Eurographics Symposium on Parallel Graphics and Visualization (2023)
R. Bujack, D. Pugmire, G. Reina (Editors)

Extended Visual Programming for Complex Parallel Pipelines in ParaView

Marvin Petersen[1], Jonas Lukasczyk[1], Charles Gueunet[2], Timothée Chabat[2], Christoph Garth[1]

[1]RPTU Kaiserslautern-Landau, Kaiserslautern, Germany
[2] Kitware, Villeurbanne, France

Abstract

Modern visualization software facilitates the creation of visualization pipelines combining a plethora of algorithms to achieve high-fidelity visualization. When the complexity of the pipelines to be created increases, additional techniques are needed to ensure that reasoning about the pipelines structure and its performance remains feasible. This paper presents three additions to ParaView with the goal of improving presentation of complex, parallel pipelines benefiting pipeline realization. More specifically, we provide a runtime performance annotation visualization integrated in a visual programming node editor, allowing all users to reason about basic performance and intuitively manipulate the structure and configuration of pipelines. Further, we extend the list of available filters with control flow filters, supporting for- and while-loops with a comprehensible representation in the node editor. Our extension is based on graphical manipulation of a node graph that expresses the flow of data and computation in a VTK pipeline, and draws upon a long tradition and positive experience with similar interfaces across a wide range of software systems such as the visualization tools SCIRun and VTK Designer, or the rendering systems Blender and Houdini. The extension we provide integrates seamlessly into the existing ParaView architecture as a plug-in, i.e., it does not require any modifications to ParaView itself or VTK's execution model.

1. Introduction

Over the past decade, the growing complexity of research questions pursued by computational scientists and engineers has led to a commensurate increase in the complexity of interactive visualization systems that frequently take on key roles in scientific discovery, hypothesis validation, and science communication. The state of the art among such systems combine complex analysis algorithms with advanced rendering techniques, while employing parallelization to allow scaling to very large problem sizes and complexities.

Early it was recognized that visualizations of scientific data can frequently be expressed in terms of a data flow graph: a directed acyclic graph (DAG) that describes how data is propagated and transformed from data sources via processing steps (*filters*) to the final image. Obtaining a good understanding of a complex data flow graph and its performance becomes an increasingly difficult task, since data flow graphs become larger, exhibit more complex graph structures and parallelization complicates straightforward performance reasoning. Furthermore, performance analysis and pipeline realization are deeply intertwined, meaning that the complexity of performance analysis can depend largely on the complexity of the respective pipeline. Also, performance insights about parts of the pipeline, may lead to structural optimizations changing the pipeline itself. Thus, a fitting representation of these two aspects is essential.

In regard to visual pipeline representations, the data flow graph that constitutes the foundation of early interactive visualization systems, such as *IBM*'s *Open Data Explorer (OpenDX)* and the *Visualization ToolKit (VTK)* [SML06], was exposed and utilized as a primary user interface metaphor to interactively create and manipulate visualizations by allowing direct interaction with a node-link representation, facilitating a measure of visual programming that is intuitive for both novices and experts. Corresponding user interfaces are found in many interactive visualization systems such as *SCIRun* [Sci16], *VTK Designer* [LB08], *VisTrails* [BCC*05], *Inviwo* [JSS*19], *OpenWalnut* [EHWS10], among many others. The *node editor* user interface is also enormously popular in other domains that are data flow-centric, e.g., in rendering systems such as *Blender* [Com], *Maya* [Aut], and *Houdini* [Sid].

A notable exception in this context is *ParaView* [AGL05]: a state-of-the-art visualization system that is routinely used in production science. Historically, *ParaView* displays the processing pipeline—i.e., the data flow DAG—via a tree view widget to prioritize ease-of-use for mostly linear workflows. However, this abstraction has proven brittle as *ParaView* has evolved from a moderate number of relatively simple, general-purpose filters to encompass a large number of special-purpose filters, in part due to increased availability of powerful plugins (e.g. the *Topology Toolkit (TTK)* [TFL*18] for topological data analysis). The resulting non-

DOI: 10.2312/pgv.20231084

linear data flow graphs are difficult to express in the tree UI, and managing complex pipelines with tens to hundreds of filters borders on the infeasible. While *ParaView* does allow complete control of visualization pipelines through an additional low-level *Python* interface, its use is beyond the typical user skill set.

