Scientific data analysis and visualization at scale in VTK/ParaView with NumPy

Utkarsh Ayachit, Berk Geveci
Kitware, Inc.
28 Corporate Drive
Clifton Park, NY 12065

***Abstract*—The Visualization Toolkit (VTK) is an open-source, freely available software system for 3D computer graphics, data processing and visualization written primarily in C++. ParaView is an end-user application based on VTK that is designed for remote, distributed data processing and visualization at scale on large supercomputers and desktops alike. Traditionally, adding new algorithms for data processing in VTK and ParaView has required users to write C++ code. With widespread adoption of Python (fueled by NumPy and SciPy) within the scientific community, there is growing interest in using Python to do parts of the data processing workflow, leveraging the large set of utility functions provided by these packages. In this paper, we discuss the mechanisms for doing the same. Python modules newly added to VTK make it possible to seamlessly transfer data objects between VTK and NumPy, with minimal copying, while preserving relationships between data elements, including mesh connectivity. Subsequently, scripts that combine VTK filters and NumPy ufuncs can be easily supported even when executing at scale in parallel and distributed environments on HPC systems.**

I. Introduction

The Visualization Toolkit (VTK)[1] is an open-source, freely available software system for 3D computer graphics, data processing and visualization. VTK consists of a C++ class library and several interpreted interface layers including Tcl/Tk, Java, and Python. VTK supports a wide variety of visualization algorithms including: scalar, vector, tensor, texture, and volumetric methods; and advanced modeling techniques such as: implicit modeling, polygon reduction, mesh smoothing, cutting, contouring, and Delaunay triangulation. VTK has been used as the underlying data processing engine for several applications including ParaView[2], VisIt[3], and MayaVi[4]. The reason for this has been VTK's data model that can describe complex data structures encountered in scientific domains easily and efficiently, as well as VTK's pipeline-based execution model, that enables demand-driven execution with a rich request qualification vocabulary to control what data gets read in and processed.

ParaView is an open-source, multi-platform data analysis and visualization application built on top of VTK. Besides adding the graphical interface for setting up visualization pipeline, ParaView also manages the remote client-server and distributed processing infrastructure, enabling data analysis and visualization at scale from desktops to supercomputing clusters.

Using Python as the scripting language for application to bind together components is nothing new. Python has been the de-facto scripting language for both VTK and ParaView since the early days. Using Python to describe the pipeline connections between processing modules is popular for both these tools. Besides such batch execution modes, ParaView also uses Python scripting for enabling macros to provide shortcuts for user interactions in the user interface.

In past decade, with large community efforts behind SciPy[5] and NumPy[6], Python has evolved into more than just a scripting language to put things together. The powerful N-dimensional array processing infrastructure with a large set of linear algebra, Fourier transform, and random number capabilities has made writing data processing code in Python a practical and efficient solution.

As a result there has been an increasing interest in writing processing modules (or algorithms) in VTK and consequently, in ParaView using Python and NumPy. In this paper we will discuss the enhancements to VTK that enable seamless data flow between VTK data objects and NumPy arrays.

Section II provides an overview of the VTK data model and how it differs from NumPy's multidimensional array. In Section III we walk through various examples that illustrate the interoperability between VTK and NumPy with discussion on the underlying implementation.

II. Understanding data in VTK

NumPy relies on an array centric data model. At its core is the `ndarray`, which is a homogeneous, multidimensional array. This a surprisingly flexible view of the data which when used along with the slicing, indexing, and shape manipulation techniques can be used for solving a wide array of mathematical and statistical problems.

While array-based operations are indeed useful for data analysis and visualization in domains targeted by VTK and ParaView (e.g. structural analysis, fluid dynamics, astrophysics, and climate science), several of the popular techniques require additional information about the structure of the elements e.g. the mesh structure, including the geometry and topology (what are the points? and how are they connected to each other?). For example, for computing a gradient of a variable defined on points in an unstructured mesh, we need to know the now the connectivity i.e. how the points are connected together to form the cells, from which we can deduce the neighbouring points needed in gradient computation.

