This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3091387, IEEE Transactions on Cloud Computing

Interactive Visualization of Large Turbulent Flow as a Cloud Service

Tanner Hobson, James Hammer, Preston Provins, and Jian Huang

Abstract—Many scientific communities today have community datasets that are continuously created, curated, and maintained for community use. Such datasets are often hosted and shared through cloud-based data repositories. In this work, we propose a lightweight and affordable visualization cloud service that can be deployed as a companion service of a community dataset. Our target visualization use case is parallel flow visualization, which is crucial for understanding planet-scale phenomena such as the Earth's atmosphere and ocean. As a core research topic of scientific visualization, parallel flow visualization typically uses HPC computing platforms. It is complex to implement with scalability, deploy with efficiency, and is often considered an advanced form of scientific visualization. Because of the heterogeneous nature of cloud platforms, in this work, we use a swarm-based parallel design to replace traditional HPC designs that assume homogeneity and rely upon conventional methods such as Message Passing Interface (MPI). This design enables interactive visualization of large flow fields in a way that is lightweight, efficient and easily deployable as a cloud service. We demonstrate our proposed system using NOAA's NCEP ensemble data, which captures turbulent planet-scale atmospheric flows in observed forms, as well as in forecast forms for varying time scales. We evaluate the performance and efficacies of our system on Amazon Web Services (AWS) for three use cases, where remote users can use their laptops to (i) interactively explore global atmospheric flow patterns in general, (ii) to specifically compare how a forecast is different from the observation, and (iii) to explore flow patterns in a typical information visualization dashboard.

Index Terms—Cloud, cloud computing, Amazon AWS, parallel flow visualization, scalability, interactivity

1 Introduction

Scientists organize their research around data. When a research community starts to continuously accumulate, curate and share community datasets, community data repositories become a catalyst of future research. They are also a powerful source to engage the public, make science relevant, and deepen societal impact of science. Such continuously growing and open datasets are precious assets of the whole world. As an example, the data used in this work is the Climate Forecast System (CFS) ensemble data repository [52] from NOAA National Centers for Environmental Prediction (NOAA NCEP) that captures annual global atmospheric patterns at spatial 0.5×0.5 degree precision and a temporal resolution of 6 hours.

While cloud has already become the leading solution for creating distributed systems to support large data repositories, in this work, we propose to extend cloud-based functionalities of a data repository beyond data services. Our primary proposal suggests interactive visualization services can become a component of cloud-based data repositories too.

Such a broadened scope of data repositories can help lower many barriers of adoption by diverse user communities. For example, researchers can accelerate their work by being able to always see the latest data to test and refine their hypotheses without needing to maintain an up-to-date local replica of the entire dataset. They can also collaborate with more people, by being able to share their findings with others who do not have, or cannot afford to have, an entire local copy of the data.

Recently, interactive volume visualization as a service has been shown to be feasible [49], [50]. In this work, we focus on creating interactive parallel flow visualization as a service, because flow visualization is important to many disciplines, including atmosphere, ocean, fusion, petroleum, aerodynamics, and cardiovascular biophysics, where "seeing" the flow is often the first step of scientific research.

In addition, interactive parallel flow visualization offers a unique opportunity to study how to use heterogeneous cloud resources to achieve consistent parallel accelerations in support of synchronous user interactions. Cloud resources are a promising alternative to traditional HPC computing and visualization resources that require scientists to have a working relationship with supercomputing centers. Due to this reason, our work focuses on using cloud platforms to make leading-edge scientific datasets interactively usable at an incremental cost.

Our prototype system is called Visualization Cloud Instances (VCIs). VCIs work collaboratively as a self-organizing swarm for parallel computing. Each swarm appears as a single cloud service, i.e. a VCI Service, which can be used locally on an institutional cluster, or remotely on a public cloud such as Amazon AWS. Our results show that the VCI approach is able to support large flow data and ensembles of data and still maintain crucial cloud-based characteristics: (i) built for large flow data and ensembles of flow data; (ii) achieves fast interactivity; (iii) instantaneously available; (iv) serves multiple users concurrently; (v) serves users locally and remotely; (vi) supports a variety of user devices, and (vii) lightweight to integrate into applications.

Desktop applications can use VCI Service through a JavaScript library, *vci.js*, which transparently manages parallelism, performance, and fault-tolerance. By hosting NOAA NCEP CFS data in the cloud, we show (1) an application that can provide interactive visualization of 3D global atmospheric flow field to students (Figure 1); (2) a comparative visualization for scientists to analyze deviations between forecast vs. observed ground truth (Figure 8); and (3) a public-awareness application that integrates a VCI Service

• *All the authors are with the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, 37996. E-mail: {thobson2,jhammer3,pprovins}@vols.utk.edu, huangj@utk.edu*

2168-7161 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF TENNESSEE LIBRARIES. Downloaded on July 11,2021 at 18:42:46 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3091387, IEEE Transactions on Cloud Computing

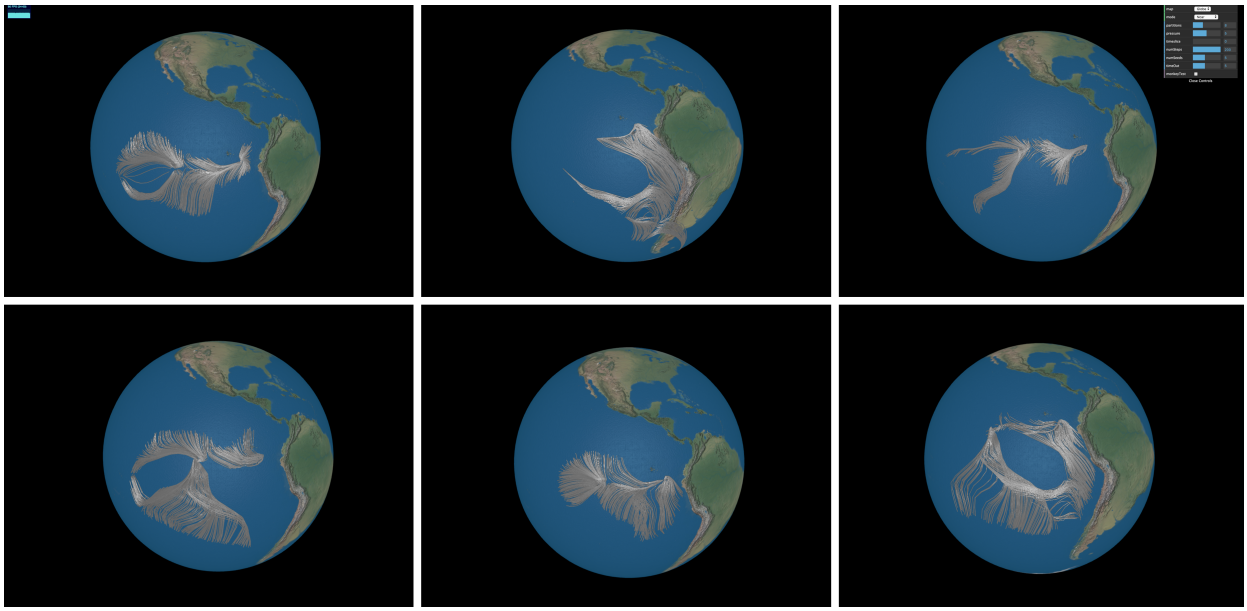


Fig. 1: Example results using a web browser to interact with parallel flow visualization service supported by a self-organizing swarm of cloud instances. The system can support multiple concurrent users. Each subfigure shows 1,000 particle traces of 200 time-steps each. They are extracted interactively from an observational dataset of global 3D atmospheric flow for the year of 2012; a total of 150 GB with 1463 time-steps at 0.5 degree geo-precision and 6-hour time-precision. The data is from the NOAA NCEP CFS repository. Exploration is instantaneously available in an on-demand manner. The service can use dedicated machines as well as public cloud platforms. When hosted on-demand on Amazon Web Services (AWS), for example, the total cost is personally affordable at less than $1/hour.

into the popular D3 [1] library. This last example enables any citizen to investigate how pollution from nuclear power plants in the United States, on any particular day in the year, can impact the entire globe (Figure 9).

All applications use a year-long observation dataset of the Earth's 3D atmospheric flow (1,463 timesteps, $720 \times 361 \times 36$ spatial resolution, 150 GB) from the NCEP CFS repository [52]. The second application adds a corresponding forecast dataset of the same dimensionality from the same CFS repository. In all cases, the AWS setup required is less than $0.70/hour.

Our results suggest cloud services, like the VCI Service, can conduct parallel computing using heterogeneous resources and support flexible, interactive, general use of large datasets on a desktop. These interactive use cases, coupled with efforts to quantify the exact cost-performance benefits of the cloud [22], expand the application potential of cloud hosted data resources.

All software of VCI will be open-sourced upon publication of the paper. The remainder of this paper is organized as background in Section 2, system architecture in Section 3, and application development in Section 4. We show results in Section 5 and discuss conclusion, and future works in Section 6.

2 Related Work

2.1 Web- vs. Cloud-Based Scientific Visualization

In order to handle large datasets, visualization applications commonly use the client-server architecture; well-known examples include Paraview [6], VisIt [18], and ViSUS [44]. As the web browser has become universal and has displaced native applications, many systems (e.g. ParaView Web [31], the GeoVizCloud [66], and the Arctic Viewer [2]) now support web browsers clients and can handle large datasets. Similar efforts are increasing in domain science communities too [14], [36]. Meanwhile, there are also significant successes in transitioning the server-side of workflow systems to web services, examples include Giovanni [9], Pathomx [24], Pegasus [19], and others [62].

While scientific visualization in a web-based form is more commonly available now, cloud-based scientific visualization is still less common and there are some key distinctions to notice.

First, server-side heterogeneity. To date, parallel visualization methods tend to have roots in HPC and often use Message Passing Interface (MPI), where it's assumed that the computing resources in use are homogeneous. On a symmetrically configured platform, this assumption is valid as long as the machine in use is not shared. When there is heterogeneity, either because the platform consists of different kinds of resources or because the platform is in shared use, performance of traditional parallel visualization packages can suffer. In addition, tightly coupled HPC applications that require significant global communication have been shown to perform relatively poorly on Amazon EC2, and variability introduced due to how virtualized environments are shared is also a challenge to traditional load-balancing methods in HPC [28]. In sum, even though HPC-based methods can be hosted using the cloud, systems based on HPC-based methods are not sharable and fault-tolerant as typically expected in cloud services.

