AIAA SciTech Forum
11–15 & 19–21 January 2021, VIRTUAL EVENT
AIAA Scitech 2021 Forum

10.2514/6.2021-0140

Development of a discontinuous Galerkin solver using Legion for heterogeneous high-performance computing architectures

Kihiro Bando*[1], Steven R. Brill[1], Elliott Slaughter[1], Michael Sekachev[2], Alex Aiken[1], and Matthias Ihme[1]
[1]*Stanford University, Stanford, CA 94305, United States*
[2]*Total E&P Research & Technology, United States, LLC*

This work discusses the development, verification and performance assessment of a discontinuous Galerkin solver for the compressible Navier-Stokes equations using the Legion programming system. This is motivated by (i) the potential of this family of high-order numerical methods to accurately and efficiently realize scale-resolving simulations on unstructured grids and (ii) the desire to accommodate the utilization of emerging compute platforms that exhibit increased parallelism and heterogeneity. As a task-based programming model specifically designed for performance portability across distributed heterogeneous architectures, Legion represents an interesting alternative to the traditional approach of using Message Passing Interface for massively parallel computational physics solvers. Following a detailed discussion of the implementation, the high-order convergence of the solver is demonstrated by a suite of canonical test cases and good strong scaling behavior is obtained. This work constitutes a first step towards a research platform that is able to be deployed and efficiently run on modern supercomputers.

I. Introduction

SCALE-RESOLVING simulations constitute a promising way towards improving the accuracy of computational fluid dynamics (CFD) tools for turbulent flows of industrial relevance involving complex geometries [1–3]. The modeling limitations of Reynolds-averaged Navier-Stokes (RANS) approaches for unsteady turbulent flows motivate research in order to make large-eddy simulations (LES) more accurate, robust and affordable. Numerical schemes play an important part in that effort as evidenced by recent advances in high-order methods that are compatible with unstructured mixed irregular grids, such as the discontinuous Galerkin (DG) method [4]. They are characterized by a high-order convergence regardless of the shape of the elements, a compact stencil and have potential for sophisticated adaptation strategies. Their dissipation/dispersion characteristics when combined with upwind numerical fluxes mimic to some extent the physical sub-grid scale dissipation which lead to their use in the context of implicit LES [5–7]. Other works explored a low dissipation approach combined with an explicit sub-grid scale model [8, 9]. Although these methods demand more floating-point operations per degree of freedom compared to a typical formally second-order finite volume method for general grids and are affected by more stringent time-step restrictions with explicit time-stepping, several studies have shown that the cost-to-accuracy ratio can be in their favor when unstructured meshes are employed [10, 11].

DG implementations are characterized by a higher arithmetic intensity and the use of standard linear algebra operations making them well suited for application to modern computing hardware such as graphics processing units (GPU). Several studies have shown the possibility and the performance benefits obtained by porting computational kernels to accelerators [12–14]. Meanwhile the increasing heterogeneity within new high-performance computing nodes and between supercomputers makes the development of performance portable solvers increasingly challenging. The PyFR code [12] is one example that achieves performance portability across hardware types by using a domain-specific low-level code generation system from a single template. It uses the Message Passing Interface (MPI) programming system for memory transfers, which requires explicit communication directives. While MPI remains the most popular parallel programming system used by CFD codes, guaranteeing correctness and good performances on different architectures is a potential burden due to its explicit nature. In order to overcome this, recent developments by the computer science community have shown the potential of using runtimes to manage the extraction of parallelism and memory operations.

In this paper, the design and development of a DG method to solve the compressible Navier-Stokes equations using the Legion programming system [15] is presented. Legion is a runtime-based alternative to MPI for applications

*bandok@stanford.edu

targeting distributed heterogeneous systems. The mathematical framework is presented in section II. The Legion implementation is detailed and discussed from the perspective of an application developer in section III. Section IV presents a complete verification of the solver. Code performance is assessed in section V. The paper finishes with conclusions and discusses future work in section VI.

II. Mathematical formulation

A. Governing equations

The compressible Navier-Stokes equations are solved and are written in vector form as

$$\partial_t \mathbf{U} + \nabla \cdot \mathbf{F}_c = \nabla \cdot \mathbf{F}_d, \tag{1}$$

where $\mathbf{U} : \mathbb{R}^+ \times \mathbb{R}^{N_D} \to \mathbb{R}^{N_U}$ is the vector of conservative variables, $\mathbf{F}_c : \mathbb{R}^{N_U} \to \mathbb{R}^{N_U \times N_D}$ is the convective flux and $\mathbf{F}_d : \mathbb{R}^{N_U} \times \mathbb{R}^{N_U \times N_D} \to \mathbb{R}^{N_U \times N_D}$ is the diffusive flux. Here, N_U denotes the number of state variables and N_D is the number of spatial dimensions. The vectors can be expanded as

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho \boldsymbol{u} \\ \rho E \end{pmatrix}, \quad \mathbf{F}_c = \begin{pmatrix} \rho \boldsymbol{u}^T \\ \rho \boldsymbol{u} \otimes \boldsymbol{u} + P\mathbb{I} \\ (\rho E + P)\boldsymbol{u}^T \end{pmatrix}, \quad \mathbf{F}_d = \begin{pmatrix} 0 \\ \boldsymbol{\tau} \\ -\boldsymbol{q}^T + (\boldsymbol{\tau} \cdot \boldsymbol{u})^T \end{pmatrix}, \tag{2}$$

where ρ is the density, $\boldsymbol{u}$ is the velocity vector, E is the total specific energy, P is the pressure, $\mathbb{I}$ is the identity tensor, $\boldsymbol{\tau}$ is the viscous stress tensor and $\boldsymbol{q}$ is the heat flux vector. $\boldsymbol{\tau}$ and $\boldsymbol{q}$ are written as

$$\boldsymbol{\tau} = \mu \left(\nabla \boldsymbol{u} + (\nabla \boldsymbol{u})^T - \frac{2}{3} (\nabla \cdot \boldsymbol{u}) \, \mathbb{I} \right), \tag{3a}$$

$$\boldsymbol{q} = -\kappa \nabla T, \tag{3b}$$

where μ is the dynamic viscosity, κ is the thermal conductivity and T is the temperature. The equations are closed with the calorically perfect gas equation of state which relates thermodynamic variables as

$$P = \rho R T, \tag{4a}$$

$$\rho E = \frac{P}{\gamma - 1} + \frac{1}{2} \rho \boldsymbol{u}^2 \tag{4b}$$

where R is the specific gas constant and $\gamma = 1.4$ is the ratio of specific heat capacities, both of which are assumed constant. Extensions to thermally perfect gases and non-equilibrium systems are topics of future work.

