Abstract—ParaViewWeb uses ParaView, an open-source, parallel, data analysis and visualization framework, to generate data products on the server-side and rapidly deliver those data products over the Internet to a web client. Since version 4.0 of ParaView, using ParaViewWeb technologies provides access to all of ParaView through Python interfaces using `pvpython` or `pvbatch`. Using a Python script and leveraging the Python interpreter, ParaViewWeb exposes ParaView’s HPC large data analysis and visualization capabilities through a web server. ParaViewWeb allows the user to perform computationally intensive analysis and visualization tasks within a Web browser by relying on a remote, and possibly distributed, ParaView server for parallel processing and/or rendering. In this paper, we present the new python-enabled ParaViewWeb framework and demonstrate several use cases including its integration with iPython.

I. INTRODUCTION

Seamless integration with Python began in ParaView [1] version 3.0. Simply load the paraview.simple module from Python to get full access to all of ParaView’s large data visualization and analysis capabilities. This includes the ability to create, on the fly, scripted readers and filters that run, in parallel, on the server. ParaView scripts are easy to write, especially if you choose to simply record your work in the desktop application in the form of a python script. Python scripts can be played back with or without the GUI in order to create reproducible, easily customizable, and scalable visualizations.

In the last decade, large tier-1 and tier-2 shared high-performance computing (HPC) resources have been delivered in regional centers, such as the National Science Foundation’s (NSF) TeraGrid/XSEDE NICS, NCSA, SDSC, and TACC, or leadership computing facility (LCF) centers, such as NERSC, Argonne (ALCF) and Oak Ridge (OLCF). This enhanced computational power has made it possible for researchers across the United States, and beyond, to gain new insights from running large simulations, which produce correspondingly large results or data sets. The data sizes of these exceedingly larger and larger runs, typically, have made remote visualization a necessity.

As a natural evolution, ParaViewWeb [2], [3] was developed as a framework used to leverage the power of VTK [4] and ParaView on the Web in an interactive manner. ParaViewWeb uses ParaView, an open-source, parallel, data analysis and visualization framework, to generate data products on the server-side and to rapidly deliver those data products over the Internet to a web client. In cases of small 3D geometry, ParaViewWeb can send the geometry to the client to allow local rendering using WebGL.

Since version 4.0 of ParaView, ParaViewWeb relies on Python to expose the ParaView framework as a service using modern Web protocols. Hence, ParaViewWeb is present in any of the ParaView binaries, leveraging both ParaView Python interpreters `pvpython` and `pvbatch`. Through a simple Python script, ParaViewWeb exposes ParaView’s HPC large data analysis and visualization capabilities through a web server. ParaViewWeb allows the end-user to perform computationally intensive analysis and visualization tasks within a Web browser by relying on a remote, and possibly distributed, ParaView server for parallel processing and/or rendering.

ParaViewWeb development started around 2010. This paper will briefly define the historical architecture and highlight the major improvements of ParaViewWeb.

II. RELATED WORK

Traditional High-Throughput Computing (HTC) - Traditional HTC platforms have typically focused on data mining, management, and analytics, as well as information visualization to a somewhat lesser extent. One of the prominent solutions in this area is Statistical Analysis System (SAS) [5], which was originally developed at North Carolina State University and is now being developed and maintained by SAS Institute, Inc. The R programming language [6] is both a language and a runtime environment allowing statistical computing, data manipulation, and graphical display. The R parallel package provides libraries that support high-performance computing with R. MapReduce [7] is a programming model and runtime system introduced by Google for parallelizing programs and executing them on a cluster of commodity machines. Hadoop [8] is a well-known open-source implementation of MapReduce. The main concept that distinguishes ParaViewWeb from the work mentioned here is our focus on three-dimensional (3D) visualization through the use of High-Performance Computing (HPC) resources for the purpose of visualization and analytics. SAS, R, and MapReduce might be used to produce analytical data for display in ParaViewWeb.