Second, stateless server-side. Cloud-based designs can improve scalability by decoupling server-side components from client-side user logic, and make the server-side stateless. In this manner, cloud-based designs are inherently for "multi-user" scenarios, even for situations where there is a single human user. By breaking apart the traditional monolithic designs, synchronous communication can be replaced by asynchronous communication. As a result, cloud-based designs can process many concurrent user-requests

2168-7161 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF TENNESSEE LIBRARIES. Downloaded on July 11,2021 at 18:42:46 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3091387, IEEE Transactions on Cloud Computing

in parallel. These concurrent user-requests can be from the same user or many simultaneous users. The client-side application needs to partition a large user-request into smaller simultaneous user-requests. The partition can be done transparently by a client-side API or an application library, however. By leveraging browser automatic request-resend such cloud-based designs can achieve fault-tolerance, acceleration, and scalability at a lower cost.

In sum, even through web-based scientific visualization applications that are monolithic can work remotely for selected users, those applications cannot achieve the level of cost scalability and instant availability for large-scale user bases as expected in Cloud Computing. To this end, whether the server-side uses virtualized containers (e.g. Docker [3] and Singularity [34]) is not a deciding factor of whether a visualization application is cloud-based, it is instead whether the design assumptions go beyond homogeneity and monolithic application architectures.

Tapestry supports interactive volume visualization as a cloud-based microservice [49], [50]. The design assumed heterogeneity and decoupled application architectures. Of course, parallel flow visualization is much harder than volume visualization because of the nondeterministic runtime workloads. To the best of our knowledge, no previous works have attempted using independent cloud instances to provide interactive parallel flow visualization.

2.2 Parallel Flow Visualization

Flow visualization starts by placing seeds in the flow and extracts flow lines by tracing the advection of the seeds through the flow field over time. Seeding is important because of the need to reduce visual clutter and to reduce the computation load.

Batch-Mode Flow Advection. Due to the difficulty to achieve interactivity, many previous researchers have studied seeding strategies in a batch-processing context. For instance, for static flow fields, researchers have developed methods based on critical points [56], information theory [60], topology and distancing information [37], [48], [59], [67], geometry clustering [55], or specific domain-science hypotheses, such as flux [12], [58]. For turbulent flow fields, which play a much bigger role in leading-edge science, there are few proven seeding strategies so far. In a batch-processing mode, it's not uncommon to pre-compute a dense set of flow lines and then down-select a set of flow lines, for example, according to geometric similarity measures [38].

Parallel Processing. Parallel flow tracing in large flow fields requires both data-parallel and task-parallel. Coupled with evolving hardware architectures and application software architectures, many creative and successful solutions have been proposed to scale batch-mode parallel particle tracing to 16K or 32K processors and beyond [33], [45]. The field has also explored new processor architectures [16] as well as new memory architectures [15].

From an algorithmic perspective, mapping computing tasks onto a parallel architecture is a core problem of parallel processing [8]. By nature, flow line advection has an unpredictable data access pattern, which the makes parallel particle tracing hard and attracted a plethora of past research. The following are just a subset of the most notable recent works on this topic: data- and task-partitioning strategies [10], [45], [47], [61], [65], dynamic load balancing [42], [64], runtime job management [17], [40], runtime data management [30], and coupling advection together with analysis [26], [33],

Parallel speedup is the result of algorithmic improvements where the typical optimization criteria is parallel speedup. However, the necessary tuning process can be system dependent, application dependent, and workload dependent. The corresponding process for each application can be lengthy.

Instead of focusing solely on parallel speedup, we use a different set of optimization criteria in this work. First, to ensure that the parallel computing and communication time for extracting flow lines from turbulent flows are below 100 milliseconds so that users can sustain front-end interactivity. This metric is consistent with that of existing remote visualization systems [32]. Front-end rendering rates need to be above 30 frames per second. Second, to ensure that many concurrent user-requests can self-balance through Docker's load balancer and within our VCI swarm. Third, to minimize the runtime footprint so that the server-side parallel computing incurs low costs (e.g. $1/hr), even though the entire turbulent flow field data are 100s of GBs in size. These are evaluated in detail in Section 5 (Results).

2.3 Aspects of Interactivity

Interactive flow visualization is particularly valuable for scientific exploration. Regardless of how large their data is, scientists wish to place seeds interactively, see the flow geometry immediately, control the visualization intuitively, and navigate flexibly. They also wish to share what they see with their cross-disciplinary teams so that their teams can explore interactively too.

Interactivity. According to the interactive analytics framework from [23], the "interactivity" of interactive particle tracing would be incomplete until users have an on-the-fly ability to modify data transformation (i.e. seeding flow geometry extraction), visual mappings (i.e. rendering methods and parameters), and view transformations (i.e. spatiotemporal navigation) at the same time.

Accessibility. Particle tracing in large datasets often depends on HPC systems that require reservation. Unfortunately, such use is only available to selected scientists working on pre-approved projects by the administrators of those systems. In contrast, immediate on-demand accessibility of cloud systems provides a new avenue for boosting productivity and flexibility.

Portable Interactivity. Parallel flow visualization is one of the most difficult to make interactive, due to complexities in data, I/O, flow line advection, parallelism, load balancing, data management, and rendering. The performance optimizations are not easy to generalize across platforms. When data is large, ensuring a high level of performance that is portable across platforms is hard.

Cost Effective Reproducibility. Reproducibility of scientific results is now a priority. In the reproducibility spectrum introduced for computational sciences, it suggests that publications should come with executable code and data to meet the "gold standard" [5], [7]. However, due to how interactivity is important for parallel visualization of turbulent flows, reproducing the same computing environment may need to be required because of performance requirements. But that option is unaffordable to most. In this regard, interactive visualization services as a part of community data repositories can provide a better and more sustainable solution.

3 System Architecture

3.1 Design Overview

When designing VCI, we separated the complexities into three categories: (i) those due to front-end interaction needs by domain scientists, (ii) those due to back-end parallel computing, and (iii) those required to bridge the front-end and the back-end.

2168-7161 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF TENNESSEE LIBRARIES. Downloaded on July 11,2021 at 18:42:46 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3091387, IEEE Transactions on Cloud Computing

Accordingly, we consider three separated spaces: the application space, the system library space, and the swarm space. The three spaces are illustrated in Figure 2.

In this section, we discuss front-end and back-end separation, communication mechanisms, and related design decisions to make VCI efficient. At the core of VCI's design is its use of Docker Swarm. Accordingly, all references to "swarm" in this section refer to Docker Swarm which is characterized by its ability to dynamically increase or decrease the number of running services and transparently load balance between them. We discuss these more in Section 3.2.

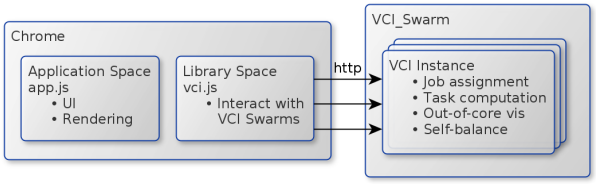


Fig. 2: System Diagram

Front-End. Application logic is in the application space, where the focus is user interaction and rendering. The system library space is concerned with accessing and managing interactions between the application space and larger-scale computing resources in the swarm space. Even though we use JavaScript herein as the target language of the front-end, the separation of these spaces can be equally applicable to C/C++ or Python front-ends.

Back-End. The computing sources in VCI is a self-organizing swarm. Each instance inside the swarm is a Docker [3] instance. The notion of swarm is to highlight that there is minimum "cluster-level" orchestration, which makes the swarm model a more natural fit with cloud platforms like AWS. To this end, we should also note a swarm can just as easily run on a user's many-core workstations or small-scale clusters. We show results for both situations.

Communication. VCI has two distinct modes of communication. The first is for transient connections that need to be opened and closed on demand, which is primarily used within the swarm. These transient connections are implemented through HTTP. The second mode of communication is for persistent connections primarily between front-end and back-end, so that many requests can be made simultaneously. These persistent connections are through WebSocket. After a VCI Service receives requests through WebSocket, they are transparently transformed into HTTP requests that the swarm uses internally.

Lightweight System Design. Both client- and server-side designs are kept minimal to operate alongside other compute resources. In Chrome, the entire memory footprint, combining the application and library spaces, of an on-demand flow visualization application with functionality as in Figure 1 is less than 10 MB. The server-side swarm's data management uses a thrifty out-of-core scheme to lower memory footprint, typically using only 50 to 100MBs of memory in total. This greatly increases the overall system's portability.

Always Parallel Design. Even when there is a single front-end using a back-end swarm, the operation of the front-end and the back-end are both parallel. To this end, instead of treating each user interaction as a request to be answered in a step-locked synchronous cycle, VCI treats user interactions as a continuous stream of asynchronous requests.

When a user moves the mouse on the globe (Figure 1), particle tracing requests are sent by Chrome to the back-end swarm as HTTP requests immediately and continuously. As Chrome continues to manage all outstanding requests transparently, the requests received by the swarm are distributed to VCI instances, which work together in parallel. During the process of extracting traces, incremental results are sent back to users. In wide area tests, the time to start receiving traces is under 0.1 seconds. *vci.js* running inside Chrome transparently manages the receiving of the parallel and continuous streams of extracted traces. Hence, although the application space makes a single-user assumption, the user always benefits from the multi-user assumption in the swarm.

Development Challenges. The parallelism inside of VCI offers benefits while it also presents several challenges. First, it is non-trivial to capture the global state of the swarm. For this reason, VCI uses distributed event logging in order to precisely record the state of individual instances as requests go through the system. Second, the dynamic nature of requests going through the system, and the related stochastic characteristics, makes it hard to exactly reproduce an exact flow of requests and events at runtime. Third, the same randomness in the system can introduce significant noise when benchmarking system performance.

3.2 Swarms of VCI Instances

Figure 3 shows an overview of the swarm architecture. The number of nodes and containers can vary as needed, even at runtime. Docker Swarm's manager receives and distributes incoming requests and is required on conventional computing clusters, but not required on public clouds like Amazon AWS, where the AWS load balancers serve the same purpose.

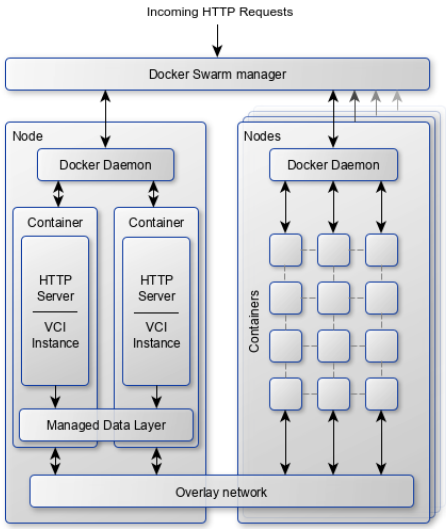


Fig. 3: Overview of an VCI Swarm.