ParaView serves as a standard visualization solution for many scientific codes (e.g. *OpenFOAM* [JJT*07]), and—due to its scalability—is one of the visualization components of the flagship Exascale Computing Project (ECP) [Mes17]. Despite the focus in this area, the inbuilt performance analysis capabilities are limited and lack a visual representation. Thus users have to process this data themselves or must employ other tools for analyzing their pipelines. There are a multitude of tools specifically tailored to the visual analysis of parallel and distributed applications. However, for the general user of the visualization system, the additional effort in the setup or the complexity of the tool for in depth performance analysis might be prohibitive. Depending on the goal with respect to performance, the required data for the analysis is beyond basic time-to-solution measurements, especially in complex pipelines. However, detailed performance measurements referencing code parts may also become overwhelming for the pipeline creator, who, in general, does not change the implementation of specific modules. Thus, we strike a compromise between the ease of use and the amount of insights that can be gathered from a performance visualization, tailored to the use case of pipeline creation.

In this paper we make the following contributions:

- We present in Section 3 a *Node Editor* implementation, an open-source plugin that has already been integrated into *ParaView* and provides a modern visual programming interface based on the node-link paradigm. Our implementation exposes the full gamut of *ParaView*'s internal processing model, while remaining non-invasive and completely optional. We discuss specific design and implementation choices, and compare the use of the Node Editor UI with the traditional tree view on several examples.
- We present an augmentation of the node-link representation in the Node Editor UI with the display of performance data, allowing users at all levels to reason about the performance of visualization pipelines in a straightforward and accessible manner, especially for MPI-based parallel processing (Section 4). We illustrate performance insights on real-world examples.
- In Section 5, we describe UI mechanisms, integrated into the Node Editor, to simplify constructing and working with complex pipelines that involve iterative processing, such as for-loops available through specific filters, and illustrate these mechanisms on real-world problems. The UI mechanisms expose the cyclical behavior and additional filters allow for iteration of non-linear pipeline parts, which was previously impossible with only ParaView's UI.

The Node Editor is the first approach to fundamentally enable complex visual programming for the *ParaView* visualization system.

2. Related Work

Node editors have been employed in many data flow-based systems in order to provide a clear view of data or workflow and to provide an intuitive way to manipulate both. Prominent examples of 3D computer graphics software where the usefulness of node editors is exemplified are *Blender* [Com], *Houdini* [Sid], or *Maya* [Aut]. There they are used in multiple ways, from compositing images to creating and manipulating materials for rendering or linking and grouping objects or their parameters. In scientific visualization, this concept has been explored in software like *OpenDX* [Wik21], *AVS/Express* [Ava], *VisTrails* [BCC*05], *SCIRun* [Sci16], *MeVisLab* [MeV], *Inviwo* [JSS*19], *VTK Designer* [Udu15] and *OpenWalnut* [EHWS10], among others.

In the *OpenDX* systems [Wik21], scientific visualizations can be created using a visual program editor, showing the interactors and how they are combined in a network view. Here, nodes in the network represent processing steps, with special nodes representing data sources (e.g. the *FileSelector* node) and sinks (*Image* node). Each node exposes connections for processing inputs and outputs (ports), which are connected using a drag and drop metaphor. *OpenDX* networks are typically laid out such that data flow vertically, from top to bottom.

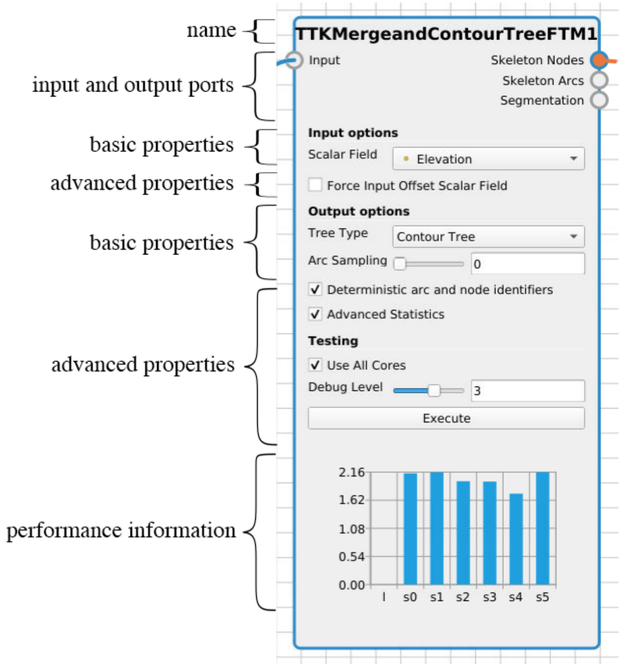
A further, earlier node-based visualization system is *SCIRun* [PJ95]. It utilizes a node editor to compose scientific visualizations and simulations. Furthermore, *SCIRun* also provides performance information on each node via the used CPU time that was needed for completing the specific module.

