Received: 1 September 2017 | Revised: 24 March 2018 | Accepted: 3 May 2018
DOI: 10.1002/cpe.4703

WILEY

SPECIAL ISSUE PAPER

Building a scientific workflow framework to enable real-time machine learning and visualization

Feng Li | **Fengguang Song**

Department of Computer Science, Indiana University-Purdue University Indianapolis, Indianapolis, IN, USA

Correspondence
Fengguang Song, Department of Computer Science, Indiana University-Purdue University Indianapolis, 723 W. Michigan St., SL 275, Indianapolis, IN 46202, USA.
Email: fgsong@iupui.edu

Funding information
Purdue Research Foundation; NSF, Grant/Award Number: 1522554

Summary

Nowadays, we have entered the era of big data. In the area of high performance computing, large-scale simulations can generate huge amounts of data with potentially critical information. However, these data are usually saved in intermediate files and are not instantly visible until advanced data analytics techniques are applied after reading all simulation data from persistent storages (eg, local disks or a parallel file system). This approach puts users in a situation where they spend long time on waiting for running simulations while not knowing the status of the running job. In this paper, we build a new computational framework to couple scientific simulations with multi-step machine learning processes and in-situ data visualizations. We also design a new scalable simulation-time clustering algorithm to automatically detect fluid flow anomalies. This computational framework is built upon different software components and provides plug-in data analysis and visualization functions over complex scientific workflows. With this advanced framework, users can monitor and get real-time notifications of special patterns or anomalies from ongoing extreme-scale turbulent flow simulations.

KEYWORDS

computational fluid dynamics, DataSpaces, high performance computing, machine learning, real-time online data analytics, scientific workflows

1 | INTRODUCTION

A wide variety of scientific applications are driven by advancements in high performance computing (HPC). Important examples include adverse weather forecasts,[1] new aircraft design,[2,3] earthquake simulation,[4,5] radiation-treatment planning,[6] and evacuation in the case of the release of hazardous materials.[7] Those real-world applications not only involve intensive computations but also generate massive amount of data. For example, S3D, which is a parallel flow solver to compute direct numerical simulation (DNS) of turbulent combustion, requires extreme-scale computing power and outputs tens of terabytes of data.[8] In an S3D workflow, 30-130TB of data is generated per simulation.[9] Given the extreme-scale data generated, it is challenging to answer certain fundamental questions such as, "did any unusual phenomena happen or not during the simulation?"

In this paper, we extend our previous work[10] and design a computational fluid dynamics (CFD) specific machine learning method to automatically detect anomaly flows. In addition to targeting at automated data analysis (ie, one of the objectives of this work), we also aim to expedite the process of online simulation-time data analysis. Analyzing the simulation-computed results often includes 1) loading massive amounts of data from distributed storage systems; 2) reading data to active disks; and 3) loading data into the main memory for data analysis. However, expensive I/O overhead is involved in this procedure.

To alleviate the expensive I/O overhead, scientists start to use the in-situ (or in-memory) data processing approaches in which data analysis functions can directly interact with the simulation application. In order to implement this approach, different applications are required to be recompiled, linked again, and run in the same process or address space. However, such a deep integration of different applications has the following issues: 1) different applications are designed by experts from different domains, and it can be difficult and unfeasible for them to integrate multiple large codebases; 2) applications may have various memory needs and scaling properties, and such an integration can bring problems such as load balancing to all the applications.

To tackle the issues, we adopt a *tuple space* to connect different applications dynamically at runtime. This way, analysis or visualization specialists no longer need to know the details of simulation code and vice versa. In the tuple-space–based approach, data communication is realized through Remote Direct Memory Access (RDMA), which can significantly reduce the I/O overhead in traditional workflows. The integrated simulation and machine learning framework we build consists of four major applications: simulation application, data processing application, data analytics (ie, machine learning) application, and data visualization application. All the applications are coupled together through the tuple space.

In the paper, we present the following contributions in details:

1. Creation of an advanced real-time workflow system that consists of simulation, real-time data analysis, and visualization components. We focus on the domain of computational fluid dynamics and turbulence data analysis by designing an advanced computing workflow and domain-specific machine learning techniques.
2. Design and development of an interactive software infrastructure where different types of applications can efficiently cooperate. The infrastructure demonstrates an efficient way to combine different applications to enable a real-time CFD-based machine learning and visualization system.
3. Design and implementation of a novel scalable *parallel non-parametric anomaly detection algorithm* that works as an efficient real-time machine learning method for CFD turbulence analyzes. The new algorithm is a variation of the k-medoids clustering method and is augmented with a non-parametric divergence estimator and a distributed sampling scheme to achieve scalable performance.

In the rest of the paper, next section introduces the background of DataSpaces, in-situ visualization techniques, and the detection of vortex flows. Section 3 presents the related work to couple applications in scientific workflows and anomaly detection methods. Section 4 describes the new machine learning method of parallel non-parametric anomaly detection. Section 5 presents the design and implementation of the integrated workflow software framework. Section 6 shows the experimental results, and Section 7 concludes the paper with future work.

2 | BACKGROUND

This section introduces the tuple space software component, the data analysis and visualization software component, and the application of turbulence flow analysis. These components are used to build the integrated simulation and machine learning software framework, which will be presented in Section 5.

2.1 | Tuple space and DataSpaces

Tuple space is an associative memory model that is intended for high-productivity and distributed/parallel computing. In this model, *tuples* are accessed by content and type, rather than by their raw memory addresses. The strength of the model is its ability to describe data without referencing to any specific computer architecture. Unlike in-situ methods which make all applications deeply integrated with each other, tuple space makes it possible to flexibly combine different simulation and analysis applications and to provide insights to users dynamically.

DataSpaces[11] supports the tuple space model and builds a flexible interaction and coordination substrate for various applications so that they can interact frequently at runtime. Since *DataSpaces* can provide such a simple, flexible, and high-level abstraction of concurrent data, we use it in this work to combine CFD simulations with machine learning analysis and visualization automatically.

2.2 | In-situ data visualization and analysis

It often takes weeks or even months to run large-scale simulations, which generate a great amount of simulation results. By connecting simulations with big data analysis, scientists are able to steer simulations, examine predicted phenomena, verify formulated theory, and discover novel patterns or anomalies to initiate new inquiry. To enable this type of simulation-time data analysis, in-situ (or in-memory) data analysis is often used to analyze data while it still resides in memory. Instead of outputting simulation data to secondary storage, data analysis is performed in memory while data is being produced.