Each computing node runs a Docker daemon process, which interfaces with the Docker Swarm manager. There can be a variable and configurable number of Docker containers on each node. Each Docker container is a light-weight, fast to spin up, and self-sufficient virtual machine. Since the concept of "node" is virtualized on public clouds, the common hierarchy of Swarm-Node-Instance can be compressed to simply Swarm-Instance, although this difference is negligible for software development.

We refer to each instance as a Visualization Cloud Instance (VCI) and the entire Docker Swarm as a VCI Swarm. All VCI instances are identical. Each instance is a fully independent entity. HTTP is the only communication protocol used by VCI instances, regardless of communicating within or outside the swarm.

2168-7161 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF TENNESSEE LIBRARIES. Downloaded on July 11,2021 at 18:42:46 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3091387, IEEE Transactions on Cloud Computing

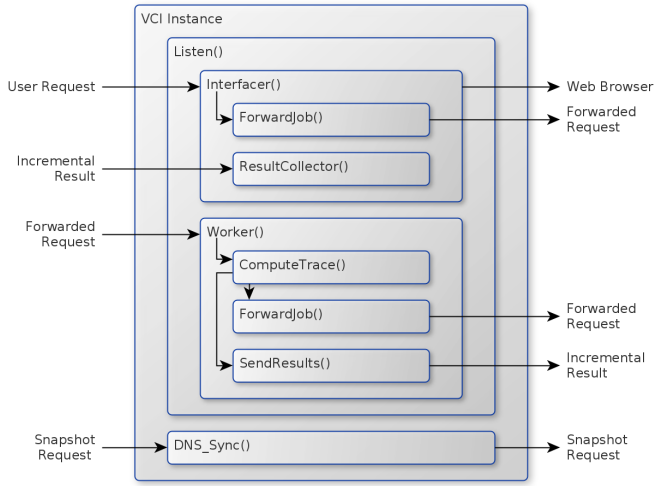


Fig. 4: The main components of a VCI instance.

Every VCI instance runs its own HTTP server, responsible for receiving and queuing incoming requests. The swarm manager manages an overlay network between nodes. VCI instances on this network are able to freely communicate with one another. The swarm manager handles all DNS and routing within this network, which is not exposed externally.

The choice of HTTP as the sole communication protocol is to leverage the proven multi-threaded solution within modern web servers that can reliably receive, queue, and managed large amounts of concurrent requests. To our knowledge, few parallel visualization systems have similarly efficient, robust, and high throughput asynchronous communication capabilities.

The swarm manager receives and distributes incoming requests, but in no other way orchestrates parallel computation. The VCI instances self-organize to work in parallel. The only global synchronization is snapshotting swarm-wide workloads, so that instances can each adjust their own self-balancing scheme accordingly. The snapshotting operation takes place every 5 seconds.

The VCI Swarm has some resemblance to stream processing in how collaborating processing threads are used, because the VCI Swarm also routes requests through a network of operators to construct results. The main difference is that VCI swarm works on stored data, as opposed to streaming data; hence, key requirements for stream processing do not apply to VCI, i.e. query mechanisms and the need to keep data moving [54]. In addition, VCI Swarm is lightweight with a much smaller functional scope, assumes only standard Linux process management, without dependence on additional scheduling frameworks or resource managers. Lastly, in comparison to well-known stream processing systems [4], [25], [41], [63], VCI swarms run on resources that are minuscule.

3.3 VCI Instance

3.3.1 Inherently Threaded Design

The HTTP web server run by each VCI instance manages all computation and communication tasks of the instance. In essence, we chose to use a collection of web server processes for parallel computing. This design decision has two architectural reasons.

First, when parallel particle tracing is both data- and task-parallel, there are many disparate yet collaborative tasks to be managed [17], [30], [33], [40], [45], [47]. Although it helps to dedicate specialized threads for each specific function, managing a large number of threads scalably is non-trivial. The high-throughput thread management used in web servers can be reused as an efficient and reliable solution to this need.

Second, when using containers, a universal interface of collaboration is HTTP. In this case, there is a performance advantage, as well as a portability advantage, if parallel computing task can be mapped onto HTTP web server model directly.

For VCI, we have developed an HTTP compliant web server in Python using a high-performance kernel in C. This design is easily adaptable for other HPC programming languages. We note two key details regarding threading mechanism here.

Threading vs. Forking. We chose threading server over TCP forking server for reasons of latency and shared memory address space. In particular, I/O is a fundamental challenge of all large-data visualization systems. In a data-parallel manner, each VCI instance is responsible for its assigned data partitions. We desire that the data loading as well as resident-memory management parts of the instance can be shared by all threads. This way, a single out-of-core visualization implementation can minimize I/O operations as well as memory footprint for an entire instance.

Our VCI instance is a derived class from the Python `ThreadingHTTPServer`, which spawns new threads for each request using `ThreadingMixIn` internally. Through evaluation many potential designs, this way of implementation offers the best efficiency, both for thread spawning and thread joins.

3.3.2 Specialized Threads

Our design philosophy comes from the domain of Unix: That is, to enforce rigorously that each thread is specialized in one type of task, and to ensure that threads can easily collaborate.

Figure 4 shows the internal parts of a VCI instance, where every box is a type of thread and arrows are HTTP requests being sent or received. These threads are based on `ThreadingMixIn` derived from Python's core `socketserver`. As threads have different functions, they have different lifespans too.

Listen(): this thread is up for the entire lifespan of the VCI instance. It runs continuously at all time and listens for new requests on the open socket. It does not perform any real task other than to determine the type of request and spawn off appropriate specialized threads to handle the requests. In particular, Listen() thread spawns off Interfacer() threads and Worker() threads.

DNS_Sync(): this thread is up for the entire lifespan of the VCI instance. It runs periodically to query the Docker swarm manager or the AWS load balancer to get an up-to-date list of instances as well as their load information. DNS_Sync() also provides the swarm membership information to any threads upon request.

Interfacer(): An Interfacer() thread is created by the Listen() thread when a client HTTP request is received. No Interfacer() threads have overlaps because they each serve a different client request. Requests from the same user are treated as independent requests and are handled by separate Interfacer() threads.

Interfacer() threads have the lifespan of a client request. When a user cancels a client request, for example, by closing their web browser, the corresponding Interfacer() thread is terminated.

Worker(): Worker() threads are created by the Listen() thread when an HTTP request is received from an Interfacer() thread. A Worker() spawns off a ComputeTrace() thread to perform the computation. As incremental computation results are ready, a SendResults() thread is spawned to send them to the ResultCollector(). Especially for flow line advection, the advection may not have reached the targeted length requirement if the flow line exits

2168-7161 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF TENNESSEE LIBRARIES. Downloaded on July 11,2021 at 18:42:46 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3091387, IEEE Transactions on Cloud Computing

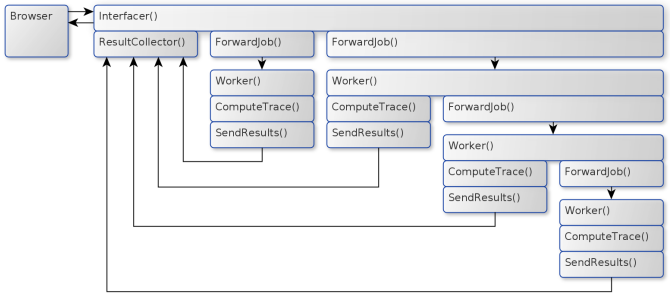


Fig. 5: Workflow tree illustrating how a user request is handled. The verticality of each box represents serial execution while boxes horizontal of each other are executed in parallel.

the assigned partition. When this happens, a ForwardJob() thread is created so computation can resume on the target instance.

3.3.3 Parallel Processing

As many-core processors become mainstream, processing power is getting condensed into smaller and smaller physical footprints. Not only are workstations with 48+ vCPUs very affordable, nowadays a cloud service with even 100s of processors can be affordably available on-demand at less than $10/hour.

Work Assignment and Data Partition. As with common practice, we partition data along the 4 spatiotemporal dimensions in a pre-processing step. We apply 5-way partitioning with 4 voxel ghost regions resulting in 625 partitions. All partitions are on the same network mounted storage and are loaded by VCI instances at runtime as needed through memory mapping.

When a VCI instance is started, a set of partitions are assigned to that instance. These are *primary assignments*. As typical in fault-tolerant systems, we allow replication (i.e. k-replica), where k can be 1, 2, or 4. When k is larger than 1, additional partitions are assigned to each instance as *secondary assignments*.

Both primary and secondary assignments remain the same throughout the lifespan of the whole swarm. The mapping between partitions and instances uses a computable data assignment. This hash is determined by two parameters (the number of VCI instances and the number of partitions) and is based on a simple deterministic round robin process yielding a mapping between instances and partitions. Although there is not an explicit predetermined partition assignment, each instance is able to compute the same list at runtime without any extra communication.

As particle tracing is computed, some particles may exit the assigned region for an instance. When this happens, that instance uses the data assignment mapping to identify candidate instances to continue the trace. These candidates are filtered based on load estimates (Section 3.4.3) and the particle is forwarded.

Workflow Tree. Through system-level routing (Section 3.4), a VCI instance is chosen at random as the main responder. The Listener() on that instance spawns off a dedicated Interfacer() thread for that request, which appears as the root of a workflow tree, shown in Figure 5. These threads may spawn other threads either within-instance (stacking) or cross-instance (arrows).

The Interfacer() thread will spawn off ForwardJob() threads, one for each of the target VCI instances, because they are assigned jobs according to swarm-level partitioning (Section 3.4). Each target VCI's Listener() receives the request and spawns off a dedicated Worker() thread to handle the computation.

