

# Scientific Visualization as a Microservice

Mohammad Raji<sup>1</sup>, Alok Hota<sup>1</sup>, Student Member, IEEE, Tanner Hobson<sup>1</sup>,  
and Jian Huang<sup>1</sup>, Senior Member, IEEE

**Abstract**—In this paper, we propose using a decoupled architecture to create a microservice that can deliver scientific visualization remotely with efficiency, scalability, and superior availability, affordability and accessibility. Through our effort, we have created an open source platform, Tapestry, which can be deployed on Amazon AWS as a production use microservice. The applications we use to demonstrate the efficacy of the Tapestry microservice in this work are: (1) embedding interactive visualizations into lightweight web pages, (2) creating scientific visualization movies that are fully controllable by the viewers, (3) serving as a rendering engine for high-end displays such as power-walls, and (4) embedding data-intensive visualizations into augmented reality devices efficiently. In addition, we show results of an extensive performance study, and suggest how applications can make optimal use of microservices such as Tapestry.

**Index Terms**—Scientific visualization, visualization systems, cloud computing, web applications, and distributed visualization

## 1 INTRODUCTION

WHILE the web has transformed the way our society uses computers and computing technology over the past 20 years, the web architecture has undergone rapid and accelerating improvements itself, gradually becoming one of the most robust, scalable, and general client-server architectures.

Since client-server is a core design in scientific visualization, our field has dedicated much effort to map the scientific visualization pipeline onto the web architecture; for example, to increase the mobility of scientific visualization applications, to reach a wider audience, and to use the rich web ecosystem for building interactive tools and websites. To this end, our field first explored ways to adopt the web browser as a general client in scientific visualization applications. Para-ViewWeb [1], ViSUS [2], XML3D [3], ArcticViewer [4] are a few of the most well known successes in that regard.

The next step is to explore whether web services can serve as performance servers in the client-server settings of scientific visualization. This work extends our previous efforts [5] in which we presented Tapestry, a platform for the scalable delivery of scientific visualization. Tapestry is an open source platform<sup>1</sup> that separates application states from the server and makes the server stateless.

The benefits of using stateless servers are multi-fold: (1) make it feasible to use virtualized containers, parallel swarm management, auto load balancing, and auto-scaling in scientific visualization applications, (2) make it practical to expand application design goals to also include expanding accessibility of the visualizations, i.e., to improve the

scalability of audience, (3) formalize and unify the communication between client and server as simplified *rendering requests*. The Tapestry system treats each rendering request independently and automates request handling through the use of Docker Swarm [6].

Using rendering requests as the unifying interface, we were able to further simplify the complexity on the client-side and introduce 3D scientific visualizations into the standard Web Document Object Model (DOM) as very lightweight objects called *hyperimages* [5]. A web page can contain as many hyperimages as the web developer would like. When not interacted with, the performance cost of a hyperimage is equal to that of any static image, and its impact on web page load time is negligible. When a user interacts with a hyperimage, the in-browser user experience is as if the whole scientific visualization is local. A hyperimage transparently translates user interactions into web requests, issues the requests, and receives and caches new images from the server. Hyperimages are not only interactive themselves. Interactions on different hyperimages embedded in the same DOM can be linked and synchronized via *hyperactions* [5].

While web services directly serve users, microservices serve application developers, so that new applications can make agile and scalable use of a suite of highly available and accessible services, instead of having to remain tightly-coupled with certain hardware or platforms.

In this work, we extend Tapestry into an independently deployable microservice, in particular, by increasing controllable granularity and parallel end-points in the system. We also release Tapestry in the form of a Docker image (details in Section 3.4), which is portable across cloud platforms such as Amazon AWS.

Herein, we show that the Tapestry microservice can facilitate the following application innovations: (1) embed volume visualization into lightweight web pages that work interactively even on mobile phones, (2) create portable and user-controllable movies of scientific visualizations, (3) serve as an efficient and cost-effective parallel rendering engine for large

1. <https://github.com/seelabutk/tapestry>

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

Manuscript received 21 Feb. 2018; revised 16 Oct. 2018; accepted 29 Oct. 2018. Date of publication 5 Nov. 2018; date of current version 4 Mar. 2020.

(Corresponding author: Mohammad Raji.)

Recommended for acceptance by J. Ahrens.

Digital Object Identifier no. 10.1109/TVCG.2018.2879672

power-wall displays, (4) embed interactive volume renderings into augmented reality devices (e.g., HoloLens) in a very lightweight manner.

Besides enabling novel applications, scalability is another key aspect of microservices. To that end, we have conducted an extensive performance study of the Tapestry microservice. For deployment platforms, we tested on an institutional cloud and Amazon AWS cloud. Our scalability tests included up to 100 geographically distributed test workers. Regarding test methodology, we included batch-mode stress test, realistic user behavior based monkey testing, and actual application usage testing. Our test datasets come from typical domains of scientific visualization.

We will discuss related works in Section 2, Tapestry's architecture in Section 3, demonstrative applications in Section 4, and results in Section 5. The conclusion is in Section 6.

## 2 BACKGROUND

Delivering scientific visualization with mobility is full of challenges, regardless of whether web is involved, because scientific visualization is resource heavy. With today's ever expanding needs for analytics on datasets of growing sizes, scientific visualization's simultaneous requirement of computing capabilities, data bandwidth, and interactivity calls for our field to rethink the traditional client-server delivery mechanisms of scientific visualization.

### 2.1 Using Web Browsers as a Visualization Client

Over the years, visualization researchers have made much effort to make interactive visualizations work inside web browsers. The creation of D3 in 2011 became one of the most recognized milestones [7] in that regard. Other notable successes include: (i) visualization system interfaces, such as Visualizer and LightViz [1]; (ii) API-based scientific data management applications, such as MIDAS using the ParaViewWeb API [8]; (iii) plugin-based web browser systems backed by a high-end resource, such as ViSUS using an IBM BlueGene [2]; and (iv) plugin free implementations backed by custom clusters, such as XML3D [9], [10].

Within a browser, the scientific visualizations can be implemented in *data-space* or *image-space*. A naive data-space implementation is to download the dataset into a browser and then render it using libraries such as WebGL or VTK.js [11]. This has the obvious limitations of available bandwidth and working memory on the local machine. More scalable data-space systems perform data processing and rendering on a remote server and transmit final or intermediate data to the client web browser for further handling. Those solutions fit better with the high-end computing community [1], [2], [8], [9], [10], where server-side computing and networking resources tend to be abundant. Regardless, when data is large, the networking and in-browser computing overhead can still be overwhelming. In particular, render performance would be slow on mobile devices, leading to an unresponsive web page.

Image-space techniques can be used for remote visualization in general [12]. As a more recent success, ArcticViewer is a web visualization system that improves in-browser user experience by serving pre-rendered images of datasets

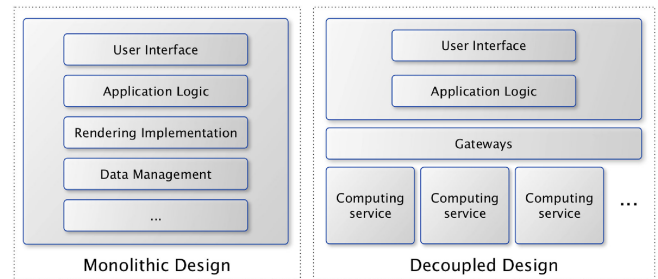


Fig. 1. Comparing a monolithic design (left) and a decoupled design (right). In a monolithic design, application states exist throughout the component stack. In a decoupled design, compute-intensive tasks such as rendering and data management are encapsulated in stateless services and accessed through a unified cloud-hosted gateway.

on demand [4]. Paired with Cinema [13], ArcticViewer addresses needs by the in-situ visualization community particularly well. However, pre-rendering typically generates a large amount of rendered images that may or may not be used by the end-user.

### 2.2 Architectural Designs of Client-Server

Regardless of whether a web browser is used, delivering visualizations with mobility should ideally combine the best of both data-space and image-space designs. For example, it should be “expressive and flexible” like in data-space systems [7], and be “immediately available” like in image-space systems [13].

This need can benefit from having a more clear separation of concerns, where the expressive and flexible interactions are handled separate from the highly available and efficient computing.

Existing applications often take a monolithic approach (Fig. 1-left). In result, the 1-to-1 mapping between client and server has become a standard design in existing systems, such as VisIt[14], ParaView[15], and web-based systems like ParaViewWeb. Previous works have also explored adding staging nodes in between the client and the server, in order to achieve better system performances [16], [17], while continuing to abide by the monolithic 1-to-1 mapping between client-server.

A decoupled design (Fig. 1-right) can separate the highly responsive visual application from the compute-intensive components such as rendering. In that regard, Tapestry [5] was the first example to our knowledge that implemented such decoupling and allow the client-server to have an  $m$ -to- $n$  relationship, where  $m$  and  $n$  can be any number above 1.

By simplifying the client-server interface into rendering requests, Tapestry's server-side is responsible for rendering and is completely oblivious to anything application related; that is, it has become stateless, free of application logic. The server can answer simultaneous requests from  $m$  different clients. On the client-side, a web-page can contain visualizations of many datasets, potentially hosted on  $n$  different servers.

While computing services commonly use HPC platforms, the Tapestry server was instead created as a web service managed by Docker [5], to reap the benefits of automatic load balancing, auto-scaling, and automatically parallelized data transfers that are standard in today's web technology.

In this work, we further extend Tapestry so that the Tapestry server can run as an Amazon AWS deployed microservice,

which is independently deployable, fine-grained, and lightweight. For example, as we will show in Section 5.6, using the Tapestry microservice, applications can achieve interactive performance at minuscule costs. Hence, scientific visualization can be more accessible and available than before.

Established tools that have played a pioneering role in today's computational science, such as VisIt [14], ParaView [15], and Arctic Viewer [4], should be able to adopt Docker based microservice model for resource-scaling purposes too.

### 2.3 Server-Side Rendering

Volume visualization is well understood from an algorithm perspective [18]. Highly efficient implementations using many-core processors, either GPU or CPU, are available as community-maintained open-source renderers [14], [15], [19], [20]. In this work, we use OSPRay [19] because of its rendering performance. Additionally, its software-only nature makes it easier to manage in a typical cloud-managed container. A GPU-based renderer that exhibits similar throughput to OSPRay can also be used. Unlike in our previous version where PNG was used, we now encode OSPRay-rendered framebuffers as JPG images, because of its better compression rate and faster in-browser decompression.