Paraview Catalyst (also known as *Paraview Coprocessing Library*)[12] is one of the early attempts to analyze or visualize large-scale datasets. It provides an adaptable application programming interface (API) between simulation and visualization applications. Unlike certain specialized systems such as the hurricane prediction visualization,[13] it offers a generic in-situ visualization framework. In order to instruct or steer simulations at runtime, *Paraview Catalyst* only requires developers to implement three subroutines: *initialize*, *coprocess*, and *finalize*. The major *coprocess* subroutine is responsible for converting raw simulation data to a ready-to-visualize format and performing different visualization functions in each time step.

In this work, we utilize the *Paraview Catalyst* library to fulfill the specific online CFD visualization in real time.

2.3 | Real-world application of vortex detection in turbulence flows

Besides computation-intensive simulations, this paper also targets big data analysis to look for interesting features that can be regarded as patterns occurring in data. Though some features and patterns are common, they may not have precise definitions. Vortex is one of such features in the CFD domain.

Generally speaking, a vortex can be characterized by the swirling motion of fluid around a central region. Our work focuses on the analysis and detection of the important vortex problem in real time.

Vortex-finding methods can be divided into two categories: region based or line based.[14] Region-based vortex detection is used to identify whether continuous cells belong to a vortex, while line-based vortex detection is used to identify vortices by locating vortex core lines. In practice, region-based methods are easier to realize and are less computationally expensive than line-based methods. In this paper, we introduce a parallel region-based vortex detection algorithm using a non-parametric divergence estimation method (details are provided in Section 4).

3 | RELATED WORK

There are several approaches to coupling multiple applications of scientific workflow in HPC systems and to providing an all-in-one interface to end users. In this section, we compare our work with MapReduce frameworks, workflow systems, adaptive I/O, and anomaly detection methods.

MapReduce Frameworks. MapReduce frameworks (eg, Apache Hadoop[15] and Spark[16]) have been widely used nowadays for big data processing. They usually provide simple programming interfaces with data parallelism and fault tolerance. However, applying those frameworks to modern HPC systems remains a challenge mainly because of the unique infrastructure features of leading-edge HPC systems (eg, Infiniband, RDMA).[17] For example, natively, Apache Spark uses Java socket interface with IPoIB (IP over Infiniband) network to enable communications between different nodes. Compared with native verbs interface, the extra TCP/IP and IPoIB layers in the kernel space could bring noticeable overhead. RDMA-SPARK[18] is targeting at this limitation of Spark, and by leveraging advanced features on high-performance networks, a significant performance improvement has been achieved. This is also a sign that, even though the popular MapReduce frameworks can provide simpler programming interfaces, they may not make full usage of modern HPC infrastructure. In contrast, HPC-oriented frameworks, like DataSpaces (which also supports RDMA verbs natively), have advantages in utilizing advanced hardware features in HPC systems and supporting data intensive operations in extremely large-scale efficiently.

Workflow Systems. Another approach to enabling large-scale data processing using HPC resources is using workflow systems, which can define, compose, manage, and execute multiple applications on distributed computing resources to achieve an overall goal. *Kepler* is a widely used workflow framework where scientists, analysts, and software developers may share data and models over the Internet (via Web Services).[19] *Kepler* provides a graphical user interface (GUI) where users can simply select and connect different data sources and analytical components to create a scientific workflow. Such workflows involve applications which can be data-intensive, computation-intensive, or visualization-intensive. Web service extensions are required by workflow systems to access remote resources and services seamlessly. *Pegasus*[20,21] is another popular workflow framework that can map complex scientific workflows to distributed resources. Pegasus is built on HTCondor,[22] a production high-throughput distributed batch computing system. Users can describe the workflow using several popular languages (python, java, perl), then the generated DAX (Directed Acyclic Graph in XML) file can be passed into a "planner", which maps the abstract representation (DAX) of workflow to HPC resources. Nevertheless, these traditional workflow frameworks have to use files and slow disk I/Os to exchange data between applications. They also require querying file existence or flags to detect whether new data is available or not.

Adaptive I/O. *ADIOS* (Adaptive I/O System)[23] supports a range of data transfer methods and supplies scalable, portable, and efficient componentization of the I/O layer on both Linux clusters and supercomputers. *ADIOS* also provides I/O componentization for different data transport methods, which makes switching I/O methods in different infrastructures simpler. In fact, *ADIOS* also offers *DataSpaces* as a data transport method (implemented with a wrapper), since *DataSpaces* can provide low-overhead, high-throughput data extraction from running simulations. Our framework uses *DataSpaces* to integrate CFD simulations with machine learning applications. Unlike *ADIOS* and *DataSpaces*, our system is a high-level application-specific framework, which focuses on simulation-time CFD anomaly detections.

Anomaly Detection Methods. Anomaly detection (or outliers detection) is the process to find data objects with behaviors that are different from expectations.[24] It has been widely used in areas such as intrusion detection, fraud detection, medical/public health detection, and image processing.[25] There are mainly three types of methods that can be used in anomaly detection: *statistical methods*,[26,27] which assume normal data can fit into a statistic model, while the outliers cannot; *proximity-based methods*,[28,29] which regard an object as an anomaly based on proximity (or distance measure); and *clustering-based methods*,[30,31] which assume normal data reside in larger groups, while anomalies can be found in smaller clusters or do not belong to any clusters. As for the real-time CFD turbulence data we are targeting at, training a model using statistical approach can be unrealistic. There is a lack of domain-specific training data and the training process cannot keep pace with the real-time data, which is generated relatively fast by simulations. In our approach, a customized and optimized clustering-based method is designed and developed, and more details will be shown in Sections 4 and 5.3.

4 | AN UNSUPERVISED MACHINE LEARNING ALGORITHM TO ANALYZE CFD FLOWS

Unlike conventional data science applications in which each data point has a fixed number of finite-dimensional features, dynamic fluid flows can be analyzed in a different way using continuous probability distributions, in which case a *group* of data points (ie, a flow region) are processed as independent and identically distributed (i.i.d.) samples. Our basic idea is to use an unsupervised machine learning method to classify flow regions into different categories that exhibit different properties.

We choose to use the k-medoids clustering method[32] to cluster all regions. The less computation-intensive k-means method cannot be simply applied here because it is not suitable to get the means (or "averages") of distributions (flow regions). Instead, we can measure the difference of those distributions, then k-medoids method is used here. The k-medoids clustering method has the advantages of being robust to noise and outliers and uses the general Manhattan Norm[33] to compute the distance between objects instead of the Euclidean distance only. While Euclidean distance measures the shortest distance in the plane, the Manhattan distance measures the shortest path if you are only allowed to move along one dimension. Mahattan distance can be especially useful when different dimensions are not directly comparable.