The most fundamental data structure in VTK is a data object. Data objects can either be scientific datasets such rectilinear grids or finite elements meshes or more abstract data

structures such as graphs or trees. These datasets are formed of smaller building blocks: mesh (topology and geometry) and attributes. VTK provides rich collection of specializations of data object that leverage inherent structure in the mesh to optimize for both memory and performance for common operations such as accessing elements, and neighbours. Along with preserving relationships between elements in a dataset, there are cases (e.g. adaptive-refinement meshes (AMR), and assemblies encountered in structural analysis) where relationships between datasets, and not just elements in a dataset, are as important and necessary for visualization. Such use-cases are supported by composite datasets in VTK, which are collections of datasets themselves. Currently, there are over 20 concrete sub-types of the VTK data object that used for representing and processing various datasets.

Besides the mesh, the VTK data object also stores attributes for elements in the mesh for example, there may be data values defined at sample points (point data attributes) as well as over interpolation elements or cells (cell data attributes). Representing the mesh and the complex relationships between elements as well as datasets can be challenging in a multidimensional array-based model. These element attributes themselves, however, are not much different from multidimensional arrays encountered in NumPy. We use this similarity as the stepping stone for the inter-operation between NumPy and VTK data types.

III. VTK-NumPy integration

A. Accesssing arrays

Let's start with a simple example: accessing attribute arrays in Python. First, let's create a basic pipeline to produce a VTK data object. For this, we will use the Python scripting API for VTK.

```
>>> import vtk

# create a data source.
>>> w = vtk.vtkRTAnalyticSource()

# execute this trivial pipeline
>>> w.Update()

# access the generated data object.
>>> dataRAW = w.GetOutputDataObject(0)

>>> print type(dataRAW)
<type 'vtkobject'>
```

The `dataRAW` is a Python wrapped instance of the VTK data object. Using the Python wrapping, we can indeed access the C++ API to access the data elements. However, that's not the most elegant approach as in that case any operation over all the elements would entail looping over the elements in Python which can have serious performance impacts. Instead, we would like to use NumPy, which uses highly optimized C/-Fortran routines for data iteration to alleviate this performance hit.

To work well with NumPy, VTK provides a new package, `vtk.numpy_interface`. To wrap a raw VTK data object into a Python object for NumPy interoperability, we do the following:

```
>>> from vtk.numpy_interface \
      import dataset_adapter as dsa
>>> import numpy as np

# wrap the data object.
>>> data = dsa.WrapDataObject(dataRAW)

>>> print type(data)
<class 'vtk.numpy_interface.dataset_adapter.DataSet'
    >

>>> print type(data.PointData)
<class 'vtk.numpy_interface.dataset_adapter.
    DataSetAttributes'>

# To access point data arrays,
# one can simply do the following:
>>> print data.PointData.keys()
['RTData']

>>> print data.PointData['RTData']
[ 60.76346588  85.87795258  72.80931091 ...,
  67.51051331  43.34006882 57.1137352 ]
```

The wrapped data object provides properties `PointData`, `CellData`, etc. that provide access to the attribute arrays associated with various elements in the dataset. These return an object of type `DataSetAttributes` that implements the dictionary interface providing access to the arrays available as well as their names.

Let's take a closer look at the array:

```
>>> rtData =  data.PointData['RTData']
>>> print type(rtData)
<class 'vtk.numpy_interface.dataset_adapter.VTKArray
    '>

>>> print isinstance(rtData, np.ndarray)
True
```

As we can see, the array object returned is of the type `ndarray`. In other words, it a NumPy array. Under the covers, it uses `numpy.frombuffer` to pass the raw C-pointer from the VTK array to NumPy to avoid creating an in-memory copy. Since it is a NumPy array, we can now use any of the NumPy and SciPy routines to operate on this array.

```
# using ufuncs (universal functions)
>>> np.max(rtData)
VTKArray(276.8288269042969, dtype=float32)

>>> np.mean(rtData)
VTKArray(153.15403304178815)

# using operators
>>> rtData / np.max(rtData)
VTKArray([ 0.21949834,  0.31022042,  ...,
  0.15655909,  0.20631427], dtype=float32)
```