Level-of-detail is a proven approach to manage the trade-off between speed and quality for time-critical visualization [21], [22], [23]. Tapestry uses a similar approach. When a user interacts with the 3D visualization in the web document, rendering requests are made at a lower resolution. After a user pauses, rendering requests are made at a higher resolution. This is detailed in Section 3.1.2.

Parallel visualization generally takes three approaches: data-parallel, task-parallel, and a hybrid of the two [24], [25]. Our primary concern is system throughput (i.e., rendering requests/sec). We chose the task-parallel approach to process rendering requests in parallel. As is commonly done [26], we group worker processes into a two-level hierarchy: (i) the computing cluster as a whole, (ii) each computing node. Worker processes on the same node share datasets via memory-mapped regions of disk. Using known methods to resolve I/O bottlenecks [27], we have a dedicated I/O layer as the data manager on each node to manage pre-loading the data once Tapestry starts (detailed in Section 3.2.1).

## 3 ARCHITECTURE

Tapestry decouples client and server and separates the application space from the system space. We do so by formalizing rendering requests as a reduced and restricted interface, and the only interface, between the two spaces. As shown in the system diagram (Fig. 2), the generation of rendering requests in the application space is asynchronous and distributed. On the server side, rendering requests are automatically distributed to many disparate endpoints through typical web server load balancers and ensures scalability.

The application space maintains the *dynamic states* related to the application and interaction. The system space is dedicated to answering rendering requests and stays stateless without maintaining any application state information.

The two spaces have different life cycles. The system space stays up as long as the cloud service is up. The application space exists as individual instances, with one instance per

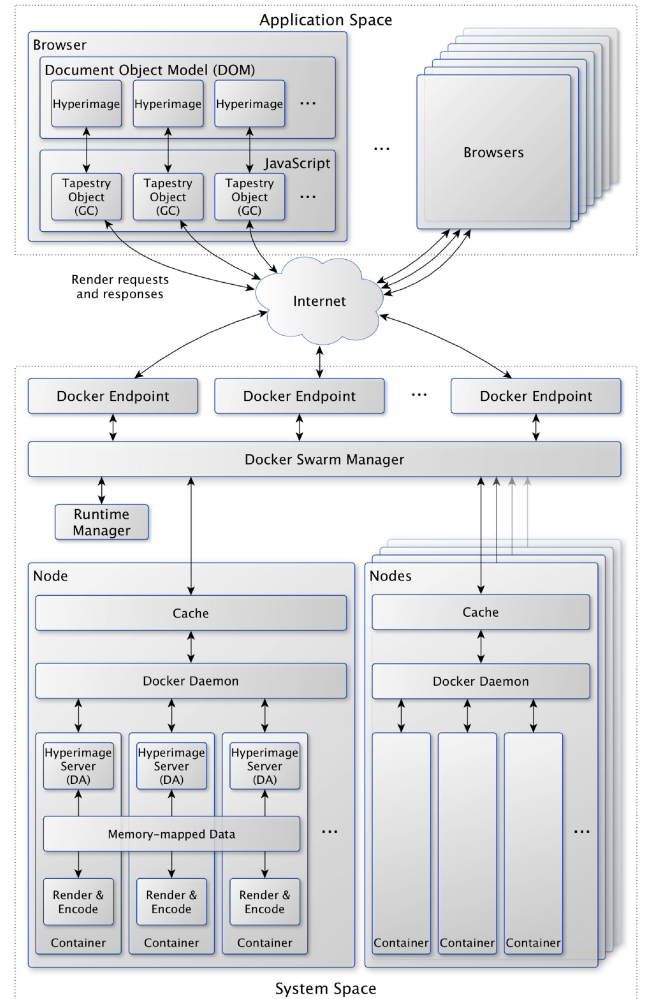


Fig. 2. The Tapestry system architecture, which separates the application space and system space.

each session when a user accesses the application, e.g., a web page with embedded 3D visualizations. The application space can have many instances. The system space is a single entity shared by all instances of the application space.

In the application space, a hyperimage is the universal interactive visualization object. Each hyperimage is controlled by an attached Tapestry object in JavaScript, which presents the 3D interactions and automatically requests services from the server, by way of issuing rendering requests. Details in Section 3.1.

The system space is cloud hosted on a cluster of nodes. These nodes comprise a Docker Swarm [6]. The swarm abstracts handling of rendering requests into a cluster of microservices implemented in virtualized containers, which the swarm manages altogether as a collection. The system also includes elastic task handling, request routing, and automatic resource scaling. Details in Section 3.2.

Connections between the two spaces are simple, short, and transient rendering requests. An application instance can generate many rendering requests concurrently. The system space can answer a large amount of rendering requests simultaneously. The system space does not relate one rendering request with another, and treats each request independently, even when the rendering requests are from the same application instance.



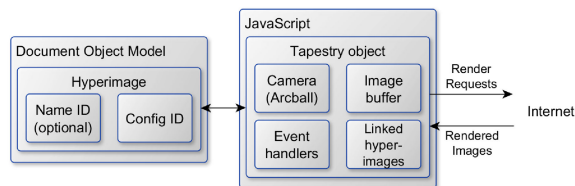


Fig. 3. Hyperimages are DOM elements. In the application space, each hyperimage element is paired with a Tapestry object, which handles user interaction and communicate with the Tapestry server.

### 3.1 Application Space

Using Tapestry, the presentation of the visualization resides in a desktop/mobile web browser as an embedded object.

Within a browser, we could consider using the HTML5 canvas or the 3D-enhanced WebGL canvas [28]. However, we chose to use a simple image tag (`<img>`) instead for several reasons. First, HTML5 and WebGL canvases are heavyweight elements with initialization costs. Their performance also relies on the user's hardware. Second, the output of many visualizations is an image and therefore an `<img>` tag is a natural medium that does not need any post-processing and is widely used across the web ecosystem. We refer to our enhanced `<img>` tags as *hyperimages*.

#### 3.1.1 Control of Visualization Objects

Fig. 3 shows a closeup of Tapestry's application space in a web setting. An application can use as many hyperimages as the developer desires. In this example, we show a single hyperimage in the DOM, but multiple may be present. In essence, a hyperimage is a simple `<img>` tag with extended capabilities. As a user interacts with a hyperimage, a controlling JavaScript object generates and submits rendering requests to the server automatically, updates the received renders and updates the hyperimage's `src` attribute.

The Graphics Context (GC) of each hyperimage is controlled by an attached Tapestry object in the `tapestry.js` JavaScript code. The GC information includes: camera management through arcball, an image buffer for received images, event handlers and a list of other hyperimages that may be linked to the object. Optional settings such as initial camera position can be sent to the Tapestry constructor if needed.

#### Listing 1. Sample Code for Adding a Hyperimage into a Webpage

```
<script>$(".hyperimage").tapestry({});</script>
<img class="hyperimage" data-dataset="supernova"/>
```

Listing 1 shows the full HTML code to embed a 3D visualization on a web page. The second line of Listing 1 shows a simple hyperimage of a supernova. The `class` attribute identifies the tag as a hyperimage, and the dataset being rendered is added in the `data-dataset` attribute. Note, `data-*` is the standard prefix for custom attributes in HTML5 [29]. Hyperimages become interactive by replacing the source attribute of the tag. When the user is not interacting, a hyperimage is effectively a simple image.

For time varying data, a hyperimage can take an optional `data-timerange` attribute. The value of this attribute represents the time step range through which the volume can animate. This range is formatted as `<integer>..<integer>`. For example, a value of 5 .. 15 would mean that

TABLE 1  
Supported Hyperactions

Action	Description
<code>position(x, y, z)</code>	Sets the position of the camera
<code>rotate(angle, axis)</code>	Rotates the camera angle degrees about the given axis
<code>zoom(z)</code>	Sets the relative camera Z position
<code>link(id1, ...)</code>	Links the viewpoint of other hyperimages to the current hyperimage's camera
<code>unlink(id1, ...)</code>	Unlinks the viewpoint of other hyperimages
<code>play()</code>	Animates the time steps of a time series dataset
<code>stop()</code>	Stops the time series animation
<code>time(t)</code>	Changes the timestep to t
<code>switch_config(name)</code>	Switches to a new hyperimage configuration

the hyperimage cycles through time steps 5 to 15 when animated.

In addition to mouse and hand gestures, Tapestry allows a customizable type of interaction: *hyperactions*. Hyperactions provide a way for the DOM to manipulate a hyperimage without user intervention. A simple use case of a hyperaction is a hyperlink in a text that rotates a hyperimage to a specific viewpoint. Hyperactions essentially provide a simple connection between textual content and volume renderings. Any standard DOM element can be converted to a hyperaction by adding three attributes: the class `hyperaction`, a `for` attribute that denotes which hyperimage should be associated with the action, and a `data-action` attribute describing the action itself. For example, a hyperlink that sets the camera position of a hyperimage is shown in Listing 2.

When clicked on, this hyperaction sets the camera position of the hyperimage with the `id` of `teapot1` to (10, 15, 100). A list of supported actions and their syntax is shown in Table 1. The logic behind what hyperactions do is also controlled by Tapestry objects. When a Tapestry object is initialized, it looks at the DOM for hyperimages and their corresponding hyperactions and sets up event handlers for the hyperactions' action. Two example applications in Section 4 make use of hyperactions.

#### Listing 2. An Example Hyperaction that Sets the Camera Position to the Given Position for the Teapot Dataset

```
<a class="hyperaction" for="teapot1" data-action="
position=10,15,100">a new viewpoint</a>
```

#### 3.1.2 Generation of Rendering Requests

The DOM defines the structure of a web page, and the JavaScript provides interactivity and control. The relationship between a hyperimage (a DOM element) and the related Tapestry object is no exception to that. When a user interacts with a hyperimage through mouse or touch gestures, the corresponding Tapestry object manages callback functions and generates rendering requests as needed. While interaction is happening, it continues to send new requests to the server-side and asks for updated renders.

During interaction (e.g., when rotating), the object requests interaction resolution images ( $256^2$  by default) to allow for smoother movement. When interaction stops, the object requests a viewing resolution image ( $1024^2$ ).