Our domain-specific k-medoids clustering algorithm is shown in Algorithm 1. The algorithm takes all regions as input data objects and minimizes the sum of in-cluster dissimilarities. The k-medoids method is executed for `num_runs` times, each with a set of different initial random medoids. The corresponding clustering result with the smallest in-cluster dissimilarities will be returned as the final output.

The key point to make the k-medoids method work is the method to compare the dissimilarity between two flow regions. We use the metric of *divergence* and the *non-parametric divergence estimation*[34,35] to compare different regions, where the density of each data point can be estimated using the distance to its k-nearest neighbors, and divergence of two regions can be calculated based on comparison of densities over the whole region.

To cope with the fluid flow data analysis problem, we use three dimensions to represent each point in a fluid region: v_x and v_y for the point's velocity in the x and y directions, respectively, and d_c for the distance from the point itself to the fluid region's center. We choose to add the d_c dimension because the velocity in a vortex most often changes with the distance from the center, which has also been reported in the work of Póczos et al.[36] Since the third dimension (d_c) is not directly comparable with the other two, it is an obvious choice to utilize the Manhattan distance to give more robust results.

Next, to measure the divergence between two groups of data points, we use the commonly used L_2 divergence, which is explained in Definition 1.

Definition 1. Let p and q be densities over R^d, then the L_2 divergence is

$$L(p||q) = \int (p(x) - q(x))^2 \mathrm{d}x)^{1/2}.$$

Moreover, to avoid recomputing the divergence between the same pair of regions, Algorithm 1 initially constructs a dissimilarities matrix and feeds it to the k-medoids method.

Algorithm 1 CFD specific K-medoids clustering based on the Non-parametric Divergence Estimation (*npdivs*).

Input:
regions of data points $R_{1:n}$, each point has (v_x, v_y, d_c)
k, number of clusters
num_runs, number of runs of k-medoids method
Result:
cluster_ids, cluster id of each region
Begin
estimate all pairwise divergences: $divs(R_1, R_2), divs(R_1, R_3) \ldots divs(R_{n-1}, R_n)$ among all regions from $R_{1:n}$ using npdivs method ;
construct a distance matrix D from $divs(R_1, R_2), divs(R_1, R_3) \ldots divs(R_{n-1}, R_n)$;
// run k-medoids for multiple times
for $i = 0; i < num\_runs; i++$ **do**
  randomly choose k initial medoids ;
  run k-medoids until sum of in-cluster dissimilarity doesn't change ;
  update *cluster_ids* if smaller sum of in-cluster dissimilarity is achieved in this run ;
end
// return best clustering results
return *cluster_ids* **End**

An overall illustration of our algorithm is shown in Figure 1, with four stages present as follows:

1. **Region division**. Fluid flows are divided into *flow regions* based on the geometric information.
2. **Divergence calculation**. Divergences between *flow regions* are measured using the L_2 divergence metric, and a distance matrix is generated.
3. **K-medoids grouping**. Regions are assigned into different groups using k-medoids, given the information from the distance matrix.
4. **Label assigning**. Based on the grouping information, a label is assigned to each group (for further visualization).

Remark 1. A naive implementation of the Algorithm 1 cannot scale well for large size datasets due to the expensive calculation of pairwise divergence among all the regions. Hence, we design and develop an improved algorithm to solve the issue, which will be presented in Section 5.3.

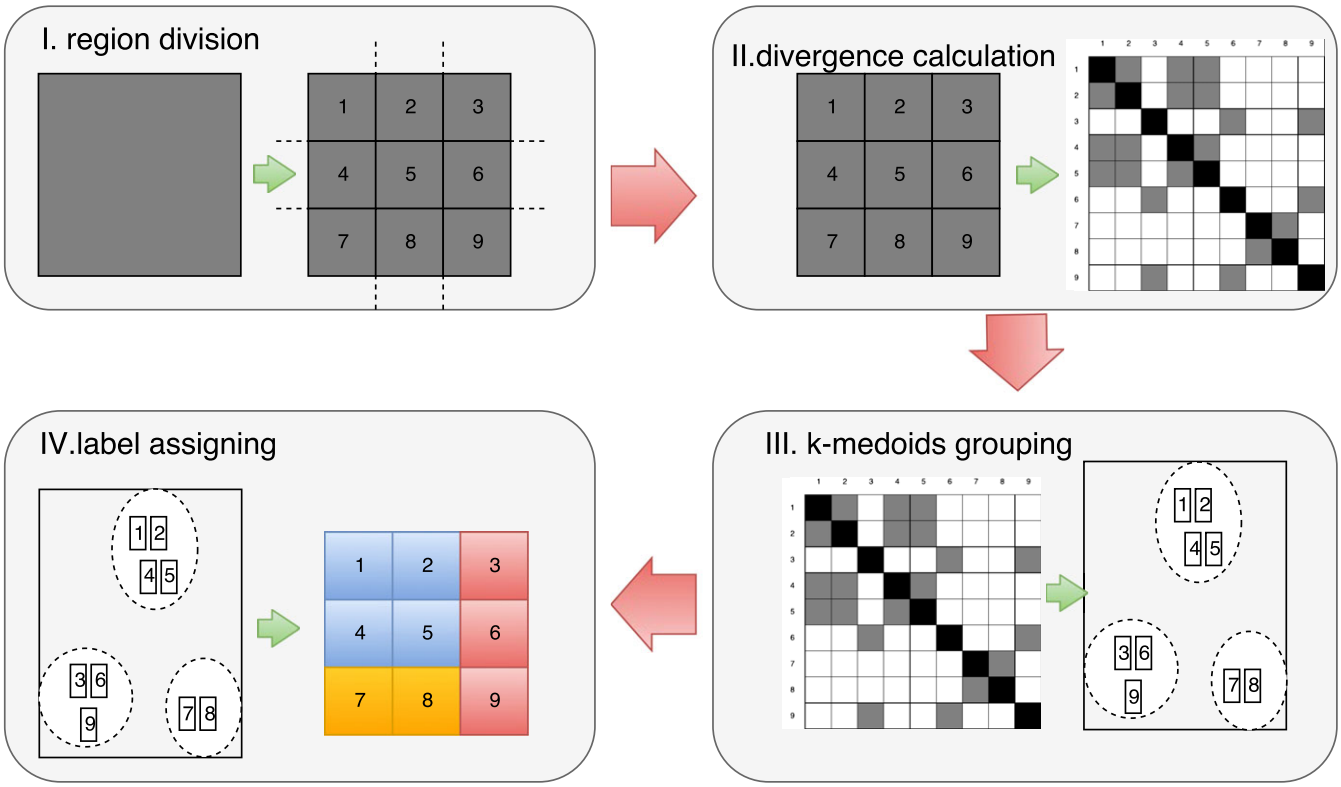

FIGURE 1 Demonstration of the proposed dynamic fluid anomaly detection algorithm

5 | SOFTWARE FRAMEWORK

In a typical scientific workflow system, multiple components can coexist and each of them performs specific functionalities. In our vortex detection problem, we use four components, ie, *simulators*, *data processing*, *data analysis*, and *Catalyst* (visualization), as shown in Figure 2. *DataSpaces* is used as communication channel among all the components due to its high efficiency and portability. All components are coupled together and no disk I/O is involved.