Two Dimensional (2D) Visualization - Data visualization on the Web has evolved dramatically over recent years. A few years ago, website visualization was limited to static charts and images. However, with the advances of HTML5 standards and modern browsers in recent years, the Web is quickly gaining the tools necessary for sophisticated visualizations. D3 [9],...
with its roots in Proovis [10], was a pioneer in this area and provided the groundwork for interactive and animated plots of all types using Scalable Vector Graphics (SVG) [11]. Other libraries such as Raphael [12] and NVD3 [13] use SVG for their rendering. The 2D Canvas API enabled another set of tools such as Processing.js [14] and Flot [15]. Vega, a new specification for visualizations, has alternate SVG and Canvas backends [16]. This large and growing set of tools provides a great capacity for data visualization and has rapidly made the Web the premier environment for showcasing intricate and novel visualizations, as demonstrated by many New York Times storytelling graphics [17]. These tools are not typically used within HPC platforms or for 3D visualization, but could be used in combination with ParaViewWeb.

Three Dimensional (3D) Visualization - WebGL [18] provides a low-level javascript API for performing 3D graphics in a browser. WebGL is based on the OpenGL ES 2.0 specification, which means it is optimized for resource-constrained mobile devices. The API is very similar to modern desktop OpenGL (v2.0 and later), but does have limitations such as number of rendering targets. As the geometry and data increase in size, WebGL performance degrades rapidly. For large problems, WebGL may not be an optimal solution. The three.js library [19] is a abstracted Javascript library that provides support for things like scenes, importing/exporting files, level-of-detail, morphing, and keyframe animation, etc. Three.js is built on top of technologies such as WebGL, SVG, and CSS3D. Another Javascript library built on top of WebGL is Google's O3D [20]. It provides an abstracted interface to programming 3D graphics for the Web.

The Java3D (JOGL) [21] wrapper library is a fairly direct mapping of the OpenGL library, and it allows Java programs to create 3D graphics and visualizations. Using JOGL, interactive 3D Java applets can be developed for the browser, which leverage the OpenGL API. The drawbacks of this approach include having to install Java on client machines as well as having to install and correctly configure the Java plugin in target browsers.

ParaViewWeb leverages WebGL for small-scale visualization and analysis, and JOGL can be used by VTK.

The derived architecture interconnected a WebServer with a ParaView C++ back-end, via a Java Message Service (JMS), which was acting as a broker. The WebServer’s role was to provide an HTTP front-end to the JMS broker.

The visualization server (PWServer) was a ParaView-based headless application that responded to JavaScript Object Notation (JSON) messages sent from the WebServer (PWService). The PWServer was responsible for both data processing and rendering, and it generated the visualization either by itself or by connecting to a remote ParaView server running over a cluster using the message passing interface (MPI).

Fig. 2. The ParaViewWeb remote visualization pipeline.

The Web-service component (PWService) managed communication between remote visualization servers (PWServer) and Web clients. The PWService was packaged as a Web-application that was deployed on a Web server and accessible using a specific URL determined at configuration time. Communication was based on JSON-RPC, a simple JSON-based protocol for remote procedure calls, even though the communication was transmitted over JMS between the Web Server and the PWServer. In addition to client/server communication, PWService handled the management of the PWServer instances, whether they ran locally or on a remote cluster.

The client used a JavaScript library to instantiate and manage remote visualization environments. Clients could make calls to the PWService in order to start new instances of PWServer, monitor running PWServer instances, and send JSON messages to a PWServer. Messages sent to a PWServer allowed for the construction of visualization pipelines and end-user interaction using a mouse.

Therefore, ParaViewWeb’s architecture became a framework of reusable components, which could be combined to implement both server and client functions. Figure 2 depicts the ParaViewWeb architecture in action to create a remote visualization pipeline.

The JavaScript library that came along with the ParaViewWeb framework was wrapping all the ParaView proxies into JavaScript objects. Each change on those wrapped proxies was generating a network communication in order to propagate changes to the PWServer process. This provided great flexibility for JavaScript client development because this exposed everything to the client. However, it produced a great amount of unnecessary communication.

Using these components, developers could build complete websites or Web portals with data analysis and visualization capabilities. The implementation required supporting Java-based Web server, such as Apache Tomcat, a free and open-source implementation.

To get the best interactive rendering possible over standard HTTP, several renderer implementations were investigated. We had a pure JavaScript implementation along with other technologies that required browser plugins such as Flash and Java.
applet. Their respective performance analyses are displayed in Table I, which shows the frame rates obtained and the different network speeds.

### B. Architecture Evolution

While redesigning ParaViewWeb, we followed a completely new path based on a few overarching goals. ParaViewWeb should:

- Be easy to build
- Be easy to deploy, develop with, and use.
- Be easy to secure.
- Leverage new technologies.
- Enforce best practice for client/server architecture.