Instead of directly using ufuncs from NumPy, for now let us stick with accessing the ufuncs from `algorithms` module provided by VTK. The need for this indirection will become clear when we look into composite datasets and parallel execution. Thus, the above code can be rewritten as follows:

```
>>> from vtk.numpy_interface \
      import algorithms as algs
# using ufuncs (universal functions)
>>> algs.max(rtData)
VTKArray(276.8288269042969, dtype=float32)
```

```
>>> algs.mean(rtData)
VTKArray(153.15403304178815)

# using operators
>>> rtData / algs.max(rtData)
VTKArray([ 0.21949834,  0.31022042, ...,
  0.15655909,  0.20631427], dtype=float32)
```

We can also pass the array computed in Python back to VTK.

```
>>> result = rtData / algs.max(rtData)
>>> data.PointData.append(result, "Result")
```

This `data` can be pass back into the VTK pipeline be setting it as an input to another filter.

```
>>> contour = vtk.vtkContourFilter()
>>> contour.SetInputData(data.VTKObject)
# note since data is the wrapped data object,
# we need to pass the rawData array back to
# VTK pipeline.
    ...
```

As with bringing data into NumPy, these methods try to avoid creating a new copy of the data array is possible. Since VTK expects a fixed ordering for data arrays, however, in certain cases, deep copy becomes necessary. This is handled within the implementation. When not deep copying, the implementation ensures that a reference a held on to the NumPy array so that it does not get garbage collected until the data object is released by VTK. This is necessary since VTK uses its own reference counting mechanism for managing object allocations.

B. Accessing arrays in composite datasets

Composite datasets are datasets comprising of other datasets. Let's consider a multi-block dataset consisting of two blocks.

```
>>> s = vtk.vtkSphereSource()
>>> s.Update()
>>> sdataRAW = s.GetOutputDataObject(0)

>>> w = vtk.vtkRTAnalyticSource()
>>> w.Update()
>>> wdataRAW = w.GetOutputDataObject(0)

>>> mbRAW = vtk.vtkMultiBlockDataSet()
>>> mbRAW.SetBlock(0, sdataRAW)
>>> mbRAW.SetBlock(1, wdataRAW)

# Let's wrap each of these raw data objects
# to make it easier to handle.
>>> sdata = dsa.WrapDataObject(sdataRAW)
>>> wdata = dsa.WrapDataObject(wdataRAW)
>>> mb = dsa.WrapDataObject(mbRAW)

# Let's look at the PointData arrays available for
# each
>>> sdata.PointData.keys()
['Normals']
>>> wdata.PointData.keys()
['RTData']
```

In this case, each block in the multi-block dataset has a different set of arrays for PointData. We can of course iterate over all blocks in the multi-block. In that case, we are dealing with one dataset at a time and hence we can perform array operations as in Section III-A.

```
counter = 0
for d in mb:
...   print "Block", counter, d.PointData.keys()
...   counter += 1
...
Block 0 ['Normals']
Block 1 ['RTData']
```

`numpy_interface.dataset_adapter`, however makes it possible to access the arrays directly on the wrapped multi-block dataset.

```
>>> print mb.PointData.keys()
['Normals', 'RTData']

>>> print mb.PointData
<vtk.numpy_interface.dataset_adapter.
    CompositeDataSetAttributes instance at
    0x1193fe5a8>

>>> mbrtData = mb.PointData["RTData"]
>>> print type(mbrtData)
type(mb.PointData['RTData'])
<class 'vtk.numpy_interface.dataset_adapter.
    VTKCompositeDataArray'>

>>> isinstance(mbrtData, np.ndarray)
False
```

In this case, instead of getting the `DataSetAttributes` instance, we get a `CompositeDataSetAttributes` instance. `CompositeDataSetAttributes` exposes an arrays list which is a union of all the arrays available on all the blocks in the dataset. Naturally, you'd expect to perform operations on the arrays. However, the returned arrays of the type `VTKCompositeDataArray` which no longer is a `ndarray`. Since each block in a composite dataset can have different sets of arrays with different number of elements in them, it cannot be represented as a single multidimensional array. However, it supports the usual arithmetic operators for addition, subtraction, etc. At the same time, the `algorithms` module wraps several of the *ufuncs* provided by NumPy to properly handle iterator over `VTKCompositeDataArray`. Hence, the code for earlier still continues to work. This is one of the reasons why we go through this indirection, rather than directly using NumPy ufuncs. Implementation wise, `algorithms` module does little more than *decorate* the NumPy ufuncs with code to handle iteration over composite dataset (and distributed processing, explained later).

```
>>> algs.max(mbrtData)
276.82882690429688

>>> algs.mean(mbrtData)
VTKArray(153.15403304178815)

>>> result = mbrtData / algs.max(mbrtData)
>>> mb.PointData.append(result, 'Result')
```