In the proposed framework, *simulators* will run the computation-intensive simulations and flush resulting data into *DataSpaces* at each time step. Instead of consumed by one single application, simulation data can be simultaneously obtained by both *data processing* and *Catalyst* components.

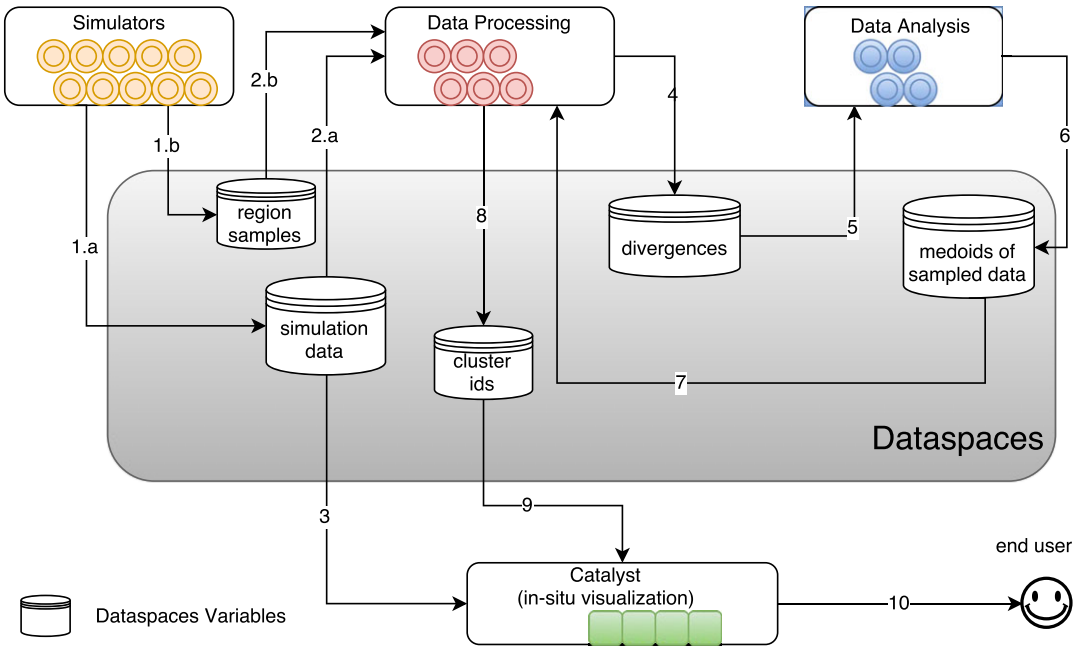

FIGURE 2 Architecture of the software framework. All applications are connected together using *DataSpaces* variables

Data sent to *data processing* component is preprocessed first and then further analysis will be conducted using the intermediate data. In the same time, simulation data also flows to *catalyst*, which will visualize data in a proper way and give users a view of both turbulence flow (through path 3 in Figure 2) and data analysis results (through path 9 in Figure 2).

Among all the four components, *data processing* and *data analysis* are both related to extracting and predicting information from simulation dataset; however, they are placed in different components since they are aiming at separate procedures and run in different sales. More detailed information of each component will be introduced in the rest part of this section.

5.1 | Simulators

Our system targets at real-time and large-scale scientific simulations, where data needs to be extracted and analytics information should be given as soon as possible. However, archived, static simulation data is used first to verify the correctness of our algorithm design and system implementation. More specifically speaking, the archived simulation dataset we choose, which is from turbulence database,[37] is well structured and there are existing analysis tools which we can refer to. More details about this dataset are provided in Section 6.

The other type of simulation data we use is the real-time data flow from CFD software (eg, *OpenFoam*[38]). Widely used in production in both engineering and science fields, openfoam can run in extremely large scale and generate a huge amount of data for specific workload. It provides a large range of solvers for different problems.

IcoFoam, which is a transient solver for incompressible, laminar flow of Newtonian fluids,[39] is used by us to simulate *2D lid-driven cavity flow* problem[40] To connect *simulators* with other applications, we add an alternative I/O routine to *IcoFoam* so that simulation results can be transferred into *DataSpaces* instead of written into persistent storage.

5.2 | Data processing and data analysis

Once we get all the simulation results and send data to *DataSpaces*, the challenge becomes 1) how we can consume the data in a proper and efficient way; and 2) how to produce useful information based on limited data input. Because the size of data generated in each iteration is large, keeping all records in memory can be very expensive. It will be attractive if data can be automatically extracted and hidden information can be given in real time.

The naive unsupervised machine learning method has been discussed in Section 4, which involves four steps: *region division*, *divergence calculation*, *k-medoids grouping*, and *label assigning*, as shown in Figure 1. Algorithm itself is straightforward and what really matters is how to nicely distribute workloads among all computing resources. To have better understanding of how computing resources are consumed in this algorithm, we implemented sequential version first and found out that pair-wise divergence calculation uses most of CPU time.

Based on this observation, we divide the knowledge discovery functionality into two applications: 1) *data processing* component which uses massive nodes to calculate divergences between region pairs; and 2) *data analysis* component, which uses smaller amount of resources to analyze the reduced data (divergence matrix).

The two applications work as follows: 1) simulation data is first divided into regions and distributed to different *data processing* processes; 2) *data processing* processes then reads the assigned regions and computes the divergences between all pairs of regions; 3) after step 2 is complete, *data analysis* processes will read the computed divergences and perform k-medoids clustering to search for the k global medoids, where k is the number of clusters specified by users; and 4) *data processing* processes assign a cluster ID to each of its allocated regions based on their distances (ie, divergence) to the k medoids.

The rest of the section provides more details about the *data processing* and *data analysis* applications.

5.2.1 | Data processing

One concern when running the divergence calculation is how we can split all the raw simulation data among massive *data processing* processes. DataSpaces can provide application developers a very intuitive programming interface, which asks for logical area where the I/O operations will be performed on. However, carelessly using this interface may produce significant performance degradation. For instance, DataSpaces generally performs better when the operation has good spatial locality, so in the *data processing* application side, we divide data into lengthy strips instead of square blocks, as shown in Figure 3. The layout of strips is also beneficial to the sampling technique, which will be introduced in Section 5.3.

