Task-based framework for physics-based ensemble simulation and in situ data processing

By K. Maeda AND T. Teixeira †

1. Motivation and objectives

There is an ever-growing demand for integrated computational modeling and analysis of complex systems in various areas of science and engineering. In addition to massive high-fidelity simulations, combinations of a large number of simulation runs at various levels of fidelity and parameters as well as parallel data processing are required for control, design, and optimization for practical engineering applications. The effective use of high-performance computing (HPC) systems is a key challenge to address these demands. Meanwhile, modern HPC systems are rapidly evolving into heterogeneous machines featuring different kinds of processors, such as GPUs and CPUs, and deep memory hierarchies (Mittal & Vetter 2015). Many physics-based simulation and data-analysis tools, such as computational fluid dynamics (CFD) solvers, have been designed for a single kind of processor based on the single program, multiple data (SPMD) technique realized by explicit parallel programming models like the message passing interface (MPI) (Gropp *et al.* 1999). Adaptation of these tools to heterogeneous machines often requires cumbersome programming since one has to explicitly implement parallelism to specify, for instance, domain decomposition, inter-processor communication, and data movement among the memory hierarchies. The twofold complexity of targeted applications and machine architectures is becoming a critical drawback for work efficiency.

Implicit parallelism is a characteristic of programming models that allows a compiler or interpreter to automatically exploit the parallelism inherent in the computations expressed by the language's constructs, easing the need for the programmer to manually implement parallelism. In practice, implicit parallelism is realized by task-based programming, in which a program is split into a set of tasks which can be distributed and executed concurrently (e.g., Thoman *et al.* 2018; Slaughter *et al.* 2020). Since the tasks can be in principle heterogeneous and their execution can be asynchronous, various kinds of task ensembles can potentially be efficiently parallelized without extensive programming efforts. Implicit parallel programming models can therefore be suitable for building complex software applications on heterogeneous systems. Of the various programming systems, the Legion programming system (hereafter Legion) is an implicit, task-based parallel programming model conceived for distributed heterogeneous architectures (Bauer *et al.* 2012). Legion features abstraction to describe properties of program data (e.g., independence, locality) as well as high-level languages and interfaces to address its application programming interface (API), notable of which in the present context are Regent, Pygion, and the C++ mapping interface (mapper). Regent is a generative, task-based programming language interfacing Legion's API (Slaughter *et al.* 2015). Pygion is a Python-API of Legion which enables the direct use of general Python libraries in tasks executed in the Legion runtime (Slaughter & Aiken 2019). The mapper provides programmer controlled

† Computer Science Department, Stanford University

placement of data in the memory hierarchy and assignment of tasks to various kinds of processors. For users of heterogeneous machines who are primarily interested in engineering applications, Legion can be appealing in the following two aspects. First, Legion can be a programmer-friendly alternative to the previous MPI-based programming models to perform large-scale simulations on heterogeneous machines. This aspect has been explored in CFD applications. S3D, a massively parallel framework for turbulent combustion, has its version built on Legion (Chen *et al.* 2009; Treichler *et al.* 2017). Soleil-X and HTR are Legion-based parallel flow solvers for particle-laden flows with radiative heat transfer and high-speed combustion, respectively (Torres *et al.* 2019; Di Renzo *et al.* 2020). Various other mini-apps have also been developed. Second, Legion can enable implementation of highly complex tasks in nontrivial workflows to open up new applications of supercomputers which are otherwise challenging for previous programming models. This aspect has remained relatively unexplored and requires particular attention. Due to its task-based nature, the Legion runtime can execute ensembles of various kinds of tasks simultaneously on different kinds of processors without intensive programming. Task heterogeneity can be particularly useful for in situ processing of simulation data. For instance, fluid flow simulations with in situ data compression has been performed (Pacella *et al.* 2022). In situ visualization of CFD data has been carried out at scale by simultaneously executing tasks of Soleil-X and rendering tasks on GPUs (Heirich *et al.* 2017). Moreover, ensemble simulations can potentially take advantage of heterogeneous machines by distributing tasks for various kinds of models over heterogeneous processors. Bi-fidelity ensemble simulations with a small number of ensemble members have been demonstrated by optimally mapping tasks on GPUs and CPUs independently using both Soleil-X and HTR (Papadakis 2019; Maeda & Teixeira 2021; Maeda *et al.* 2022). The focus of these studies was nevertheless placed on computer science exploration as well as on verification and performance assessment using specialized software configurations. The critical use of implicit parallelism and Legion for complex task ensembles has not been fully proven for practical applications and for general users.