A Worker() thread spawns a ComputeTrace() thread to handle the job. This thread always creates a SendResults() thread to send

```python
from requests import post
from concurrent.futures import ThreadPoolExecutor, wait

hosts = ['http://1.2.3.4:8840', 'http://1.2.3.5:8840']
seeds = [(0.0, 0.0, 50.0, 0.0), (80.0, 40.0, 50.0, 0.0)]
executor = ThreadPoolExecutor(max_workers=len(hosts))
futures = []
for seed, host in zip(seeds, hosts):
    kwargs = { 'url': host + '/trace/',
               'json': { 'seeds': [seed] } }
    future = executor.submit(post, kwargs=kwargs)
    futures.append(future)

done, _ = wait(futures)
for future in done:
    future.result()
```

Listing 1: VCI's server code for creating and managing runtime threads using Python's `ThreadPoolExecutor`.

```c
void rk4(float *seed, float *newseed) { /* omitted */ }

int trace(size_t nseeds, float *seeds,
          size_t nsteps, float *trace) {
  #pragma omp parallel for
  for (size_t i=0; i<nseeds; ++i) {
    memcpy(OUTPUT(trace, i, 0), SEED(seeds, i), 16);
    for (size_t j=1; j<nsteps; ++j) {
      rk4(OUTPUT(trace, i, j-1), OUTPUT(trace, i, j));
    }
  }
}
```

Listing 2: The particle tracing kernel with OpenMP parallelization.

the newly computed incremental results. A ForwardJob() thread may also be created if the trace is not yet complete so the other instance can compute the rest of the trace.

Threads Spawning. The workflow tree depth depends on the expected length of flow lines, the number and distribution of the seeds, and how data partitions are assigned. To spawn many threads efficiently, we use Python's `ThreadPoolExecutor` as shown in Listing 1. The ForwardJob() thread uses a thread pool to distribute forwared jobs to all of the VCI instances in parallel.

UUID. When Interfacer() threads generate jobs, we avoid collision between job identifiers using universally unique identifier (UUID), a proven concept in distributed systems and databases [35]. We use Python's `uuid.uuid4` function to generate UUIDs as 32 byte random strings, seeded with the process start time.

Working Set Minimization. Since cloud architectures incur much lower costs when instance sizes are small, it is beneficial to minimize the in-core memory overheads. Hence, during processing, we design each instance to only load as small a spatial-temporal partition in the dataset as possible. To this end, in order to maintain generality, we decided to manage such data access patterns on the granularity of memory pages. This is done through memory mapping mechanisms provided by all modern Unix-flavored operating systems. As a result, rarely used pages get swapped back to disk while commonly accessed ones stay loaded.

We use runtime communication, i.e. request forwarding across VCI instances, to ensure that each instance focuses on small partitions; whereas collectively the instances have a full coverage of the entire spatio-temporal domain with as small an overlap as possible. This is managed by having primary and secondary assignments for each partition, as shown in Figure 6. This reduces total in-core memory needs to 10's of GBs even when the full turbulent flow dataset amounts to 100's of GBs.

Scatter-Gather Design Pattern All communication in a VCI swarm follows the scatter-gather pattern and happens point-to-

2168-7161 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF TENNESSEE LIBRARIES. Downloaded on July 11,2021 at 18:42:46 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3091387, IEEE Transactions on Cloud Computing

point without using explicit collective communication primitives like barriers. The request forwarding mechanisms scatter requests through the system as necessary and then the responses are gathered. Additionally, all instances poll their DNS_Sync() thread periodically for updates in the swarm-wide membership information, including whether there are newly added or removed instances.

3.3.4 Worker Thread Lifecycle and Results Streaming

Computational Kernel. The computation kernel of the Worker() threads is implemented in C to be usable from Python. The main computation task is flow line advection, using 4th-order adaptive size Runge-Kutta, modified from Numerical Recipes [46] to be thread-safe. We further accelerate the kernel using OpenMP as in Listing 2 to process multiple traces at once.

Interfacer() Heartbeat. A unique design requirement for VCI swarm is fault-tolerance in relation to the client's status. A client can cancel outstanding requests in several ways, such as through new interactions or through closing the web page.

The Interfacer() thread is the root of the workflow tree (Figure 5). Cancelled visualization requests trigger the Interfacer() thread and all of its within-instance threads to be terminated.

Across other instances, the corresponding Worker() threads need to be terminated too. This is done through the mechanism of a heartbeat signal. Specifically, while a Worker() thread operates, the Worker() thread checks the heartbeat of the Interfacer() thread.

The heartbeat information is collected by making an HTTP request to the ResultCollector() thread before running the low-level kernel. With this, the Worker() thread can detect that a user request is no longer valid and prunes the workflow tree accordingly.

Results Streaming and Termination. Two threads are involved in continuous results streaming: ResultCollector() collects partial results and Interfacer() sends partial results back to the requester. The former maintains a key-value store of all jobs created by the Interfacer() and tracks each job's status by their UUIDs. As ResultCollector() receives computed results from various Worker() threads, each job's status is updated, either marked as complete due to a termination condition being met, or marked as on-going. The termination conditions can include reaching the targeted length of flow line, or the flow line exiting the domain of the dataset.

The Interfacer() is the only thread that serves data back to the requester, and does so as soon as partial results are available. The application space, i.e. app.js, is the only place where the geometry of flow lines are assembled based on the UUIDs.

The ResultCollector() tracks per job completion status until computation for the whole request completes. The Interfacer() manages the start of the workflow and sends results with the UUIDs back to the requester. These two threads share a mechanism of double-buffering: one buffer for ResultCollector() in write-only mode, and the other for Interfacer() in read-only mode.

When request processing is completed, the ResultCollector() thread initiates the termination of the entire Interfacer() thread. We note that, in the normal scenario, no worker thread should be still alive serving that request. In other words, the entire workflow tree for that request should have only the Interfacer() thread, ie. the root node of the tree, remaining alive.

3.4 Self-Organizing Swarm

3.4.1 Distributed Request Management

We distributed request management in three tiers. First, the Docker Swarm manager, or the AWS load balancer, randomly assigns incoming user HTTP requests to a different VCI instance. Second, the Interfacer() thread created by the assigned VCI instance's Listen() thread polls the DNS_Sync() thread on the same VCI instance to get the most recent snapshot of system's operating status. The Interfacer() creates the jobs and decides how to optimize the mapping between jobs and all VCI instances. Third, individual instances make job distribution decisions in ForwardJob() threads. Each of these decisions seek to distribute system load as evenly as possible, as illustrated in Figure 6, each VCI instance can be assigned primary and secondary partitions.

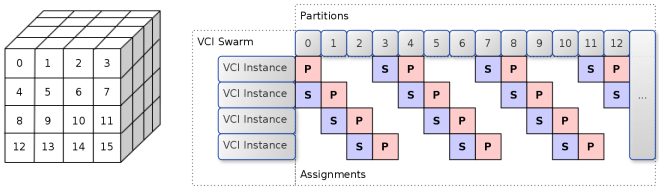


Fig. 6: Round-robin partition assignment with $k = 2$ replication. (Red P) are primary partitions; (Blue S) are secondary partitions. Each VCI instance volunteers to cover its secondary partitions only when the corresponding primary instances are under load.

Primary Partition. Upon initialization, each VCI instance is assigned a preprocessed partition as its primary partition. Altogether, the primary partitions collectively form a complete coverage of the entire spatial domain. There are no overlaps between primary partitions. The problem domain is partitioned by a grid pattern, each instance takes one cell from the grid.

Secondary Partition. In addition to the primary partition, instances may also be assigned other partitions to be used when at increased load. These partitions cover the same area as the primary ones, essentially forming another complete covering of the problem domain. When a job is forwarding to an instance based on its secondary partition, it is handled the same as it would be for its primary partitions. Secondary partitions are only used when the k-replicas (Section 3.3.3) is larger than 1.

Flexible Assignment. Data partitions are not pre-distributed to any VCI instance. Instead, the partitions are centrally stored and only mounted into an instance when the instance is created. The instances use OS-level memory mapping to access the data but do not pre-load anything until data is accessed. In this way, all VCI instances implement out-of-core visualization and use a very limited memory footprint.

3.4.2 Runtime System Snapshot

Periodically, for example every minute, DNS_Sync() queries the Docker Swarm manager or the AWS load balancer to get the latest information about swarm membership. After that, DNS_Sync() spawns off asynchronous threads to obtain and record the latest load information on all VCI instances.

When answering a status query from DNS_Sync(), each VCI instance reports the "ps" system load parameters, of which the CPU usage is the most important. This information is recorded in a per-instance hash, easily accessible by any function in the instance.

Thereby the DNS_Sync() is a specialized thread dedicated to keep track of how computing load is distributed among the entire swarm. Before any ForwardJob() threads make job distribution decisions, they query the latest snapshot from the DNS_Sync() thread so that their decisions adapt with variations in system load. Due to the polling rate of the DNS_Sync() thread, varying system

2168-7161 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF TENNESSEE LIBRARIES. Downloaded on July 11,2021 at 18:42:46 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3091387, IEEE Transactions on Cloud Computing

load within 5 seconds do not factor into forwarding decisions which helps smooth momentary bursts of increased load.

3.4.3 Adaptive Load Balancing

For each job request, the ForwardJob() thread uses the same procedure to decide which VCI instance should receive the job. During the assignment phase, we get a round-robin mapping between partitions and instances that are responsible for that region, sorted so that the primary host is before the secondary hosts.

To decide the best host, the sorted list is traversed and the first one matching all criteria is selected. For our load balancing, this criteria is a simple comparison between the CPU percentage and a pre-determined threshold. We use 70% so that a very overloaded host does not get more load assigned to it.

We don't have finite pre-defined workload, hence pre partitioning is hard. As a result, it is about work distribution rather than work stealing, because the workload/request is unknown. Each unit of work completes within milliseconds as shown in our results.

The random data access pattern of particle tracing causes considerable performance challenges. The proven solution is to trade communication complexities to ensure good locality of computing tasks. Fortunately, we benefit from the freedom offered by virtualized containers. That is, accessing data via memory-mapping does not involve explicit traditional I/O operations, and at the same time includes a reliable resident memory management capability. In this way, minimizing memory footprint and providing effective caching is one of the same, which achieves great results.

4 Application

As popularized by d3.js [13], Vega-Lite [53], and other toolkit libraries, interactive information visualization inside web browsers is now a commodity. These toolkits have transformed the standard of portable interactivity for information visualization by managing the non-trivial interaction paradigm transparently.

For scientific visualization, portable interactivity is meaningless without data scalability. In this work, VCI's swarm space addresses data scalability, the library and application spaces address portable interactivity. We have implemented *vci.js* for the library space, and *app.js* as an example of application space, respectively. The design intention is that *vci.js* is a general reusable library, and that *app.js* will vary from one application to another. The separation between back- and front-end spaces enables independence and autonomy between them.

4.1 vci.js and app.js

vci.js is a compact front-end JavaScript library that helps developers to easily interact with multiple VCI swarms. Let's motivate the need to consider multiple swarms through a few use scenarios.

First, even for a single user using a single dataset, the incremental cost of $1/hr per VCI swarm makes it easy for a user to afford multiple swarms, which has both fault-tolerance and performance benefits. Such horizontal scaling is trivial in the cloud setting but very hard in traditional settings.