MeVisLab [MeV] is an image processing and visualization framework that employs a node editor focusing on the medical domain. A modern node-based visualization system also focusing on medical imaging is *OpenWalnut* [EHWS10].

A more recent software framework for visualization of scientific data is Inviwo [JSS*19]. The node editor used there models the data flow from top to bottom, while horizontal connections are used to synchronize property states. Similar to Inviwo, our approach also makes use of a distinction between horizontal and vertical flow, modelling data processing and rendering respectively.

VisTrails [BCC*05] was designed with provenance visualization for scientific visualizations in mind. With it visualizations can be created in its own *VisTrails Builder*, which also uses a node-link representation for the data flow. *VisTrails* is built on top of the *Visualization Toolkit* (*VTK*). A tool made specifically for *VTK* was presented with *VTK Designer* [Udu15]. Its primary goal was to ease the process of developing a pipeline of VTK filters. It uses a node editor to represent the *VTK* pipeline structure with sources, filters, actors and mappers as nodes. Created pipelines could afterwards be transformed to C++ source code.

Analogously to, e.g., *Blender* [Com] and in contrast to, e.g., *VisTrails* [BCC*05], we include module parameters directly in their nodes in the presented interface. Furthermore, we include performance data about the represented pipeline and its modules, which most of the previous system do not—with the notable exception of *SCIRun* [PJ95]. But in contrast, the plugin presented here visualizes the execution times in plots that facilitate comparison between modules and gives insight into the performance on distributed machines by including timings per MPI rank. Additionally, the user interface that is presented here can be easily used with the open-source visualization application *ParaView*, building upon a complex and powerful visualization software. Thus, a user does not



TTKMergeandContourTreeFTM1
Input | Skeleton Nodes, Skeleton Arcs, Segmentation
Input options
Scalar Field: Elevation
☐ Force Input Offset Scalar Field
Output options
Tree Type: Contour Tree
Arc Sampling: 0
✓ Deterministic arc and node identifiers
✓ Advanced Statistics
Testing
✓ Use All Cores
Debug Level: 3

Figure 1: A fully verbose filter node. The name and its ports are always visible. Visibility of properties can be cycled through (none, basic, basic+advanced). Performance information is individually toggleable.

have to worry about importing/exporting pipelines into another software—much less, recreating it there—to get the benefits of a node-link representation of the data flow, unlike the aforementioned *VTK Designer* [Udu15]. Opposed to the aforementioned systems, the prominent visualization systems *ParaView* [AGL05] and *VisIt* [CBW*12] represent data flow of their pipelines in a tree view. Modules that receive data from another module are children to this module in the tree view.

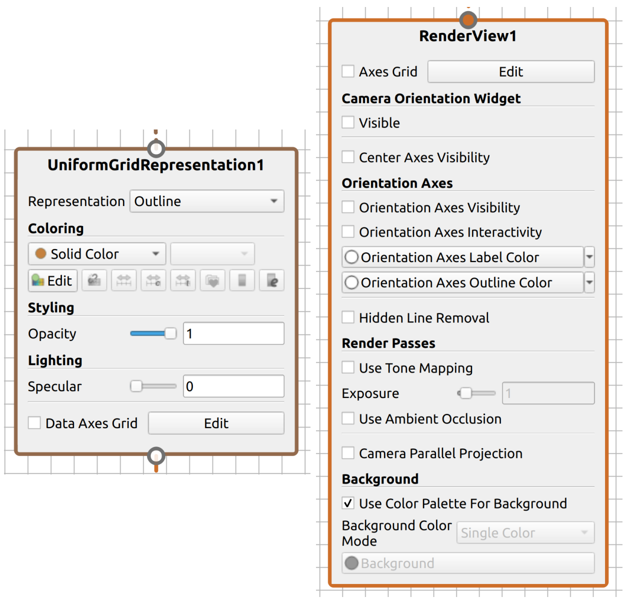
For performance analysis/visualization, there is a plethora of tools for parallel applications like *Vampir* [NAW*96], more recent additions like *Traveler* [SBT*23] or *CallFlow* [NBJ*21] and many other [IGJ*14]. In contrast to most of these, our approach is directly coupled with the interactive visualization system to be analyzed, thus requiring no effort on the user side to set up. Furthermore, the scope of the performance visualization used here is directly coupled to the task of pipeline realization. Many performance visualization systems provide deep insights into the execution, potentially down to the hardware level. In contrast, the work presented here is not targeted at domain experts. Since our approach is targeted at the general user, it visualizes only basic performance information for the purpose of general comprehension [IGJ*14] and simple problem detection tasks. Therefore, visualizing, e.g., execution traces in gantt charts [NAW*96, SBT*23] is limited in its value in this application, since the user cannot change the scheduling of pipeline modules further than with their position in the pipeline. Similarly, visualizations of the call context [NBJ*21], while potentially revealing a more detailed explanation of an inefficiency, are also limited in their value, since the pipeline pattern directly encapsulates the high level call structure. While the low level call structure might be interesting for the performance, the average user of *ParaView* does not change the filter implementation, meaning the information is less relevant with respect to pipeline realization. Although basic, the visualization techniques used here have been employed in performance visualization successfully. For example, as a means to visualize load imbalance bar charts [DHJ07, NBJ*21] and heat maps [VSLNMS20, SLB*11] have been used.

