

One direct consequence of this assumption is that we may obtain an approximation to the original PDE data matrix $\boldsymbol{A}$ via the matrix-matrix product

$$\boldsymbol{A} \approx \boldsymbol{A}(:, \mathcal{I})\boldsymbol{P}, \tag{1}$$

where $\boldsymbol{P} \in \mathbb{R}^{k \times n}$ is referred to as the *coefficient matrix* and $\boldsymbol{A}(:, \mathcal{I}) \in \mathbb{R}^{m \times k}$, which is a subset of the columns

of $\boldsymbol{A}$ indexed by $\mathcal{I}$, is called the *column skeleton* of $\boldsymbol{A}$. The decomposition (1) is the so-called column interpolative decomposition (column ID) of a matrix $\boldsymbol{A}$ [3]. The number of columns used in the decomposition, k, is determined by the user by providing either (1) an error tolerance stopping criterion or (2) fixing k ahead of time. The columns are selected using a pivoted QR routine based on a modified Gram-Schmidt procedure.

There are numerous schemes to produce a column ID; we present two different algorithms: the sub-sampled ID (SubID) and the single-pass ID (SPID). SubID is a two pass algorithm for computing the column ID of a PDE data matrix: one pass is needed to compute the indices $\mathcal{I}$ and coefficient matrix $\boldsymbol{P}$ using a coarse grid representation of the data –compatible with the PDE solver– to expedite computation, then a second pass to obtain the column skeleton $\boldsymbol{A}(:, \mathcal{I})$. When the amount of data generated in the PDE solve exceeds the storage available in RAM or disk, the user may be required to run a second simulation in order to construct $\boldsymbol{A}(:, \mathcal{I})$. In order to avoid this second pass over $\boldsymbol{A}$ – or need to run a second simulation – we present a single-pass variation of SubID: the so-called single-pass ID (SPID) [4]. In SPID, the column skeleton is computed by interpolating the coarse grid data back onto the fine grid, as opposed to running a second simulation to obtain the original fine grid column skeleton. Our presentation will focus solely on the *in situ* implementation of the SPID algorithm, since it is more efficient than the subID algorithm.

III. ID Methods and Task-based Parallelism

A. Parallelism of CFD codes

Computational fluid dynamics (CFD) simulations are concerned with the numerical solution of a set of partial differential equations on a discretized grid representation of the physical domain of interest. For large simulations, this discretized domain is partitioned into subdomains (blocks) of data, and the blocks are distributed across the available computing resources. This strategy of extracting concurrency from a given computational approach is referred to as data-parallelism. In general, 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), and then there is a communication stage as relevant data is exchanged between the blocks of the domain (e.g., information from stencil points for a finite difference method). 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.

B. Parallelism of ID methods

From a computational perspective, ID greatly differs from CFD simulations. ID algorithms can be applied independently to each block of the decomposed domain, so that no communication between the blocks is required. (In fact, applying ID individually to blocks of data can help to increase the overall compression. Generally, due to the structure of the data matrices of interest, adjacent entries in the matrix correspond to adjacent points in the spatial domain. Because of this, there tends to be more correlation (low-rankness) between adjacent data in the matrices (due to neighboring spatial points) than the entire matrix, so partitioning the data matrix is a favorable approach.

For CFD analyses, the independence of the ID algorithm means that, in theory, the computations required to perform the ID analysis on the most recent solution step can be performed while waiting for the data synchronization required to compute the fluid solver update for the next simulation step. Thus, the overhead cost of performing the ID approximation can be hidden if the computing system has a large amount of communication latency (as is often the case for extremely large simulations). Additionally, if the quantity of interest 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. For both of these reasons, implementing ID within a synchronous or barrier-synchronous parallelization model (traditionally implemented using OpenMP, MPI, CUDA, etc.) like those mentioned above will lead to large reductions in performance as the simulation becomes increasingly complex and/or large. Instead, we use an *asynchronous* parallelization model that removes these barriers. We select a programming model that is well-suited to simulations that are run on modern, heterogeneous supercomputers, as ID is generally applied to large simulations that require data compression.

C. Task-based Parallelization of ID methods in CFD codes

Task-based (or functional) parallelism is one such model that meets these requirements [5]. A task-parallel CFD code will have a layout like that shown in Fig. 2. Each processor contains local data about a subsection of the physical domain. At the i^{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 advance the fluid solver in time. This process repeats until the final simulation time is reached.

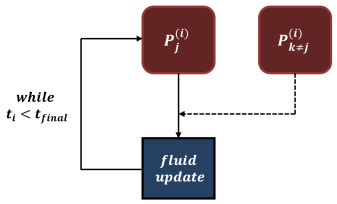

Fig. 2: CFD Analysis

As shown in Algorithm 1, 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 only depend upon the communication of the stencil data between processors.