The new architecture allows ad-hoc usage of ParaViewWeb services with the distributed binaries. How easy is that?

**ParaViewWeb Server** - The ParaViewWeb server is run via `pvpython`, which provides full access to all of ParaView through Python.

```bash
$ ./bin/pvpython lib/site-packages/paraview/web/pv_web_visualizer.py --port 8080 --content .share/paraview/www
```

Listing 1. Command to launch a single end-user ParaViewWeb server.

ParaViewWeb is a single Python script that could be executed by `pvpython` or the provided python interpreter with the correct environment settings (see Listing 1). The script will be responsible for starting a Web server and listening to a given port. The following command line illustrates how to trigger such a server:

Figure 3 illustrates how a single end-user or developer of an existing ParaViewWeb application (e.g. `pv_web_visualizer.py`) can begin interacting with a local ParaViewWeb server.

![Fig. 3. Single end-user ParaViewWeb technologies and layers.](image)

This setup allows for multiple end-users, but will force them to share the same visualization session. This can be useful for collaboration, but is not practical if you want to provide a dedicated service for each end-user. For dedicated multi-user service, you will need a slightly different setup/deployment.

In Listing 2, three main sections: importing ParaViewWeb Python modules; defining the server application class; and the main method.

Listing 2. An example ParaViewWeb application server.

First, the ParaViewWeb modules `wamp`, `protocols`, and `web` provide: WAMP-based client/server communication, a python wrapping of the ParaView services, and the Web server, respectively. `wamp` module is based on Autobahn/Python, a WebSockeT/WAMP library for Python 2 and 3 implemented on Twisted and asyncio. `protocols` are used to predefine the methods needed by the Web client on the server side, `web` provides the Web server methods for the specific application.

WebSocket is a protocol that provides full-duplex communications channels over a single HTTP connection. The WebSocket protocol makes possible more interaction between a browser and a website, facilitating live content. This is one of the new technologies leveraged in ParaViewWeb and Auto-bahn’s implementation in Python. JavaScript are significantly responsible for reducing the layers and technologies from the old ParaViewWeb architecture, and the Python WAMP library made it simple to reuse the seamless integration of ParaView with Python.

Next, the application class defines variables and initializes (registers) specific protocols/services required by the application. The two static methods are convenience methods for parsing and configuring command-line arguments to variables for customizing the application. To reduce interaction communication between the client and the server, we stopped wrapping all of the ParaView proxies into a JavaScrip object. Instead, we allow the developer to register necessary ParaView services from an external python module (`protocols`), which provides a cleaner design, escalation of flexibility, improved performance, and increased security.

ParaViewWeb `ParaViewWebPortGeometryDelivery` is a service required to facilitate geometry delivery to a client employing, for instance, WebGL. WebGL (Web Graphics Library) is a JavaScript API for rendering GPU accelerated
interactive 3D graphics and 2D graphics within any compatible Web browser without the use of plugins. WebGL is another new technology leveraged in the redesign of ParaViewWeb.

Finally, the main method is used to start the Web server. Given the convenience methods, the main method reduces to the following four simple steps: adding server arguments (--content, --port, --debug), adding application arguments, configuring the application, and starting the Web server.

```
var config = {  
    "sessionManagerURL": "http://localhost:8080/paraview",  
    "application": "visualizer",  
    "key1": "value1",  
    "key2": "value2",  
    "key_n": [1,2,3]  
};
```

Listing 3. ParaViewWeb client launcher code.

**Process launcher RESTful API**

ParaViewWeb comes with a JavaScript library, which allows the end-user to trigger a new process on the server side in a configurable manner.

```
listner start(config, function(connection) {  
    // Success callback  
    function(error) {  
        // Error callback  
    });
```

Listing 3 illustrates triggering a new process on the client side, and we will explain what should be expected by the server.

The client code will trigger a POST request on `http://localhost:8080/paraview` with the given `config` object as payload. As a response, the server should return the same `config` object with additional keys such as:

- `sessionURL`: contains the WebSocket URL to where the client should connect in order to connect to the newly started process. `(ws://localhost:8080/proxy?id=2354623546)`
- `id`: contains the session ID that can be used to query the launcher in order to retrieve the full connection information.