3. Basic Node Editor

3.1. Design

Our implementation follows the same design as other prominent node editors, such as the ones of *Blender* and *Houdini*. Since *VTK* already follows a pipeline concept, most parts of the *VTK* architecture can directly be mapped to the node-link paradigm. Specifically, each *VTK* filter is represented as a single node (a so-called *processing node*) with the name of the associated filter instance at the top (Figure 1). Each individual input and output data object of a filter is represented via a single input and output port, respectively. Input ports are located on the left border of a processing node, and output ports are located on the right border. Hence, the processing pipeline is arranged from left to right, where connections between input and output ports are represented via blue *processing edges*. Widgets that control filter parameters and performance measurements are shown inside the node.

One design aspect in which our node editor differs from existing editors is the way in which it handles the visibility of data objects. A *VTK* render view (also known as viewport) can be seen as a special case of a *VTK* filter, which takes multiple data objects (filter outputs) as its input, and then renders them to an output image. We



UniformGridRepresentation1
Representation: Outline
Coloring
Solid Color
Styling
Opacity: 1
Lighting
Specular: 0
☐ Data Axes Grid

RenderView1
☐ Axes Grid
Camera Orientation Widget
☐ Visible
☐ Center Axes Visibility
Orientation Axes
☐ Orientation Axes Visibility
☐ Orientation Axes Interactivity
Orientation Axes Label Color
Orientation Axes Outline Color
☐ Hidden Line Removal
Render Passes
☐ Use Tone Mapping
Exposure: 1
☐ Use Ambient Occlusion
☐ Camera Parallel Projection
Background
✓ Use Color Palette For Background
Background Color Mode: Single Color
Background

Figure 2: A representation node (left) and a fully verbose view node (right).

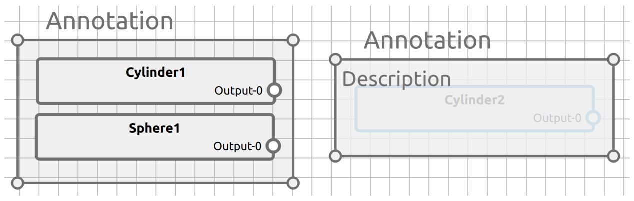


Figure 3: Example annotations: title and description (if selected).

therefore represent each view also with a *view node* (Figure 2), but a view node has only one input port (the set of all visible objects) and no output ports (since the rendered image is only displayed on screen and not explicitly represented as a data object).

Whether a data object is shown inside a view is represented by an edge connecting the output port of the corresponding filter to a *representation node* that further connects to the view via an additional edge (Figure 4). Properties of the render view—e.g., raytracing and post processing shader parameters—are shown inside the view node in the same way as for processing nodes, while properties that are tied to the representation of the specific data object, like a colormap, are shown in the representation node (Figure 2). In addition to coloring processing and view elements differently, we separate more clearly the processing from the visualization pipeline by displaying processing pipelines horizontally, and visualization pipelines vertically. This is achieved by placing view nodes always below filter nodes, and placing the input port of view nodes at the top.

ParaView also has a concept of *active* data objects (filter outputs) and views. The borders of the corresponding nodes are highlighted via color, whereas inactive nodes have a gray border.

We also allow users to control the visual verbosity of the node editor. For instance, it is possible to hide view nodes to focus only on the processing pipeline. To still indicate in this case which data objects are currently visible in the active view, we render the output ports of the corresponding data objects in the color of the active view (Figure 4). It is also possible to toggle the visibility of basic, advanced, or all filter parameters for the filter nodes, as well as their performance information (Figure 1).

The node editor also features a layout engine that is based on `GraphViz` [EGK*02]. Every time the graph is modified the layout engine computes an optimized layout that is as much as possible consistent with the previous layout. This feature can of course be turned off and users always can freely move nodes. If turned off, the layout of the graph is stored alongside the state file when exported.