B. Discontinuous Galerkin discretization

Let Ω_h be the union of elements of a non-overlapping discrete partition of the physical domain Ω into N_E non-empty cells Ω_e, $e \in \{1, \ldots, N_E\}$. Let $\mathcal{V}_h^p$ be the set of test functions

$$\mathcal{V}_h^p = \left\{ \phi \in L_2(\Omega_h) \;\middle|\; \phi|_{\Omega_e} \in \mathcal{P}^p(\Omega_e), \; \forall e \in \{1, \ldots, N_E\} \right\}, \tag{5}$$

where $\mathcal{P}^p$ denotes the space of polynomials of degree at most p. The discrete solution $\mathbf{U}_h$ is expanded as follows:

$$\mathbf{U}_h = \bigoplus_{e=1}^{N_E} \mathbf{U}_h^e \;\text{ with }\; \mathbf{U}_h^e = \sum_{n=1}^{N_p} \widetilde{\mathbf{U}}_n^e \phi_n, \tag{6}$$

where $\widetilde{\mathbf{U}}_n^e : \mathbb{R}^+ \to \mathbb{R}^{N_U}$ is the vector of the n-th polynomial coefficient of the discrete solution in element e and $\{\phi_n, \; n \in \{1, \ldots, N_p\}\}$ is a basis of $\mathcal{P}^p(\Omega_{\text{ref}})$ where Ω_{ref} is the reference element. For details regarding the formulation of reference-space DG discretizations, the reader is referred to [16]. The discrete variational problem reads: find $\mathbf{U}_h$ such that $\forall \phi \in \mathcal{V}_h^p$,

$$\frac{d}{dt} \int_{\Omega_h} \mathbf{U}_h \phi \, d\Omega + \mathcal{L}_c(\mathbf{U}_h, \phi) = \mathcal{L}_d(\mathbf{U}_h, \phi). \tag{7}$$

The convective part is written as

$$\mathcal{L}_c(\mathbf{U}_h, \phi) = -\int_{\Omega_h} \mathbf{F}_c(\mathbf{U}_h) \cdot \nabla\phi \, d\Omega + \int_{\mathcal{E}_i} \widehat{\mathbf{F}}_c(\mathbf{U}_h^+, \mathbf{U}_h^-, \boldsymbol{n}) \, (\phi^+ - \phi^-) \, d\Gamma + \int_{\mathcal{E}_b} \widetilde{\mathbf{F}}_c(\mathbf{U}_h^+, \mathbf{U}_b, \boldsymbol{n}) \, \phi^+ \, d\Gamma, \tag{8}$$

where $\mathcal{E}_i$ is the set of interior faces, $\mathcal{E}_b$ is the set of boundary faces, the + and − signs denote respectively each side of a face and $\boldsymbol{n}$ is the outward normal vector with respect to cell +. $\widehat{\mathbf{F}}_c$ is the inviscid numerical flux and the HLLC flux [17] is used in the present study. $\widetilde{\mathbf{F}}_c$ depends on the type of boundary and boundary treatment. The viscous part can be treated with a mixed formulation where the original system is recast into a system of first-order equations by introducing an auxiliary variable. Here a flux-based formulation is used for which the auxiliary variable is the discrete representation of $\mathbf{F}_d$. The final form of the viscous flux discretization reads

$$\begin{aligned}
\mathcal{L}_d(\mathbf{U}_h, \phi) = &-\int_{\Omega_h} \mathbf{F}_d(\mathbf{U}_h, \nabla_h \mathbf{U}_h) \cdot \nabla\phi \, d\Omega \\
&- \int_{\mathcal{E}_i} \left\{ \left[(\boldsymbol{\mathcal{A}}^+ : (\widehat{\mathbf{U}} - \mathbf{U}_h^+) \otimes \boldsymbol{n}) \cdot \nabla\phi^+ \right] - \left[(\boldsymbol{\mathcal{A}}^- : (\widehat{\mathbf{U}} - \mathbf{U}_h^-) \otimes \boldsymbol{n}) \cdot \nabla\phi^- \right] \right\} d\Gamma \\
&+ \int_{\mathcal{E}_i} \widehat{\mathbf{F}}_d \cdot \boldsymbol{n} \, (\phi^+ - \phi^-) \, d\Gamma \\
&+ \int_{\mathcal{E}_b} \left(\widetilde{\widetilde{\mathbf{F}}}_d \cdot \nabla\phi^+ + \widetilde{\mathbf{F}}_d \cdot \boldsymbol{n}\phi^+ \right) d\Gamma,
\end{aligned} \tag{9}$$

where $\boldsymbol{\mathcal{A}} = \partial \mathbf{F}_d / \partial(\nabla \mathbf{U})$ is the homogeneity tensor and $\{\widehat{\mathbf{U}}, \widehat{\mathbf{F}}_d\}$ are numerical fluxes associated with the viscous discretization. The symmetric interior penalty scheme [18] is used in the present study. Similar to the treatment of inviscid boundary fluxes, $\widetilde{\widetilde{\mathbf{F}}}_d$ and $\widetilde{\mathbf{F}}_d$ depends on the type of boundary and boundary treatment.

With this, the mass matrix can be formed from the first term in the left-hand side by using the expansion from Eq. 6 and the semi-discrete form can be discretized using the method of lines. A third-order explicit strong-stability-preserving Runge-Kutta scheme [19] is used for temporal integration. All integrals are computed in reference space using quadrature rules that exactly integrate all polynomials of order less than or equal to $2(p_{\text{sol}} + p_{\text{geo}})$ where p_{sol} is the highest polynomial order of the basis functions and p_{geo} is the highest polynomial order to represent the mapping from reference to physical space.

III. Legion implementation

A. The Legion runtime

Legion [15] designates both a task-based programming model and its implementation into a C++ runtime targeted for applications running on distributed heterogeneous systems. It provides abstractions to the programmer to describe data and their usage in order for the runtime to reason about the program, extract available parallelism, understand dependencies, and perform all memory allocations, communications and synchronizations required to guarantee the correct behavior of the program thus achieving portability. In contrast to the MPI programming system, explicit data transfers for distributed execution are, in most cases, not needed, mitigating the challenging task of ensuring a correct execution when running on architectures with complex memory hierarchies. In order to achieve performance portability across clusters, one core idea in the Legion system is the separation of the logical description of the program and its actual execution pattern on a given architecture. These two components are clearly separated in the source code allowing the developer to maintain different execution policies for different architectures with the same core application code. This is clarified in more details in the next sections.

At runtime, Legion constructs a dependency graph of tasks, the units of work in the Legion system, based on the information provided through its programming interface. This task graph is the basis for orchestrating their asynchronous execution. The throughput and the overhead of the runtime analysis are optimized via task parallelism while conforming to the underlying dependencies of the program.

Legion has been successfully used for several production-grade computational physics solvers [20, 21], to improve an existing MPI-based implementation [22] or as a possible backend for higher-level runtimes [23]. The latter application is unstructured while the former ones are structured.

B. Data description: regions and partitions

As a data-centric system, the first building block of a Legion program is the description of the program data. They are encoded in the form of *regions* which are potentially multi-dimensional containers of *fields*, similar to arrays of structs in a language like C. However the placement and the layout of regions are not fixed: regions are automatically copied and their layout transformed as required by the description of the tasks executed by the program. In the Legion nomenclature, regions are created from an *index space* (which describes their dimensionality) and a *field space* (which describes the set of fields they contain).