How `append` works in the code above is trickier than it appears at first glance. Notice that `RTData` array is only available on the second block, and hence all the operations only make sense on the second block. So one would expect the `Result` array to be added only to the second block in the multi-block dataset as well.

```
counter = 0
for d in mb:
...   print "Block", counter, d.PointData.keys()
```

```
...   counter += 1
...
Block 0 ['Normals']
Block 1 ['RTData', 'Result']
```

To make this possible, we use a special placeholder called `NoneArray` for every location where the array is missing in the traversal of the multi-block dataset. `NoneArray` implements all the standard operators to return a `NoneArray`. Also, all the ufuncs implemented in `algorithms` module handle operations with `NoneArray` appropriately. Finally, `VTKCompositeDataArray.append` handles `NoneArray` by skipping the blocks where a `NoneArray` is present. Thus, users can continue to work with arrays in composite-datasets as without explicitly handing them.

C. Indexing and slicing arrays

`VTKArray` as well as `VTKCompositeDataArray` support the same indexing/slicing abilities as any NumPy array.

```
>>> rtData[0]
60.763466

>>> rtData[-1]
57.113735

>>> rtData[0:10:3]
VTKArray([  60.76346588,   95.53707886,
    94.97672272,  108.49817657], dtype=float32)

>>> rtData + 1
VTKArray([ 61.76346588,  86.87795258, ...,
        44.34006882,  58.1137352 ], dtype=float32)

>>> rtData < 70
VTKArray([ True , False, False, ...,  True,  True],
    dtype=bool)

# We will cover algorithms later. This is to
# generate a vector field.
>>> avector = algs.gradient(rtData)

# To demonstrate that avector is really a vector
>>> algs.shape(rtData)
(9261,)

>>> algs.shape(avector)
(9261, 3)

>>> avector[:, 0]
VTKArray([ 25.69367027,   6.59600449, ...,
        -5.77147198,  13.19447994])
```

A few things to note in this example:

- Single component arrays always have the following shape: (ntuples,) and not (ntuples, 1).
- Multiple component arrays have the following shape: (ntuples, ncomponents).
- Tensor arrays have the following shape: (ntuples, 3, 3).
- The above holds even for images and other structured data. All arrays have 1 dimension (1 component arrays), 2 dimensions (multi-component arrays) or 3 dimensions (tensor arrays).

It is possible to use boolean arrays to index arrays. So the following works just as well:

```
>>> rtData[rtData < 70]
VTKArray([ 60.76346588,  66.75043488, ...,
  55.39360428,  67.51051331,
  43.34006882,  57.1137352 ], dtype=float32)

>>> avector[avector[:,0] > 10]
VTKArray([[ 25.69367027,   9.01253319,
    7.51076698],
  [ 13.1944809 ,   9.01253128,   7.51076508],
  [ 25.98717642,  -4.49800825,   7.80427408],
  ...,
  [ 12.9009738 , -16.86548471,  -7.80427504],
  [ 25.69366837,  -3.48665428,  -7.51076889],
  [ 13.19447994,  -3.48665524,  -7.51076794]])
```

`VTKCompositeDataArray` is a collection of `VTKArray` instances corresponding to each block in the composite dataset. In needed, one can individually access the arrays of each block as follows.

```
>>> mbrtData.Arrays[1]
VTKArray([  60.76346588  85.87795258, ...    ],
    dtype=float32)
```

Note that indexing is slightly different.

```
>>> print mbrtData[0:2]
[<vtk.numpy_interface.dataset_adapter.VTKNoneArray
  object at 0x1191d0150>,
 VTKArray([ 60.76346588,  85.87795258], dtype=
     float32)]
```