Rendering requests are sent using the HTTP GET method. As a result, renderings can be saved or shared after interaction

just like any image with a valid address. A rendering request takes the form of `http://HOST/DATASET/POS_X/POS_Y/POS_Z/UP_X/UP_Y/UP_Z/RESOLUTION/OPTIONAL`. The DATASET parameter denotes which configured dataset should be rendered. The camera position is given by `<POS_X, POS_Y, POS_Z>`, and the up vector is given by `<UP_X, UP_Y, UP_Z>`. RESOLUTION denotes the rendering's resolution. Finally, additional optional parameters can be added as a comma separated string of key-value pairs. For example, to specify the time step in a temporal series.

**Listing 3.** Two Rendering Requests for a Well-Known Supernova Simulation [30]. The Values Represent Camera Position, up Vector, and Image Size, Respectively. The Second Request Includes an Optional Time step Parameter

```
http://host.com/supernova/128.0/-256.0/500.0/0.707/0.0/0.707/256
http://host.com/supernova/128.0/-256.0/500.0/0.707/0.0/0.707/256/timestep,5
```

Tapestry objects also control the volume of rendering requests. For example, a user's mouse can typically emit up to 125 *move* events per second (on a common 125 Hz mouse). We set a default policy: let every fifth event trigger a rendering request. This policy generates up to 25 rendering requests per second.

Due to the minimal interface between the client and server, requests can also be generated in batches and by scripts, for more complicated applications. Section 4 shows this in more detail through several applications.

### 3.1.3 Non-Invasive Embedding

From an application developer perspective, Tapestry provides non-invasive integration in clients. In other words, it is simple to integrate and customize and does not cause any global changes in the host web application.

More specifically, hyperimages in the client are self-contained and do not share state with each other. This means that they can be independently added or removed in a page.

Another aspect of non-invasiveness are hyperactions. Hyperactions are behaviors, not objects. In other words, they can be added to a variety of HTML elements (e.g., buttons, hyperlinks, images, etc.) and enable interaction with a hyperimage. Those HTML elements can be freely styled and edited by the developer.

Users of scientific visualization often need to tweak and edit visualization tools to add new capabilities. To facilitate this, the Tapestry server can take an optional *app* directory as input at runtime. JavaScript, HTML, or CSS source code in the *app* directory overrides those of Tapestry's default, allowing for easy hot-swappable functional changes. In other words, client-side changes to a user's application do not require a re-compile or restart of the Tapestry service.

## 3.2 System Space

The sole concern of the system space is to process rendering requests. It is a task-parallel computing system, using distributed resources that auto-scale on demand.

In system space, we make a distinction between a *physical node*, a *Docker container*, and a *hyperimage server instance*.

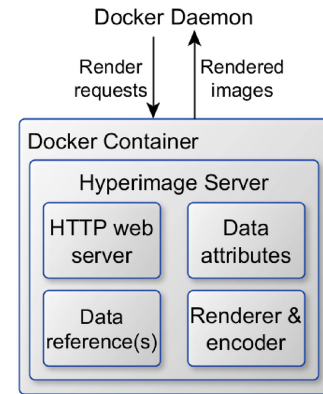


Fig. 4. A container is the basic processing unit in Tapestry's system space. Each container runs an instance of the hyperimage server.

A physical node refers to the real machine on which multiple Docker containers may be launched. There may be multiple physical nodes. A Docker container is an in-memory virtual operating system.

Fig. 4 shows a single Docker container. Each container includes an instance of a hyperimage server, which is a web server that manages attributes of given datasets, and handles any rendering requests it receives in sequence.

### 3.2.1 Container-Based Rendering Services

Virtualization and containerization are classic concepts in software architecture [31]. Open-source software container platforms have become popular, including for HPC computing services [32].

We chose Docker [6] containers because they are lightweight, and provide a robust and simple interface. Each Docker container includes a small, stripped-down version of an operating system as well as all the dependencies needed to run an application independently. Multiple containers can run on the same node.

Each physical node runs a local Docker daemon, which manages all running containers on that node. Across nodes, we use Docker Swarm as another layer of abstraction on top of a collection of physical nodes, allowing a pool of containers to have unified entry points as well as leverage Docker Swarm's load balancer.

In Tapestry, each Docker container is based on a stripped down version of Ubuntu, which runs a hyperimage server instance inside. The Docker Swarm Manager monitors and manages the containers, routes incoming rendering requests, and load balances the containers using its internal Ingress load balancer [33].

When a hyperimage server starts, it loads all pre-configured datasets into memory using a memory-mapped loading operation. In other words, containers that reside in the same worker node only load the data once and only during system startup.

### 3.2.2 Hyperimage Server and Data Attributes

A hyperimage server is initialized once and lives for the lifetime of the cloud service. A hyperimage server takes a configuration directory during initialization. All valid configuration files – properly formatted JSON files – within this directory are used to provide data attributes for the server instance.

These configuration files, provide basic information about the datasets. An example configuration file is shown in Listing 4.

**Listing 4.** Example Configuration File Providing Data Attributes

```
{
  "filename" : "/path/to/data/magnetic.bin",
  "dimensions" : [512, 512, 512],
  "colorMap" : "cool to warm",
  "opacityAttenuation" : 0.5,
  "backgroundColor" : [38, 34, 56]
}
```

The configuration files are a list of key-value pairs. A complete list of keys and possible values for configuration files can be found in our previous work [5]. These parameters are standard visualization data attributes. Basic information about the dataset, such as filename and dimensions are required, but most others are optional and can revert to default values. Different transfer functions require different configuration files. However, they can all point to the same dataset. Memory-mapping assures that the dataset used by different configurations are only loaded to memory once for each node.

Additional configuration keys available also include *isosurfaceValues* and *specular* to control isosurface rendering if desired. Note that Tapestry uses OSPRay's *implicit isosurface rendering* to provide images of surfaces. Implicit isosurfaces avoid the need to explicitly compute and store surface geometry, which allows the server to remain stateless.

Currently, the server handles raw binary and NetCDF files, two common formats for scientific data. The filename provided may be a path to a single file, i.e., a static volume, or a path with wildcard characters to describe multiple volumes, i.e., a time-varying series. Example filenames for a time-varying series could be: `"~/supernova/*.bin"` for all available time steps or `"~/supernova/time_[2-7].bin"` for 5 specific time steps.

During initialization, the datasets referred to by the configurations are loaded. Since each physical node may run multiple server instances, we memory-map the datasets when loading. This allows the physical node's host operating system to maintain an in-memory map of a file that can be given to each server instance. This reduces I/O costs and allows using multiple configuration files to reference the same dataset without additional overhead.

Attributes about the dataset from the configuration, such as transfer function or data variable, are kept alongside the reference to the data. Multiple configuration files may reference the same dataset, for example, using varying transfer functions. This flexibility allows for more power in the rendering requests.

### 3.2.3 Handling of Rendering Requests

After being routed from a unified endpoint to a specific Docker container, a rendering request is handled by a hyperimage server. Rendering requests from the client ask for an image URL in which various parameters are embedded. Image requests are processed by the C++ web server, built with the Pistache library [34], by first parsing the

options and then rendering the requested image using the OSPRay renderer.

Each incoming rendering request contains the dataset, camera position, up vector, the resolution of the render, and potentially time-step. Camera and renderer settings are updated accordingly.

OSPRay performs the rendering according to the above parameters. The life-cycle of the OSPRay rendering objects in each server are equal to that of the hyperimage server itself. Data and rendering attributes are pre-configured per volume during hyperimage server initialization. When the render completes, we composite the OSPRay framebuffer onto the appropriate background color and encode as a JPG image. There is no need to store the image to disk on the server, so the encoding is done to a byte stream in memory. At this point, all information about the camera position and other dynamic state parameters are no longer needed nor held.

The web server sends the rendered image as JPG byte stream (e.g., `image/jpeg` MIME type) from the rendering module. The Docker Swarm Manager, which routed the request to this container, handles responding to the appropriate user. *The hyperimage server itself remains oblivious to whom it has communicated with.*

### 3.2.4 Elastic System Operation

#### *Job Assignment and Runtime Management.*

Using a single container, rendering requests from  $n$  users will be queued up by the web server. Each request will occupy the container until rendering and network transfer of the image is complete. With multiple containers, any container available can be selected for any given rendering request. Sequential requests from a single user can be routed to different containers on different physical nodes. This has two main benefits: (i) new rendering requests can be processed while other requests are blocked for I/O, network transfer, or rendering; and (ii) elastic routing provides fault tolerance when a hyperimage server or physical node goes down.

The volume of rendering requests is variable over time and hard to predict. We monitor the current load on all containers and scale the number of containers up or down accordingly, through the runtime manager (RM) shown in Fig. 2.

Our RM, like RMs on typical cloud platforms, implement elasticity by periodically checking CPU usage across all containers, and start new containers or close idling containers as needed. In our previous work, we showed how Tapestry leveraged such auto-scaling on an institutional cluster [5]. In this work, we deploy Tapestry on Amazon AWS as a microservice and, to this end, benefit from Amazon's auto-scaling RMs transparently.

*Cache Container.* In each physical node, we have added an Nginx cache container intercepting all messages between hyperimage servers and the outside. In a completely transparent manner, this enables caching for the Tapestry microservice instances. Server responses are now cached based on the incoming request. This improves efficiency and scalability for many use cases. For example, commonly used view angles, isovalues, etc. in repeated batches of renderings for hypervideos and tiled renderings can now be simply reused, saving hyperimage servers to handle new rendering requests. Note that client-side caching inside web browsers also take place transparently by browsers themselves.



**Controllable Granularity.** Tapestry's server-side is a task-parallel engine. As known for task-parallel systems in general, the granularity of the tasks can affect the parallel efficiency of the overall system. In this work, we have added a tiling mechanism to Tapestry as an option so that an application can choose to use finer granularity to achieve better performance.

With tiling, a single hyperimage can be divided into many `<img>` tags on the client-side. Each tile represents a portion of the final render and is rendered on a different container in parallel to other tiles. Using tiling, the client-side creates a render request for each tile and sends them to the server-side. Once the response comes back, the appropriate `<img>` is updated with the result.