Synchronization between the data processing application and the next data analysis application is supported by customized DataSpaces locks (details will be discussed in Section 5.5).

5.2.2 | Data analysis

Once all the divergences are calculated by the *data processing* application, the *data analysis* application will take over and apply k-medoids clustering methods to construct the distance matrix, thus determining new medoids in each group of regions. The new medoids information will immediately be sent back to the *data processing* processes so that they can assign a cluster ID to every region.

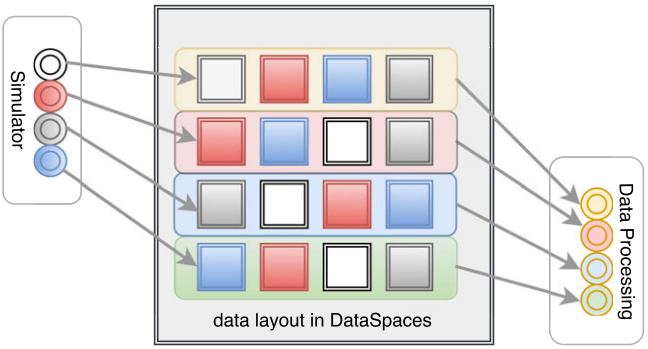

FIGURE 3 Data layout of simulation data in *DataSpaces*. *Simulators* store simulation data in DataSpaces using 2D logical geometrical locations, while *data processing* processes read simulation data in stripes

5.3 | Distributed sampling

Applying k-mediods method using divergence matrix seems straightforward in Section 5.2.2, but maintaining such divergence information becomes very challenging for large dataset. For a data input containing n square regions, there will be $O(n^2)$ divergences to be calculated for all the pairs. In this section, a distributed sampling method is proposed, which is specially optimized for *DataSpaces* and also greatly reduces the computation time in divergence calculation.

The idea of sampling in k-mediods is not new. CLARA (Clustering LARge Applications)[41] is an example to use sampling to reduce computation complicity for clustering tasks. Instead of getting new medoids from original dataset, CLARA will first randomly select a smaller amount of samples, update new medoids from samples, and use distances to these "estimated medoids" to assign the original data objects into different classes. Experiments show applying such sampling can give higher efficiency and while not losing much accuracy, when compared with the original k-medoids method.[41]

To apply the idea of sampling into our system, the *naive* approach could be: 1) *simulators* flush all simulation data into *DataSpaces*; 2) each time when a *data processing* process wants to calculate the divergence for a region pair, it reads that pair to local memory from *DataSpaces*; and (3) algorithm in Section 4 will be used to figure out how different those two regions are with each other.

Note that *DataSpaces* needs to issue one I/O (get or put) operation for each logically continuous block, and each I/O operation involves synchronization among all application processes, thus reading such two randomly selected regions from *DataSpaces* is expensive, especially when there needs to be a large amount of samples. Instead of using such small-granularity *DataSpaces* operations, a better way is combining small I/O operations together and reading a large chunk of data at once.

In our solution, this is actually done in a 3-phase fashion, as shown in Figure 4:

i. Each simulator will select several regions as *samples*, concatenate them together, and insert them into *DataSpaces* in one "put" operation.
ii. Central *DataSpaces* will gather sampled regions from all the *simulators*.
iii. Each *data processing* process needs only one single "get" operation to get a full copy of the central samples. After that, it can calculate a subset of the divergences of those samples.

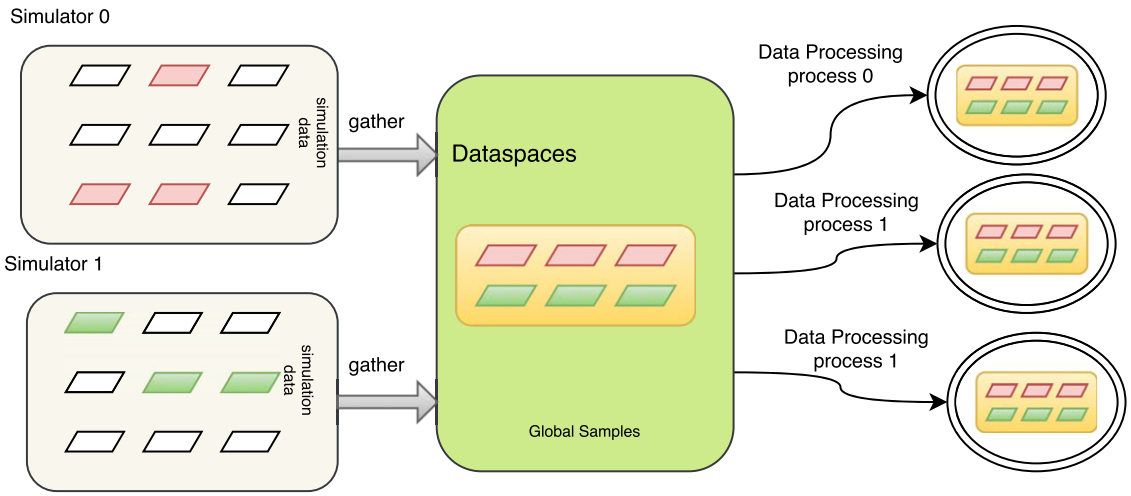

FIGURE 4 Distributed sampling using *DataSpaces*. Each simulator will perform local sampling first and samples from all simulators will be gathered in central *DataSpaces* variable. Each *data processing* process can get a full copy of samples from DataSpaces

5.4 | In-situ visualizations with catalyst

The goal of designing this workflow system is to provide user with a vision of both simulation flow and analysis results. As we have mentioned, Paraview Catalyst is used as the visualization component to give real-time view of both CFD simulations and clustering analysis results. Catalyst is a library which can be conveniently linked to simulation application. One modification we have made is, instead of directly linking catalyst with simulation, we combine input from both simulation and data analysis (through *DataSpaces*) and then visualize them together. A dedicated server is also settled so that multiple user connections can be simultaneously served.

Catalyst needs to be "instructed" with simulation so that it can give the desired visualization output for different kinds of requests. To achieve that, application developers usually follow two steps to work with Catalyst:

i. Pre-process step, where user needs to define how he wants the simulation data to be visualized, which is done by defining visualization pipelines in python scripts.
ii. Execution step, where *Catalyst* is linked with simulation application and visualization data structure is generated based on the rules defined in step i.

The advantage of separate configurable python file is that, if a user wants to present data in a different view or layer, simply modifying the python file is sufficient, instead of recompiling applications. Another advanced feature of the Catalyst tool is a user does not even need to "write" the python file from scratch if he uses the "Catalyst Script Generator plugin". To achieve this, the following steps are required: 1) open a piece of *sample data* (usually a data cutout of one time step) in Paraview GUI; 2) add visualization pipelines, which includes dragging and dropping different visualization components, and modifying default configurations in the GUI; and 3) use the plugin to generate scripts.