The return value is a composite array consisting of two `VTKArrays`. The `[]` operator simply returned the first 2 values of each array. Since the first array in the collection is a `NoneArray`, it returns a `NoneArray`. In general, all indexing operations apply to each VTKArray in the composite array collection. Same is true when indexing using mask (or boolean) arrays:

```
print mbrtData[mbrtData < 50]
[<vtk.numpy_interface.dataset_adapter.VTKNoneArray
    object at 0x1191d0150>,
 VTKArray([ 49.75050354, ...,
    49.39894867,  43.34006882], dtype=float32)]
```

D. Differences with NumPy

So far, all of the functions from `algorithms` module that we discussed are directly provided by NumPy. They all work with single arrays and composite data arrays. `algorithms` also provides some functions that behave somewhat differently than their NumPy counterparts. These include `cross`, `dot`, `inverse`, `determinant`, `eigenvalue`, `eigenvector` etc. All of these functions are applied to each tuple rather than to a whole array/matrix. For example:

```
>>> amatrix = algs.gradient(avector)
>>> algs.determinant(amatrix)
VTKArray([-1221.2732624 ,  -648.48272183,
    -3.55133937, ...,    28.2577152 ,
       -629.28507693, -1205.81370163])
```

E. Algorithms using connectivity

One of VTK's biggest strengths is that its data model supports a large variety of meshes and its algorithms work generically on all of these mesh types. The `algorithms` module exposes some of this functionality. Other functions can be easily implemented by leveraging existing VTK filters.

```
>>> avector = algs.gradient(rtData)
>>> amatrix = algs.gradient(avector)
```

Functions like these require access to the dataset containing the array and the associated mesh. This is one of the reasons why we use a subclass of `ndarray` in `dataset_adapter`:

```
>>> rtData.DataSet
<vtk.numpy_interface.dataset_adapter.DataSet at 0
    x11b61e9d0>
```

Each array points to the dataset containing it. Functions such as gradient use the mesh and the array together. NumPy provides a gradient function too. To understand the difference between that and the one provided by `algorithms`, let's look at this example:

```
>>> algs.gradient(rtData2)
VTKArray([[ 25.46767712,   8.78654003,
    7.28477383],
       [  6.02292252,   8.99845123,   7.49668884],
       [  5.23528767,   9.80230141,   8.3005352 ],
       ...,
       [ -6.43249083,  -4.27642155,  -8.30053616],
       [ -5.19838905,  -3.47257614,  -7.49668884],
       [ 13.42047501,  -3.26066017,  -7.28477287]])
>>> rtData2.DataSet.GetClassName()
'vtkUnstructuredGrid'
```

`gradient` and algorithms that require access to a mesh work irrespective of whether that mesh is a uniform grid or a curvilinear grid or an unstructured grid thanks to VTK's data model. NumPy's `gradient`, on the other hand only works with what VTK would refer as uniform rectilinear grids.

Under the covers, `gradient` indeed uses the `vtkDataSetGradient` filter. Since `gradient` mimics NumPy ufuncs, one can combine it with other NumPy ufuncs, or operators:

```
>>> algs.max(algs.gradient(rtData2) / 2.0)
```

F. Parallel data processing

For addressing large data processing needs on supercomputing environments, ParaView and VTK rely on distributed data processing using MPI[7]. Data is distributed among MPI ranks, with each rank processing part of the data. When we are dealing with data distributed among ranks in such a fashion, we often encounter cases where:

- ranks may not have any part of the data i.e. certain ranks can have no datasets at all.
- ranks that have data may still have differences in the number of elements as well as attribute arrays present.
- when dealing with composite datasets, these disparities manifest themselves at block level as well i.e. certain blocks may be present on certain ranks and not on others.

Now, if data on the local process is not complete, certain operations like `min`, `max` will yield different results on different ranks! This is where the indirection of using ufunc implementations in `algorithms` module comes in handy. We can explicitly handle such operations that need special reduction operations or communication among ranks here.