The setting `"tiling,TILE_NUMBER-N_TILES"` is an optional parameter in the rendering request to specify tiling. For example, `tiling,0-16` denotes that the rendering request is for the first tile out of a 16-tile render. Once this rendering request reaches a hyperimage server, the server calculates the portion of the volume that it needs to render and updates the OSPRay camera's clip space.

When rendered tiles are returned to the client-side, the tiles are placed in the DOM in their own corresponding `<img>` tag. Because each tile request can be sent independently and routed to the correct position in the hyperimage, there is no explicit compositing step required. That is, we provide *stitch-free* tiling.

**Multiple Endpoints.** Docker Swarm uses an Ingress load balancer [33]. The setup allows any physical node to be an endpoint for incoming requests. The requests are then routed to a free container. As a new addition, in this work, we have added support for multiple endpoints in the client (`tapestry.js`). The host parameter in a Tapestry object can be set to an array of host addresses. Endpoints are then chosen using a round-robin approach in the client in Tapestry objects. This achieves two purposes. First, the problem of bottlenecking at a node's inbound traffic is alleviated. Second, browsers typically only open a limited number of sockets per host address (e.g., Chrome currently defaults to opening 6 connections per destination host (endpoint) [35].) By using multiple endpoints, Tapestry objects can take advantage of more open sockets.

In the case of Amazon's cloud, AWS also has a load balancer that provides the same effect as Docker Swarm's and is called the Elastic Load Balancer (ELB). Multiple ELBs can target the same set of machines to provide a similar effect on AWS as on our institutional Docker Swarm. The address of the ELBs can be used as endpoints in Tapestry clients.

### 3.3 Deployment on Institutional Clouds

Tapestry's source code comes with a command-line interface (CLI) named `tapestry.sh` that simplifies setting up and running the backend on institutional clouds. Linux and Docker Swarm are the only requirements for running the Tapestry system. With Docker Swarm installed, users can simply run `./tapestry.sh build` and `./tapestry.sh run` to run the system. Since Tapestry is built inside Docker containers, the build is guaranteed to be successful on machines that run Docker. In that regard, Docker has simplified portability. The command-line interface also contains other sub-commands such as `scale` (for manually scaling the system), `example` (to download and run the

examples), `cache_report` (to view the number of cache hits and misses) among others. Extra features of the interface can be seen using the `help` subcommand.

### 3.4 Deployment on Amazon AWS as a Microservice

Although the achieved performance metrics on public clouds may be lower than on institutional clouds, public facing cloud platforms, such as Amazon AWS, provide true Internet-scale availability and accessibility at very affordable cost levels.

To create a Tapestry service on AWS from scratch, only a few steps are needed. AWS provides a load balancer that is instrumental in distributing rendering loads across multiple machines. For the setup, an AWS load balancer needs to be started with its listening port set to a publicly accessible port for the service; typically the default HTTP port 80. The load balancer must then be configured to forward traffic to some alternative port (e.g., 8,080).

After that, an AWS Elastic Container Service (ECS) service can be created. Tapestry's Docker image then needs to be uploaded to Amazon's cloud-based registry and needs to include any necessary data and configurations. The ECS service needs to point to this image and use the previously specified private port (8,080). Finally, the user needs to scale the service as necessary; often a higher number than would be used on an institutional cloud because AWS shares the resources with other users and services.

In studying the performance of Tapestry on Amazon AWS, we were mostly interested in choosing the optimal type of machine and measuring the price for a desired frame-per-second performance. In our tests, we spawned various numbers of different machines and sent rendering requests of different image sizes and measured the round trip time. As a summary of the outcome, we found to support a large number of simultaneous users, using a large number of small T2 type instances is more cost effective. However, for super resolution renderings for a few users, the Compute-Optimized machines are more suitable. More detailed results are shown in Section 5.4. Additionally, to simplify usage on cloud services, we have released a Docker image of Tapestry.<sup>2</sup>

## 4 APPLICATION DEVELOPMENT

In this section, we describe three application development settings enabled by using the Tapestry microservice. Specific application performance results are in Section 5.6.

### 4.1 Embedding Visualizations into Web Pages

Hyperimages can be easily added to a web page using HTML tags and a short JavaScript function call. To integrate hyperimages into a page, the developer must include the `tapestry.js` file and its dependencies: `arcball.js`, `sylvester.js`, `math.js` and `jQuery.js`. Then, one line of JavaScript needs to be called to initialize all hyperimages: `$(".hyper-image").tapestry();`

This call creates a Tapestry object per hyperimage tag. Parameters such as default size of the hyperimage and camera position can be sent to the object through the constructor.

2. <https://hub.docker.com/r/seelabutk/tapestry>

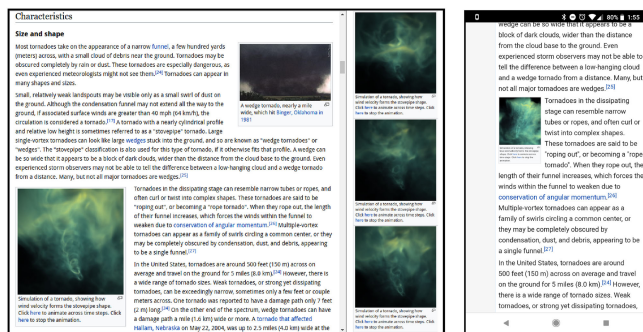


Fig. 5. Left: Embedded a volume rendering of tornado (dataset details in Table 3) in a Wikipedia page on tornadoes. Users can start and stop an animated temporal sequence. Right: The same page also works on mobile phones. The page used to hold a static image showcasing the shape of a stovepipe tornado. Now users can interactively see the temporal progression of the natural phenomenon.

#### 4.1.1 Time-Varying Data Animation (Wikipedia Example)

Listing 5 shows the changes needed to include a hyperimage of a time-varying dataset into a Wikipedia page.

#### Listing 5. Code for Adding a Hyperimage of a Time Varying Simulation into the Wikipedia Tornado Page

```
$( ".hyperimage" ).tapestry({
  "host": "http://host.com:port/",
  "width": 256, "height": 256, "zoom": 300,
  "n_timesteps": 20
});
<img id="timeseries" class="hyperimage"
  data-volume="tornado" data-timerange="0..20"/>
<a class="hyperaction" for="timeseries"
  data-action="play()" "></a>
<a class="hyperaction" for="timeseries"
  data-action="stop()" "></a>
```

#### Listing 6. Code Needed to Insert the Four Linkable Hyperimages and Hyperaction into NASA's Supernova Web Page

```
<script>
  $( ".hyperimage" ).tapestry({
    "host": "http://host.com:port/",
    "width": 128, "height": 128, "zoom": 300
  });
</script>
<img id="s1" class="hyperimage" data-dataset="nova1" />
<img id="s2" class="hyperimage" data-dataset="nova2" />
<img id="s3" class="hyperimage" data-dataset="nova3" />
<img id="s4" class="hyperimage" data-dataset="nova4" />
<a class="hyperaction" for="s1"
  data-action="link(s2,s3,s4)" "></a>
```

Fig. 5 shows the Wikipedia page on tornadoes after the modification. The page includes a hyperimage linked to a series of time steps from a tornado simulation dataset. Two hyperactions can be seen in the code. Users can click a hyperaction to play or stop the animation, while still having the ability for 3D interaction with the volume rendering.

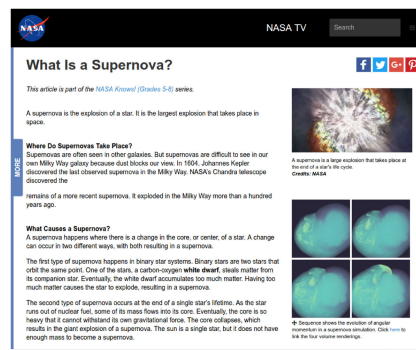


Fig. 6. Embedding four time steps of a supernova simulation into a NASA educational web page (dataset details in Table 3). The four hyperimages (bottom right) can be linked or unlinked using the hyperaction in the caption below it. Previously, the page had only a static figure (top right) showing an artist's rendition. Now users can also interactively explore how a supernova evolves over time.

#### 4.1.2 Multiple Linked Views (NASA Example)

Here we show a NASA educational outreach page explaining supernovae. The relevant code changes are in Listing 6. The modified page is shown in Fig. 6.

The page now contains four hyperimages showing consecutive time steps of a supernova simulation. The views can be linked and unlinked with the hyperaction in the caption. When linked, all four hyperimages move together when a user interacts with any one of them.

## 4.2 Controllable Movies of Scientific Visualization

By unifying the interface of the Tapestry microservice as simple rendering requests, we can achieve more complex application logic, for example, for making movies of scientific visualization.

#### Listing 7. Sample Script for a Hypervideo with Two Keyframes

```
<script id="video" type="text/json">
{
  "keyframe0": {
    "rotation": [-0.72, 0.30, 0.62, 0.51, 0.83, 0.19, -0.46, 0.45, -0.75],
    "zoom": 500, "timestep": 0, "isovalue": 0.2
  },
  "keyframe1": {
    "rotation": [0.44, -0.16, 0.88, 0.43, 0.90, -0.05, -0.78, 0.40, 0.46],
    "zoom": 200, "timestep": 20, "isovalue": 0.7
  }
}
</script>
<div class="hypervideo" data-keyframeid="video"
  data-dataset="supernova"></div>
```

Traditionally, making a visualization movie requires creating the keyframes first. Then, a movie is created by rendering all of the intermediate frames sequentially. Making changes to an already-made movie requires a user to have access to significant computing resources, and is usually a very time consuming process.



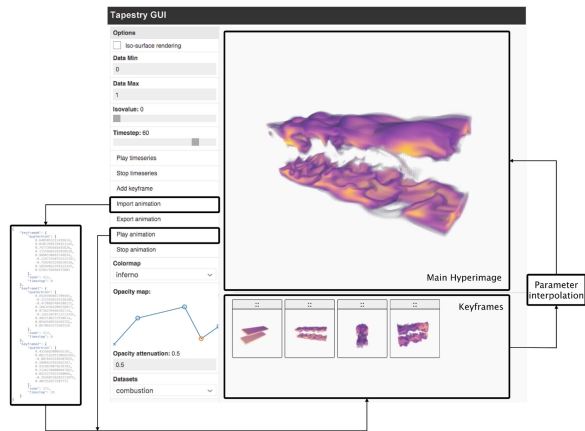


Fig. 7. A webpage for creating and manipulating hypervideos. A user can add keyframes, edit existing keyframes, and export movies. Editing the movie can be to modify camera angle, time step, isovalue, color map, etc.