This brief introduces the task-based ensemble framework (TEnF), an implicitly parallel computational software framework built on Legion for physics-based ensemble simulation and in situ data processing, with CPU/GPU co-operative capabilities. For its top-level interface, the framework combines a physics solver written in Regent and data processing tasks implemented in the Pygion API which directly interact with the solver, enabling the high-level implementation of complex modeling and in situ data processing in short source codes for heterogeneous machines. The framework is portable to various machines in which Legion can be installed. This paper focuses on providing an overview of this framework for general readers who are not necessarily experts in computer science. The rest of this brief is organized as follows. In Section 2, we provide high-level descriptions of each component of the framework. In Section 3, we present simple examples of an ensemble simulation of a high-speed, multi-component jet and in situ data processing using Python libraries including NumPy and scikit. In Section 4, we state conclusions.

2. Framework description

The framework is illustrated in Figure 1. The top-level interface of the framework is composed of the solver written in the Regent language (hereafter solver), co-operative Pygion tasks which can import external Python libraries (hereafter Pygion module), and optional C++ mapper. Legion's original API is object-oriented C++, which allows flexi-

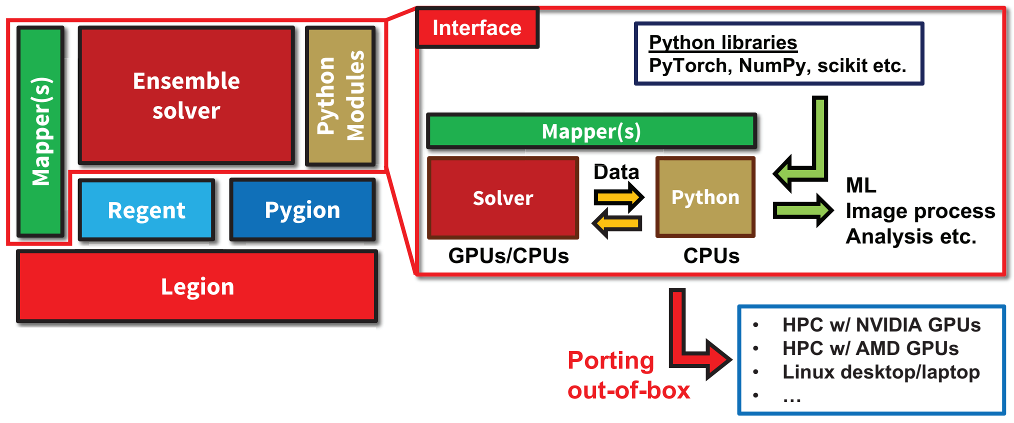

FIGURE 1. Schematic of the framework.

bility in the solver development and data handling, but requires programming experience and deep familiarity with the Legion runtime. The present framework is designed such that development can take place entirely through the higher-level interface without facing the C++ API. The complexity of the solver can be highly dependent on targeted applications, just like any other software. Meanwhile, Regent is a task-based programming language which does not primarily require the specification of parallelisms. It extends Terra's generative high-level descriptions of scalar, array, and matrix arithmetic (DeVito *et al.* 2013), enabling programming of algorithms for solving general partial differential equations (PDEs) and ordinary differential equations. The Pygion module can import almost arbitrary sets of external Python libraries, such as PyTorch, scikit, and NumPy. It can directly access data handled in the solver at runtime through Regent-Pygion interoperation, enabling real-time data processing. The mapper is written in C++ and directly interfaces with the Legion runtime. The development of the mapper may require expertise in computer science. The mapper is optional since the default mapper built in Legion can be used to compile and execute the solver and the Pygion module.

Currently, the solver tasks in Regent can be executed on both CPUs and GPUs, and Python data processing can be executed on CPUs. In CPU/GPU heterogeneous machines, the solver tasks and data processing tasks can respectively use GPUs and CPUs in common compute nodes. Such GPU-CPU co-operation is important from the perspective of efficient resource usage (Maeda & Teixeira 2021). Many existing solvers are designed to use only a single kind of processor. In heterogeneous machines, however, if only one kind of processor is used, the other kind is left idle. The simultaneous use of both kinds of processors can increase the total amount of computation per node unit. The configuration of GPU-CPU cooperation in heterogeneous machines may require highly non-trivial programming efforts for explicit parallel programming models, and the use of Legion may become of critical use. In the meantime, with future updates of Legion, the Python data processing could be executed on GPUs concurrently with the solver and other tasks.

From a developer's point of view, it may be unrealistic for a single programmer to develop all components of the software stack. Rather, the framework assumes interdisciplinary collaboration in its development, where computer scientists, physicists/engineers, and data scientists who would develop, in order, the mapper, solver, and Pygion modules, respectively. From a user's point of view, the interoperation of the Pygion module is the key to high usability of the framework by data scientists. The Regent language is statically typed and requires compilation before execution whenever modifications are made in its source code. The tasks in Pygion module are, on the other hand, dynamically typed by Pygion at runtime (Slaughter & Aiken 2019), meaning re-compilation of the software is in princpile not required after the Pygion module is modified. It is ultimately possible for users who are not familiar with the solver to edit only the Pygion modules and perform data processing, by using the other components as a pre-compiled black-box for real-time data generation on HPC systems. The following sections describe the construction of each component and clarify the terminologies and techniques used in the framework.