Algorithm 1 Task-Parallel CFD Solver

```
while t < t_final do
    for id = 0, N do
        update_fluid_step(block[id], stencil[id])
    end for
end while
```

To compute the SPID, each processor must again perform the fluid update at each time step, but now the data must also be stored, which requires the addition of the $store\_subsampled\_data$ task. Often, if the number of time steps, n, is very large (on the order of the number required for an explicit time-advancement scheme), then the amount of memory required to store and perform ID on the fluid data matrix $\boldsymbol{A}$ greatly exceeds what is available, even on large distributed-memory systems. To circumvent this problem, we apply the ID algorithm in a two-step, piece-wise manner. The first step applies the ID algorithm to N individual blocks of time steps of the domain, so that the following matrix, which we denote $\boldsymbol{A^1}$, is formed:

$$[\boldsymbol{A_0}(:,\mathcal{I}_0)\boldsymbol{P_0}\cdots\boldsymbol{A_{N-1}}(:,\mathcal{I}_{\mathcal{N}-1})\boldsymbol{P_{N-1}}], \tag{2}$$

where index 0 corresponds to the first $\frac{n}{N}$ time steps, index 1 corresponds to the next successive $\frac{n}{N}$ time steps, and so on until index $N-1$.

We may then decompose $\boldsymbol{A^1}$ to be the product of the following two matrices

$$\boldsymbol{A^1} = \boldsymbol{A^{1'}}\boldsymbol{P^{1'}} \tag{3}$$

$$= [\boldsymbol{A_0}(:,\mathcal{I}_0)\cdots\boldsymbol{A_{N-1}}(:,\mathcal{I}_{\mathcal{N}-1})]\boldsymbol{P^{1'}}, \tag{4}$$

where $\boldsymbol{P^{1'}} \in \mathbb{R}^{k(N-1)\times N}$ is a sparse matrix comprised of rectangular blocks given coefficient matrices $\boldsymbol{P}_i$:

$$\boldsymbol{P^{1'}} = \bigoplus_{i=0}^{N-1} \boldsymbol{P}_i. \tag{5}$$

To eliminate any redundant columns in the horizontally concatenated column skeleton matrix $\boldsymbol{A^{1'}}$ and achieve further compression, we apply the ID algorithm to $\boldsymbol{A^{1'}}$ to obtain the following approximation:

$$\boldsymbol{A^{1'}} \approx \boldsymbol{A^{2'}}\boldsymbol{P^2}, \tag{6}$$

where

$$\boldsymbol{A^{2'}} = \boldsymbol{A^1}(:,\mathcal{I}_{\mathcal{U}}), \tag{7}$$

such that $\mathcal{I}_U$ indexes the columns of the original matrix $\boldsymbol{A^1}$ extracted following the second application of column ID; the subscript U suggests that these indices come from the union of the index vectors $\mathcal{I}_j$ in $\boldsymbol{A^{1'}}$. A similar approach to this is employed in [6].

Then, the final ID approximation of the domain is

$$\boldsymbol{A} = \boldsymbol{A^{1'}}\boldsymbol{P^{1'}} \approx \boldsymbol{A^{2'}}\boldsymbol{P^2}\boldsymbol{P^{1'}} = \boldsymbol{A^{2'}}\boldsymbol{P^{2'}}. \tag{8}$$

Where $\boldsymbol{A^{2'}} \in \mathbb{R}^{m\times|\mathcal{I}_U|}$ and $\boldsymbol{P^{2'}} \in \mathbb{R}^{|\mathcal{I}_U|\times n}$.

The structure of a CFD solver with this ID implementation is shown in Fig. 3. 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.

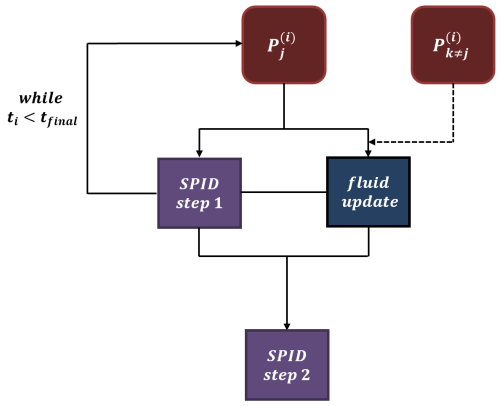

Fig. 3: CFD Analysis with modified SPID

This is shown in Algorithm 2. 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}, where the time step at t_{crit} is some multiple of $\frac{n}{N}$. 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 $second\_SPID$ is not launched until the completion of the fluid solve, it is operating on a much smaller matrix than the entire original data matrix.