Data parallelism is achieved by partitioning the regions into sub-regions. A typical pattern for launching tasks is to execute as many copies of the same task as sub-regions, each copy being assigned and working on one sub-region. Several partitions can be created for a given region as illustrated in Fig. 1. Three different partitions for the same region are illustrated. Each color represents a sub-region. The characterization of the nature of the partitions is another key component for the dependence analysis. For instance, the top partition is said to be complete (the union of the sub-regions recovers the entire region) and disjoint (no pair of sub-regions intersects). The middle partition is also disjoint but incomplete. Finally the bottom partition is complete and aliased (not disjoint) because sub-regions overlap with one another. Elements filled with several colors belong to several sub-regions. This kind of partition is typical for a computational fluids code where data from direct neighbors need to be read in order to advance the solution in time. It is apparent that two identical tasks that write the same fields in the orange and blue sub-regions respectively cannot be executed concurrently (at least in the most general context and without any specific treatment) since the behavior for elements in the intersection would be undefined.

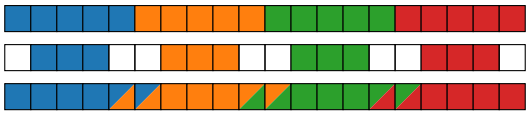


Fig. 1 Example of different partitions associated with a region. A color represents a sub-region in a partition. The top partition is complete-disjoint, the middle partition is incomplete-disjoint, and the bottom partition is complete-aliased.

C. Kernels description: tasks

Units of work in the Legion system are called tasks. A task implementation is comprised of (i) the declaration of the regions it requests access to with associated privileges, (ii) its registration where specific layouts can be enforced for these regions and (iii) its body which contains its functional behavior. For example, a task `T` can enforce a structure-of-array layout on a region `R` and request fields `f1` and `f2` with read-only and read-write privilege respectively. Several different versions of a task for specific target hardware can be implemented. For instance, a task can exist in a CPU and a GPU version and Legion will use the right version for the actual hardware where it gets executed. A better alternative for software maintainability is to choose a machine-portable paradigm to implement one single body that can run on many hardware types. Based on the regions a task accesses and the associated privileges, Legion constructs the underlying dependency graph and schedules the execution of tasks accordingly. The task body can assume that dependencies are satisfied and layout constraints enforced. For example, if a given task writes to a sub-region, that change becomes visible to any subsequent (in the sense "depends on") task that accesses any other sub-regions that overlap with the former. This does not mean that for every task each sub-region gets its own distinct allocation, rather that if two sub-regions logically overlap, copies will be triggered if needed.

D. Performance portability: mapper

Legion achieves performance portability across supercomputers through its mapping interface. A mapper is a component of a Legion program which implements methods that the runtime queries for mapping decisions. The Legion library is also equipped with generic mappers with reasonable default behaviors. Mapping is the process of (i) choosing a processor (in the sense of a machine that can do computations like a CPU or a GPU) for every task where it

is going to be executed and (ii) creating/choosing physical instances in memory with expected layout for every field of every region that tasks request. It becomes clear that the optimal set of decisions strongly depends on the nature of the application and the machine it runs on. While these decisions generally interfere with the core application code when explicit approaches such as MPI are employed, the mapper is a piece of code that is entirely separate from task bodies. This allows for a very flexible and non-intrusive performance tuning and many different mapping policies can be experimented in a productive way without affecting code correctness. An application can thus be ported to a new architecture, assuming task implementations exist for the hardware, and good performance can be obtained by implementing a new mapper without changing task bodies.

E. DG algorithm implementation

The implementation presented in this work focused on building the general infrastructure as well as the independent modules necessary for a DG solver such as classes related to basis functions or the physics. Emphasis was put on correctness and rapid development of a working solver. Therefore we do not use all aspects of Legion in an optimal manner yet and the full potential of a Legion implementation is discussed in the next section. Future work will emphasize on optimizing performance, demonstrating performance portability and taking advantage of all features offered by the Legion system.

The DG method is naturally amenable to a task-based representation and fits the Legion framework well. In this section the implementation is described using the Legion nomenclature as well as some specifics that makes it different from a traditional MPI implementation. For ease of presentation, a fully periodic domain will be considered noting that boundary faces can be treated as a separate region.

Two one-dimensional index spaces $\mathcal{I}_e$ and $\mathcal{I}_f$ are first created with size corresponding to the number of elements and the number of faces respectively. The field space associated with $\mathcal{I}_e$ contains all relevant data such as the solution coefficients that are solved for. The field space associated with $\mathcal{I}_f$ contains meta-data such as pointers to the left and right elements. From these, regions for the elements $\mathcal{R}_e$ and faces $\mathcal{R}_f$ are created. Then the regions are partitioned.

In order to understand this procedure, some terminology is introduced first. Globally, given a partition (in the traditional sense, i.e. disjoint and complete) from a mesh partitioner such as METIS [24], an element can either be private or shared. Private elements' direct neighbors are contained within the same sub-region while shared elements are a neighbor to at least one element owned by another sub-region. For a given sub-region, ghost elements which are elements owned by other sub-regions but are neighbors to the current sub-region need to be considered. Fig. 2 gives an example of these different element types. When computing the residual, data from shared elements need to be sent while data from ghost elements need to be received. The point of making this separation is that private elements don't need to move unless performing load balancing across distributed memories. From the output of a mesh partitioner, Legion dependent partitioning functions [25] are used to create the private, shared and ghost partitions following a specific sequence of calls, the first two being disjoint while the third one being aliased.

The main tasks are associated with the volume integrals and the face integrals in Eq. 7. The volume integral task is purely local and only involves sub-regions that are disjoint. The task computing the face integrals needs to access ghost element information and thus requires this data to be communicated prior to execution. This requirement is automatically discovered by Legion which can overlap the volume residual task with communication. The current default execution pattern consists of creating as many sub-regions as compute nodes and using OpenMP for intra-node parallelism. This provides good performance for compute nodes made of multi-core processors while coarsening the task granularity compared to a core-level partition of the computational domain, which mitigates the overhead due to the runtime.

F. Discussion

All Legion programs benefit from the automatic memory management by the runtime which reduces the likelihood of bugs due to missing or misplaced communication directives. The mapping interface also eases the process of performance optimization and portability. Finding the best mapping policies often entails finding the strategy that keeps the machine as busy as possible. A key ingredient in order for Legion to achieve this goal is to expose sufficient task parallelism while keeping the task granularity relatively coarse so that the latency associated with a task launch can be hidden. We recognize that the current implementation does not expose sufficient task parallelism to take full advantage of Legion (three tasks per Runge-Kutta stage: one for the volume integrals, one for the face integrals and one for the update of the unknowns). This is not an issue for perfectly balanced simulations with homogeneous compute nodes but will prevent optimization in the presence of load unbalance or hardware heterogeneity. It is worth noting that

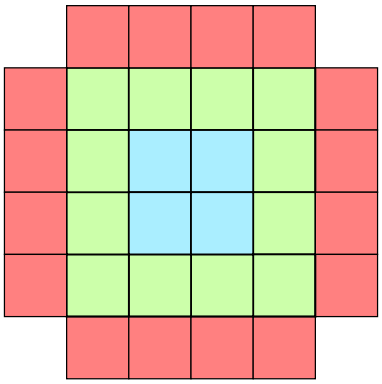


Fig. 2 Private (blue), shared (green) and ghost(red) elements for a given set of owned elements (blue & green). Shared elements are ghost to owners of the red elements. Shared and ghost elements are globally shared.

an all-GPU mapping is not necessarily the optimal choice and using the host CPU for a subset of tasks can result in improved throughput [22]. Reformulating the decomposition of the DG algorithm tasks with finer granularity is the subject of future work.