2.1. *Solver*

2.1.1. *Task-based programming*

For the solver, programmers are essentially required to only define data and a set of operations which read/write the data, respectively termed as region and task, without the need to explicitly specify processor allocation and data movement in memory hierarchies in a source code. A region can be decomposed into multiple sub-regions, termed as partitions. The operations over the sub-regions are can be parallelized in an efficient manner, semi-automatically (Slaughter *et al.* 2017). The direct interface of Legion is C++ API, and tasks and regions can be expressed as function objects and field data. Regent is an additional layer of abstraction to enable high-level programming, as an alternative interface to the C++ API. In Regent source codes, tasks are simply defined by arguments including operating regions, access privileges to the regions, and a task body. The syntax of Regent is highly interpretable for non-experts. Regent is designed as an extension of the Terra language and is compatible with Lua meta-programming (Ierusalimschy *et al.* 1996) to allow additional flexibility with the back end of LLVM (Lattner & Adve 2004). Embedding of C standard functions is also supported, which can be practically useful for operations including simple string Input/Output for debugging. Various compiler and runtime options are available for optimization. Further formal descriptions and computer science background are left to the original developers and literature (Slaughter *et al.* 2015).

2.1.2. *Task asynchronicity*

One of the core features of Legion is in the asynchronicitu of programmed tasks. For programmers who are accustomed to explicit parallel programming using languages such as Fortran and C++, asynchronous programming may sound unfamiliar. Functions/modules in Fortran and C++ typically follow synchronous programming models, meaning that they are executed in order as written in a source code. In Legion, in contrast, the task launch in runtime is based on the dependencies of operating data and may not follow the order of tasks written in the source code. Rather, the order is automatically determined by runtime, unless explicitly specified by programmers, for example, with a barrier statement. Representative example tasks are given below.

```
task Op1(r_A: region(int2d,fc), r_B: region(int2d,fc))
   update_A(r_A) --task_A updates region_A
   update_B(r_B) --task_B updates region_B
end

task Op2(r_A: region(int2d,fc))
   update_A(r_A) --task_A updates region_A
   read_B(r_A) --task_B reads region_A
end

task Op3(r_A: region(int2d,fc), r_B: region(int2d,fc))
   copy(r_A,r_B) -- copy region_A to region_B
   update_A(r_A) --updates region_A
   read_B(r_B) --reads region_B
end
```

In *Op*1, *task_A* and *task_B* are asynchronous; they are not necessarily executed in order if *region_A* and *region_B* are independent, but runtime decides the timing of execution based on the availability of resources. Meanwhile, in *Op*2, *task_A* and *task_B* are executed in order since *region_A* read by *task_B* is dependent on the operation of *task_A* over *region_A*. In *Op*3, *copy* and *task_A* are asynchronous since they are independent of each other. Similarly, *task_A* and *task_B* are asynchronous. Meanwhile, *task_B* is executed only after the completion of copy. To generalize, tasks are synchronized only when dependencies are identified. This asynchronicity is a critical factor to configure ensemble simulations.

2.1.3. *Ensemble simulations*

We configure ensemble simulations by taking advantage of the task-based, asynchronous programming model. An example task is shown below to configure an ensemble of three simulation samples with a non-trivial workflow.

```
task Ensemble(r_A: region(int2d,fc), r_B: region(int2d,fc), r_C: region(int2d,fc))
where reads writes (r_A,r_B,r_C) do

   for i=1:block

      for j=1:step_A/N_b do
         Iter1(r_A) --Advance the field of r_A by a single step
      end

      for j=1:step_B/N_b do
         Iter2(r_B) --Advance the field of r_B by a single step
      end

      copy(r_A,r_Acp) --Copy r_A to r_Acp
      copy(r_B,r_Bcp) --Copy r_B to r_Bcp
      py_out(r_Acp, r_Bcp) --process r_Acp and r_Bcp using a Pygion module

      Oper(r_A, r_B) --Updates the field of r_A and r_B by certain operations

   end
```

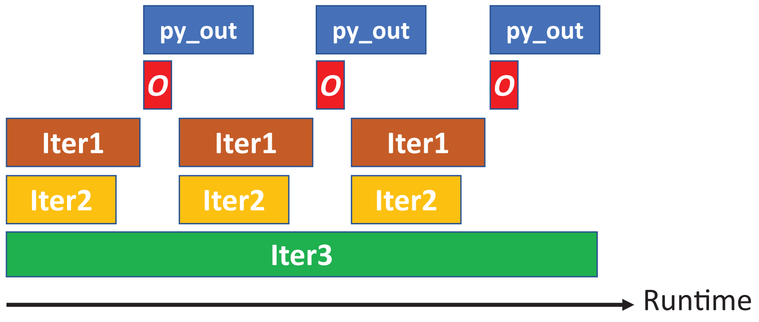

FIGURE 2. Schematic of the task execution in the Ensemble task. "O" represents the *Oper* task.

```
   for j=1:step_C do
      Iter3(r_C) --updates r_C by a single iteration
   end

end
```

In this example, an ensemble simulation of three samples is considered. r_A, r_B, and r_C are independent regions of field data which correspond to the three distinct samples. These field data can contain solutions of PDEs, each of which models a physical system of interest. The field is sequentially updated by tasks $Iter1$, $Iter2$, and $Iter3$ in the for loops. These tasks perform a unit increment of the numerical integration of the field in time, using, for example, a Runge-Kutta method.