In the case of a two-step connection, a client may want to trigger a GET request on `http://localhost:8080/paraview/$sessionID` in order to get the full `config` object illustrated earlier. This is typically useful in cases of collaboration when the person who initiates the visualization session wants to invite other end-users to join. He/she just needs to share his/her `sessionID` to allow other clients to connect. The launcher should also be capable of stopping a running process by triggering a DELETE request on `http://localhost:8080/paraview/$sessionID`. This will return the same `config` object illustrated earlier. Currently, that method is not exercise in our ParaViewWeb usage, as our applications always let the server know when the end-user leaves the application. Moreover, if a server is running without any connected end-user, it will automatically timeout and quit by itself.

**Configuration**

The launcher server will rely on a configuration file that will provide all of the information required for the visualization service. Listing 4 depicts the configuration information to be read by the launcher.

```
{ 
    "configuration": {  
        # Directory where log files for each visualization session will be stored  
        "log_dir": "/data/pv/logs",  
        # Host name for the launcher Web server  
        "host": "localhost",  
        # Endpoint that will be used to handle launcher type of requests  
        # (POST, GET, DELETE)  
        "endpoint": "paraview",  
        # Pattern for the URL that the client should use to connect  
        # to its newly created session  
        "sessionURL": "ws://localhost/proxy?sessionId=${id}",  
        # Time to wait before assuming the launched process is ready  
        # to process request if the ready_line was not found before.  
        "timeout": 25,  
        # Session fields that can be returned to the client  
        "fields": ["sessionId", "host", "port", "udpdir", "key1", "key_n"],  
        # Port to use for web server  
        "port": 8080,  
        # Path for the file that keep track of the mapping between  
        # sessionID and host:port. This is typically used by apache  
        # to handle the WebSocket forwarding.  
        "proxy_file": "/data/proxy.txt",  
        # Optional argument which can be used to server the static  
        # content of your applications  
        "content": "/data/www"  
    },  
    # Additional information returned to the client  
    "sessionData": {  
        "udpdir": "Home",  
        "key1": "value1",  
        "key_n": [1,2,3]  
    },  
    # Set of available resources where a ParaViewWeb process can live  
    "resources": [  
        # Allow up to 3 concurrent session on localhost  
        {  
            "port_range": [9001, 9005],  
            "host": "localhost",  
        },  
        # Allow only 1 session on node2  
        {  
            "port_range": [9001, 9001],  
            "host": "node1",  
        },  
        # Allow only 1 session on node2  
    ]
```

Fig. 4. Evolution of ParaViewWeb’s architecture.

**Multi-User Setup** - In order to support, transparently, the connection of several end-users in different visualization sessions, the server must provide a single entry point to establish a connection, as well as a mechanism to start a new visualization session on demand.

Figure 4 illustrates a multi-user setup where Apache is used as a front-end application to deliver the static content (HTML, JavaScript, CSS, images), as well as to forward the WebSocket communication to the appropriate back-end visualization session. Moreover, a launcher process is used to dynamically start the `pvpython` process with appropriate arguments for the visualization session.

Even though this setup is more complex than the ad-hoc, single end-user setup, it still remains both easy and practical to implement for a variety of institutions.

**Python Launcher** - When deploying ParaViewWeb for multiple end-users, a launcher module is needed to start a new visualization process for each end-user that requests one. This task can be achieved by any Web server that can spawn a new process based on a POST request. However, we wanted to provide a simple answer that did not require the use of any external component not already available within the ParaView binaries. Hence, we built a Python-based process launcher that follows the ParaViewWeb RESTful API for launching a new visualization process.
Listing 4. Example launcher.json.

Editing a launcher.json file is relatively straightforward, given the following object overview:

- The configuration object is designed to allow for general launcher configuration such as host, sessionURL, timeout, ....
- The resources object determines the resources available to the service.
- The sessionData object specifies that certain arbitrary key/value pairs should be included in the data returned to the client upon successful creation of the session.
- The properties object provides a place to define environment variables.
- The apps object defines the application’s command line for the launcher.

Listing 5. Python launcher command-line.

In order to run this service, you will need to execute the following command line.

Then, once the service receives a POST request, it will trigger a new command line, which will have its output redirected to /data/pv/logs/${session_id}.log where the ${session_id} will be a unique generated string that will ID the given process.