IV. Implementation and Numerical Results

Legion, an open-source, collaborative effort between Stanford University, LANL, NVIDIA Research, and SLAC, is a parallel programming model that uses task-based parallelism to create highly efficient and portable code for high-performance computing applications. The main unit of abstraction in Legion is the *logical region*. These logical regions allow precise definition of how data is being used by the various tasks of an application, and can be further partitioned into *subregions*. Legion builds a task graph that analyzes the relationship between tasks and these subregions, and the Legion runtime *dynamically* assigns tasks to computing units (CPUs, GPUs, etc.) [7]. We chose the Legion programming

Algorithm 2 Task-Parallel CFD Solver with modified SPID

```
while t < t_final do
    for id = 0, N do
        update_fluid_step(block[id], stencil[id])
        save_subsampled_data(block[id])
        if t == t_crit then
            first_SPID(subsampled_block[id])
        end if
    end for
end while
for id = 0, N do
    second_SPID(subsampled_block[id])
end for
```

model to implement our CFD + ID applications for several reasons:

- Legion, unlike applications using MPI+X, allows the user to write a single code that can then be ported to multiple architectures. A single Legion application can be run on a laptop or a leadership-class machine, on CPUs, GPUs, or a combination of the two.
- Regent, a high-level programming language within Legion, allows for implicit parallelism to be extracted from a serial code.
- The Legion runtime dynamically schedules tasks; this allows for dynamic load balancing with little to no user input.

Our presentation will focus on the successful Legion implementation of the SPID algorithm into two fluid dynamics examples: the analytical solution of the incompressible 2D Taylor-Green vortex, and the numerical solution of a compressible 3D Taylor-Green vortex using a high-order Cartesian Navier-Stokes solver [8].

Results presented for the incompressible 2D Taylor-Green vortex solution will demonstrate that, for both a structured and an unstructured grid, the ID implementation will reconstruct to machine precision the exact vortex solution independent of physical partitions in the domain.

More extensive attention will be paid to the Navier-Stokes + ID solver. This implementation is of particular interest because of the communication-intensive nature of the numerical scheme used for the fluid solver, a high-order three-stage Runge-Kutta method. This discussion will include:

- The accuracy of the reconstructed ID solution.
- Visualizations from the Legion Prof profiling tool, which visualizes exactly how the application runs of the machine (how the tasks are executed across the available processors), as well as overall resource utilization.
- Performance and scaling studies. An example of this is Fig. 4, which shows strong scaling results (normalized to the single node Navier-Stokes solve) for both the Navier-Stokes solver and the ID-augmented Navier-Stokes solver for 10 time steps of a 128^3 point domain with 4^3 domain partitions. The increase in runtime due to the introduction of the ID algorithm was, at most, 10% that of the Navier-Stokes solver.

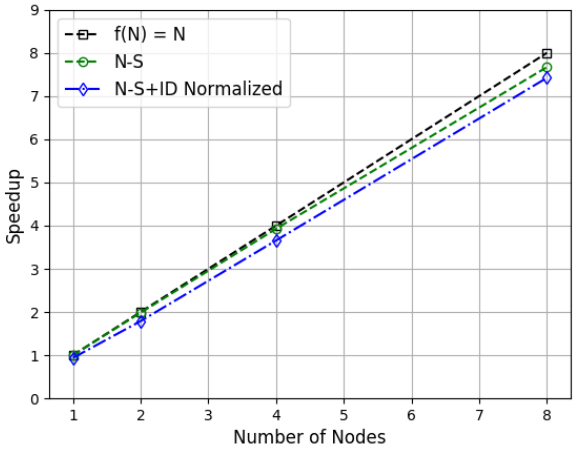

Fig. 4: Strong scaling normalized w.r.t. one node runtime; runs were performed on Stanford University's Yellowstone cluster

References

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

[2] H. Torres, M. Papadakis, L. Jofre, W. Lee, A. Aiken, and G. Iaccarino, "Soleil-x: Turbulence, particles, and radiation in the regent programming language," in *Parallel Applications Workshop, Alternatives To MPI+X*, 2019.

[3] H. Cheng, Z. Gimbutas, P.-G. Martinsson, and V. Rokhlin, "On the compression of low rank matrices," *SIAM Journal on Scientific Computing*, vol. 26, no. 4, pp. 1389–1404, 2005.

[4] A. M. Dunton, L. Jofre, G. Iaccarino, and A. Doostan, "Pass-efficient methods for compression of high-dimensional turbulent flow data," 2019.

[5] T. Rauber and G. Rünger, *Parallel Programming for Multicore and Cluster Systems*. Springer, 2010.

[6] Y. Pi, H. Peng, S. Zhou, and Z. Zhang, "A scalable approach to column-based low-rank matrix approximation," in *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, 2013, pp. 1600–1606.

[7] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Supercomputing Conference (SC12)*, 2012.

[8] J. Leffell, J. Sitaramen, V. Lakshminarayan, and A. Wissink, "Towards efficient parallel-in-time simulation of periodic flow," in *The 54th AIAA Aerospace Sciences Meeting*, 2016.