The schematic of task execution in the Ensemble task is shown in Figure 2. The iterations of r_A and r_B are blocked into common N_b blocks, by the outer for loop. r_A and r_B are respectively advanced by $step\_A$ and $step\_B$ times in each block. After iterations in each block, r_A and r_B are copied to r_Acp and r_Bcp. r_Acp and r_Bcp are processed in a task exported to Pygion, py_out. Since $Iter1$ and $Iter2$ do not depend on r_Acp and r_Bcp, py_out can be executed asynchronously, possibly in parallel, with the next block of iterations. On the other hand, $Oper(r\_A, r\_B)$ is a task which directly operates on r_A and r_B. Therefore, the next block of iteration has to wait until $Oper(r\_A, r\_B)$ is complete. $Iter3(r\_C)$ is independent and is asynchronous with any other tasks above, since r_C does not appear in the block.

In practical applications, each region can be decomposed into partitions, and these partitions can be processed by multiple processors in parallel. This parallelization can be optimally controlled by a custom mapper, as addressed in the following section.

2.2. *Pygion modules*

The Pygion modules are essentially Python files. The module includes a Legion task (or tasks) defined in Python. General Python libraries can be imported to work with the data and tasks of Legion within the module. In the Ensemble task above, py_out is defined in the Regent code and its body is exported to Pygion by using the following syntax.

```
extern task py_out(r_A: region(ispace(int2d),fc),r_B: region(ispace(int2d),fc))
where reads (r_A,r_B) end
py_out:set_task_id(99)

task Ensemble(r_A: region(int2d,fc), r_B: region(int2d,fc), r_C: region(int2d,fc))
where reads writes (r_A, r_B, r_C) do
...
      py_out(r_Acp, r_Bcp) --send data to Pygion module
...
end
```

In the Pygion module, py_out can be defined as follows.

```
#!/usr/bin/env python3
from __future__ import print_function
import pygion
from pygion import task, Region, R
import numpy as np
import matplotlib.pyplot as plt

@task(task_id=99,
argument_types=[Region,Region],
privileges=[R],
return_type=void,
calling_convention='regent')
def py_out(r1,r2):
   r1_r=r1.r #field "r" in 2D region r1
   r2_r=r2.r #field "r" in 2D region r2
   U,s,V=np.linag.svd(r1_r) # singular value decomposition of r1_r using NumPy
   plt.imsave("1.png",r2_r.transpose()) # save r2_r as an image using matplotlib
```

Multiple tasks can be defined in a single or multiple Pygion modules.

2.3. *C++ mapper*

The set of the solver and the Pygion module can be, as mentioned above, compiled and executed using the default mapper built in Legion, without the need for a custom mapper. Depending on the application, the default mapper can be functional enough in both performance and configuring ensembles. The default mapper can distribute tasks of multiple simulation samples over GPUs and CPUs. For optimizing the performance of highly complex configurations of ensembles, a custom mapper can be desirable. For instance, one can consider an ensemble of a bi-fidelity set of simulation samples and in situ data processing in heterogeneous machines where a small number of high-fidelity, expensive samples are executed in a large number of GPUs, a large number of low-fidelity samples occupy additional GPUs with multiple samples per GPU, and the Pygion modules are executed on CPUs to process the ensemble data in common nodes. Such a complex task mapping is presently not supported by the default mapper.

2.4. *Portability*

An additional critical advantage of the use of Regent and Pygion is that the software package is highly portable. From the same source code, the Regent compiler generates Legion codes which are executable in various computing environments with minimal modifications, by automatically generating arch-specific kernels and source codes for the low-level API and running the Legion compiler to compile them. This generative programming frees programmers from having to manually modify the source code for each environment, as far as Legion and Regent support the environment. This feature is particularly useful to gain portability to different architectures of systems, for example, those of NVIDIA GPUs and AMD GPUs, which typically require distinct kernels for GPU programming. Direct use of Legion C++ API does not provide this portability unless combined with specialized ecosystems like Kokkos (Edwards *et al.* 2014).

3. Illustrative examples

In this section, we describe an example use case. The software package used here is original and includes an in-house light weight solver and Pygion modules, both of which were written by the first author. In the solver, an ensemble of two-component, high-speed fluid flows are simulated. The dynamics of the fluid is modeled by the compressible multi-component Navier-Stokes equations. The equations are discretized on uniform Cartesian grids. The numerical scheme used here is second-order in space in smooth regions and third-order in time. Technical details of the solver, including numerical methods used therein, are the subject of a separate study. An early version of this package, which was named as the Regent-based ensemble flow code † , has been utilized in a recent study with application to online machine learning (Laurent & Maeda 2022*a*,*b*).