Using the Tapestry microservice, we can make the movie-making process interactively controllable by a user from within a simple web browser. While offloading all rendering tasks to the microservice, we simplify the application space of the movie-making process to just the textual representations of the keyframes (i.e., the corresponding rendering requests). We call these application-space constructs, *hypervideos*.

Hypervideos can be embedded in HTML with the `class` attribute set to `hypervideo`, and their `data-keyframes` set to a JSON file. Alternatively, developers can set the `data-keyframeid` attribute to the id of a script tag that contains the JSON. Listing 7 shows an embedded hypervideo with two keyframes.

Using and interacting with hypervideos is different from traditional movies in important ways.

First, each keyframe can be presented on a web page as a hyperimage, which has all of the interactivity described in Section 4.1, including allowing the user to alter the keyframe by changing the view. The generation of intermediate frames is automatic. We use linear interpolation for changes in timesteps, isovalues, and zoom levels; we interpolate camera rotations using *slerp* [36].

Second, in a traditional movie, only the keyframes are controllable. In contrast, due to the Tapestry microservice treating all rendering requests in the same way, we turn each individual frame in the movie into a hyperimage. In this way, when a viewer watches the movie, he or she can pause the movie at any time to interact and navigate around the dataset freely.

Third, because of the microservice's availability, the movie, i.e the hypervideo, can remain text-only, and hence remain compact, easily editable, sharable, and version controlled. In addition, while changing number of frames, screen resolution, splitting and re-joining movies etc., are hard for traditional movies, they are trivial tasks for hypervideos.

For creating hypervideos, Fig. 7 shows a GUI that is essentially a web page. A user can interactively add and control the key frames. When the keyframes are set, the user can play the animation or export the video in the form of JSON text or as MP4 (rendered and encoded server-side using *ffmpeg*). At all times, the Tapestry microservice serves as the rendering engine.



Fig. 8. A volume rendering of the turbine blade dataset shown through HoloLens.

The performance of hypervideo renderings is presented in Section 5.6.

### 4.3 Augmented Reality and Power-Wall

The endpoints of the Tapestry microservice is served by Docker Swarm following standard HTTP protocols. This kind of generality allows any application to simply access the endpoints (e.g., via Linux's *curl*). When using the Tapestry microservice, the application space does not have to be related to web browsers at all. We further provide two demonstrative examples as follows.

For the first example, we developed HoloTapestry, a C# application for augmented reality using the Tapestry microservice. This prototype runs on a Microsoft HoloLens device and performs stereo renderings using two textured planes, rotated so they stay normal to the viewer's eyes. Each plane independently updates its texture by making rendering requests to the microservice based on the current camera parameters from the HoloLens. Transparency is achieved by setting the background color of the renders to black as is standard in HoloLens applications.

In result, Tapestry microservices allowed us to deliver volume renderings of a 7.5 GB dataset to an AR device with 2 GB of memory by writing about 100 lines of code. Fig. 8 shows a view of the turbine blade dataset on a desk. The performance of HoloTapestry is in Section 5.6. HoloTapestry is open-source.<sup>3</sup>

For the second example, we target power-wall displays, which is arguably one of the most prized tools for demonstrating advances in science and engineering. Traditionally, each power-wall facility is accompanied by its own computing cluster. Due to the typical tiled nature of power-walls, producing super-resolution renderings using the Tapestry microservice is straightforward. One can use a short Shell script that batch-generates rendering requests through *curl*. Or, one can run a web browser across the power-wall and have the browser transparently issue the batch of rendering requests, one per each tile in the image, in order to achieve parallel acceleration on the server side. In both cases, a lightweight single-node can deliver data-intensive visualizations onto the whole power-wall.

Fig. 9 shows a user using Tapestry to inspect defects in a 3D printed wind turbine blade on a  $4 \times 3$  power-wall display. The volume is created by scanning the actual 3D

3. <https://github.com/seelabutk/holotapestry>



Fig. 9. A user using Tapestry to inspect a 3D printed wind turbine on a  $4 \times 3$  power-wall. Renderings are  $2048 \times 2048$  in resolution.

printed model using neutron scattering [37]. The renderings are at  $2048^2$  resolution, rendered in  $256$  tiles ( $128^2$  pixels per tile) in parallel. The tiles are synchronized using a global barrier. Tiled-based performance is detailed in Section 5.1.

## 5 RESULTS AND DEPLOYMENT

Our testing platforms include our institutional cloud and Amazon AWS instances. Our institutional cloud setup includes three machines each with 24 physical cores (dual-socket Xeon E5-2650 v4, 2.9 GHz, 128 GB memory) and three machines each with 28 cores (dual-socket Xeon E5-2650 v4, 2.9 GHz, 256 GB memory).

On AWS, we tested seven different types of instances. Table 2 shows the detailed list. The “d” suffix (e.g., c5d.xlarge) refers to AWS instances with SSDs. For our system, the SSDs do not affect the runtime performance, only micro-service initiation time.

Our testing includes: (i) using 1 single container to serve 1 rendering request (Section 5.2), (ii) using an institutional cluster to serve a varying number of emulated streams of rendering requests (Section 5.3), (iii) using Amazon AWS cloud to serve a varying number of emulated request streams (Section 5.4), (iv) using AWS cloud to serve a varying number of simulated users (Section 5.5), and (v) performance of demonstrative applications as experienced by a user (Section 5.6).

Among the above tests, (i) - (iii) are to understand how the Tapestry server performs, independent of user behavior. (iv) is to understand the quality of service received by a cohort of simultaneous users performing exactly the same kinds of operations. (v) is to understand how a single user experiences applications supported by the Tapestry micro-service. Note that end-users are not affected by dataset load time in these tests because all datasets are pre-loaded before the service starts.

### 5.1 Configuring the Tapestry Microservice

This section discusses application policies to consider when deploying Tapestry on the cloud.

When deploying on Amazon AWS, because virtual instances have to share their physical nodes with others, Amazon by default sets a low cap on the number of containers. For example (as shown in Table 2), on c5d.18xlarge (with 72 vCPUs), the Amazon imposed container count cap is 14, which translates to a 0.2 container/core ratio. Because

TABLE 2  
Amazon AWS Instances Used in this Work

Instance	Core Cnt	Memory	# Containers
t2.micro	1 vCPU	1 GiB	1
t2.medium	2 vCPUs	4 GiB	2
c5d.large	2 vCPUs	4 GiB	2
c5d.xlarge	4 vCPUs	8 GiB	3
c5.2xlarge	8 vCPUs	16 GiB	3
c5d.2xlarge	8 vCPUs	16 GiB	3
c5.9xlarge	36 vCPUs	72 GiB	7
c5d.18xlarge	72 vCPUs	144 GiB	14

The t2 prefix (e.g., t2.micro) refers to general purpose instances, while the c5 prefix refers to compute optimized instances. The containers column shows the maximum number of containers allowed by AWS on each particular instance.

this is much lower than the 0.8 ratio on institutional cloud (explained in Section 5.3), we use the max number of containers allowed by AWS.

For applications to run optimally on the cloud, there are three accelerations to consider, all of which are independent of Tapestry. Instead, they are solely application-side policies.

First, use tiling. Instead of sending a rendering request for a  $1024^2$  image, send 16 rendering requests of  $256^2$  tiles. These per-tile rendering requests will be answered by the Tapestry microservice in parallel. For example, a t2.medium instance has 2 vCPU and 2 GB memory, each available for 4.6 cents/hour. It’s easily affordable, and beneficial for fault tolerance, to get a cohort of 100 t2 mediums to use for Tapestry.

We have found a simple and general heuristic to set tiling factor to 16. A tiling factor of 4 still limits the amount of parallelism that can be exploited. A tiling factor of 64 creates too much management overhead for the client. Based on our tests, a tiling factor of 16 reliably leads to 3 to 4 times faster rendering performance, as compared to when tiling is not used. Tile size or image size of  $64^2$  or smaller is too fine grained. In all our demo applications, we lower bound tile size to  $128^2$ .

Second, use a lower interaction-resolution and a higher viewing-resolution. As discussed in Section 2.3, level-of-detail is very effective to ensure user-experience. Specifically, when needing a visualization at a viewing resolution of  $1024^2$ , during interaction for faster response time, it is helpful to use a lower interaction resolution. Regardless of whether rendering for interaction- or viewing-resolutions, all of our demo applications use tiling (to benefit from parallel server-side rendering).

Third, use multi-threaded downloading. Most modern web browsers implement this by default. For example, Chrome automatically opens 6 asynchronous socket connections for each destination host. When accessing Tapestry from a non-browser client (e.g., curl), we have also found parallel connections helpful.

Hence, we have set up our tests of Tapestry microservices, in Sections 5.4, 5.5, and 5.6, using the following assumptions: (1) each user has 6 concurrent request streams, (2) tile-based rendering requests, (3) when testing for user experience, use a viewing-resolution of  $1024^2$  and a interaction-resolution of  $256^2$ .

### 5.2 Rendering Pipeline Performance

We benchmarked the rendering and encoding process using three variables that affect render time: image size, level of

TABLE 3  
The Datasets Used in this Work

Dataset	Size per Volume	Spatial Resolution	Time Steps
Boston teapot with lobster	45 MB	$356 \times 256 \times 178$	1
Isotropic turbulence [39]	64 MB	$256 \times 256 \times 256$	1
Jet flames [40]	132 MB	$264 \times 396 \times 66$	122
Superstorm [41] (1 run)	201 MB	$254 \times 254 \times 37$	49
Tornado [42] (wind velocity)	257 MB	$480 \times 480 \times 290$	600
Supernova [30]	308 MB	$432 \times 432 \times 432$	60
Magnetic reconnection [43]	512 MB	$512 \times 512 \times 512$	1
Turbine blade [37]	7500 MB	$1589 \times 698 \times 1799$	1

For time-varying data, varying time steps were used during testing.

attenuation of a ramp opacity map, and number of samples per pixel. We used 6 image sizes ( $64^2$ ,  $128^2$ ,  $256^2$ ,  $512^2$ ,  $1024^2$ , and  $2048^2$ ), 4 attenuation values (1.0, 0.5, 0.1, and 0.01), and 4 sampling rates (1, 2, 4, and 8). The target hardware was a 24-core node of our institutional cluster with a single container. We then tested each combination of these parameters, resulting in 96 test cases. We repeated each of the 96 cases 10 times with the camera at a randomized positions to simulate the effects of the volume being at different distances and angles. We calculate the average time taken for 10 renders for a given test case. To see the effect of image sizes, we then averaged the times for each image size. This simulates possible variation in image quality within same-sized images.