Furthermore, the node editor allows adding annotations in form of resizable rectangles with titles and descriptions for grouping filters and explaining design choices, as shown in Figure 3.

3.2. Implementation

Since the *ParaView* user interface is based on *Qt*, we decided to implement the node editor via the *Qt Graphics View Framework*. The primary reason for this is the reuse of existing code, since this framework makes it possible to simply embed the parameter widgets of a filter inside a node in the same way they can be embedded in the `Properties Panel`. Note that in Figure 7 the same widgets of the contour filter are shown in the properties panel as well as in the corresponding node. We did not have to adjust existing widget code, and widgets are automatically synchronized when shown and adjusted simultaneously in both views.

The software architecture of the node editor implements an observer pattern to react to changes of the pipeline state. Hence, the node editor only manages the visual representations of pipeline elements, which are created, modified, or deleted based on events emitted by the `pqServerManagerModel`, the `pqActiveObjects`, and `pqApplicationCore` instances. The event-driven design of *ParaView* made this usually difficult task rather simple. Specifically, the node editor listens to the following events:

- `pqServerManagerModel::sourceAdded`
- `pqServerManagerModel::sourceRemoved`
- `pqServerManagerModel::viewAdded`
- `pqServerManagerModel::viewRemoved`
- `pqServerManagerModel::connectionAdded`
- `pqServerManagerModel::connectionRemoved`
- `pqActiveObjects::viewChanged`
- `pqActiveObjects::selectionChanged`
- `pqApplicationCore::stateLoaded`

Note, the `sourceAdded/Removed` events are also triggered for the creation of filters, since in *VTK* the filter class is derived from the source class (which consumes no input data objects). The node editor listens to events emitted from the `pqActiveObjects` class to highlight different parts of the node editor.

The node editor never explicitly changes the pipeline state, instead changes are requested through the existing `pqServerManagerModel`. If the state changes after such a request, the `pqServerManagerModel` will emit events which are captured by the observer pattern. This architecture has the advantage that the logical component of the node editor is minimal and that everything is by design consistent to the pipeline state.

4. Runtime Annotation

A good understanding of the pipelines employed in these systems is necessary to be able to reason about its performance and make informed decisions about restructuring the pipeline and changing parameters. In order to reason about the performance of the pipeline in shared-memory parallel or even distributed execution, it often does not suffice to examine only time-to-solution of the whole pipeline. Single modules can be the bottleneck of a pipeline. While the runtime performance of a specific filter cannot be influenced by the user directly, the choice of filter and its parameters and dependencies inside the pipeline can.

Including performance information about each module is beneficial in multiple ways. First, the user gains a better understanding of the cost of each individual filter and in relation to other filters in the pipeline. This enables the user to spot bottlenecks and apply potential optimizations or change the pipeline structure. Furthermore, by comparing the current runtime of a filter to the runtimes of previous executions of the same filter, one can estimate the influence of specific parameter changes or how a change in a filter earlier in the pipeline influences a later stage. This transfers to the evaluation of data redistribution to mitigate load imbalance in a distributed case.

4.1. Design

Using the additional screen space that is available in the node editor, we complement the nodes of the node graph with runtime performance information (see Figure 1). Focusing on the pipeline realization task for end users of *ParaView*, we identified the following basic tasks that the runtime annotation should support:

(T1) Compare the performance of two or more modules
(T2) Further identify the filter(s) with the largest contribution to the overall runtime
(T3) Identify load imbalance when executing modules on multiple MPI ranks
(T4) Compare the runtime of modules before and after a parameter change (e.g. filename, iso-values etc.)

In order to support these tasks, we include performance information at the module level. While more detailed information could yield additional insights about the exact source of bottlenecks or similar, the average user is not expected to change the inner workings of VTK or ParaView.