In each sample, a high-speed, co-flowing jets of methane and oxygen are injected into a closed rectangular chamber from one of its boundary walls. Although the case is not intended to mimic a specific engineering system, this case can be seen as a canonical model of gaseous fuel injection into a closed combustion chamber. For simplicity, we assume that the viscosity and heat capacity of the gases are constant and heat conduction is negligible. In the example shown here, the ensemble consists of 128 samples. For all samples, the dimension of the chamber is $2.0\times1.0\times0.4$ m on x-y-z Cartesian coordinates with its center located at the coordinate origin. The chamber domain is discritized with $320\times160\times40$ uniform Cartesian grids. The chamber is filled with oxygen at an ambient pressure and temperature. The density of oxygen is randomly perturbed at each grid with a relative variance of 0.01 against the ambient density. The perturbation of the density field is made independent among the samples in order to model the stochasticity that leads to the development of non-identical turbulent flow fields among distinct realizations during the simulation. The inlet is square-shaped with dimensions of 0.1×0.1 m. Oxygen is injected from the inner square region of the inlet with dimensions of 0.075×0.075 m, and methane is injected from the outer region. The pressure is kept constant with zero velocity at the boundary in the inlet region, for oxygen at 12.5 bar and for methane at 10.0 bar. After the simulation begins, the jet of two gases is constantly injected into the chamber in a high-speed, turbulent jet. The two gases are gradually mixed in the chamber, and the chamber pressure is constantly elevated. For the simulation, we use Lassen, a hybrid GPU-CPU machine at Lawrence Livermore National Laboratory which equips four NVIDIA V100 GPUs and an IBM Power9 processor in each node. 64 nodes are used for the ensemble such that two samples are simulated in each node.

At runtime, the tasks for time marching the simulation samples are uniformly distributed across the GPUs available. At every specified time step, the three-dimensional field data of all samples are concatenated in the x-direction and transferred to the Pygion tasks as a set of NumPy arrays. The data are then processed on CPUs in common nodes

† A light version of this code which is configured to work on macOS is available at https://gitlab.com/unpyoukz/r-enfc_light_macos_beta1

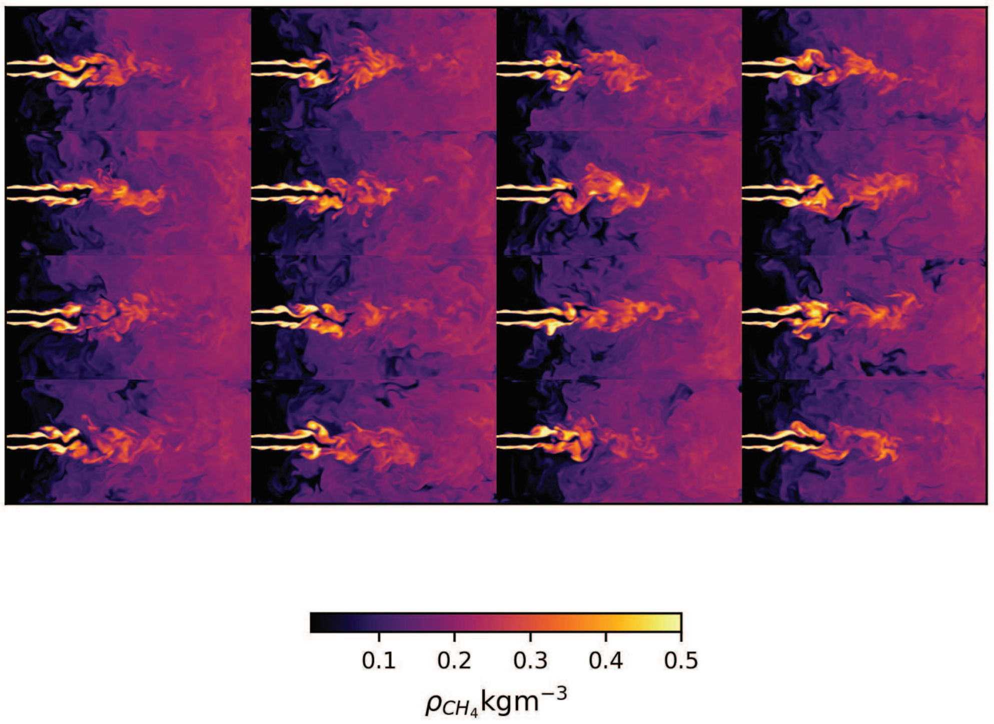

FIGURE 3. Snapshots of the CH_4 density contour on the x-y central cross section at $t = 25$ ms for 16 representative samples in the ensemble of 128 samples.

by using Python libraries in parallel with the simulation. Here we present a set of simple, representative examples of data processing.