The only problem left is how to get such *sample data*. Actually, *Paraview* provides template scripts which can transform structured data into VTK files and those produced files can be used as sample input.

5.5 | Inter-application synchronizations using DataSpaces

DataSpaces can help connect various applications, but problems may arise when synchronization is not handled carefully. Using default configurations, data (to be transfered) is saved in the main memory of staging server, which turns out to have limited capacity. It is not an issue most of the time because *DataSpaces* ensures that the next iteration of simulation would not run until all the consumers have successfully read the data from the staging server. This policy is enforced by using very strict collective locks, which are not what we actually want, since they will stop simulations from continuous running and slow down the throughput of the whole workflow.

To address this problem, we use separate locks in each iteration so that the simulation will never get blocked and data will be immediately inserted into *DataSpaces* once it finishes one step of the simulation.

In this way, simulation will run at full speed and users will also get corresponding analysis and visualization results just in time.

6 | EXPERIMENTAL RESULTS

We evaluate our scientific workflow framework using the *Karst* computer system located at the Indiana University.[42] As a high-throughput computer system, *Karst* has 228 general access compute nodes (IBM NeXScale nx360 M4), and each node is equipped with two Intel Xeon E5-2650 v2 8-core processes and 32GB memory. All compute nodes are installed with Red Hat Enterprise Linux 6 with 10-gigabyte Ethernet interconnection.

The three experiments are designed as follows. Firstly, we show how our framework can provide real-time flow visualization and clustering-based anomaly detections. Then, communication overhead and efficiency of our framework are evaluated and analyzed in experiment 2 with a real simulation generator. Finally, we run our framework using various configurations and demonstrate the speedup and scalability of the framework.

6.1 | Turbulence analysis with JHTDB dataset

We run the first experiment with the forced isotropic dataset (coarse)[43] from Johns Hopkins Turbulence Databases (JHTDB). In this archive/static database, all the simulation data is generated from a 1024×1024 grid and is well formatted using the VTK/HDF5 format.

The data we actually use is a portion of the original isotropic data, which has 100 frames for 200×200 fluid data elements. A crawling script first fetches these frames from JHTDB datacut service and saves all of them into filesystem in advance. Then at each step, a file reader will take one frame (namely, a separate VTK/HDF5 file) as input, extract information, and feed the transformed data to our data analytics applications. The data processing algorithm will partition data in to small regions and each of those regions will be assigned with a label once divergences of all region pairs are calculated.

Figure 5 shows the demo of the analysis/visualization results of the JHTDB isotropic dataset. The right side is directly rendered from the VTK file of the current step, in which warmer color means higher pressure and the glyphs represent velocities in different directions; in the left, the corresponding colored output is given by our clustering algorithm. Small colored blocks are fix-sized regions, and each region contains certain number of data points (in Figure 5, region size of 20×20 is used). Note that, in this experiment, we do not use the distributed sampling because dataset itself is relatively small and the main purpose of this experiment is to demonstrate how our system can give in-time and meaningful visualization from continuous data input. Distributed sampling is designed to address scalability challenges and will be discussed in the next two experiments.

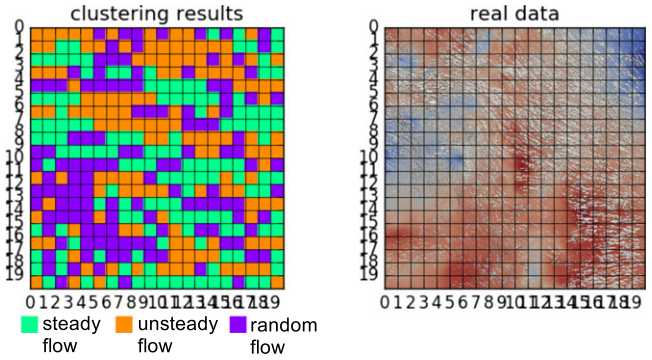

FIGURE 5 Demo on the JHTDB isotropic dataset

Three types of flow patterns are recognized from Figure 5:

1) steady flow, where all elements in this region have similar moving directions and velocity; 2) unsteady flow, where there are swirling motions of fluid; and 3) random flow, where fluid shows unexpected patterns over time.

Those three patterns illustrate typical behaviors in fluid flows, and when used along with the directly visualized simulation data (the right part in Figure 5), scientists can get clearer insights into what is happening during simulations.

6.2 | Execution time breakdown

The second experiment is designed to examine the communication overhead and efficiency of our framework, provided that different applications usually execute quite various tasks and there are also data dependencies between those tasks.

We begin this experiment with the configuration, as shown in Table 1. There are in total 16 simulation processes and each of them is responsible to solve the dynamic fluid problem in a grid with a size of 1024×1024. As for *data processing* components, we use 32 processes to get stripped data and calculate all the divergences. Then, one *data analysis* process and one *Catalyst* process are used respectively, since the workloads in those two components are relatively smaller once all divergences are obtained from *data processing* components.

Average data transfer and computation time of all the four applications (*simulator*, *(data) processing*, *(data) analysis*, and *Catalyst*) for 30 time steps is shown in Figure 6. Several conclusions can be easily drawn from the observation:

1. Communication time is much less than computation time in all the four components.
2. Among all applications, *simulator* has the longest computation time, which suggests that the data source would not be delayed by the other applications, and this will make sure the overall workflow can proceed in rather fast speed.
3. Even though other applications do run slower, they have comparable computation time with *simulator* components. This is a sign that the computation resources in those components are busily occupied most of the time.

6.3 | Scalability evaluation

For the workflow users, they usually care more about the overall latency of the whole workflow system. This latency is the time interval since the data source application starts the first step until the last data consumer application finishes its last step. This metric can be a reference to the walltime of the job for the whole workflow. Since highly parallel HPC resources are used, it is interesting to examine how this latency will change when more nodes are used.

Here, we use a same dataset as that in experiment 2, and the 4096×4096 grid can generate 1GB of simulation data in each step. Table 3 shows behaviors of our system under different configurations. The metrics we use are explained in Table 2, where the entry we are most interested in *latency_all*, which describes the end-to-end time spent between the beginning of simulation and the termination of all data consumer applications.