The target datasets were: supernova, isotropic turbulence, and magnetic reconnection (described in Table 3). All three datasets are structured grids of floating point values. To measure rendering time, each image was rendered to OSPRay's internal framebuffer and was then discarded to avoid buffer copy or encoding time. We then tested the encoding time (without saving to disk) separate from render time. Results are shown in Table 4. Note that rendering time does not necessarily increase linearly with image size (a known characteristic of ray-tracing [38]).

The fastest rendering case was unsurprisingly  $64^2$  image size. Within the test cases that used a  $64^2$  image, attenuation of 0.1 and sample rate of 1 resulted in the fastest renders at 0.001 seconds, approximately 1,000 frames per second. On the other hand, the slowest renders occurred with  $2048^2$  images.

We also compared the encoding time of PNG vs JPG (at 100 percent quality). PNG was the image format used in our previous work [5]. On average, JPG was 2.5 times faster in encoding than PNG and generated byte streams were generally smaller.

TABLE 4  
Average Benchmarking Results for Rendering Requests Using the Supernova, Isotropic Turbulence, and Magnetic Datasets

Image size	Rendering time (s)	PNG Enc. time (s)	JPG Enc. time (s)	Round-trip time (s)
$64 \times 64$	0.003	0.005	0.003	0.009
$128 \times 128$	0.004	0.011	0.005	0.016
$256 \times 256$	0.009	0.035	0.012	0.030
$512 \times 512$	0.024	0.122	0.037	0.092
$1024 \times 1024$	0.083	0.452	0.147	0.284
$2048 \times 2048$	0.338	1.651	0.580	1.066

The round-trip time for each request includes render, encode, and transfer time to and from the server with JPG encoding.

Performance Results for a Single Container

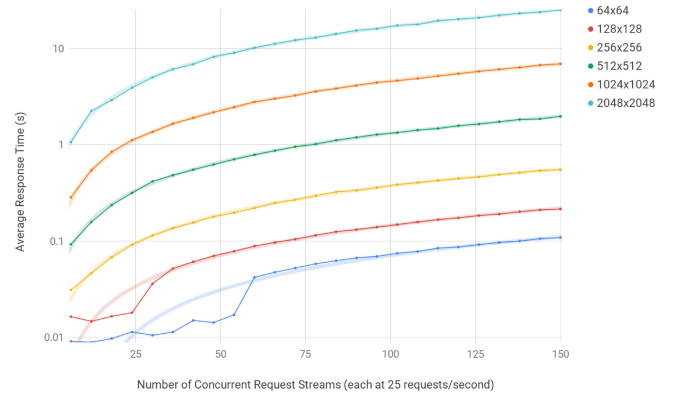


Fig. 10. System throughput results showing request rate versus response time for various image sizes in log scale. The linear regression trendlines are over-plotted indicating the linear growth of response time in relation to the number of concurrent request streams.

In our experiments, the size of the rendered images varied between a few kilobytes for low resolutions up to under 300 KB for  $2048^2$  images. The exact size of the generated images depends on the content of the rendering.

### 5.3 Tapestry Server Throughput

In order to evaluate our system's throughput, we implemented a stress test of Tapestry microservices running on our institutional cluster. We orchestrated multiple test machines to send rendering requests to Tapestry simultaneously. In other words, each test machine sends a different request stream to the server.

The testing master starts by spawning testing workers on the test machines. The master then waits until all test workers have finished their tests. Test workers use `curl` to send rendering requests at a rate of 25 requests/second, while randomly changing rendering parameters (e.g., camera position) for each request. Finally, the master reads off the test logs from a shared queue and saves to disk. The logs list request-sent and response-received times that allow us to measure the average time it takes our system to respond to rendering requests. This throughput testing suite is written in Python and is included in the Tapestry repository.

To increase the load on the system, we simply increase the number of test workers. Like in Section 5.2, initially our test target was one Tapestry container in a single 24 core node of our cluster. We ran each test 100 times on the supernova, turbulence and tornado datasets (Table 3). For each dataset, we generated rendering requests for six image sizes:  $64^2$ ,  $128^2$ ,  $256^2$ ,  $512^2$ ,  $1024^2$ , and  $2048^2$ .

We then averaged the response time collected, to show an overall system throughput under a mixture of different sizes of rendering jobs. Fig. 10 shows the scaling curves for various image sizes. When doubling image size, average response time approximately increased by a factor of 4, which is expected.

Then, we tested for the effect of the number of containers per node. In this test, we kept the number of testing workers constant (150), and varied the number of Tapestry containers. Fig. 11 shows the results for three image sizes. For all image sizes, as we gradually increase the number of containers from 1 towards 20, average response time improves.



Performance Results On a Varying Number of Containers

150 Concurrent Request Streams

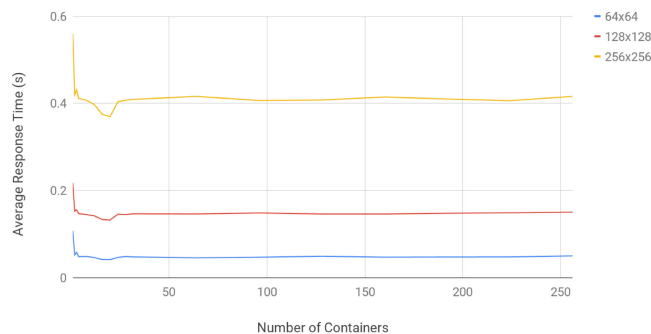


Fig. 11. Results showing the relationship between the number of containers in our institutional cloud and average response time. The optimal number of containers is shown to be 20 for a machine with 24 physical cores.

After reaching 20 containers, adding more containers did not yield noticeable improvements.

With the hardware being a single node with 24 physical cores, getting best performance with roughly 20 containers suggests roughly a 0.8 container/core ratio. Through additional testing, we found this ratio to be quite consistent on institutional cloud.

#### 5.4 AWS Microservice Throughput

Next, we evaluated Tapestry's performance on Amazon AWS. In particular, we looked at the relationship between FPS versus Price over various tile sizes:  $64^2$ ,  $128^2$ , and  $256^2$ .

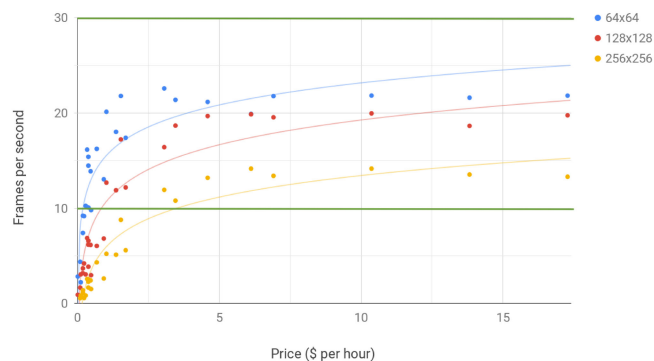
Since Tapestry is a compute-intensive service, we tested Amazon's Compute-Optimized instances as well as T2 Performance instances [44]. We chose T2 machines because of their ability to sustain CPU workload and low costs [44]. For each instance type, we ran different number of machines. For more powerful machines we were limited to lower quantities due to Amazon's policies.

For the supernova dataset, Fig. 12 shows FPS vs Price for 6 and 120 concurrent request streams with all of our tested AWS instance types. Each point in the scatter plot represents an AWS instance type and configuration.

For example, Fig. 12-top shows the cost to sustain 10 FPS when rendering tiles of  $256^2$  is approximately \$4/hour. Please note, tiling lets applications transparently leverage

FPS vs. Price In Various AWS Instances

6 Concurrent Request Streams



FPS vs. Price In Various AWS Instances

120 Concurrent Request Streams

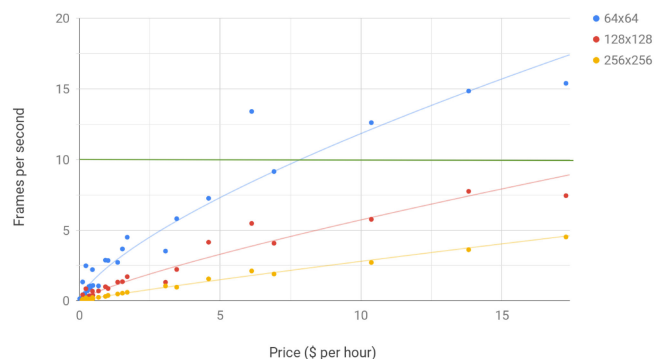


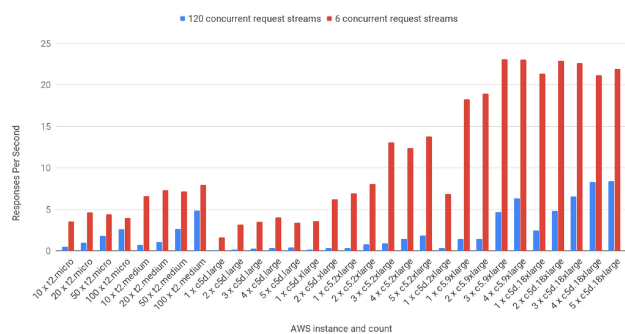
Fig. 12. Graphs showing FPS versus price on Amazon AWS for 6 (top) and 120 (bottom) concurrent request streams. Each point in the scatter plots belong to a different AWS instance and configuration. 10 FPS and 30 FPS are marked in green.

server-side parallel rendering; when an application requests tiles of  $256^2$ , the target image resolution is actually  $1024^2$ .