Figure 3 shows snapshots of the CH_4 density contour on the x-y central cross section at $t = 25$ ms for 16 representative samples in the ensemble. The snapshots are generated and saved by using the Matplotlib library. In all snapshots, the jet is highly disturbed around the center of the domain due to flow instabilities. The field of mixture is developed and spread in the downstream region. The flow recirculates to the left wall after impingement on the right wall. In the snapshot, the right end of the chamber is filled with the mixture, but the back flow has not reached the left wall near the inlet yet. It is evident that the turbulent flow fields are different for each sample.

In order to assess the coherence among the instantaneous flow fields of the samples, we perform the singular value decomposition (SVD) of the concatenated density field. The SVD function in the NumPy library is used (Harris *et al.* 2020). The linear algebra operations of NumPy are accelerated by the OpenBLAS library. Since the concatenated field is periodic, we expect that the relative variance of the principal component (normalized spectrum) converges in the limit of a large ensemble member. Figure 4 shows the relative variance of the principal components obtained from the online SVD of the concatenated cross-sectional density fields obtained from 1, 16, and 128 samples in the ensemble at $t = 25$ ms, in the descending order. At all indexes, the variance decreases with the number of samples in the data. The difference is much smaller between the results of 16 samples and 128 samples, compared to that between 1 sample and 16 sam-

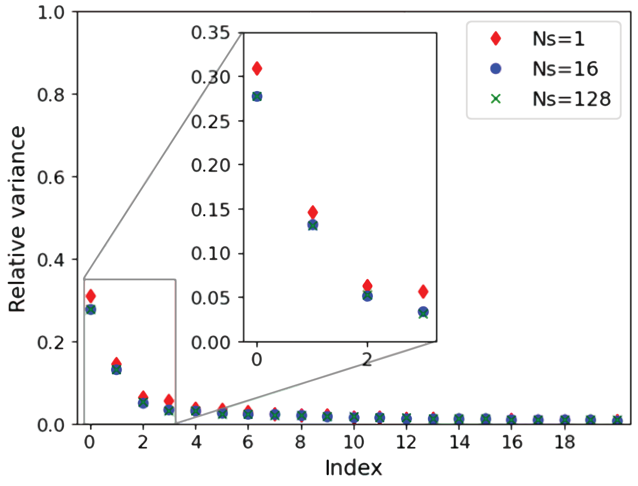

FIGURE 4. The relative variance of the principal components obtained from the online singular value decomposition (SVD) of the CH_4 density field on the x-y central cross section obtained from the ensemble at $t = 25$ ms. Results for 1 sample, 16 samples, and the entire ensemble (128 samples) are shown. Ns stands for the number of samples.

ples. Therefore, the result of the 16 samples can be seen as being nearly converged. $O(10)$ samples are therefore expected to be sufficient to obtain statistical convergence for this specific simulation. The SVD can also be used for lossy compression of the image. The example Pygion module outputs images reconstructed from the first principal component and those reconstructed from the first 10 principal components.

Figure 5 shows scatter plots of the equivalence ratios sampled at the center of the top wall (Φ_t) and that of the bottom wall (Φ_t), obtained from the 128 samples at t =15, 20, and 25 ms. The data points are categorized into three clusters using the K-means algorithm using the scikit-learn library (Pedregosa *et al.* 2011). In all plots, the data points are largely scattered. This scattering is due to the turbulent stochasticity. The distribution is shifted in the top-right direction with the time advancement, showing that methane is increased at the sample locations due to the injection. In the plot of $t = 25$ ms, the horizontal and vertical distributions of the points are evident, as represented by clusters 1 and 2. These clusters show that when the methane is concentrated on the upper wall, it is dilute in the bottom wall, and vice versa. The result implies that the recirculating flow has large-scale coherent structures which make the methane concentration in-homogeneous between the near-wall regions. This is not surprising since the jet and the shed vortexes, which drive the recirculation, are not visibly vertically symmetric visibly (Figure 3). Although this specific case may not have direct connections to specific engineering applications, the consideration of such spatial inhomogeneity of the fuel mixture can be an important factor for the success of ignition by a localized energy deposition in a combustor, for instance, for the laser-induced ignition of an in-space rocket (Maeda *et al.* 2022).