TABLE 1 Configuration for the time-breakdown experiment

Simulation processes	16
Fluid grid size/process	1024×1024
Data processing processes	32
Data analysis processes	1
Catalyst processes	1
Region size	16×16
Number of clusters (k)	3
Max number of versions buffered	30

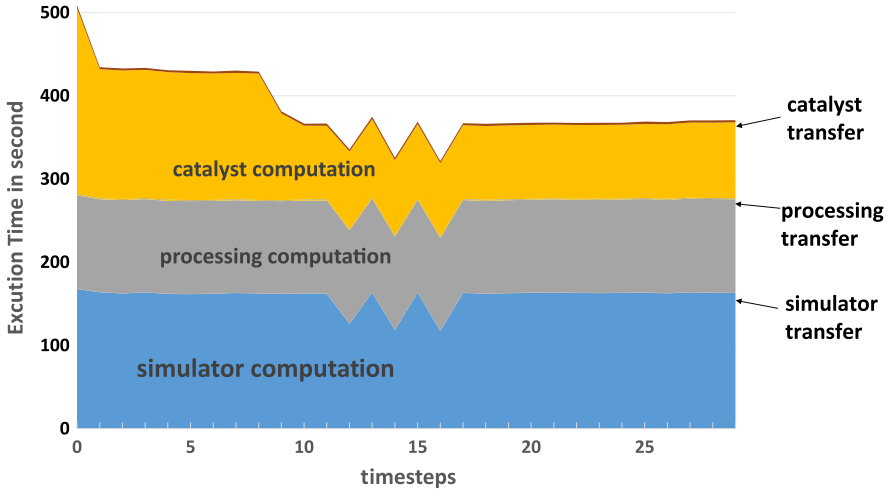

FIGURE 6 An execution time breakdown

TABLE 2 Selected metrics for scalability evaluation

Metric Name	Description
np_sim	Number of processes for simulation application
np_dp	Number of processes for data processing application
time_divs_cal	Average time spent on divergence calculation in each time step
time_cluster_assign	Average time spent on assigning cluster ids using medoids information
time_catalyst	Average time used by catalyst application in each timestep
latency_produce	Latency produced by the producers: simulation application
latency_consume	Latency produced by the consumers: data processing, analysis, and visualization
latency_all	Overall end-to-end latency

TABLE 3 Configurations and results of scalability evaluation

np_sim	np_dp	time_divs_cal	time_cluster_assign	time_catalyst	latency_produce	latency_consume	latency_all
16	32	59.3 s	88.9 s	93.75 s	340.5 s	453.0 s	793.5 s
64	128	14.1 s	20.1 s	91.6 s	53.5 s	140.2 s	193.7 s

problem size is 4096×4096

The first configuration uses 16 simulation processes and 32 data process processes, which is actually the same configuration as in experiment 2. In this case, the overall latency is 793.5 s. We four time the number of processes for both *simulator* application and *data processing* application in the second configuration and it gives a much lower latency of 193.7s. A sample size of 512 is used in both configurations.

To dig deeper into this significant drop in latency, we can take a closer look at the *latency_produce* and *latency_consume* items in Table 3, where we can find that both of those two latencies are greatly reduced. This tells us that the overall shorter end-to-end time is contributed by applications in both data source and data consumer sides. The *time_divs_cal*, *time_cluster_assign*, and *time_catalyst* entries are the average timing of specific applications. From those three metrics, we can infer the reasons for differences in end-to-end time between two configurations: 1) divergence calculation time (*time_divs_cal*) is reduced since more *data processing* processes are working on the same simulation problem size; and 2) *time_cluster_assign* is also decreased because label-assigning is also actually done by the *data processing* application.

The catalyst simulation time remains the same because we do not change either the number of catalyst processes or the data input of *Catalyst* (simulation size and number of regions to be labeled).

7 | CONCLUSION AND FUTURE WORK

The world of scientific computing is in the process of being transformed by large-scale efforts to combine HPC simulations with advanced big data analytics techniques. However, it is rarely studied how to apply machine learning to extreme scale scientific simulations for obtaining new knowledge rapidly. Also, current methods for achieving efficient combination of HPC and big data analysis are still unsettled.

With the aim of providing an efficient integrated simulation-time machine learning framework, this work creates and builds a workflow software framework to combine simulation application with various data analysis applications. The framework consists of a parallel simulation application, a data processing application, a machine learning application, and an in-situ visualization application, which are coupled together through a globally shared tuple space. Each of the applications is running on a separate subset of compute nodes and interacts with each other. In the framework design,

we use DataSpaces as the shared tuple space to enable communication between applications, and we also avoid the expensive disk I/O overhead by using RDMA techniques.

Along with the enabling integrative software framework, we also design and develop a parallel non-parametric clustering method to perform online machine learning for CFD simulations, which is based upon a new density estimation method optimized for flow regions. The parallel non-parametric clustering method has also been extended with distributed sampling so that the data analysis time can be significantly reduced. We apply the workflow framework to the CFD application of vortex/anomaly detection in turbulence flows. With the new non-parametric divergence estimation method, we are able to cluster fluid regions into various categories. As a result, output from the online machine learning and related fluid properties such as velocity and pressure can be displayed side-by-side by an in-situ visualization tool in real time. By using this new framework, scientists can get real-time notifications of special patterns or anomalies that are happening in turbulence flows. Also, the experimental results show that the integrated framework is efficient and can scale well when more computing resources are used.

Our future work along this line will be studying new parallel machine learning algorithms and statistical methods that target extreme-scale simulation data analysis and designing dynamic methods to schedule computing resources to different workflow stages in a balanced manner.

ACKNOWLEDGMENTS

This material is based upon research partially supported by the Purdue Research Foundation and by the NSF Grant No. 1522554. We would like to thank the DataSpaces team in the Rutgers Discovery Informatics Institute for their assistance to our framework design and deployment.

ORCID

Feng Li http://orcid.org/0000-0002-8505-5208
Fengguang Song http://orcid.org/0000-0001-7382-093X