For example, if the client sends the given JSON payload:


The client, shown in Listing 6, is all basic HTML. Inside the <body> section, we encounter an instance <div class="viewport-container"></div>. This div is the document object model (DOM) element that will contain the ParaViewWeb produced image/WebGL objects. This is followed by both short and longer JavaScripts.

The first one simply leverages vtkweb-all.js to load vtkweb-all.js with all its dependency. The second script can be broken into three fundamental parts:

- Choose what type of visualization session we want to create on the server side.
- Define what we want to do once the connection is established.
- Make the request for the visualization session.

First, we need specify where we should issue our request (sessionManagerURL) for a specific visualization session (application). In this example (see Listing 6), the “cone” application simply serves a vtkConeSource pipeline. Next, we define a callback function, named “start,” that will be triggered once the connection is established with the visualization session. In that callback, we create a viewport, binds that viewport to the viewport-container div defined earlier, handle resizing events, and override the stop method to properly exit the remote visualization session when the Web page gets unloaded. Finally, a connection to the visualization server is made using the smartConnect method, which takes two callback functions depending on whether or not the request for the visualization session was successful.

The developer can add endless amounts of client-side interaction by using jQuery or other JavaScript functionality and linking with the registered services defined in the server application (see Listing 2). Autobahn’s implementation in JavaScript is responsible for the WAMP/WebSocket server/client communication.

C. Results

The timing results shown in Table II, Table III, and Table IV were measured running the ParaViewWeb server on
TABLE II: PARAViewWeb Timings - Measured on a LAN.

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framerate (fps)</td>
<td>3</td>
<td>200</td>
<td>54</td>
</tr>
<tr>
<td>Round trip (ms)</td>
<td>3</td>
<td>259</td>
<td>13</td>
</tr>
<tr>
<td>Processing time (ms)</td>
<td>0</td>
<td>69</td>
<td>17</td>
</tr>
</tbody>
</table>

three different systems. Tests were run using the ParaViewWeb WebVisualizer application, with 600px x 600px image delivery. The three rows in each table give the minimum, maximum, and average values for framerate (in frames per second), round trip time (in milliseconds), and processing time (server side processing, also in milliseconds).

Table II shows results where the browser running the WebVisualizer was on the same local area network (LAN) as the ParaViewWeb server, connected entirely by Gigabit-capable CAT-6 cabling. Table III and Table IV show results where the browser was running on the local network, and the ParaViewWeb server was outside the firewall, many network hops across the internet. The bandwidth between the local network and the internet was measured at the time the tests were taken and indicated roughly 22 Mbps download and 11 Mbps upload speeds.

TABLE III: PARAViewWeb Timings - Measured on an Amazon EC2 Instance.

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framerate (fps)</td>
<td>1</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>Round trip (ms)</td>
<td>99</td>
<td>866</td>
<td>117</td>
</tr>
<tr>
<td>Processing time (ms)</td>
<td>0</td>
<td>18</td>
<td>8</td>
</tr>
</tbody>
</table>

While average framerates shown in Table III and Table IV are significantly lower than the one in Table II, they still represent an interactive experience for the end-user. Additionally, attention in these cases should be given to the high network latency, which explains the degraded frame rates.

TABLE IV: PARAViewWeb Timings - Measured at ALCF.

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framerate (fps)</td>
<td>1</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>Round trip (ms)</td>
<td>93</td>
<td>941</td>
<td>108</td>
</tr>
<tr>
<td>Processing time (ms)</td>
<td>0</td>
<td>38</td>
<td>8</td>
</tr>
</tbody>
</table>

IV. USE CASES

Using the latest HTML 5.0-based technologies, such as WebSocket and WebGL, ParaViewWeb enables communication with a ParaView server running on a remote visualization node or cluster using a light-weight JavaScript API. Using this API, Web applications can easily embed interactive 3D visualization components, and the application developers can write simple Python scripts to extend the server capabilities including creating custom visualization pipelines.

In the following sub-sections, we will show three diverse use cases of the ParaViewWeb framework.

A. WebVisualizer

WebVisualizer is a ParaViewWeb reference application that essentially provides a Web-based front-end to ParaView. With WebVisualizer, end-users can quickly build visualizations to analyze their data using qualitative and quantitative techniques. This data exploration can be done interactively in 3D. As a full relative application of ParaView, WebVisualizer, like ParaView, was developed to analyze extremely large datasets using distributed memory computing resources. It can be run on supercomputers to analyze datasets of exascale size as well as on laptops for smaller data.