Second, data-enabled science is gradually gaining momentum. Maintainers and curators of a community-centric data repository may wish to stand up a web service for their repository. For example, the forecast component and observational components of the CFS data are obtained through different means and are updated at different intervals. If these are kept as separate VCI services, scientists can choose which services to use flexibly.

Third, when it comes to reproducibility, it is common to expect that a user may need to compare a new result with results from previous works. Practically, it has been very hard to reproduce another researcher's computing environment. If published results can all have separate web services, which may be spun up at very low costs, such comparisons can be prompt, convenient, reproducible and sharable.

For these reasons, *vci.js* always assumes there are multiple VCI services being used at the same time and transparently manages flow-control and fault-tolerance of each request in parallel. *vci.js* is highly concurrent but presents itself as a single-function interface. The basic settings include (i) the URL to each VCI swarm, and (ii) a callback function to execute when a response is received from each swarm.

Developers can customize their policy settings anytime. The key parameters are: (i) how many services are used at the same time; (ii) how long a single request can take before being counted as a failure; (iii) how many retries a request can have before stopping re-sends; and (iv) how long a single request with retries should wait before giving up and not re-sending. The default policy setting in *vci.js* is geared towards high interactivity (i.e. low timeout and low maximum attempts)

All services registered with *vci.js* are tracked individually. Specifically, *vci.js* tracks outgoing request traffic of each service, and receive, along with each result, the overall load of each VCI swarm. Depending on the policy choices set by the application developers, *vci.js* may add or reduce loads on each VCI swarm selectively. In each response, the CPU load of the service handling the request is returned. vci.js selects the lowest loaded service to dispatch the following requests to. This method guarantees natural load balancing for any task type performed service-side.

Upon sending each request, a configurable timeout can abort the request. If a request is aborted, vci.js tries re-sending the request until the max timeout or max number of attempts has been reached, according to the customized policy. When configured in multi-host mode, once any response is received, all other identical requests to other hosts are aborted. In addition, each service is monitored for failed or aborted requests. Continually failing services are dropped from the hosts list and no longer utilized.

When application developers make use of the VCI Service, application logic needs to be separated into an *app.js*. In the following sections, we show three examples, each targeting a different kind of user and a different use case. Accordingly, each application has its own *app.js*, which contains the front-end application that performs rendering and UI functions. For example, in a smooth tracking mode, *app.js* tracks mouse locations upon every single move, automatically maps seeding points onto the global map, and calls *vci.js* accordingly.

4.2 Application: 3D Flow Visualization

This application targets students who may be unfamiliar with flow visualization. By offering an interactive way to seed the streamlines, it enables students to experiment and learn.

This application uses the JavaScript library THREE.js for creation, control, and rendering of a 3D scene graph using WebGL, and primarily maps data between the *vci.js* API and THREE.js.

Figure 7 shows the control panel of the application shown in Figure 1. The control panel includes temporal starting point of traces (i.e. "Seed Time"), the maximum length of flow lines (i.e. "nSteps/Trace"), the vertical height of seeding location (i.e. "Seed Pressure," measured in isolevels).

2168-7161 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF TENNESSEE LIBRARIES. Downloaded on July 11,2021 at 18:42:46 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3091387, IEEE Transactions on Cloud Computing

We also assume bundles of flow lines are more useful in turbulent flows. Hence, once a seed location is given, it is jittered by small random amounts (controlled by "nSeeds/Req") and sent together in a single VCI request. Obviously, we can reduce a bundle to a single flow line by setting "nSeeds/Req" to 1. In our tests, we set "nSeeds/Req" to 5. If *app.js* is handling 1,000 user requests/minute, 5,000 flow lines/minute are extracted.

Extracted flow line segments are received in the form of vertex arrays, which we then converted into stream tubes by using Catmull-Rom splines in order to allow for better illumination.

4.3 Application: Comparative Flow Visualization

Comparing models vs. observations, or models vs. models, is crucial for answering fundamental scientific questions. In climate research, such comparisons have led to a deeper understanding of climate extremes [21] and general atmosphere circulation [27]. Such comparisons are also indispensible in model validation [29] calibration [39], and scenario and sensitivity analyses [51].

As datasets become larger, such comparative studies are increasingly challenged by the data deluge, however. Interactive comparative visualization is becoming more difficult too.

Our design of the swarm-service model was in part motivated by this need. VCI makes it easy to use specialized swarms to manage each large dataset separately, while the swarms appear uniformly as standard web services. An analysis application can then have the feasibility and flexibility to adopt new datasets depending on needs, instead of depending on what a user's computer can manage.

To show this possibility, we compare how the NCEP forecast differs from the actual observations. To do so, *app.js* registers a second VCI web service with *vci.js*, and easily transitions to using two datasets at the same time. Each is 150GB in raw storage.

Figure 8 shows how the forecast model differs from actually observed wind velocity fields. The cool color shows forecast, specifically the `forecast3` data product of NCEP CFS. The warm color shows the observed data. The visualization shows that the model predictions have a more drastic shearing effect than the observed. The particle traces are seeded from January 1, 2012.

VCI's interactive visualizations in Figure 8 confirm global atmospheric circulation patterns overall. However, both the observed and the modeled patterns show unique variations on multiple levels. It is worthwhile for any scientific user to have such an interactive ability to investigate the scientific reasons behind model agreement as well as model divergence, which are general needs in all disciplines of computational science and engineering.

4.4 Application: D3 Flow Visualization

This application shows a possible integration with the popular JavaScript library, *D3.js* [13]. Its intended use case is for citizens

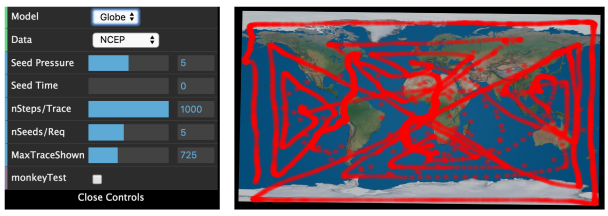


Fig. 7: Left: UI controls of the tracing app (Section 4.2). Right: Monkey testing request locations. Each test makes 3,481 requests in 1 minute at roughly a rate of 60 requests/second. Each request includes 5 seeds for particle tracing.

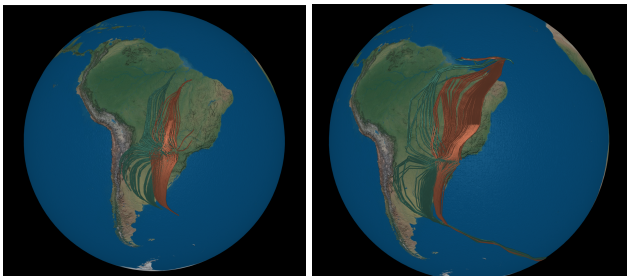


Fig. 8: Comparative visualization showing differences between actual observation vs. forecast atmospheric wind velocity patterns. Observation in cool color and forecast in warm color.

without any training in flow visualization to be able to pose questions and answer them in an explorable way. We used D3 for this application because it is the de-facto standard for information visualization targeting general audiences.

This application prioritizes data analysis with multiple linked views. The flow seeding locations reflect 60 nuclear power plants in the US. The first view is a relational Sankey diagram, which describes the relation between individual power plants and the respective states the flow have travelled through. The second view is a line chart showing the number of states affected by a single power plant over time. The third view is a map that shows potential flow emitted from the power plants. A calendar selector is included with the third view so that users can flexibly experiment with daily varying atmospheric flow patterns.

An example of the kind of insights enabled by this application is to examine the effect of a power plant disaster if it occurred on January 1st, 2012. In aggregate, we found that 60 power plants have a combined coverage of effect on 37 different states. If we instead focus on a single power plant in Michigan, then it alone is capable of affecting 9 different states in as little as 67 minutes. These analyses are very user directed and will depend on what exactly the user is interested in.

5 Results and Discussion

5.1 Overview

Test Dataset. We chose the NCEP CFS atmospheric community dataset [52] because it represents the leading edge of assimilating observational data together with model predictions. Our results are collected using a subset of this dataset that spans the entire year of 2012 at a 6-hour time interval (i.e. 1,463 time measurements), a

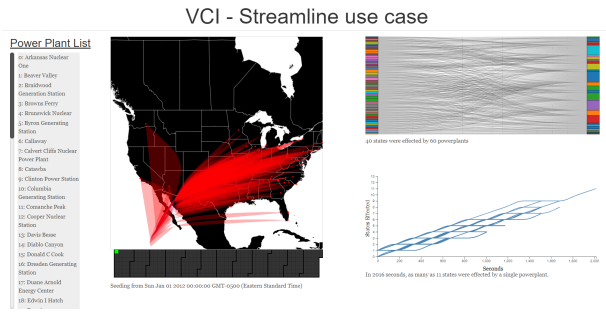


Fig. 9: D3 application showing flow emitted from US nuclear power plants on January 1st, 2012. Left: Power plant selection list. Middle: Map view. Middle Bottom: Calendar date selection widget. Top Right: Sankey relational view. Bottom Right: Line chart view.

2168-7161 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF TENNESSEE LIBRARIES. Downloaded on July 11,2021 at 18:42:46 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3091387, IEEE Transactions on Cloud Computing

global spatial resolution of $720 \times 361 \times 36$, along the dimensions of longitude, latitude, and pressure isolevels, respectively. This dataset has 3 attributes: latitudinal wind velocity, longitudinal wind velocity, and vertical wind velocity.

For most results in this work, we use a 150 GB piece of the CFS data which is based on the actually observed wind velocity. Results in Section 4.3 show interactive a comparative visualization between both the observed and the forecasted data. The additional forecasted component amounts to 150 GB of raw storage as well and has the same format as the observed data.

As a pre-computing step, data is partitioned, where each partition is $144 \times 73 \times 8$, for a total of 625 partitions. A ghost zone of 4 voxels has been added along each dimension to each partition because we use adaptive size 4th-order Runga Cutta to compute the flow advection. Each partition amounts to 679 MBs, whereas the partitioned CFS forecast dataset amounts to 417 GBs. The same applies to the observed data as well.

Interactive Test. Since this paper is concerned with portability, we test wide-area reliability via pre-recorded "monkey" tests, which have been recorded to emulate typical user interactions. This style of testing is effective at evaluating complex applications [57]. The test generates $3,481$ requests in a hand drawn path which traverses the globe over a period of 60 seconds. This test is run once for each configuration. Average request rate is about 60 requests/second. Each request includes 5 seeds for particle tracing. Data is collected on the back end utilizing the logs generated by VCI which catalog request across the distributed system.