In the present example, data are transferred from the solver to the Pygion module unidirectionally. For different applications, it is entirely possible for the Pygion module to interfere with the solver. For instance, based on the result of the clustering analysis of the simulation samples performed in the Pygion module (Figure 5) at runtime, time marching of samples in a selected cluster(s) can be terminated if they are categorized as no

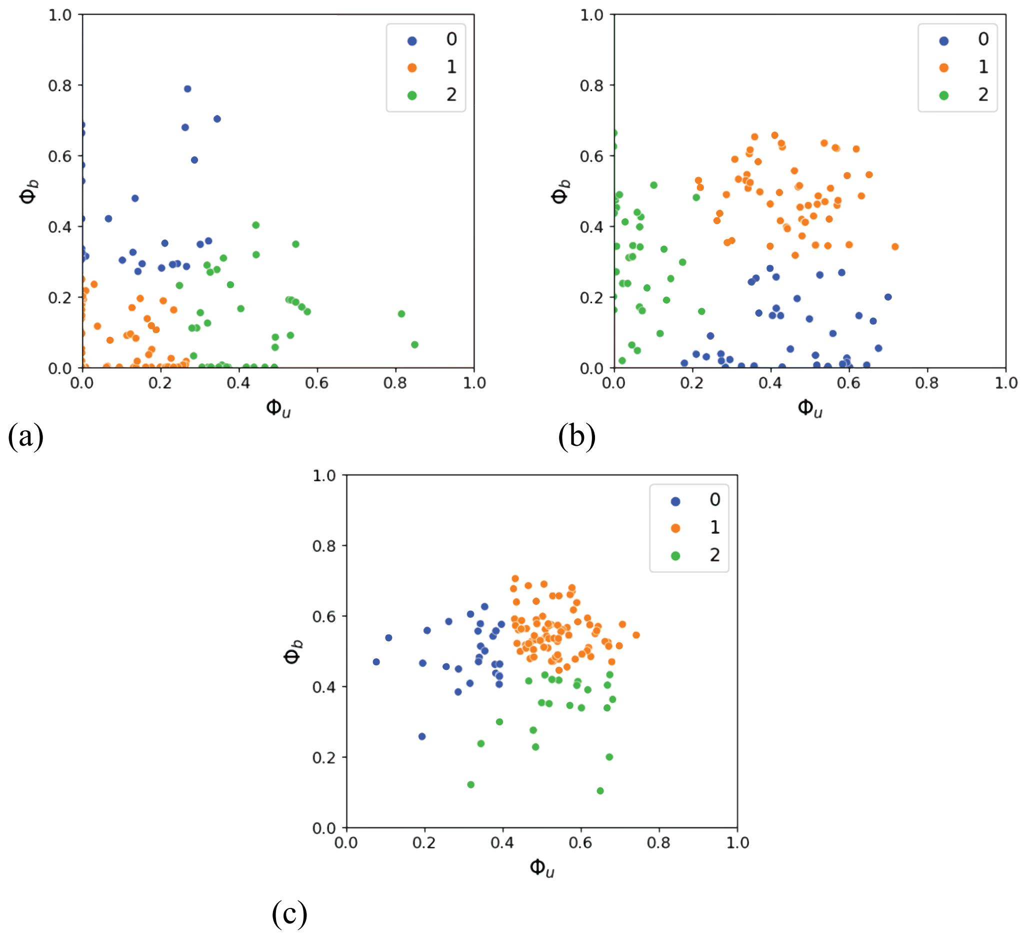

FIGURE 5. Scatter plots of the equivalence ratios, one sampled at the center of the top wall (Φ_t) and the other sampled at the center of the bottom wall (Φ_t), obtained from an ensemble of 128 samples at $t =$(a)15, (b)20, and (c)25 ms. The data points are categorized into three clusters using the K-means algorithm in the scikit-learn library.

longer necessary, while the iteration can continue for the remaining samples. Such a dynamic management of ensemble can realize the efficient resource usage by re-distributing tasks and data of active samples across the resource after removing the terminated ones, without much programming effort.

4. Conclusion

We have introduced TEnF, a GPU-CPU co-operative, implicitly parallel computational software framework for physics-based ensemble simulations and in situ data processing built on the Legion programming system. The framework is primarily designed to enable complex workflows of coupled simulation and data processing on heterogeneous machines in a programmer/user-friendly and portable manner. To this end, in the top-level interface, the Regent language and the Pygion API are respectively used for the task-based programming of the solver and concurrent data processing via Python libraries. A minimal set of illustrative examples was presented for online processing of an ensemble of transient, compressible multi-component jets using external libraries, including the NumPy and scikit libraries. Depending on targeted applications, the framework can accommodate different Regent-based solvers addressing various physical systems, and it can use various Python libraries for online data processing. The implicitly parallel GPU-CPU co-operation for coupled simulation and data analysis could also be realized by implicit programming systems other than Legion. To our knowledge, the critical advantage of our framework is in its high portability and compactness. The framework can be extended in various ways, following updates in Legion. For instance, the data processing could be performed on GPUs in the future for machine learning and other applications.

Acknowledgments

This investigation was funded by the Advanced Simulation and Computing (ASC) program of the US Department of Energy's National Nuclear Security Administration (NNSA) via the PSAAP-III Center at Stanford, Grant No. DE-NA0002373. The authors acknowledge Dr. Elliott Slaughter for valuable discussions and bringing Pygion to their attention. KM also thanks Dr. Slaughter for updating Legion to enable the Regent-Pygion interoperation used in this study. The authors also thank Prof. Gianluca Iaccarino and Prof. Alex Aiken for fruitful discussions, and Dr. Charlelie Laurent for testing an early version of the software package used in the illustrative examples.