![Fig. 5. ParaViewWeb’s reference implementation, WebVisualizer.](image)

1) WebVisualizer User Interface: The WebVisualizer application utilizes the same user-interface concepts as ParaView, but translates these concepts into the Web. It properly handles various screen sizes and allows seamless interaction from a smart phone to a 30’ inch display. Key to the user interface is the inspector, which provides various types of controls to the end-user that are grouped and control by the following toolbar buttons:

- The ParaViewWeb logo toggles the visibility of the inspector in order to maximize the available space for the 3D renderer, while still providing access to the controls.
- The pipeline icon activates the pipeline browser and editor panel.
- The file icon enables file and directory browsing, allowing any data that can be found on the server side to be loaded.
- The plus icon displays the source menu. A source can be any basic 3D shape like a box, sphere, cone, cylinder, or a customizable text field that will show up in the 3D view.
- The funnel icon provides access to the filter menu. This allows the end-user to process data from the pipeline by filtering them in some way. Common filters such as Clip, Contour, Stream Tracer, Slice, and Calculator are available.
- The information icon enables the information panel to be presented. This panel provides insight on the data that is currently active in the pipeline (e.g., number of points, cells, memory used, data array ranges, etc.).
- The arrows icon resets the 3D camera, which will make the current 3D object fit into the screen and re-adjust the center of rotation.
- The clock icon toggles the visibility of the time toolbar that provides a set of VCR controls to navigate across time and display the current time value.
- The gears icon displays the preference panel that gives access to custom control on the renderer such as the rendering mode (Local vs Remote), statistics visibility, and other application wide controls.
**Pipeline browser** - The pipeline browser provides a graphical representation of the pipeline topology while allowing the end-user to select a given source or filter that he/she wants to control. Toggling the visibility of any filter is also available by clicking on the little circle in front of the filter name. The circle will be filled in white if the filter is not rendered in the 3D scene.

**Editor panel** - The editor panel is similar in many ways to the “Proxy Editor” panel in ParaView. The top toolbar provides the following functionality:

- The tools icon toggles the “Advanced properties” visibility.
- The bookmark icon toggles the visibility of the scalarbar (color bar).
- The extended arrow rescales the lookup table to use the current active data range.
- The title shows the currently active filter, and when allowed, a trash icon is displayed. Clicking on that icon will result in the deletion of that filter.
- The check icon will validate any local changes and will push them to the server side.
- The cross icon will cancel any local changes and reset the user interface to its original state.

In the Editor panel, three sections can be found. The first one represents the set of properties that can be applied on the filter itself. The second one provides the control over the properties of the representation, which defines the way the filter data should be rendered. This includes color mapping and rendering mode (Surface, Wireframe, Surface with Edge...). Finally, the last section provides control over the view itself including the orientation axis, the center of rotation, and the background color.

The editor panel controls all the properties that can be tuned in the same manner as the editor panel in the Paraview Qt application.

2) **WebVisualizer protocols** - While designing a nice Web interface for the WebVisualizer application, we also spend some time developing a simple but powerful protocol to handle and control any type of proxy, as well as, the proxy’s properties. The protocol relies on seven methods listed below, with additional information available online within the ParaViewWeb documentation at the following address:


- `pv.proxy.manager.available` - list what source or filter the server will allow you to create.
- `pv.proxy.manager.create` - create a new source or filter.
- `pv.proxy.manager.create.reader` - create a new reader to open the provided file(s).
- `pv.proxy.manager.get` - return the current state of a given proxy.
- `pv.proxy.manager.update` - update a set of properties among any number of proxies.
- `pv.proxy.manager.delete` - delete the provided proxy only if no other proxy is using it as input.
- `pv.proxy.manager.list` - return the list of proxies that compose the pipeline.

**B. Interactive Pipelines**

The Listing 2, from the previous section, fails to demonstrate how a client front-end would interact with the visualization pipeline of the server application. For example, if the data has more than one field (e.g., temperature, salinity, ...), then a natural interaction would be to switch the coloring of an object by the various fields.

![Fig. 6. Clients interface generated with results returned from the getArguments method.](image)

Listing 7. Defining a ParaViewWeb application that handles client requests.