One motivation for the Legion implementation is load balancing. Load unbalance can stem from different sources. For example, localized shock-capturing methods increase the cost at so-called troubled elements where a specific treatment needs to be applied. Similarly, for reacting flow simulations, additional cost occurs for elements where chemical reaction source terms are stiff (typically at flame fronts). This results in load unbalance when the time integration is realized via operator splitting and several internal steps need to be taken for integrating the chemistry. A general and easy way for achieving load balancing in the context of runtime-based implementations is over-decomposition [26]. By partitioning the domain into more pieces than the number of available processors, lightly-loaded sub-regions are continuously scheduled for execution while a sub-set of processors work on highly-loaded sub-regions. This improves the overall machine utilization. For very localized and unsteady sources of load unbalance, this strategy might however be sub-optimal due to the required fine granularity which increases the overhead of the runtime relative to the time it takes to complete a task. This is part of ongoing investigations.

An attractive feature of high-order methods is their flexibility for adaptation. Based on some indicator, h-adaptation locally refines or coarsens the mesh while p-adaptation adjusts the polynomial order. Both adaptation strategies create load unbalance. Dynamic p-adaptation is easier to handle implementation-wise due to the fact that the mesh remains the same. Several remarks can be made regarding its implementation:

- In order to overcome the load unbalance that originates from having different polynomial orders in the domain, it is common to re-partition the mesh after every adaptation cycle and to weigh every element with its associated order [27]. This operation is quite expensive and involves substantial memory transfers. Such approach is only interesting if the cost of setting up a new partition is small compared to the rest of the cycle which seems to be the case in the context of implicit time-stepping even when performed at every time-step. The cost of setting up new partitions with Legion has not been assessed so far with this application but will be addressed in future work.
- The goal of adaptation is to increase efficiency. In particular, dynamic p-adaptation implies changing the amount of data associated with the solution in a given element. When implemented with a linked-list data structure [28], it is straightforward to resize the memory allocation for adapted elements. However this prevents the use of collective linear algebra operations (at least without requiring to copy data from the collectively treated elements into a contiguous memory location) which could impact performances on accelerators for which vendor-provided libraries are usually suited for large matrices. Legion by default does not use a per-element approach to materialize physical instances of the regions. Thus resizing the field space for specific elements of the index space is not idiomatic and not likely to be efficient. Of course one could over-allocate for the highest requested order but this entails a large memory overhead. It also does not suppress the need of re-ordering elements according to their order if a collective treatment is sought for (the operator matrix for quadrature evaluation for instance is a function of polynomial order).

The objectives of making these remarks are two-fold. First the gain in efficiency with dynamic hp-adaptation is closely linked to the base and adaptive implementations. Second, although Legion is designed for load-balancing, implementing dynamically adaptive methods in the Legion programming system presents its own challenges.

Finally, Legion naturally incurs some overhead in order to perform the dependence analysis during runtime. Moreover some resources need to be dedicated to the runtime (typically 2-3 threads on the host CPU) including those responsible for memory transfers. For unstructured applications, the ghost partition is in general very sparse in the index space when computed from the output of a general graph partitioning library. Due to the desire of using a collective treatment for linear algebra operations, both the private and the shared elements are allocated contiguously in memory. At first approximation, allocations for sub-regions in the ghost partition then use substantially more memory than needed because Legion uses the bounds in index space to construct them. Support for sparse instances has been recently added and discussions with the Legion developers team to find the optimal strategy in order to reduce the memory usage are ongoing.

IV. Verification

The solver is verified using three canonical cases. Case 1 is the advection of an isentropic vortex in a periodic domain which verifies the numerics and the implementation of the Euler equations. Case 2 is a manufactured-solution for the Navier-Stokes equations which verifies the discretization of viscous terms. Case 3 is a three-dimensional Taylor-Green vortex case and demonstrates the ability of the solver to converge to direct numerical simulations of turbulent flows.

A. Case 1: advection of an isentropic vortex

This is a canonical case used to verify numerical schemes and has been studied as a way to verify high-order methods [29]. The Euler equations in two dimensions are solved in dimensionless form. Quantities used to make the system dimensionless are the domain size L_0, the background density ρ_0, the background temperature T_0 and the background speed of sound c_0. With this the domain size is $[0, 1]^2$ with periodic boundary conditions at all sides. The initial perturbations are prescribed in primitive form as

$$\delta u = -\frac{y_1}{r}\Omega, \tag{10a}$$

$$\delta v = \frac{x_1}{r}\Omega, \tag{10b}$$

$$\delta T = -\frac{\gamma - 1}{2}\Omega^2, \tag{10c}$$

where

$$x_1 = x - 0.5, \;\; y_1 = y - 0.5, \tag{11a}$$

$$\Omega = \beta \exp\left(-\frac{1}{2\sigma^2}\frac{x_1^2 + y_1^2}{r_0^2}\right), \tag{11b}$$

$$\beta = \mathrm{Ma}_0 \frac{5\sqrt{2}}{4\pi} e^{0.5}, \tag{11c}$$

where $\mathrm{Ma}_0 = 1$ is the background Mach number, $r_0 = 1/20$ is the vortex size and $\sigma = 1$ is a parameter. The initial conditions become

$$u = \mathrm{Ma}_0 + \delta u, \tag{12a}$$

$$v = \delta v, \tag{12b}$$

$$P = \frac{1}{\gamma}(1 + \delta T)^{\frac{\gamma}{\gamma-1}}, \tag{12c}$$

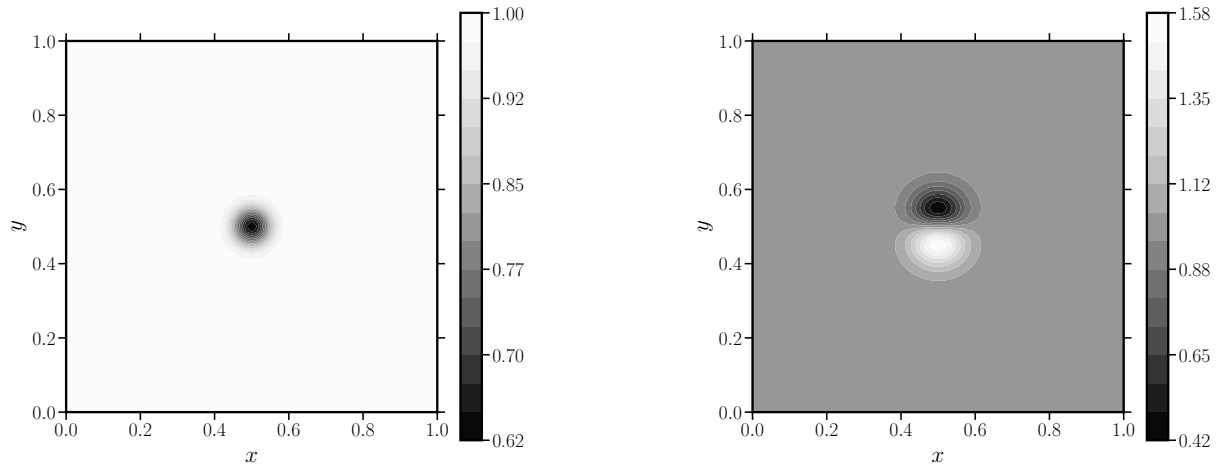
$$\rho = (1 + \delta T)^{\frac{1}{\gamma-1}}. \tag{12d}$$

Contours of initial density and x-velocity are illustrated in Fig. 3 at $t = 0$.