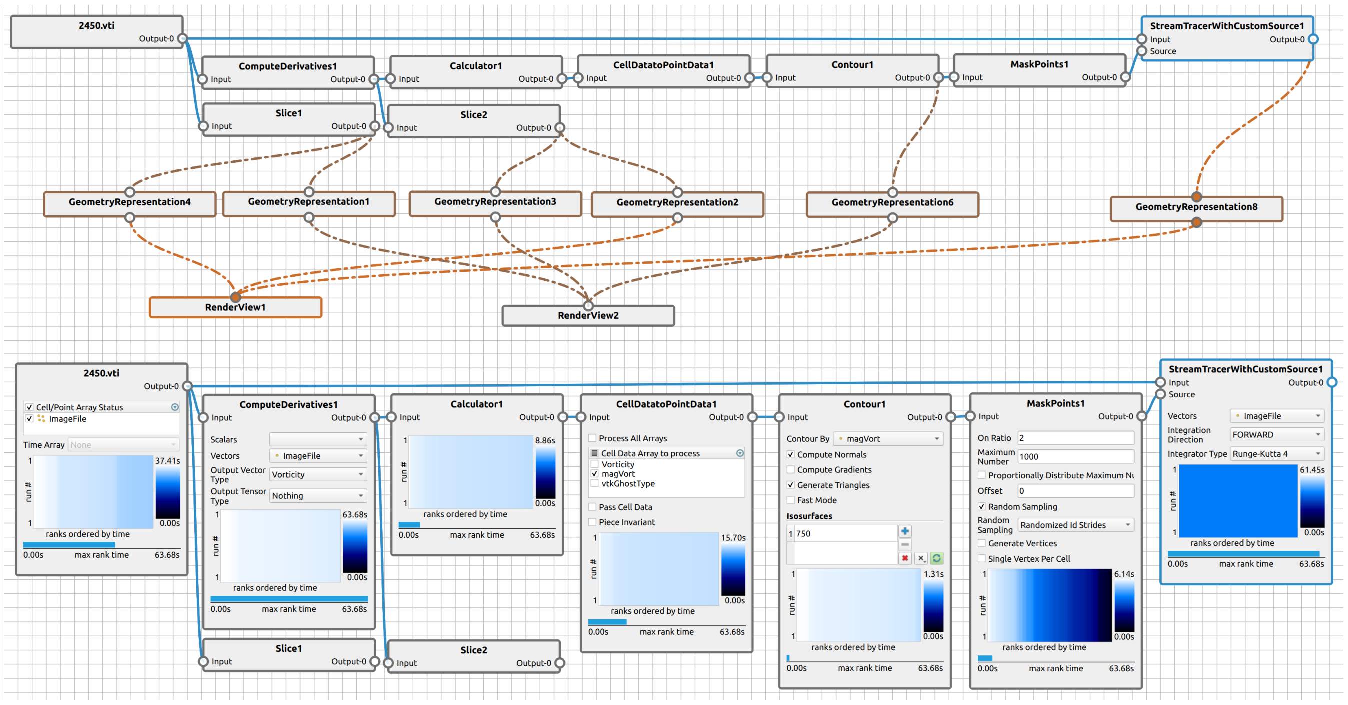
Additionally to the modules themselves and their parameters in the basic node editor, we include a runtime visualization per module that shows the modules runtime—or runtimes if executed on multiple ranks—in one of currently four representations. In the three chart representations, the y-axis represents the runtime in seconds—having the same scale for each module—and the x-axis shows the categories of the origin of the measurement, meaning whether the measurement was produced locally or on a server and if on a server, on which rank.

The first representation is a bar chart showing the runtime of the most recent execution. Thus, the first bar represents the local execution and every following bar is an additional server rank, which may further be divided into render- and data-server (Figure 1). This view enables easy comparison between module runtimes, due to the same y-scale (T1). Furthermore, load imbalances can be easily spotted in this view as they manifest in uneven distribution of the bar heights (T3). Since the maximum of the y-scale is chosen as the maximum over all modules plus a margin, identifying the modules with the largest execution time is also simple (T2).

If multiple timings should be registered but the order is not important, e.g., when processing a file series, we employ a box plot per rank and one accumulated over all ranks (see Figure 5), However, the bar chart and the box plots suffer from scalability issues when the number of ranks is too large.

To target this issue, we include a line plot where each line is getting thinner and more transparent the older the measurement is (see Figure 5). Further, the latest timing is colored differently. By connecting the measurements of each rank with a line, spikes in the plot that indicate load imbalance are still visible with larger number of ranks (T3). This representation enables comparison of consecutive runs, allowing to observe the impact of parameter changes or structural changes in the pipeline on the performance (T4). However, with increasing number of iterations it suffers from overplotting.

Hence, we integrate a heat map visualization (Figure 5d and Figure 5e). Here the iteration is shown on the y-axis—oldest (top) to latest (bottom)—and the ranks on the x-axis with color indicating the execution time. This way consecutive runs do not suffer from



2450.vti: ✓ Cell/Point Array Status; ✓ ImageFile; Time Array: None

ComputeDerivatives1: Scalars: ; Vectors: ImageFile; Output Vector Type: Vorticity; Output Tensor Type: Nothing