REFERENCES

1. Shimokawabe T, Aoki T, Ishida J, Kawano K, Muroi C. 145 TFLops performance on 3990 GPUs of TSUBAME 2.0 supercomputer for an operational weather prediction. *Procedia Comput Sci*. 2011;4:1535-1544.
2. Johnson FT, Tinoco EN, Yu NJ. Thirty years of development and application of CFD at Boeing commercial airplanes, Seattle. *Comput Fluids*. 2005;34(10):1115-1151.
3. Ball DN. Contributions of CFD to the 787 - and Future Needs. Stuttgart, Germany: HLRS High Performance Computing Center; 2008.
4. Sobhaninejad Gh, Hori M, Kabeyasawa T. Enhancing integrated earthquake simulation with high performance computing. *Adv Eng Softw*. 2011;42(5):286-292.
5. Cui Y, Olsen KB, Jordan TH, et al. Scalable earthquake simulation on petascale supercomputers. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10); 2010; New Orleans, LA.
6. Jia X, Ziegenhein P, Jiang SB. GPU-based high-performance computing for radiation therapy. *Phys Med Biol*. 2014;59(4):R151.
7. Babendreier JE, Castleton KJ. Investigating uncertainty and sensitivity in integrated, multimedia environmental models: tools for FRAMES-3MRA. *Environ Model Softw*. 2005;20(8):1043-1055.
8. Hawkes ER, Sankaran R, Sutherland JC, Chen JH. Direct numerical simulation of turbulent combustion: fundamental insights towards predictive models. *J Phys Conf Ser*. 2005;16(1):65.
9. Chen JH, Choudhary A, De Supinski B, et al. Terascale direct numerical simulations of turbulent combustion using S3D. *Comput Sci Discov*. 2009;2(1):015001.
10. Li F, Song F. A real-time machine learning and visualization framework for scientific workflows. In: Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact (PEARC17); 2017; New Orleans, LA.
11. Docan C, Parashar M, Klasky S. DataSpaces: an interaction and coordination framework for coupled simulation workflows. *Clust Comput*. 2012;15(2):163-181.
12. Fabian N, Moreland K, Thompson D, et al. The ParaView coprocessing library: A scalable, general purpose in situ visualization library. Paper presented at: 2011 IEEE Symposium on Large Data Analysis and Visualization; 2011; Providence, RI.
13. Ellsworth D, Green B, Henze C, Moran P, Sandstrom T. Concurrent visualization in a production supercomputing environment. *IEEE Trans Vis Comput Graph*. 2006;12(5):997-1004.
14. Jiang M, Machiraju R, Thompson D. Detection and visualization of Vortices. *The Visualization Handbook*. Burlington, MA: Elsevier Butterworth-Heinemann; 2005:295.
15. Apache Hadoop Project. Apache Hadoop website. http://hadoop.apache.org. Accessed March 12, 2018.
16. Apache Spark Project. Apache Spark website. http://spark.apache.org. Accessed March 12, 2018.
17. Lu X, Rahman MWU, Islam N, Shankar D, Panda DK. Accelerating spark with RDMA for big data processing: Early experiences. Paper presented at: 2014 IEEE 22nd Annual Symposium on High-Performance Interconnects; 2014; Mountain View, CA.
18. Lu X, Shankar D, Gugnani S, Panda DK. High-performance design of apache spark with RDMA and its benefits on various workloads. Paper presented at: 2016 IEEE International Conference on Big Data (Big Data); 2016; Washington, DC.
19. Ludäscher B, Altintas I, Berkley C, et al. Scientific workflow management and the Kepler system. *Concurrency Computat Pract Exper*. 2006;18(10):1039-1065.
20. Deelman E, Singh G, Su M-H, et al. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Sci Program*. 2005;13(3):219-237.

21. Pegasus Workflow System Project. Pegasus website. http://pegasus.isi.edu. Accessed March 10, 2018.
22. Thain D, Tannenbaum T, Livny M. Distributed computing in practice: the Condor experience. *Concurrency Computat Pract Exper*. 2005;17(2-4):323-356.
23. Lofstead JF, Klasky S, Schwan K, Podhorszki N, Jin C. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In: Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments (CLADE '08); 2008; Boston, MA.
24. Han J, Pei J, Kamber M. *Data Mining: Concepts and Techniques*. Waltham, MA: Elsevier; 2011.
25. Chandola V, Banerjee A, Kumar V. Anomaly detection: a survey. *ACM Comput Surv*. 2009;41(3):15.
26. Portnoy L, Eskin E, Stolfo S. Intrusion detection with unlabeled data using clustering. In: Proceedings of ACM CSS Workshop on Data Mining Applied to Security; 2001; Philadelphia, PA.
27. Ye N, Chen Q. An anomaly detection technique based on a chi-square statistic for detecting intrusions into information systems. *Qual Reliab Eng Int*. 2001;17(2):105-112.
28. Eskin E, Arnold A, Prerau M, Portnoy L, Stolfo S. A geometric framework for unsupervised anomaly detection. *Applications of Data Mining in Computer Security*. Boston, MA: Springer; 2002:77-101.
29. Bay SD, Schwabacher M. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining; 2003; Washington, DC.
30. Sequeira K, Zaki M. ADMIT: Anomaly-based data mining for intrusions. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '02); 2002; Edmonton, Canada.
31. Ertöz L, Steinbach M, Kumar V. Finding topics in collections of documents: a shared nearest neighbor approach. *Clustering and Information Retrieval*. Boston, MA: Springer; 2004;83-103.
32. Kaufman L, Rousseeuw P. Clustering by means of medoids. *Statistical Data Analysis Based on the L_1-Norm and Related Methods*. Amsterdam, The Netherlands: North-Holland; 1987.
33. Bourbaki N. *Topological Vector Spaces: Chapters 1-5*. Berlin, Germany: Springer-Verlag Berlin Heidelberg; 2013.
34. Póczos B, Xiong L, Schneider J. Nonparametric divergence estimation with applications to machine learning on distributions. 2012. arXiv preprint. arXiv:1202.3758.
35. Loftsgaarden DO, Quesenberry CP. A nonparametric estimate of a multivariate density function. *Ann Math Stat*. 1965;36(3):1049-1051.
36. Póczos B, Xiong L, Sutherland DJ, Schneider J. Support distribution machines. Technical Report. Pittsburgh, PA: Carnegie Mellon University; 2012.
37. John Hopkins Turbulence Database (JHTDB). Johns Hopkins website. http://turbulence.pha.jhu.edu. Accessed September 1, 2017.
38. OpenFOAM website. http://www.openfoam.com. Accessed October 5, 2017.
39. Greenshields CJ. OpenFOAM User Guide. OpenFOAM Foundation Ltd, version 3(1). 2015.
40. Bruneau C-H, Saad M. The 2D lid-driven cavity problem revisited. *Comput Fluids*. 2006;35(3):326-348.
41. Kaufman L, Rousseeuw PJ. *Finding Groups in Data: An Introduction to Cluster Analysis*. Hoboken, NJ: John Wiley & Sons; 2009.
42. Karst at Indiana University. Indiana University website. https://kb.iu.edu/d/bezu. Accessed September 1, 2017.
43. Forced isotropic turbulence. Johns Hopkins Turbulence Database (JHTDB). John Hopkins website. http://turbulence.pha.jhu.edu/datasets.aspx. Accessed September 1, 2017.

How to cite this article: Li F, Song F. Building a scientific workflow framework to enable real-time machine learning and visualization. *Concurrency Computat Pract Exper*. 2018;e4703. https://doi.org/10.1002/cpe.4703