In order to assess the convergence with mesh refinement, the L_2 error against the analytical solution of the problem is computed after one advection period for all conservative variables using different polynomial orders. Three types of meshes are considered. The baseline bilinear quadrilateral mesh is composed of five elements in each direction and five levels of refinement are performed by doubling the resolution in every direction. A set of linear triangular meshes is generated by splitting every element of the former meshes along one diagonal. Quadratic quadrilateral meshes are also

considered and are constructed by perturbing the nodes of the baseline mesh randomly by a maximum of 20% of the unperturbed element length. Then the perturbed mesh is refined uniformly in reference space. An example of such a mesh is illustrated in Fig. 4.

Figure 5 presents the obtained convergence rates for the density variable. A rate of at least $p + 1$ is obtained in all cases in the asymptotic range. It is worth noting that this case can exhibit a super convergence behavior for longer integration times as reported for instance in [12].



(a) **(b)**

Fig. 3 Case 1. Initial conditions for the isentropic vortex: (a) density and (b) x-velocity contours.

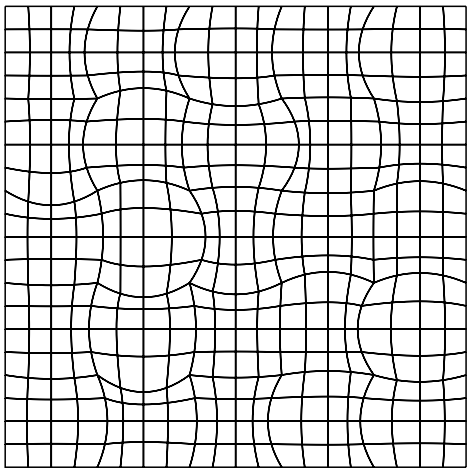


Fig. 4 Case 1. Example of a quadratic perturbed quadrilateral mesh.

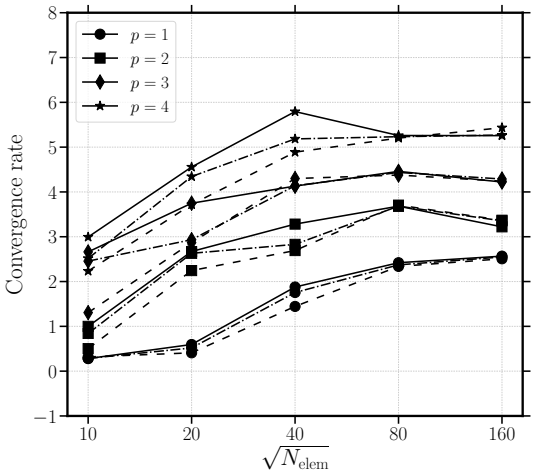


Fig. 5 Case 1. Convergence rates for the isentropic vortex case for the density variable. Solid, dashed and dash-dotted lines correspond to bilinear quadrilateral elements, linear triangular elements and quadratic quadrilateral elements respectively.

B. Case 2: manufactured solution for the Navier-Stokes equation

The test case MS-3 from [30] is used to verify the implementation of the viscous discretization. This uses the method of manufactured solution which drives the solution to a steady solution by prescribing the appropriate analytical forcing term. The initial and target solution is restated here for completeness.

The domain size is $[0, 1]^2$ and the solution is described with primitive variables as follows

$$\rho = \rho_0 + \rho_x \sin(a_{\rho x}\pi x) + \rho_y \cos(a_{\rho y}\pi y) + \rho_{xy} \cos(a_{\rho xy}\pi x) \cos(a_{\rho xy}\pi y), \tag{13a}$$

$$u = u_0 + u_x \sin(a_{ux}\pi x) + u_y \cos(a_{uy}\pi y) + u_{xy} \cos(a_{uxy}\pi x) \cos(a_{uxy}\pi y), \tag{13b}$$

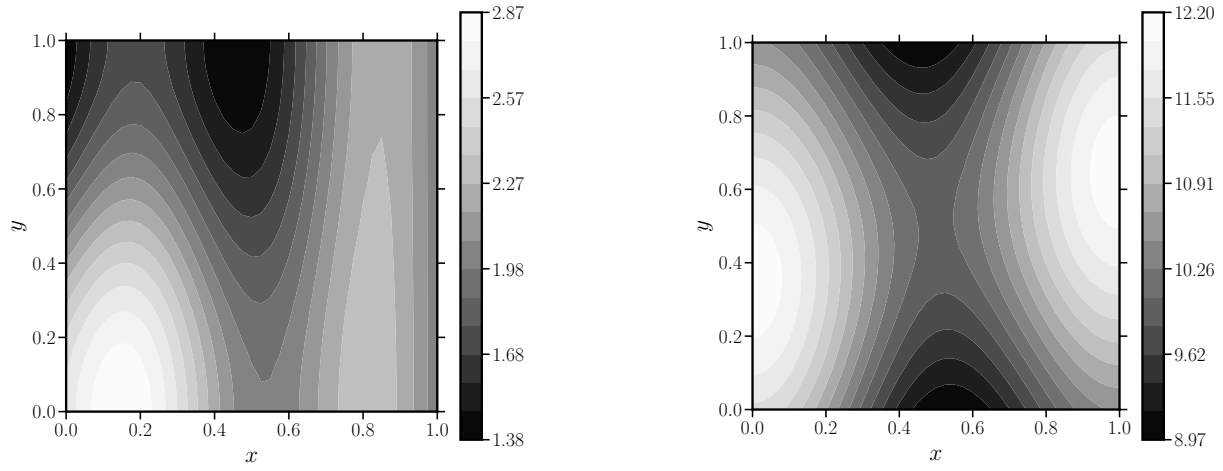
$$v = v_0 + v_x \cos(a_{vx}\pi x) + v_y \sin(a_{vy}\pi y) + v_{xy} \cos(a_{vxy}\pi x) \cos(a_{vxy}\pi y), \tag{13c}$$

$$P = P_0 + P_x \cos(a_{Px}\pi x) + P_y \sin(a_{Py}\pi y) + P_{xy} \cos(a_{Pxy}\pi x) \cos(a_{Pxy}\pi y). \tag{13d}$$

The choice of parameters is not straightforward to obtain a challenging problem that would uncover implementation errors while converging appropriately with mesh refinement at the expected rate [30]. The parameters for this particular case are provided in Table 1. Contours of the analytical solution of x-velocity and pressure are illustrated in Fig. 6 ensuring that the conditions from [30] are reproduced. A dynamic viscosity of $\mu = 0.01$ and a Prandtl number of $\mathrm{Pr} = 0.7$ are used. A bilinear quadrilateral base mesh with 10 elements in each direction is considered with 3 levels of refinement. The finest mesh has 80 elements in each direction. The simulations are advanced in time until a steady error level is obtained for all variables. Figure 7 presents the convergence rate of the L_2 error for the x-momentum variable. Optimal convergence rates of $p + 1$ are obtained for the last level of refinement.