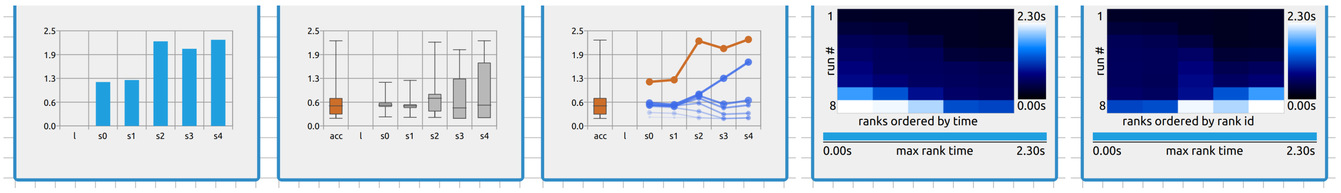
CellDatatoPointData1: ☐ Process All Arrays; Cell Data Array to process: ☐ Vorticity, ✓ magVort, ☐ vtkGhostType; ☐ Pass Cell Data; ☐ Piece Invariant

Contour1: Contour By: magVort; ✓ Compute Normals; ☐ Compute Gradients; ✓ Generate Triangles; ☐ Fast Mode; Isosurfaces: 1 750

MaskPoints1: On Ratio: 2; Maximum Number: 1000; ☐ Proportionally Distribute Maximum Nu; Offset: 0; ✓ Random Sampling; Random Sampling: Randomized Id Strides; ☐ Generate Vertices; ☐ Single Vertex Per Cell

StreamTracerWithCustomSource1: Vectors: ImageFile; Integration Direction: FORWARD; Integrator Type: Runge-Kutta 4

Figure 4: Two node editor views of the same pipeline. The top pipeline has the view nodes enabled but everything else hidden. The bottom pipeline hides the view nodes but shows the timings and parameters on some filters.



(a) (b) (c) (d) (e)

Figure 5: Different available visualizations for execution time. Figure 5a shows a bar chart with the execution time per rank represented by the respective bar height. The labels below indicate the server rank ($s1$-$s4$) or the local client l. Figure 5b and Figure 5c Show the box plot and line plot view respectively for displaying multiple timings. The line plot shows the most recent timing in orange, while older timings are thinner and have a higher transparency. Additionally there is a box plot showing the distribution over all measurements indicated by *acc*. In the heat map view (Figure 5d) the timings are ordered from oldest to newest execution from top to bottom. For each run, the execution times for each rank are shown in descending order from left to right, displaying the rank with longest time on the left. This ordering can be changed to show the execution times in ascending order of the corresponding rank id (Figure 5e), to compare timings of the same rank for multiple runs. The timings are encoded by color on a scale from 0 to the maximum time over all ranks and iterations of this filter. Beneath the heat map, a bar chart shows the longest execution time measured for a rank of the most recent execution of this filter in relation to the global maximum across all filters.

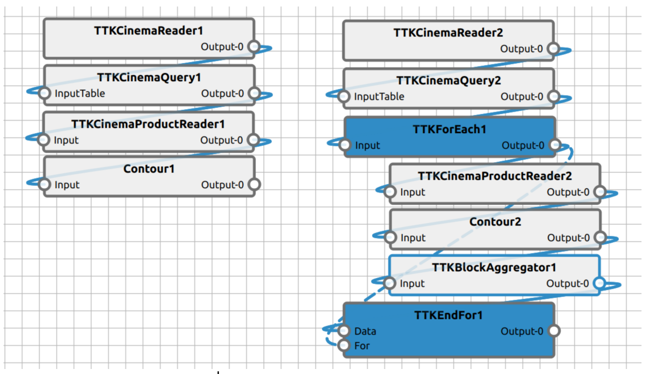
overplotting. In order to better compare consecutive runs, the rank measurements are sorted with respect to their execution time in descending order. The timings are encoded by color on a scale from 0 to the maximum time over all ranks and iterations of this filter (T4). Load imbalances manifest themselves as horizontal gradients in the heat map (T3). The varying local maxima for the heat map impedes comparison between filters. To remedy this, we include a bar chart beneath the heat map that shows the largest execution time measured for a rank of the most recent execution of this filter in relation to the global maximum across all filters (T1)(T2).

4.2. Implementation

Similar to *ParaView*'s TimerLog, performance data is retrieved via `GatherInformation` calls on the current session with `vtkPVTimerInformation` objects. Each filter has a *ParaView* internal unique id, which is used to map the timings inside the log onto the corresponding nodes. To this end, the log is processed with regular expressions. This map can be queried by the runtime annotation widget and is updated after each pipeline update. The runtime annotation widget uses *Qt Charts* add-on to visualize the runtime information for the three chart representations—bar chart, series of box plots and line plots. The heat map is implemented as a custom widget. One can iterate through the representations by clicking the widget. Currently only processing nodes are equipped with the runtime annotations.