REFERENCES

Bauer, M., Treichler, S., Slaughter, E. & Aiken, A. 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*, pp. 1–11. IEEE.

Chen, J. H., Choudhary, A., De Supinski, B., DeVries, M., Hawkes, E. R., Klasky, S., Liao, W.-K., Ma, K.-L., Mellor-Crummey, J., Podhorszki, N. *et al.* 2009 Terascale direct numerical simulations of turbulent combustion using s3d. *Computational Science & Discovery* **2** (1), 015001.

DeVito, Z., Hegarty, J., Aiken, A., Hanrahan, P. & Vitek, J. 2013 Terra: a multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pp. 105–116.

Di Renzo, M., Fu, L. & Urzay, J. 2020 HTR solver: An open-source exascale-oriented task-based multi-gpu high-order code for hypersonic aerothermodynamics. *Computer Physics Communications* **255**, 107262.

Edwards, H. C., Trott, C. R. & Sunderland, D. 2014 Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing* **74** (12), 3202–3216.

Gropp, W., Gropp, W. D., Lusk, E., Skjellum, A. & Lusk, A. D. F. E. E. 1999 *Using MPI: portable parallel programming with the message-passing interface*, , vol. 1. MIT press.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C. & Oliphant, T. E. 2020 Array programming with NumPy. *Nature* **585** (7825), 357–362.

Heirich, A., Slaughter, E., Papadakis, M., Lee, W., Biedert, T. & Aiken, A. 2017 In situ visualization with task-based parallelism. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, pp. 17–21.

Ierusalimschy, R., De Figueiredo, L. H. & Filho, W. C. 1996 Lua—an extensible extension language. *Software: Practice and Experience* **26** (6), 635–652.

Lattner, C. & Adve, V. 2004 LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pp. 75–86. IEEE.

Laurent, C. & Maeda, K. 2022*a* A heterogeneous computing approach to coupled simulation and machine-learning deployment for high-speed flows. *Bulletin of the American Physical Society* .

Laurent, C. & Maeda, K. 2022*b* Online training of neural networks using CFD simulation with application to sensing and flow inference. In *Annual Research Briefs*, Center for Turbulence Research, Stanford University, pp. 111-121.

Maeda, K. & Teixeira, T. 2021 Application-oriented investigation of task-based ensemble co-processing on heterogeneous supercomputers. In *Annual Research Briefs*, Center for Turbulence Research, Stanford University, pp. 217-224.

Maeda, K., Teixeira, T., Wang, J. M., Hokanson, J. M., Melone, C., Di Renzo, M., Jones, S., Urzay, J. & Iaccarino, G. 2022 An integrated heterogeneous computing framework for ensemble simulations of laser-induced ignition. *arXiv preprint arXiv:2202.02319* .

Mittal, S. & Vetter, J. S. 2015 A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)* **47** (4), 1–35.

Pacella, H., Dunton, A., Doostan, A. & Iaccarino, G. 2022 Task-parallel in situ temporal compression of large-scale computational fluid dynamics data. *The International Journal of High Performance Computing Applications* **36** (3), 388–418.

Papadakis, M. 2019 *New Directions in Uncertainty Quantification Using Task-based Programming*. Stanford University.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. & Duchesnay, E. 2011 Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* **12**, 2825–2830.

Slaughter, E. & Aiken, A. 2019 Pygion: Flexible, scalable task-based parallelism with python. In *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, pp. 58–72. IEEE.

Slaughter, E., Lee, W., Treichler, S., Bauer, M. & Aiken, A. 2015 Regent: A high-productivity programming language for hpc with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12.

Slaughter, E., Lee, W., Treichler, S., Zhang, W., Bauer, M., Shipman, G., McCormick, P. & Aiken, A. 2017 Control replication: Compiling implicit parallelism to efficient spmd with logical regions. In *Proceedings of the International*

Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–12.

Slaughter, E., Wu, W., Fu, Y., Garcia, N., Kautz, W., Marx, E., Morris, K. S., Cao, Q., Bosilca, G., Mirchandaney, S. *et al.* 2020 Task bench: A parameterized benchmark for evaluating parallel runtime performance. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15. IEEE.

Thoman, P., Dichev, K., Heller, T., Iakymchuk, R., Aguilar, X., Hasanov, K., Gschwandtner, P., Lemarinier, P., Markidis, S., Jordan, H. *et al.* 2018 A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing* **74** (4), 1422–1434.

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

Treichler, S., Bauer, M., Bhagatwala, A., Borghesi, G., Sankaran, R., Kolla, H., McCormick, P. S., Slaughter, E., Lee, W., Aiken, A. *et al.* 2017 S3d-legion: An exascale software for direct numerical simulation of turbulent combustion with complex multicomponent chemistry. In *Exascale Scientific Applications*, pp. 257–278. Chapman and Hall/CRC.