$(\cdot)$	$(\cdot)_0$	$(\cdot)_x$	$(\cdot)_y$	$(\cdot)_{xy}$	$a_{(\cdot)x}$	$a_{(\cdot)y}$	$a_{(\cdot)xy}$
ρ	1.0	0.1	-0.2	0.1	1.0	1.0	1.0
u	2.0	0.3	0.3	0.3	3.0	1.0	1.0
v	2.0	0.3	0.3	0.3	1.0	1.0	1.0
P	10.0	1.0	1.0	0.5	2.0	1.0	1.0

Table 1 Case 2. Parameters for the manufactured Navier-Stokes problem



(a) **(b)**

Fig. 6 Case 2. Manufactured solution: (a) x-velocity and (b) pressure contours.

C. Case 3: Taylor-Green vortex

The three-dimensional Taylor-Green vortex is a canonical problem of homogeneous isotropic turbulence. Starting from a laminar state, the flow progressively transitions to a turbulent state through vortex breakdown as illustrated in

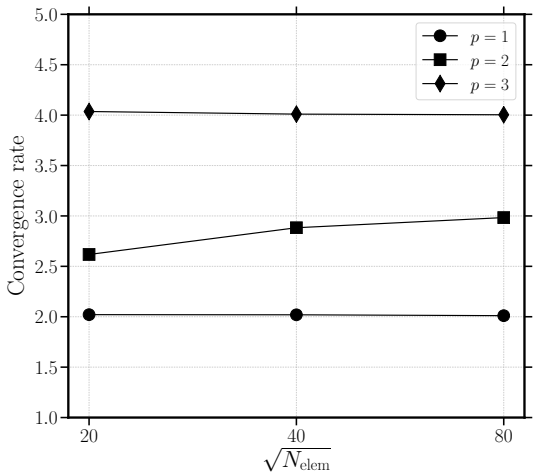


Fig. 7 Case 2. Convergence rates for the manufactured Navier-Stokes problem for the x-momentum variable.

Fig. 8. It has been used in numerous studies and is being considered as a benchmark for high-order methods as well [4]. The goal of this investigation is to verify that turbulent flows can be accurately represented with resolutions close to a direct numerical simulation (DNS). This problem is solved in dimensionless form. The reference quantities are L_0 the domain size divided by 2π, ρ_0 the background density, u_0 the background velocity and T_0 the background temperature. With this the domain size is $[0, 2\pi]^3$ and the initial conditions are given in primitive form as

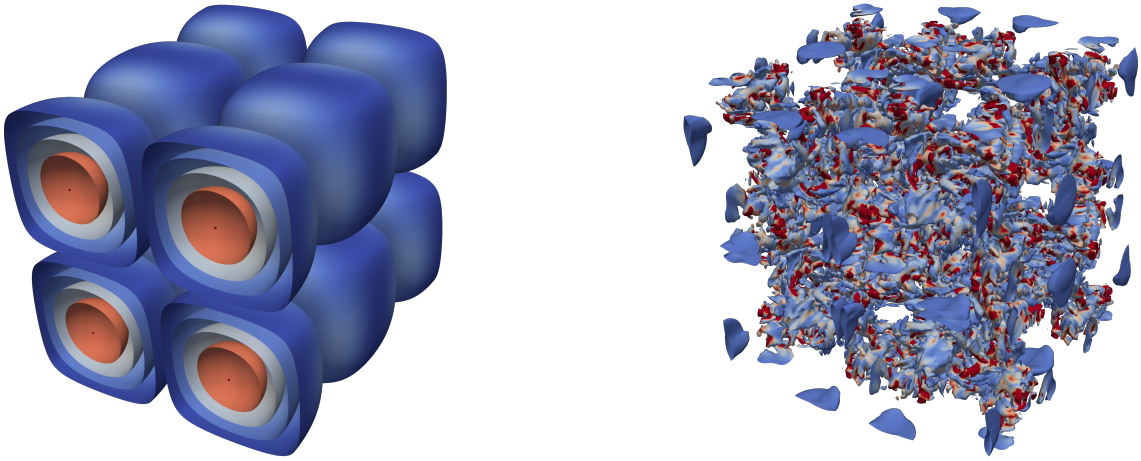
$$\rho = 1, \tag{14a}$$
$$u = \sin(x)\cos(y)\cos(z), \tag{14b}$$
$$v = -\cos(x)\sin(y)\cos(z), \tag{14c}$$
$$w = 0, \tag{14d}$$
$$P = \frac{1}{\gamma \mathrm{Ma}_0^2} + \frac{1}{16}\Big[\cos(2x) + \cos(2y)\Big]\Big[(\cos(2z) + 2\Big], \tag{14e}$$

where the pressure condition mimics the incompressibility condition. For consistency, the dimensionless specific gas constant needs to be chosen as $\frac{1}{\gamma \mathrm{Ma}_0^2}$. The parameters of the simulation are the Mach number $\mathrm{Ma}_0 = 0.1$, the Reynolds number $\mathrm{Re}_0 = 1600$ from which the dimensionless dynamic viscosity $\mu_0 = 0.625 \times 10^{-3}$ is obtained and the Prandtl number is set to $\mathrm{Pr} = 0.71$.

Three hp-resolutions are considered and summarized in Table 2. The domain integrated quantities are the density weighted kinetic energy and enstrophy

$$E = \frac{1}{(2\pi)^3}\int_\Omega \frac{1}{2}\rho \boldsymbol{u}^2 \, d\Omega, \tag{15a}$$
$$\zeta = \frac{1}{(2\pi)^3}\int_\Omega \frac{1}{2}\rho \boldsymbol{\omega}^2 \, d\Omega, \tag{15b}$$

where $\boldsymbol{\omega}$ is the vorticity vector. Direct kinetic energy dissipation are obtained by using a central difference scheme on the kinetic energy data. Results are compared against a reference DNS [31] in Fig. 9. The DG simulations converge with increasing resolution towards the reference DNS for these mean quantities. Another well-known fact about high-order methods is observed as well: at approximately equal number of degrees of freedom, smooth flows are more accurately represented with high orders. This shows the potential of high-order methods to enable more efficient simulations of turbulent flows. The wall-times for the two resolutions with approximately 256^3 degrees of freedom were comparable.