The test path is shown in Figure 7-Right. On average, each flow line has a length of $1,548$ miles and covers a timespan of 67 minutes. Each also uses data from several partitions of the dataset: 59.9% use only one partition, 45.5% use two partitions, and 1.6% use between three and five partitions. This shows that our monkey testing path exercises the system adequately as nearly half of all requests need to be load-balanced within the VCI Swarm.

Stress Test. We also designed a stress test, where each simulated user sends R requests at a time and maintains R active requests at a time by sending a new request as soon as the previous result has been received. The stress test scales up the number of simulated users as well as the value of R in order to understand the scalability of the VCI Service.

Test Logging. Our results are measured after the tests have run. To do this, we collect detailed traces of the execution of the swarm including specific messages, variables, and wall clock times. Each trace is from the context of a single user request from the browser and this context persists across forwarded jobs. By measuring the time between particular messages, we can determine how much time is spent in the tracing kernel or making cross-instance network requests, all without having to rerun the test cases.

Metrics. We collect performance metrics from the perspective of the client: latency until receiving the first byte of the result (Time To First Byte **TTFB**) and the last byte (Time To Last Byte **TTLB**). We also collect ping latency between client and server, as well as the success or cancellation status for each request. Further discussion is in section 5.5.

5.2 Interactive Test on Dedicated Server

We first evaluate how swarm configurations affect various tradeoffs among factors that include latency, throughput, system footprint, replication factors (i.e. k), I/O, and potential failures. All evaluation tests are run with the client being connected to an adjacent server.

TABLE 1: Dedicated Single-Node Performance (WebSocket)

n-k	Cores/I	Memory/I	MB/s	Syscall/s	TTFB	TTLB
4-1	12	17.3 · 42.0	0.40	3.80k	0.180s	0.252s
4-2	12	17.3 · 39.5	0.40	4.50k	0.178s	0.246s
4-4	12	17.3 · 47.0	0.37	4.29k	0.174s	0.251s
8-1	6	17.3 · 25.5	0.53	4.43k	0.082s	0.138s
8-2	6	17.3 · 25.7	0.54	5.04k	0.082s	0.139s
8-4	6	17.3 · 32.7	0.51	4.97k	0.082s	0.138s
16-1	3	17.3 · 16.9	0.57	5.38k	0.064s	0.099s
16-2	3	17.3 · 19.6	0.61	5.80k	0.072s	0.121s
16-4	3	17.3 · 24.0	0.62	5.62k	0.070s	0.113s

n-k: n swarm instances with k-replication of data. **Cores/I**: CPU Cores per instance of the swarm. **Memory/I**: Average memory usage of an instance before and after testing. **MB/s**: Peak incoming network request throughput. **Syscalls/s**: I/O operations per second. **TTFB** (**TTLB**): Average Time To First (Last) Byte for request response.

TABLE 2: Dedicated Single-Node Communication Comparison

n-k	Method	Cancel	Success	TTFB	TTLB
4-2	HTTP	1288	1951	0.201s · 0.357s	0.232s · 0.384s
4-2	WS	340	2448	0.056s · 0.178s	0.092s · 0.246s
8-2	HTTP	1086	2152	0.262s · 0.394s	0.287s · 0.415s
8-2	WS	204	3139	0.048s · 0.082s	0.078s · 0.139s
16-2	HTTP	1155	2071	0.323s · 0.424s	0.350s · 0.445s
16-2	WS	237	3192	**0.046s · 0.072s**	**0.074s · 0.121s**

n-k: n swarm instances with k-replication of data. **Method**: Method of communication. **Cancel (Success)**: Number of canceled (successful) requests. **TTFB (TTLB)**: Median and average Time To First (Last) Byte.

The testing client has an internet connection capable of sustainable rates of 500-550 Mbps down and 425-475 Mbps up.

For this purpose, we chose to use a rack-mounted machine to deploy a single swarm. The machine has dual Intel Xeon (E5-2650 v4, 12-core, 2.2 GHz) processor with a total of 48 vCPU and 128 GB memory. We test $n = 4, 8, 16$ instances in the swarm and $k = 1, 2, 4$ replication in the swarm. These configurations are denoted as *n-k*, specifically, 4-1, 4-2, 4-4, 8-1, 8-2, 8-4, 16-1, 16-2, and 16-4. The per test case results are shown in Table 1.

We seek an optimal tradeoff between request latencies and request throughput. Request latencies are measured by: time to first byte vs. time to last byte, i.e. **TTFB** vs. **TTLB** in Table 1. The VCI swarm's total bandwidth to handle incoming requests throughput is measured by peak megabytes-in, i.e. **MB/s** in Table 1. The data shows a few patterns.

First, TTFB and TTLB are consistently correlated, with the best case TTFB and TTLB being 0.064s and 0.099s. Between the client (on residential Internet) and the centralized rack-mount server, the roundtrip ping time stats are: min/avg/max/stddev = 0.0590s/0.0658s/0.0871s. Hence, if a user sets up a VCI swarm for on-premise use, we estimate the TTFB and TTLB improve by 0.059s or more, resulting in a TTFB around 0.005s which is capable of sustaining 120 frames/second of interactivity.

Second, VCI instances require sufficient processing power to function optimally, because each VCI instance is massively threaded. The data shows that allocating 12 cores to each VCI instance is unnecessary. VCI instances with 6 vCPU cores or even 3 vCPU cores can function very well and deliver low latencies.

Third, increasing replication factor k does not yield increasingly higher performance, although it does show a minor improvement. As is common in many fault tolerant applications in the wide area, $k = 2$ replication is often used.

In Table 1, there are two memory use metrics. **Base Mem/I** is

2168-7161 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF TENNESSEE LIBRARIES. Downloaded on July 11,2021 at 18:42:46 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3091387, IEEE Transactions on Cloud Computing

TABLE 3: AWS Variable-Node Performance (WebSocket)

Type	Nodes	vCPUs	Instances	Cost	Failures	TTFB	TTLB
t3.xl	2	8	8	**$0.33/hr**	41.45%	0.252s	0.295s
t3.xl	2	8	16	**$0.33/hr**	32.55%	0.268s	0.343s
t3.2xl	2	16	8	$0.67/hr	29.53%	0.203s	0.249s
t3.2xl	2	16	16	$0.67/hr	2.27%	0.171s	0.275s
c5.4xl	1	16	8	$0.68/hr	**0.09%**	0.074s	0.096s
c5.4xl	1	16	16	$0.68/hr	**0.09%**	0.072s	0.095s
c5.12xl	1	48	8	$2.04/hr	**0.09%**	**0.068s**	**0.085s**
c5.12xl	1	48	16	$2.04/hr	**0.09%**	**0.066s**	**0.083s**
c5.24xl	1	96	8	$4.08/hr	**0.09%**	0.080s	0.104s
c5.24xl	1	96	16	$4.08/hr	**0.09%**	0.082s	0.107s

Type: Amazon Web Services (AWS) compute instance type. **Nodes**: Number of AWS nodes. **vCPUs**: Total number of vCPUs in use. **Instances**: Number of VCI instances in the swarm. **Cost**: Cost of the provisioned compute instances. **Failures**: Percentage of requests that fail to return data. **TTFB (TTLB)**: Average Time To First (Last) Byte.

the per-instance base memory, measured as total memory use after the container is started but before performing any tasks. That value is consistently around 17 MB. **Net Mem/I** is the average additional memory used by VCI instances during the 3-minute long monkey test at roughly 60 requests/second.

Across the board, we see that the memory-mapped lazy management of data does offer an efficient implicit solution to out-of-core visualization. **Net Mem/I** stays under 50 MB throughout the tests. This is significant because such a footprint allows easy portability on virtually any system, even when using a VCI swarm as a background process on a user's personal desktop.

Net Mem/I drops as n increases because each instance is responsible for fewer partitions. When $n = 16$, **Net Mem/I** reduces further to less than 20 MB. Increasing k does increase **Net Mem/I** incrementally but nonlinearly.

At $k = 4$ in Table 1, we see the highest **Net Mem/I** and also a corresponding increase in I/O overheads, **Syscall/s**. The latter measures thousands of I/O operations per second which signifies higher amounts of data loading. The relatively worse **TTFB** result for higher k may be due to the increased I/O and worse caching performance within each instance.

Table 2 compares VCI's two modes of communication: HTTP and WebSocket. We chose to test only $k = 2$ for these comparisons, as it is a commonly used redundancy factor. It is immediately apparent that WebSocket provides a higher success rate as well as faster response times in all cases by a large factor. This speaks volumes to the benefits of a persistent connection to the server in continuous-demand applications. The $n = 16$ WebSocket test case is of particular interest, as it manages not only the fewest cancellations of all tests, but also the lowest TTFB/TTLB. This suggests the WebSocket implementation scales very well with increased instance count.

It's worth noting that monkey test, as done in our testing, is a typical stress testing technique in web-scale systems. Our use of a total of 3,481 user requests, each with 5 seeds, is aimed to identify worst case scenario of VCI swarm. A typical user exploration does not come close to this level of system stress.

5.3 Interactive Test on AWS Server

As a significant test of deployability, we tested 5 different AWS configurations across two hardware classes: general-purpose, "t3," and compute-optimized, "c5." All instances are cloud-managed, where the base hardware is probably shared by many users.

We repeat the same monkey testing as with our dedicated server and based on Section 5.2, we test only $n = 8, 16$ instances and $k = 2$ redundancy. The network between the client and the us-east-2 AWS instances has a round trip ping timing of: min/avg/max = 0.0495s/0.0518s/0.0600s. Table 3 shows our testing results.

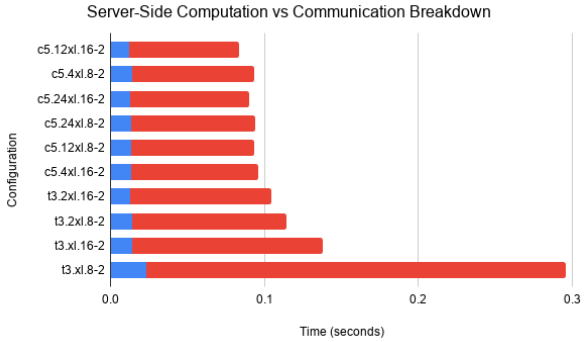


Fig. 10: Evaluating how much time an average request spends doing computation (blue) vs. communication (red) of each AWS configuration. The per-request tracking is enabled by VCI's design and use of UUID. The data shown is averaged over all requests in the monkey testing process. Reducing communication cost for better job placement can be as important as reducing computation cost for better efficiency. Experimenting with different configurations can help discover such tradeoffs.