To evaluate the choices of AWS instances, we used 120 concurrent request streams and a tile size of  $128^2$  (i.e., targeting a typical desktop visualization resolution of  $512^2$ ). Fig. 13a shows the performance of different instance types in blue for 120 streams. The cost of these instances can be seen in Fig. 13b. It appears that the cost correlates quite well with the desired FPS. The two graphs also show that although large Compute-Optimized machines (towards the

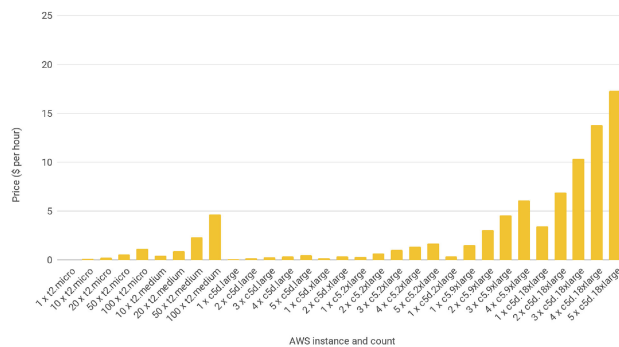
Performance of Various AWS Instances

6 and 120 concurrent request streams



(a)

Price of Various AWS Instances



(b)

Fig. 13. (a) Shows a comparison between the rendering performance of various AWS instances for 120 and 6 concurrent request streams (both at a request rate of 25 FPS). In a Chrome browser that uses 6 request streams per host, the former results in 20 users while the latter results in 1 user. Compute-optimized instances perform better with 6 request streams. (b) Shows the cost of different AWS instances.

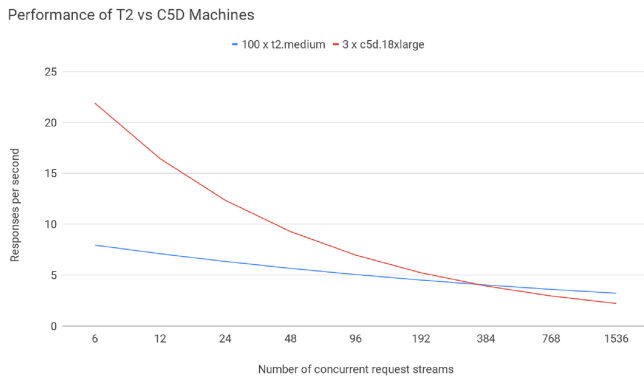


Fig. 14. Graph showing the estimated point at which T2 instances surpass compute-optimized instances at efficiency.

right) perform better, they are less cost-efficient. A reason may be that larger machines are more suited for fewer users and large tile sizes. Fig. 13a shows that by lowering the number of request streams to 6 (red bars), the rendering speed of the Compute-Optimized instances grew much more than a large number of smaller machines such as 100 *t2.medium* instances.

Furthermore, we compared the performance of 100 *t2.medium* machines and  $3 \times 72$  core *C5D.18xlarge* machines. Based on the changes in the number of concurrent requests from 6 to 120, we used the least squares fitting model to estimate where the performance of the two meet. The fitness of the model had a root mean square error of 0.019. Fig. 14 shows that at 380 concurrent request streams (i.e., about 60 simultaneous uses), 100 *t2.medium* instances become more cost-efficient.

## 5.5 User Experience Benchmarking

To test our system's performance under realistic workloads, we used "monkey testing", a standard approach to stress-test web pages. Monkey testing involves simulating interactions across elements of the page. We used this on hyperimages to simulate user interaction. We ran the "natural monkey testing" scripts in the same configuration as before [5], only that in this work 100 Amazon *t2.micro* instances were acting as testing clients. The datasets used were supernova, turbulence and magnetic (Table 3).

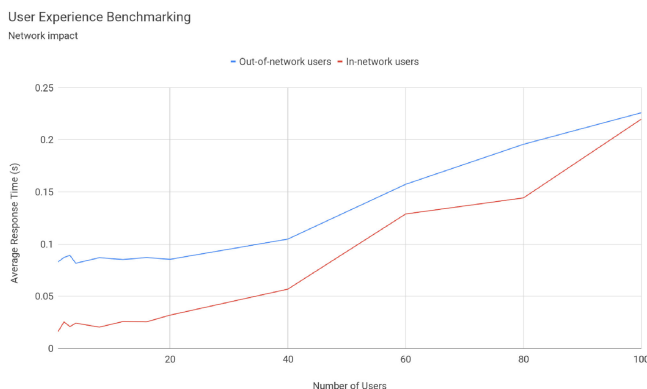


Fig. 15. Response time for a varying number of users is shown. The slower out-of-network results are from 100 simulated users on AWS, accessing our institutional cloud. The in-network results are from 100 simulated users in our local 1 Gbps network.

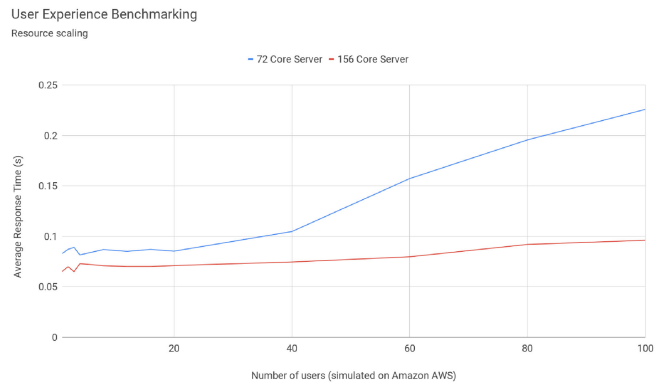


Fig. 16. Graph showing the scalability of the system. The 72 core cluster is the same as the one used in our previous work [5]. In both cases, we used 100 users (simulated on AWS with monkey-testing).

Each of the *t2.micro* instances ran a lightweight version of Ubuntu and a headless Chrome browser. Our testing script used SSH to connect to all 100 instances and run our hyperimage test page within the headless browser, and with monkey testing controlling the interactions.

When the monkey testing interactions were done, the JavaScript code within the page sent timing results to a simple Python server that log the results to a file. The timing results included request times, response times, and the resolution of requested images. On average, 3.46 percent of the images were at viewing resolution ( $1024^2$ ), and the rest were interaction resolution ( $256^2$ ).

Fig. 15 shows average response time for a varying number of testing clients. The blue line shows when the testing clients are deployed on Amazon AWS, and the red line shows when the testing clients are on the local area network as the institutional cluster. The result shows diminishing differences due to network proximity as the number of testing clients increase, which can lead to network congestion regardless of proximity.

Fig. 16 shows the same test repeated to reveal resource-scalability of our platform. We expanded the deployment from 3 nodes (72 cores, blue curve) to 6 nodes (156 cores, red curve). In both of these two cases, the testing clients were deployed on AWS.

## 5.6 Application Performance

To test the performance of the applications in (Section 4) with a single user, we used three *C5.9xlarge* AWS instances as server.

For hyperimage embedding, we conducted a single monkey-testing user test on a web page with a visualization of a dataset selected randomly (full list in Table 3). On average, interaction-resolution renderings ( $256^2$ ) were rendered at a speed of 9.43 FPS, while viewing-resolution renderings ( $1024^2$ ) achieved 2.08 FPS. In other words, when a user stops interacting, a high quality rendering is provided in less than 0.5 seconds.

We also looked at the overhead of including the client-side JavaScript code for Tapestry. On average, pages with Tapestry enabled loaded 1.29 times slower than pages without Tapestry included. For example, a Wikipedia page without hyperimages, loaded in 510 ms, while with hyperimages, it took 659 ms. Most of this overhead is due to the jQuery library.

Hypervideo performance essentially depends on the server throughput since interpolation has a negligible cost. In our tests, we created three hypervideos for different datasets (5 keyframes each). We chose to generate 50 frames between every two keyframe and therefore 200 frames were rendered for each video. Our video playback speed was set to 30 frames per second; the 200 frame videos were approximately 6 seconds long. The keyframes were chosen at random with different angles, and zoom levels. On average, it took 70.66 seconds to render a full video.

When changing one of the keyframes, on average, the readjustment of a keyframe took 21.15 seconds, since most of the intermediate frames were auto-cached by the cache container (Section 3.2.4). A user can watch the video as it renders albeit at the rendering speed. Any subsequent playback is at 30 frames/second. All hypervideo tests were done using a resolution of 1024<sup>2</sup>.

We also tested the speed of our augmented reality application. To view volume renderings of the 7.5 GB sized turbine dataset on a HoloLens (Fig. 8), HoloTapestry can update renderings at a sustained speed of 4.5 FPS. The viewing-resolution in the tests was 512<sup>2</sup> (stereo, without explicit synchronization of left and right eye images), using all 6-nodes of our institutional cluster. While the speed of our prototype implementation is not sufficient for practical use yet, we believe as hardware performance on AR devices improves, better results can be achieved, and HoloTapestry can be utilized in situations where the data is large and cannot be rendered on the device.

## 6 CONCLUSION AND FUTURE WORK

Traditionally, the computing resources that can be provisioned for a scientist restricts the kind of scientific visualization he or she can use. This limitation also hampers efforts to make scientific visualizations accessible to large collaborating teams of users. Moreover, there are even more barriers to share interactive scientific visualizations with the general public.

In this work, we describe Tapestry as an example to map the delivery architecture of scientific visualization onto a cloud platform, and have scientific visualization appear as a microservice with rendering requests being the only API.

Tapestry's architectural design stems from decoupling and shielding the visualization server-side away from application logic and application states. Through deploying the stateless Tapestry servers on Amazon AWS, we show that high-end visualization needs can be met by a cloud-hosted service in an efficient, on-demand, and cost-effective manner.

We believe there is a future potential that general data analysis and visualization tasks, beyond simple rendering, can leverage decoupled architectures in similar ways to achieve high performance, high availability, and high accessibility. In particular, we would like to further combine Tapestry with the area of augmented reality as well as improve its performance with AR applications. Another area for pursuing in the future is adding specialized support for 3D graphics and animation development. For general computer graphics concepts, dedicated rendering engines have been the norm. It may be feasible to use microservices to support such use cases on-demand.

## 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 US National Science Foundation Award CNS-1629890, Intel Parallel Computing Center (IPCC) at the Joint Institute of Computational Science of University of Tennessee, and the Engineering Research Center Program of the National Science Foundation and the Department of Energy under US National Science Foundation Award Number EEC-1041877.