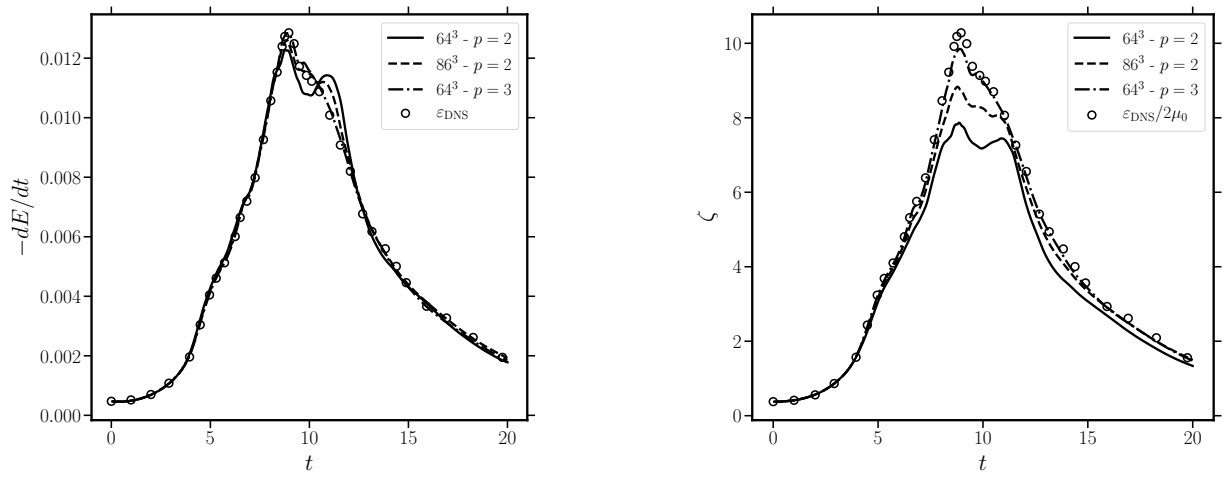


(a) (b)

Fig. 8 Case 3. Iso-surfaces of z-vorticity colored by enstrophy at (a) $t = 0$ and (b) $t = 10$.

Number of elements	Polynomial order	Number of degrees of freedom
64^3	2	192^3
64^3	3	256^3
86^3	2	258^3

Table 2 Case 3. Resolutions used for the Taylor-Green vortex simulations.



(a) (b)

Fig. 9 Case 3. Volume integrated (a) density weighted kinetic energy dissipation and (b) enstrophy.

V. Performance assessment

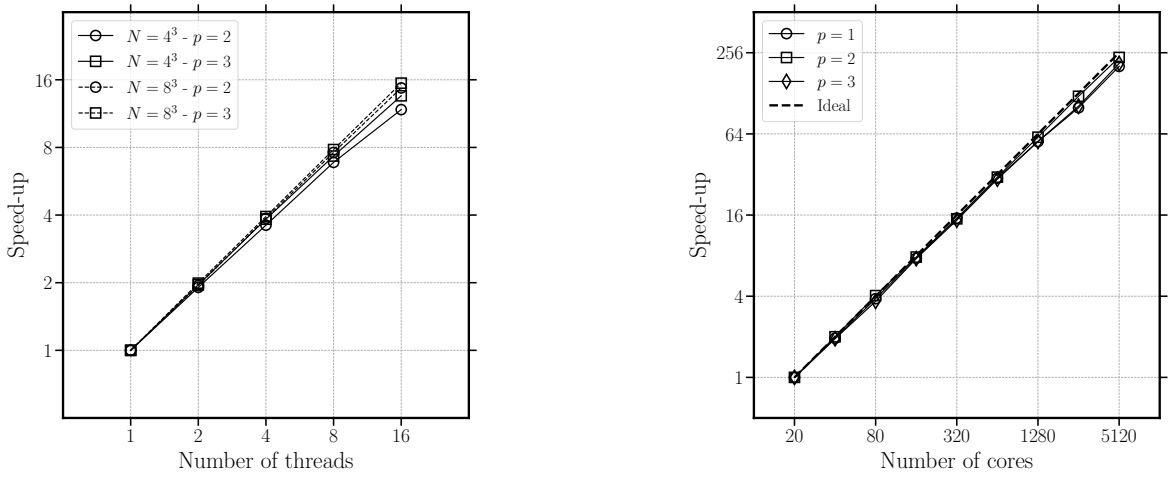
Performance of the solver is assessed by conducting three-dimensional simulations of the Navier-Stokes equations in a triply periodic box on NASA's Pleiades supercomputer [32]. In particular, the Ivy Bridge nodes composed of two ten-core Intel Xeon E5-2680v2 processors are used.

Strong scaling results are presented in Fig. 10. For single-node scaling, the number of threads is varied between 1 and 16 (recall that some threads need to be dedicated to the runtime which results in a maximum number of usable threads for the application of 17 on these nodes) and small meshes are used in order to test the limits of scalability. In particular, the 4^3 and 8^3 meshes result in 4 and 32 elements per thread at 16 threads respectively. Good scaling characteristics that improve with polynomial order are observed. Specifically, with 512 elements to process, a speed-up of 14.7 and 15.5 is observed for $p = 2$ and $p = 3$ respectively. For the multi-node strong scaling, the 64^3 mesh is used with different polynomial orders and simulations are performed up to 256 compute nodes for a total of 5,120 cores. Good scaling behavior is observed overall.

The performance index (PID) used in [33] is also computed. This can be defined as

$$\mathrm{PID} = \frac{\tau_{\mathrm{wall}} N_{\mathrm{cores}}}{N_{\mathrm{DOF}} N_{\mathrm{steps}} N_{\mathrm{stages}}}, \tag{16}$$

which represents the time required to integrate one degree of freedom normalized by the number of cores. In the saturated regime, PIDs of 9 µs for $p = 1$ and 6-7 µs for $p = 2$ and $p = 3$ were obtained. For comparison, [33] reports a PID of 1 µs for $p = 6$. Differences can be explained by the polynomial order, the absence of sum-factorization type optimization for tensor-product basis in the present work as well as the use of a modal basis instead of a co-located approach (Lagrange basis with nodes located at the quadrature points). Naturally, less time was spent so far optimizing our application in comparison.



(a) **(b)**

Fig. 10 (a) Single node and (b) multi-node strong scaling.

VI. Conclusions and future work

A DG solver was developed and implemented using the Legion programming system. Specific details concerning the Legion model and programming interface were discussed with emphasis on the potential features of Legion for further enhancement of the efficiency of the solver. Good strong scaling characteristics were observed on compute nodes made of multi-core processors. Future work will focus on the following aspects:

- the formulation of the DG algorithm into tasks will be redesigned in order to expose more task parallelism so as to broaden the space of mapping strategies when running on heterogeneous nodes;
- improving the solver in order to scale at much higher core count;
- machine portability will be sought for by using the Kokkos model [34] for task bodies;

- cases with load unbalance and/or hardware heterogeneity will be investigated in order to assess the benefits of using Legion for this application.

This work is a first step of a continuing effort to develop a research platform for high-order numerical methods that fully exploits modern compute architectures. The code is available upon request and can be provided by contacting the lead author.

Acknowledgements

Financial support through Total E&P Research & Technology with award No. 134818 and the National Science Foundation (NSF) with award No. 1909379 is gratefully acknowledged. The authors acknowledge the computing resources granted by the National Aeronautics and Space Administration (NASA) on the Pleiades supercomputer.

References

[1] Witherden, F. D., and Jameson, A., "Future directions of computational fluid dynamics," *23rd AIAA Computational Fluid Dynamics Conference, 2017*, 2017, pp. 1–16.

[2] Spalart, P. R., and Venkatakrishnan, V., "On the role and challenges of CFD in the aerospace industry," *The Aeronautical Journal*, Vol. 120, No. 1223, 2016, pp. 209–232.

[3] Slotnick, J., Khodadoust, A., Alonso, J., Darmofal, D., Gropp, W., Lurie, E., and Mavriplis, D., "CFD vision 2030 study: a path to revolutionary computational aerosciences," *NASA Report*, 2014.