In Listing 7, the interactive pipeline manager class, `_InteractivePipeline`, exposes methods to the client front-end using `@exportRpc()` annotation.
getArguments returns the metadata associated with the visualization data and calls the server convenience method getMetadata of the Pipeline object. The pipeline object is created when the file is opened. Metadata is added to the pipeline as the file is opened through the pipeline convenience method add_key. Figure 6 depicts one possible interface created based on the results from a client front-end get Arguments call.

updateActiveArgument changes various properties of the data analysis and visualization, and it calls the server convenience method update_argument of the Pipeline object. These arguments are used to change items in the Pipeline object to be rendered by the server. Figure 7 shows two possible states of the graphical user interface during an interactive session.

Fig. 7. Two GUI states in an interactive ParaViewWeb session.

C. iPython notebook

The use of the iPython notebook ranges from jotting down notes using a Python snippet on the end-user’s laptop to a high-performance computing cluster deployment and parallel computation. Our ParaViewWeb integration focuses on the later use case. In fact, we assume that iPython is used to perform parallel simulation using its MPI engines on a 3D mesh, and, for this use case, iPython/notebook lacks a scientific visualization capability. This is exactly where ParaViewWeb helps.

In the following paragraphs, we explain the concepts behind such integration but additional technical details can be found online for reproducibility at the following url:


We assume that iPython/notebook is configured to work on a cluster using MPI for inter-engine communication, and a simulation code is performing computation on a 2D or 3D mesh. Our goal is to follow the evolution of the data, in an interactive manner, while the simulation is running within the iPython/notebook webpage.

To achieve this integration, we provide a helper module paraview.web.ipython, which can be used inside a profile to define convenience methods for initialization, configuration and data update. The listing 8 depicts such profile and a possible subset of convenience methods.

Listing 8. Adding convenience methods accessible.

This module provides methods to start a ParaViewWeb engine in a separate thread across all MPI engines, which allows the simulation code to share data with the visualization engine without any additional memory consumption or copy. ParaViewWeb will run a Web Server on MPI rank 0 while the others ranks will act as data processing and rendering satellites. One convenience method will create an iframe inside the notebook that will connect to the root node serving the interactive web application.

As depicted in Figure 8, two additional actions will be required to transfer simulation data to the ParaViewWeb. First, each engine needs to asynchronously register its locally computed mesh. Then, when all engines are ready to move to the
next timestep, ParaViewWeb needs to be synchronous notified
to update the trivial producers. Once updated, ParaViewWeb
can render and composite the resulting image in a distributed
manner across all the MPI engines. Since a majority of
the processing is done in ParaView’s C++ layer, iPython is
free to process the simulation computation. Therefore, all the
processing capabilities in ParaViewWeb are available, while
the computation is processing.

The entire process, using the convenience methods, can be
defined as:

- `ComputeNextTimeStep()` pushes data on all of the
  nodes for the Trivial producer before ParaView starts.
- `ActivateDataSet()` will activate all of the previ-
  ously pushed data inside the ParaView proxy framework.
- `StartParaView()` will start a ParaViewWeb server
  and provide an interactive Window, which will have the
  trivial producer pre-loaded.

Periodically, we dynamically update the dataset by repeating
steps (1) and (2).

V. CONCLUSION

ParaViewWeb exposes ParaView’s HPC large data anal-
ysis and visualization capabilities through a Web server us-
ing a Python script and leveraging the Python interpreters
pypython and pvbatch.

ParaViewWeb allows the end-user to perform computa-
tionally intensive analysis and visualization tasks within a Web
browser by relying on a remote, and possibly distributed,
ParaView server for parallel processing and/or rendering. Par-
aViewWeb simply uses ParaView to generate data products on
the server-side and rapidly deliver those data products over the
Internet to a web client.

In the profile (shown in Listing 8), we replaced the sim-
ulation part with the creation of a 3D cones on each MPI
engine. Then for each timestep, each engine updates its cone by
changing its position and its resolution. This setup helped us to
exercise the complex integration while creating an interactive
animation. An image of the interactive animation is displayed
in Figure 9.

The new architecture makes it simple to embed real world
– not just toy problems – three-dimensional visualizations in
any Web application.

ACKNOWLEDGMENT

This work was funded by a DOE Office of Science Phase
II SBIR contract DE-SC0007647.

REFERENCES