Since we have much lower performance guarantees on AWS than on a dedicated server, we define request failures as any re-sends or request with TTFB over 1 second. We report failure rates as **Failure %** in Table 3. The failure results clearly show that the "t3" class is not usable for our purpose.

Using "c5" instances, some patterns from Section 5.2 are again confirmed. Specifically, as long as a swarm has enough instances to ensure incoming bandwidth, further increasing instance count does not lead to lower request latency. Anticipating user's workload is key when configuring a VCI service.

Based on our testing use, an 8-instance swarm on c5.4xl for $0.68/hr is the best choice. The more expensive instances performing worse is a repeatable result. A potential reason is that larger instances on AWS are shared by more users who have more sustained heavy loads. Again, the purpose of testing on AWS in this work is to show portability. The best performance for future users will be to use VCI swarms using their own on-prem cloud.

Figure 10 breaks down average server-side latencies into: (i) computation time, i.e. time spent inside computing kernels, implemented in C with OpenMP acceleration; and (ii) communication, i.e. the time spent doing network I/O for better job placement. The t3 configurations perform worse due to its general-purpose CPUs compared to the c5 configurations.

5.4 Stress Test on Dedicated Server

We ran stress tests that varied two parameters: (i) the number of users U, and (ii) the number of concurrent requests R. Accordingly, we design Test A, where U is varied while $R = 6$ ("many user"), and Test B, where R is varied while $U = 1$ ("many request"). In each of the stress tests, every request is for 1 flow line.

The results of Test A are in Figure 11. This result highlights that VCI is scalable to many users all abiding by a request limit like web browsers have. In this way, if every VCI user limits their number of active requests, then all users can make interactive requests at a rate of 80 requests per second.

The results of Test B are in Figure 12, showing that higher degrees of performance are achievable by making more concurrent

2168-7161 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF TENNESSEE LIBRARIES. Downloaded on July 11,2021 at 18:42:46 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3091387, IEEE Transactions on Cloud Computing

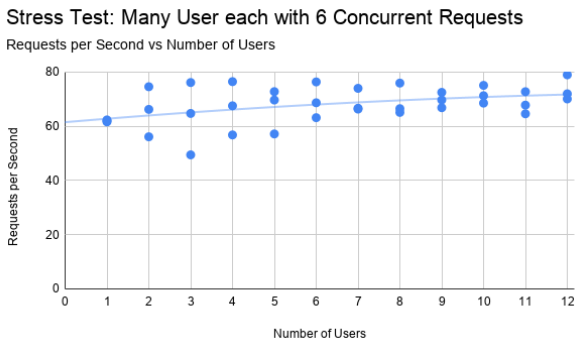


Fig. 11: Stress Test A that shows how the VCI Service handles increasing number of users where each user only makes 6 concurrent requests at once, like in a web browser.

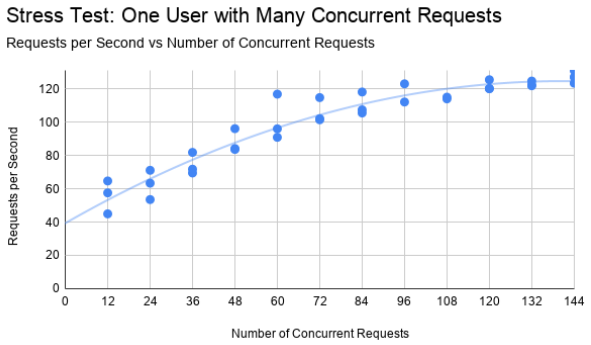


Fig. 12: Stress Test B that shows how the VCI Service handles increasing number of concurrent requests from the same user.

requests than a web browser can make. This illustrates the utility of WebSockets from a user perspective, as users are able to make up to 125 requests per second.

5.5 Discussion

Evaluating VCI's performance is tricky because VCI uses a custom-made web server for parallel computing. While works in parallel visualization evaluate efficiency and scaling from an algorithm perspective, the most applicable performance metrics for VCI are those from the literature of web systems, such as in [43] and [20]. Here, response time and incoming system bandwidth are the two primary metrics. We've reported those as MB/s-In and TTFB (Table 1 and Table 3).

Client-Side Interactivity. Using a Docker swarm on either AWS or a dedicated server, we've recorded average TTFB time below 0.070s. In addition, we have seen, in either server setup, the server can sustain answering 60 requests/second. On the client side, this translates to 60 frames/second interactive rates with 70ms latency. With typical graphics applications considering interactive use as in the range of 10 to 20 frames/second, we feel VCI is sufficient for practical use in single-user cases.

Computational Efficiency. An important paper [11] in IEEE Cluster'2020 benchmarked different parallel flow advection algorithms on Cori, one of key new HPC systems at NERSC. Due to how our user requests are generated, the only algorithm we can choose, which is also benchmarked in [11], is Parallelize Over Data (POD). The authors reported that seeding flow advection from a small spatial box is the worst case scenario for parallel flow visualization, due to great difficulties with runtime load balancing. For that use case, the authors reported POD to achieve between 70k-700k Steps Per Rank Per Second (SPRPS), when tracing streamlines that are 1000 steps each. The dataset used is NEK5000 flow of about 400GB. The computation is batch only, and far from meeting the latency requirements of interactive use.

In our stress test, the VCI Service plateaus at around 125 interactive requests per second for one user, using a 24-core dedicated server. Each request in the stress test is for a flow line of 50 steps. This corresponds to 125×50 steps per second, i.e. roughly 0.26k steps per core per second. We also designed a test with 10k requests, 5 flow lines per request, 100 steps per flow line. Ran the test 3 times through the VCI Service on the same 24-core dedicated sever. The average completion time was 23.01 seconds. This amounts to 9.05k steps per core per second.

Since VCI swarm is Python based and uses HTTP communication in order to run on the cloud, we feel this level of performance is acceptable, because VCI services can be deployed in an on demand manner for interactive use of a 150GB data set at fractional costs.

6 Conclusion

In this work, we show that a traditionally HPC-only scientific application can be ported to the cloud platform with several key benefits: (1) immediate on-demand accessibility, (2) flexible interactivity by virtually any user that has regular residential Internet connections, (3) at negligible cost compared to any current HPC based solutions. The demonstrated capability of the VCI Service further enables big data cloud resources to be easily integrated into flexible front-end applications that can operate on very low cost user devices.

We've evaluated the efficiency and portability of VCI Service using a dedicated server as well as cloud-shared instances of Amazon AWS. To test, we use leading-edge global 3D atmospheric data for the calendar year of 2012, which is made available through NCEP CFS community repositories.

There are several directions for future work. First, to consider using collaborating cloud services in scientific workflows that go beyond just visualization. Second, to consider using transiently-free resources as self-organizing swarms for parallel in-situ tasks. Third, to consider using swarm-driven cloud architectures to revisit harder scientific visualization problems such as ensemble visualization, computational steering, and uncertainty quantification. Fourth, to make use of more advanced and highly parallel flow tracing kernels that offer better guarantees of error control.

Acknowledgments

The authors would like to thank the anonymous reviewers of this and previous versions of the manuscript for their valuable comments and suggestions. The authors are supported in part by NSF Award CNS-1629890, CCRI-1925615, and USDI-NPS P14AC01485.

References

[1] http://d3js.org/.
[2] Paraview ArcticViewer - https://kitware.github.io/arctic-viewer/, 2018.
[3] Software Container Platform - Docker: https://www.docker.com/, 2018 (accessed October 14, 2018).

2168-7161 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF TENNESSEE LIBRARIES. Downloaded on July 11,2021 at 18:42:46 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3091387, IEEE Transactions on Cloud Computing

[4] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[5] D. Atkins, T. Dietterich, T. Hey, S. Baker, S. Feldman, and L. Lyon. Final Report: National Science Foundation Advisory Committee for Cyberinfrastructure Task Force on Data and Visualization, 2011.

[6] U. Ayachit. The Paraview guide: a parallel visualization application. 2015.

[7] F. Berman and R. Rutenbar. REALIZING THE POTENTIAL OF DATA SCIENCE - Final Report from the National Science Foundation Computer and Information Science and Engineering Advisory Committee, Data Science Working Group, 2016.

[8] F. Berman and L. Snyder. On mapping parallel algorithms into parallel architectures. *Journal of Parallel and Distributed Computing*, 4(5):439–458, 1987.

[9] S. W. Berrick, G. Leptoukh, J. D. Farley, and H. Rui. Giovanni: a web service workflow-based data visualization and analysis system. *IEEE Transactions on Geoscience and Remote Sensing*, 47(1):106–113, 2008.

[10] R. Binyahib, D. Pugmire, and H. Childs. In situ particle advection via parallelizing over particles. In *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pp. 29–33, 2019.

[11] R. Binyahib, D. Pugmire, A. Yenpure, and H. Childs. Parallel particle advection bake-off for scientific visualization workloads. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 381–391. IEEE, 2020.

[12] A. Biswas, R. Strelitz, J. Woodring, C.-M. Chen, and H.-W. Shen. A scalable streamline generation algorithm via flux-based isocontour extraction. In *Proceedings of the 16th Eurographics Symposium on Parallel Graphics and Visualization*, pp. 69–78, 2016.

[13] M. Bostock, V. Ogievetsky, and J. Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[14] S. A. Bozhenkov, J. Geiger, M. Grahl, J. Kisslinger, A. Werner, and R. C. Wolf. Service oriented architecture for scientific analysis at w7-x. an example of a field line tracer. *Fusion Engineering and Design*, 88(11):2997–3006, 2013.

[15] D. Camp, H. Childs, A. Chourasia, C. Garth, and K. I. Joy. Evaluating the benefits of an extended memory hierarchy for parallel streamline algorithms. In *2011 IEEE Symposium on Large Data Analysis and Visualization*, pp. 57–64. IEEE, 2011.

[16] D. Camp, C. Garth, H. Childs, D. Pugmire, and K. Joy. Streamline integration using mpi-hybrid parallelism on a large multicore architecture. *IEEE Transactions on Visualization and Computer Graphics*, 17(11):1702–1713, 2010.

[17] C.-M. Chen and H.-W. Shen. Graph-based seed scheduling for out-of-core ftle and pathline computation. In *2013 IEEE symposium on large-scale data analysis and visualization (LDAV)*, pp. 15–23. IEEE, 2013.

[18] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil. VisIt: An end-user tool for visualizing and analyzing very large data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pp. 357–372. Oct 2012.

[19] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. Da Silva, M. Livny, et al. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015.

[20] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. Vahdat. Model-based resource provisioning in a web service utility. In *USENIX Symposium on Internet Technologies and Systems*, vol. 4, pp. 5–5, 2003.