[4] Wang, Z., Fidkowski, K., Abgrall, R., Bassi, F., Caraeni, D., Cary, A., Deconinck, H., Hartmann, R., Hillewaert, K., Huynh, H., Kroll, N., May, G., Persson, P.-O., van Leer, B., and Visbal, M., "High-order CFD methods: current status and perspective," *International Journal for Numerical Methods in Fluids*, Vol. 72, No. 8, 2013, pp. 811–845.

[5] de Wiart, C. C., Hillewaert, K., Bricteux, L., and Winckelmans, G., "Implicit LES of free and wall-bounded turbulent flows based on the discontinuous Galerkin/symmetric interior penalty method," *International Journal for Numerical Methods in Fluids*, Vol. 78, No. 6, 2015, pp. 335–354.

[6] Bassi, F., Botti, L., Colombo, A., Crivellini, A., Ghidoni, A., and Massa, F., "On the development of an implicit high-order Discontinuous Galerkin method for DNS and implicit LES of turbulent flows," *European Journal of Mechanics, B/Fluids*, Vol. 55, 2016, pp. 367–379.

[7] Park, J. S., Witherden, F. D., and Vincent, P. E., "High-order implicit large-eddy simulations of flow over a NACA0021 Aerofoil," *AIAA Journal*, Vol. 55, No. 7, 2017, pp. 2186–2197.

[8] Flad, D., Beck, A., and Guthke, P., "A large eddy simulation method for DGSEM using non-linearly optimized relaxation filters," *Journal of Computational Physics*, Vol. 408, 2020, p. 109303.

[9] Navah, F., de la Llave Plata, M., and Couaillier, V., "A high-order multiscale approach to turbulence for compact nodal schemes," *Computer Methods in Applied Mechanics and Engineering*, Vol. 363, 2020, p. 112885.

[10] Lv, Y., Ma, P. C., and Ihme, M., "On underresolved simulations of compressible turbulence using an entropy-bounded DG method: Solution stabilization, scheme optimization, and benchmark against a finite-volume solver," *Computers & Fluids*, Vol. 161, 2018, pp. 89–106.

[11] Vermeire, B. C., Witherden, F. D., and Vincent, P. E., "On the utility of GPU accelerated high-order methods for unsteady flow simulations: A comparison with industry-standard tools," *Journal of Computational Physics*, Vol. 334, 2017, pp. 497–521.

[12] Witherden, F. D., Farrington, A. M., and Vincent, P. E., "PyFR: An open source framework for solving advection-diffusion type problems on streaming architectures using the flux reconstruction approach," *Computer Physics Communications*, Vol. 185, No. 11, 2014, pp. 3028–3040.

[13] Romero, J., Crabill, J., Watkins, J., Witherden, F., and Jameson, A., "ZEFR: A GPU-accelerated high-order solver for compressible viscous flows using the flux reconstruction method," *Computer Physics Communications*, Vol. 250, 2020, p. 107169.

[14] Terrana, S., Nguyen, C., and Peraire, J., "GPU-accelerated large eddy simulation of hypersonic flows," AIAA paper, 2020-1062.

[15] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A., "Legion: Expressing locality and independence with logical regions," *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2012, pp. 1–11.

[16] Hesthaven, J. S., and Warburton, T., *Nodal Discontinuous Galerkin Methods*, Springer New York, 2008.

[17] Toro, E. F., Spruce, M., and Speares, W., "Restoration of the contact surface in the HLL-Riemann solver," *Shock Waves*, Vol. 4, No. 1, 1994, pp. 25–34.

[18] Hartmann, R., and Houston, P., "An optimal order interior penalty discontinuous Galerkin discretization of the compressible Navier-Stokes equations," *Journal of Computational Physics*, Vol. 227, No. 22, 2008, pp. 9670–9685.

[19] Gottlieb, S., Ketcheson, D. I., and Shu, C.-W., "High order strong stability preserving time discretizations," *Journal of Scientific Computing*, Vol. 38, No. 3, 2009, pp. 251–289.

[20] Torres, H., Papadakis, M., and Jofre Cruanyes, L., "Soleil-X: turbulence, particles, and radiation in the Regent programming language," *SC'19: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–4.

[21] Di Renzo, M., Fu, L., and Urzay, J., "HTR solver: An open-source exascale-oriented task-based multi-GPU high-order code for hypersonic aerothermodynamics," *Computer Physics Communications*, Vol. 255, 2020, p. 107262.

[22] Treichler, S., Bauer, M., Bhagatwala, A., Borghesi, G., Sankaran, R., Kolla, H., McCormick, P. S., Slaughter, E., Lee, W., Aiken, A., and Chen, J., "S3D-legion an exascale software for direct numerical simulation of turbulent combustion with complex multicomponent chemistry," *Exascale Scientific Applications: Scalability and Performance Portability*, 2017, pp. 257–277.

[23] Loiseau, J., Lim, H., Bergen, B. K., Moss, N. D., and Alin, F., "FleCSPH: a parallel and distributed smoothed particle hydrodynamics framework based on FleCSI," *2018 International Conference on High Performance Computing & Simulation (HPCS)*, IEEE, 2018, pp. 484–491.

[24] Karypis, G., and Kumar, V., "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal of Scientific Computing*, Vol. 20, No. 1, 1998, pp. 359–392.

[25] Treichler, S., Bauer, M., Sharma, R., Slaughter, E., and Aiken, A., "Dependent partitioning," *ACM SIGPLAN Notices*, Vol. 51, No. 10, 2016, pp. 344–358.

[26] Li, W., Luo, H., Pandare, A., and Bakosi, J., "A p -adaptive discontinuous Galerkin method for compressible flows using Charm++," AIAA paper, 2020-1565.

[27] Wang, L., Gobbert, M. K., and Yu, M., "A dynamically load-balanced parallel p-adaptive implicit high-order flux reconstruction method for under-resolved turbulence simulation," *Journal of Computational Physics*, Vol. 417, 2020, p. 109581.

[28] Gassner, G., Staudenmaier, M., Hindenlang, F., Atak, M., and Munz, C.-D., "A space–time adaptive discontinuous Galerkin scheme," *Computers & Fluids*, Vol. 117, 2015, pp. 247–261.

[29] Spiegel, S. C., Huynh, H. T., and Debonis, J. R., "A survey of the isentropic euler vortex problem using high-order methods," AIAA paper, 2015-2444.

[30] Navah, F., and Nadarajah, S., "A comprehensive high-order solver verification methodology for free fluid flows," *Aerospace Science and Technology*, Vol. 80, 2018, pp. 101–126.

[31] DeBonis, J., "Solutions of the Taylor-Green vortex problem using high-resolution explicit finite difference methods," AIAA paper, 2013-0382.

[32] "Pleiades Supercomputer," `https://www.nas.nasa.gov/hecc/resources/pleiades.html`, .

[33] Krais, N., Beck, A., Bolemann, T., Frank, H., Flad, D., Gassner, G., Hindenlang, F., Hoffmann, M., Kuhn, T., Sonntag, M., and Munz, C.-D., "FLEXI: A high order discontinuous Galerkin framework for hyperbolic–parabolic conservation laws," *Computers & Mathematics with Applications*, 2020.

[34] Edwards, H. C., Trott, C. R., and Sunderland, D., "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, Vol. 74, No. 12, 2014, pp. 3202 – 3216. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.