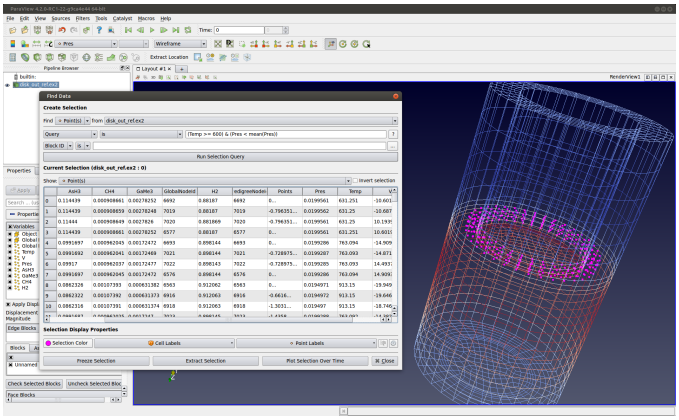

Fig. 1. *Find Data* dialog in ParaView used to find all points in the dataset where the `Temp` is greater than or equal to 600 and `Pres` is less than the mean `Pres`. Such queries and expressions can be easily evaluated by using the VTK-NumPy integration layer described in Section III.

```
# Produces same result on all ranks.
>>> algs.max(rtData)

# vtkDummyController is a replacement for
# vtkMPIController that works only locally
# to each rank.
>>> algs.max(rtdata, controller =\
        vtk.vtkDummyController())
# Will produce different results on
# different ranks based on the data
# distribution.
```

The ufunc implementations in `algorithms` for `min`, `max`, etc. are decorated versions of NumPy ufuncs with variations for handling parallel processing as well the composite datasets and `NoneArray`.

IV. Python-based data processing in ParaView

ParaView provides access to the Python-based data processing in VTK (Section III through *Programmable Source* and *Programmable Filter*. The algorithms available and API is same as discussed earlier, the only difference being that the wrapped data objects for inputs and output to the filter are bound to scope variables `inputs` and `output`. Thus, you do not need to call WrapDataObject() explicitly.

```
# Sample 'Script' for a 'Programmble Source'
# that reads in a CSV file as a vtkTable
import numpy as np
# assuming data.csv is a CSV file with the 1st
# row being the names names for the columns
data = np.genfromtxt("data.csv", dtype=None,
    names=True, delimiter=',', autostrip=True)
for name in data.dtype.names:
   array = data[name]
   output.RowData.append(array, name)
   # we directly access 'output'
   # without calling dsa.WrapDataObject().
```

Besides data processing, ParaView also uses this NumPy interoperability for providing data querying interface. The *Find Data* dialog allows users to enter Python expression that evaluate to a mask array (Figure 1). The dialog then extracts the masked elements from the dataset to present the selected elements. This leverages the infrastructure explained so far

to ensure queries work appropriately in parallel, distributed environments.

V. Conclusion

VTK and NumPy are both widely used in data processing applications. This paper describes Python modules and API recently added to VTK that makes it easier to transfer data between the two frameworks. This enables interoperability by making it possible for VTK algorithms to use NumPy utility functions and vice-versa while preserving information about the mesh topology, etc. provided by the VTK data model, which is not easily represented as NumPy's multidimensional arrays. Our implementation also addresses the complications with distributed data processing and composite datasets, commonly encountered in data analysis and visualization by VTK and ParaView users.

References

[1] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*, 4th ed. Kitware Inc., 2004, iSBN 1-930934-19-X.

[2] A. H. Squillacote, *The ParaView Guide: A Parallel Visualization Application*. Kitware Inc., 2007, iSBN 1-930934-21-1. [Online]. Available: http://www.paraview.org

[3] *VisIt User's Manual*, Lawrence Livermore National Laboratory, October 2005, technical Report UCRL-SM-220449.

[4] P. Ramachandran, "MayaVi: A free tool for CFD data visualization," in *4th Annual CFD Symposium*, August 2001.

[5] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–, [Online; accessed 2014-09-04]. [Online]. Available: http://www.scipy.org/

[6] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science and Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[7] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI – The Complete Reference: Volume 1, The MPI Core*, 2nd ed. MIT Press, 1999, iSBN 0-262-69215-5.