[21] D. R. Easterling, G. A. Meehl, C. Parmesan, S. A. Changnon, T. R. Karl, and L. O. Mearns. Climate extremes: observations, modeling, and impacts. *science*, 289(5487):2068–2074, 2000.

[22] J. Emeras, S. Varrette, V. Plugaru, and P. Bouvry. Amazon elastic compute cloud (ec2) vs. in-house hpc platform: a cost analysis. *IEEE Transactions on Cloud Computing*, 2016.

[23] B. Fisher. *Illuminating the Path: An R&D Agenda for Visual Analytics*, pp. 69–104. 01 2005.

[24] M. A. Fitzpatrick, C. M. McGrath, and S. P. Young. Pathomx: an interactive workflow-based tool for the analysis of metabolomic data. *BMC bioinformatics*, 15(1):396, 2014.

[25] M. Fu, A. Agrawal, A. Floratou, B. Graham, A. Jorgensen, M. Li, N. Lu, K. Ramasamy, S. Rao, and C. Wang. Twitter heron: Towards extensible streaming engines. In *2017 IEEE 33rd international conference on data engineering (ICDE)*, pp. 1165–1172. IEEE, 2017.

[26] H. Guo, X. Yuan, J. Huang, and X. Zhu. Coupled ensemble flow line advection and analysis. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2733–2742, 2013.

[27] P. M. Inness and J. M. Slingo. Simulation of the madden–julian oscillation in a coupled general circulation model. part i: Comparison with observations and an atmosphere-only gcm. *Journal of Climate*, 16(3):345–364, 2003.

[28] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *2010 IEEE second international conference on cloud computing technology and science*, pp. 159–168. IEEE, 2010.

[29] A. Jeuken, P. Siegmund, L. Heijboer, J. Feichter, and L. Bengtsson. On the potential of assimilating meteorological analyses in a global climate model for the purpose of model validation. *Journal of Geophysical Research: Atmospheres*, 101(D12):16939–16950, 1996.

[30] M. Jiang, B. Van Essen, C. Harrison, and M. Gokhale. Multi-threaded streamline tracing for data-intensive architectures. In *2014 IEEE 4th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 11–18. IEEE, 2014.

[31] J. Jomier, S. Jourdain, U. Ayachit, and C. Marion. Remote visualization of large datasets with midas and paraviewweb. In *Proceedings of the 16th International Conference on 3D Web Technology*, Web3D '11, pp. 147–150. ACM, New York, NY, USA, 2011. doi: 10.1145/2010425.2010450

[32] S. Jourdain, S. Wittenburg, and P. O'Leary. Python enabled paraviewweb for hpc analysis and visualization. In *Python in HPC Workshop, International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*, 2014.

[33] W. Kendall, J. Wang, M. Allen, T. Peterka, J. Huang, and D. Erickson. Simplified parallel domain traversal. In *SC11: Proceedings of the ACM/IEEE Conference on Supercomputing*, pp. 10:1–10:11, 2011.

[34] G. M. Kurtzer, V. Sochat, and M. W. Bauer. Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5), 2017.

[35] P. Leach, M. Michael, and R. Salz. RFC 4122: A Universally Unique Identifier (UUID) URN namespace. RFC, July 2005.

[36] W. Li and S. Wang. Polarglobe: A web-wide virtual globe system for visualizing multidimensional, time-varying, big climate data. *International Journal of Geographical Information Science*, 31(8):1562–1582, 2017.

[37] Z. Liu, R. Moorhead, and J. Groner. An advanced evenly-spaced streamline placement algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):965–972, 2006.

[38] T. McLoughlin, M. W. Jones, R. S. Laramee, R. Malki, I. Masters, and C. D. Hansen. Similarity measures for enhancing interactive streamline seeding. *IEEE Transactions on Visualization and Computer Graphics*, 19(8):1342–1353, 2012.

[39] M. Meinshausen, S. C. Raper, and T. M. Wigley. Emulating coupled atmosphere-ocean and carbon cycle models with a simpler model, magicc6-part 1: Model description and calibration. 2011.

[40] C. Müller, D. Camp, B. Hentschel, and C. Garth. Distributed parallel particle advection using work requesting. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pp. 1–6. IEEE, 2013.

[41] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell. Samza: stateful scalable stream processing at linkedin. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, 2017.

[42] B. Nouanesengsy, T.-Y. Lee, and H.-W. Shen. Load-balanced parallel streamline generation on large scale vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1785–1794, 2011.

[43] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Annual Technical Conference, General Track*, pp. 199–212, 1999.

[44] V. Pascucci, G. Scorzelli, B. Summa, P.-T. Bremer, A. Gyulassy, C. Christensen, S. Philip, and S. Kumar. The visus visualization framework. *EW Bethel, HC (LBNL), and CH (UofU), editors, High Performance Visualization: Enabling Extreme-Scale Scientific Insight, Chapman and Hall/CRC Computational Science*, 2012.

[45] T. Peterka, R. Ross, B. Nouanesengsy, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang. A study of parallel particle tracing for steady-state and time-varying flow fields. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 580–591. IEEE, 2011.

[46] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Numerical Recipes in C, 1988.

[47] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber. Scalable computation of streamlines on very large datasets. In *Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, 2009.

2168-7161 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF TENNESSEE LIBRARIES. Downloaded on July 11,2021 at 18:42:46 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2021.3091387, IEEE Transactions on Cloud Computing

[48] X. Qin, X. Chen, L. Chen, H. Zheng, J. Ma, and M. Zhang. Streamline uniform placement algorithm with dynamic seed points. *IEEE Access*, 7:113844–113852, 2019.

[49] M. Raji, A. Hota, T. Hobson, and J. Huang. Scientific visualization as a microservice. *IEEE Transactions on Visualization and Computer Graphics (accepted)*, 2018.

[50] M. Raji, A. Hota, and J. Huang. Scalable web-embedded volume rendering. In *IEEE Symp. on Large Data Analysis and Visualization (LDAV)*, pp. 45–54, Oct 2017. doi: 10.1109/LDAV.2017.8231850

[51] J. Rogelj, M. Meinshausen, and R. Knutti. Global warming under old and new scenarios using ipcc climate sensitivity range estimates. *Nature climate change*, 2(4):248–253, 2012.

[52] S. Saha, S. Moorthi, X. Wu, J. Wang, S. Nadiga, P. Tripp, D. Behringer, Y.-T. Hou, H. ya Chuang, M. Iredell, M. Ek, J. Meng, R. Yang, M. P. Mendez, H. van den Dool, Q. Zhang, W. Wang, M. Chen, and E. Becker. NCEP Climate Forecast System Version 2 (CFSv2) 6-Hourly Products, 2011.

[53] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics*, 23(1):341–350, 2016.

[54] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *ACM Sigmod Record*, 34(4):42–47, 2005.

[55] J. Tao, J. Ma, C. Wang, and C.-K. Shene. A unified approach to streamline selection and viewpoint selection for 3d flow visualization. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):393–406, 2012.

[56] V. Verma, D. Kao, and A. Pang. A flow-guided streamline seeding strategy. In *Proceedings Visualization 2000. VIS 2000 (Cat. No. 00CH37145)*, pp. 163–170. IEEE, 2000.

[57] T. Wetzlmaier, R. Ramler, and W. Putschögl. A framework for monkey gui testing. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 416–423. IEEE, 2016.

[58] P. T. While and L. K. Forbes. Equi-flux streamline seeding for three-dimensional vector fields. *Journal of Engineering Mathematics*, 76(1):81–100, 2012.

[59] K. Wu, Z. Liu, S. Zhang, and R. J. Moorhead II. Topology-aware evenly spaced streamline placement. *IEEE Transactions on Visualization and Computer Graphics*, 16(5):791–801, 2009.

[60] L. Xu, T.-Y. Lee, and H.-W. Shen. An information-theoretic framework for flow visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1216–1224, 2010.

[61] H. Yu, C. Wang, and K.-L. Ma. Parallel hierarchical visualization of large time-varying 3d vector fields. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pp. 1–24, 2007.

[62] P. Yue, M. Zhang, and Z. Tan. A geoprocessing workflow system for environmental monitoring and integrated modelling. *Environmental Modelling & Software*, 69:128–140, 2015.

[63] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[64] J. Zhang, H. Guo, F. Hong, X. Yuan, and T. Peterka. Dynamic load balancing based on constrained kd tree decomposition for parallel particle tracing. *IEEE transactions on visualization and computer graphics*, 24(1):954–963, 2017.

[65] J. Zhang, H. Guo, and X. Yuan. Efficient unsteady flow visualization with high-order access dependencies. In *2016 IEEE Pacific Visualization Symposium (PacificVis)*, pp. 80–87. IEEE, 2016.

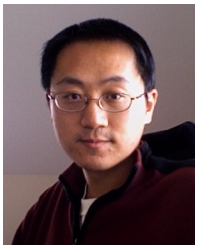
[66] T. Zhang, J. Li, Q. Liu, and Q. Huang. A cloud-enabled remote visualization tool for time-varying climate data analytics. *Environmental Modelling & Software*, 75:513–518, 2016.

[67] W. Zhang, Y. Wang, J. Zhan, B. Liu, and J. Ning. Parallel streamline placement for 2d flow fields. *IEEE transactions on visualization and computer graphics*, 19(7):1185–1198, 2012.

Tanner Hobson Tanner Hobson received his BS in Computer Science from the University of Tennessee, Knoxville, where he is currently a PhD student. His research interests include distributed computing, mixed reality visualization, and web-based systems architectures.

James Hammer James Hammer received the bachelor of science degree from the University of Tennessee Knoxville in computer science. He is currently working toward the PhD degree in computer science. His research interests are on the topics of real-time rendering, data visualization systems, and visualization architectures.

Preston Provins Preston Provins received the bachelor of science degree from the University of Tennessee Knoxville in computer science where he is currently working toward the PhD degree in computer science. His research interests are on the topics of cloud computing, GPGPU computing, and real-time rendering.



Jian Huang Jian Huang is a professor in the Department of Electrical Engineering and Computer Science at the University of Tennessee, Knoxville. His research focuses on data visualization and analytics. He received his PhD degree in computer science from the Ohio State University in 2001. Dr. Huang's research has been funded by National Science Foundation, Department of Energy, Department of Interior, NASA, UT-Battelle, and Intel.

2168-7161 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.
Authorized licensed use limited to: UNIVERSITY OF TENNESSEE LIBRARIES. Downloaded on July 11,2021 at 18:42:46 UTC from IEEE Xplore. Restrictions apply.