## REFERENCES

- [1] S. Jourdain, U. Ayachit, and B. Geveci, "ParaViewWeb, a web framework for 3D visualization and data processing," in *Proc. IADIS Int. Conf. Web Virtual Reality Three-Dimensional Worlds*, 2010, Art. no. 1.
- [2] V. Pascucci, G. Scorzelli, B. Summa, P.-T. Bremer, A. Gyulassy, C. Christensen, S. Philip, and S. Kumar, "The visus visualization framework," in *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, E. W. Bethel, H. Childs, and C. Hansen, Eds. London, U.K./Boca Raton, FL, USA: Chapman and Hall/CRC, 2012.
- [3] K. Sons, F. Klein, D. Rubinstein, S. Byelozyorov, and P. Slusallek, "XML3D: Interactive 3D graphics for the web," in *Proc. 15th Int. Conf. Web 3D Technol.*, 2010, pp. 175–184.
- [4] Paraview ArcticViewer, 2018. [Online]. Available: <https://kitware.github.io/arctic-viewer/>, Accessed on: Oct. 14, 2018.
- [5] M. Raji, A. Hota, and J. Huang, "Scalable web-embedded volume rendering," in *Proc. IEEE 7th Symp. Large Data Anal. Vis.*, Oct. 2017, pp. 45–54.
- [6] Software Container Platform - Docker: <https://www.docker.com/>, 2018. [Online]. Available: <https://www.docker.com/>, Accessed on: Oct. 14, 2018.
- [7] M. Bostock, V. Ogievetsky, and J. Heer, "D3 data-driven documents," *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 12, pp. 2301–2309, Dec. 2011.
- [8] J. Jomier, S. Jourdain, U. Ayachit, and C. Marion, "Remote visualization of large datasets with midas and ParaViewWeb," in *Proc. 16th Int. Conf. 3D Web Technol.*, 2011, pp. 147–150. [Online]. Available: <http://doi.acm.org/10.1145/2010425.2010450>
- [9] G. Tamm and P. Slusallek, "Plugin free remote visualization in the browser," in *Proc. SPIE/IS&T Electron. Imag.*, 2015, pp. 939 705–939 705.
- [10] G. Tamm and P. Slusallek, "Web-enabled server-based and distributed real-time ray-tracing," in *Proc. 16th Eurographics Symp. Parallel Graph. Vis.*, 2016, pp. 55–68.
- [11] Kitware, "Vtk.js," 2017. [Online]. Available: <https://github.com/Kitware/vtk-js>, Accessed on: Jun. 10, 2017.
- [12] J. Ding, J. Huang, M. Beck, S. Liu, T. Moore, and S. Soltesz, "Remote visualization by browsing image-based databases with logistical networking," in *Proc. ACM/IEEE Conf. Supercomput.*, 2003, pp. 34:1–34:11.
- [13] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen, "An image-based approach to extreme scale in situ visualization and analysis," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 424–434.
- [14] 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*. Boca Raton, FL, USA: CRC Press, Oct. 2012, pp. 357–372.
- [15] U. Ayachit, "The Paraview guide: A parallel visualization application," Kitware, Inc., 2015.
- [16] Q. Wu, J. Gao, M. Zhu, N. S. Rao, J. Huang, and S. Iyengar, "Self-adaptive configuration of visualization pipeline over wide-area networks," *IEEE Trans. Comput.*, vol. 57, no. 1, pp. 55–68, Jan. 2008.
- [17] R. Sisneros, C. Jones, J. Huang, J. Gao, B.-H. Park, and N. Samatova, "A multi-level cache model for run-time optimization of remote visualization," *IEEE Trans. Vis. Comput. Graph.*, vol. 13, no. 5, pp. 991–1003, Sep./Oct. 2007.



- [18] M. Meißner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis, "A practical evaluation of popular volume rendering algorithms," in *Proc. IEEE Symp. Volume Vis.*, 2000, pp. 81–90.
- [19] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navrátil, "OSPRay—A CPU ray tracing framework for scientific visualization," *IEEE Trans. Vis. Comput. Graph.*, vol. 23, no. 1, pp. 931–940, Jan. 2017.
- [20] NVIDIA IndeX, 2016. [Online]. Available: <https://developer.nvidia.com/index>
- [21] C. Zach, S. Mantler, and K. Karner, "Time-critical rendering of discrete and continuous levels of detail," in *Proc. ACM Symp. Virtual Reality Softw. Technol.*, 2002, pp. 1–8.
- [22] X. Li and H.-W. Shen, "Time-critical multi-resolution volume rendering using 3D texture mapping hardware," in *Proc. IEEE/ACM Symp. Volume Vis. Graph.*, 2002, pp. 29–36.
- [23] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "VisTrails: Enabling interactive multiple-view visualizations," in *Proc. IEEE Vis.*, 2005, pp. 135–142.
- [24] J. Gao, J. Huang, C. R. Johnson, and S. Atchley, "Distributed data management for large volume visualization," in *Proc. IEEE Vis.*, 2005, pp. 183–189.
- [25] H. Yu, K.-L. Ma, and J. Welling, "A parallel visualization pipeline for terascale earthquake simulations," in *Proc. ACM/IEEE Supercomput. Conf.*, 2004, pp. 49–49.
- [26] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke, "Condor-G: A computation management agent for multi-institutional grids," in *Proc. 10th IEEE Int. Symp. High Perform. Distrib. Comput.*, 2001, pp. 55–63.
- [27] W. Kendall, J. Huang, T. Peterka, R. Latham, and R. Ross, "Toward a general I/O layer for parallel-visualization applications," *IEEE Comput. Graph. Appl.*, vol. 31, no. 6, pp. 6–10, Nov./Dec. 2011.
- [28] A. Evans, M. Romeo, A. Bahrehmand, J. Agenjo, and J. Balt, "3D graphics on the web: A survey," *Comput. Graph.*, vol. 41, pp. 43–61, Jun. 2014.
- [29] W3C, "Embedding custom non-visible data with the data attributes," 2017. [Online]. Available: <https://www.w3.org/TR/2011/WD-html5-20110525/elements.html#embedding-custom-non-visible-data-with-the-data-attributes>, Accessed on: Mar. 21, 2017.
- [30] J. M. Blondin and A. Mezzacappa, "Pulsar spins from an instability in the accretion shock of supernovae," *Nature*, vol. 445, no. 7123, pp. 58–60, 2007.
- [31] C. Pahl, "Containerization and the PaaS cloud," *IEEE Cloud Comput.*, vol. 2, no. 3, pp. 24–31, May/Jun. 2015.
- [32] J. Stubbs, W. Moreira, and R. Dooley, "Distributed systems of microservices using docker and serfnode," in *Proc. 7th Int. Workshop Sci. Gateways*, 2015, pp. 34–39.
- [33] P. Kanuparth, W. Matthews, and C. Dovrolis, "DNS-based ingress load balancing: An experimental evaluation," *CoRR*, vol. abs/1205.0820, 2012. [Online]. Available: <http://arxiv.org/abs/1205.0820>
- [34] Mathieu Stefani, "Pistache http server," 2017. [Online]. Available: <http://pistache.io>, Accessed on: Jun. 16, 2017.
- [35] "Maximum Number of Open Connections Per Browser," (2018). [Online]. Available: <http://www.browserscope.org/?category=network&v=top>, Accessed on Oct. 2018.
- [36] K. Shoemake, "Animating rotation with quaternion curves," *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 245–254, 1985.
- [37] H. Bilheux, K. Crawford, L. Walker, S. Voisin, M. Kang, M. Harvey, B. Bailey, M. Phillips, J. Bilheux, K. Berry, et al., "Neutron imaging at the oak ridge national laboratory: Present and future capabilities," in *Proc. 7th Int. Topical Meeting Neutron Radiography. Phys.*, 2013.
- [38] M. Meißner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis, "A practical evaluation of popular volume rendering algorithms," in *Proc. IEEE Symp. Volume Vis.*, 2000, pp. 81–90.
- [39] D. Donzis, P. Yeung, and D. Pekurovsky, "Turbulence simulations on  $O(10^4)$  processors," in *Proc. TeraGrid Conf.*, 2008.
- [40] C. S. Yoo, R. Sankaran, and J. H. Chen, "Direct numerical simulation of turbulent lifted hydrogen jet flame in heated coflow," 2007.
- [41] J. Sanyal, S. Zhang, J. Dyer, A. Mercer, P. Amburn, and R. Moorhead, "Noodles: A tool for visualization of numerical weather model ensemble uncertainty," *IEEE Trans. Vis. Comput. Graph.*, vol. 16, no. 6, pp. 1421–1430, Nov./Dec. 2010.
- [42] R. Wilhelmson, M. Straka, R. Sisneros, L. Orf, B. Jewett, and G. Bryan, "Understanding tornadoes and their parent supercells through ultra-high resolution simulation/analysis," 2013.
- [43] F. Guo, H. Li, W. Daughton, and Y. H. Liu, "Formation of hard power laws in the energetic particle spectra resulting from relativistic magnetic reconnection," *Phys. Rev. Lett.*, vol. 113, no. 15, pp. 1–5, 2014.
- [44] Amazon, "Amazon AWS Instance Types," (2018). [Online]. Available: <https://aws.amazon.com/ec2/instance-types/>, Accessed on Oct. 2018.



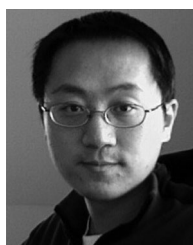
**Mohammad Raji** received the BS and MS degrees in computer engineering from Razi University, Iran, in 2008 and 2012, respectively, and the second MS degree in computer science from the University of Tennessee, in 2017. He is currently working toward the PhD degree at the University of Tennessee, Knoxville. His research interests include web-based data visualization systems, large scale visualization architectures, and deep learning.



**Alok Hota** received the bachelor of science degree in computer science from Fisk University, the bachelor of engineering degree in computer engineering from Vanderbilt University, and the master of science degree in computer science from the University of Tennessee, Knoxville. He is working toward the PhD degree at the University of Tennessee, Knoxville. His research interests include data visualization systems, large scale visualization, and heterogeneous computing. He is a student member of the IEEE.



**Tanner Hobson** received the BS degree in computer science from the University of Tennessee, Knoxville, where he is currently working toward the PhD degree. His research interests include distributed computing, mixed reality visualization, and web-based systems architectures.



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

► For more information on this or any other computing topic, please visit our Digital Library at [www.computer.org/csdl](http://www.computer.org/csdl).