5. Advanced Control Flow

The *VTK* processing pipeline can be represented as directed acyclic graph (DAG). This graph can be arbitrarily complex, and historically *ParaView* prioritized relatively simple, linear graphs. However, with the recent advances in scientific computing these graphs become more and more complex and even require new approaches to process data; especially in regard to processing ensemble data. The node editor simplifies reconnecting input and output ports via simple clicks, while at the same time displaying even complex pipelines in a comprehensible manner; in contrast to the



	execution time	peak memory usage
without for-loop	30.96 s	13.98 GB
with for-loop	36.29 s	1.13 GB

Figure 6: Two pipelines computing the same iso-contours on multiple blocks. The left pipeline does so via standard VTK multiblock execution and the right employs the for-loop filters. The pipeline with the for-loop, while being marginally slower, reduces the memory requirement for the pipeline drastically. The input is a database from which 100 blocks with size of 129 MB are read. The size of the output contours is 188 MB.

`Pipeline Browser`. In addition to these basic user interface features implemented in the *ParaView* code base, we also extended the *VTK* pipeline architecture in the *TTK* [TFL*18] code base to support *for loops* and *recursion*, i.e., partially cyclic pipelines. The node editor highlights such loop constructs by coloring the start and end nodes boundary and they become apparent as cycles in the graph view (Figure 6 and Figure 8).

5.1. Independent Iterations

It is a very common use case to apply a processing pipeline not on a single data object, but on a set of objects. Imagine a workflow that extracts features from a single timestep of a time series, or a member of an ensemble. Analysts might want to apply this workflow on every timestep or on all ensemble members. This can be expressed as a *For Each* operation where every individual pipeline execution is independent of the other executions. In the past, analysts had to implement this operation explicitly—e.g., via batch processing—but as discussed in the following it makes sense to incorporate such set operations explicitly inside the pipeline for convenience.

VTK already supports such set operations for pure linear workflows via `vtkMultiBlockDataSet`. A `vtkMultiBlockDataSet` object is a collection of `vtkDataObjects` called blocks (which can in turn be again `vtkMultiBlockDataSets`). One can think of a `vtkMultiBlockDataSet` as a list of data objects. If such a `vtkMultiBlockDataSet` is fed into a filter that was designed to process only an elementary data object type (such as `vtkPolyData`) then the pipeline executor will actually pass the blocks individually into the filter and collect the outputs in a new `vtkMultiBlockDataSet`. So the filter itself (and its code base) is unaware of the `vtkMultiBlockDataSet`, nor in which context it is executed in the pipeline. However, this very convenient feature is only available for linear pipelines where each filter must have exactly one input and one output. Additionally, the blocks of a `vtkMultiBlockDataSet` are all kept in memory. Thus, if one feeds 100 `vtkImageData` objects into a linear pipeline consisting of ten filters, then the machine has to keep 100×10 data objects in memory, which makes this feature only applicable for relatively small `vtkMultiBlockDataSets` and pipelines. A pipeline exemplifying this behavior is shown in Figure 6.

We overcome the aforementioned limitations by introducing two new *VTK* filters called `ttkForEach` and `ttkEndFor`. Note the prefix `ttk` stems from the fact that these filters are currently implemented in the `Topology ToolKit` [TFL*18], but we aim to move the code to the `VTK` code base in the future. The `ttkForEach` and `ttkEndFor` filters work in tandem to execute the pipeline spanned between them on every element of the `ttkForEach` input data object. The `ttkForEach` filter consumes a `vtkMultiBlockDataSet` and produces in every execution exactly one block as an output, i.e., the block at the index of the current iteration. To this end, the `ttkForEach` filter maintains the current iteration index and the maximum number of iterations as member variables. The `ttkEndFor` filter takes two inputs: the first must be the `ttkForEach` filter, and the second input is the supposed output of the for loop. Every time the `ttkEndFor` filter is executed, it uses the public interface of the `ttkForEach` filter to retrieve the current iteration index and the maximum number of iterations. Then the `ttkEndFor` filter checks if more iterations need to be performed (in which case the `ttkEndFor` filter asks the `ttkForEach` filter to start the next iteration), or if the current iteration was the last iteration (in which case the `ttkEndFor` filter forwards the loop output and the subsequent pipeline will be executed). The `ttkEndFor` filter maintains a `vtkMultiBlockDataSet` member variable and every time it is executed it places the last for loop output at the block of the current iteration index, i.e., it aggregates the for loop output over time. This aggregation can be turned off, in which case the filter only forwards the output of the last iteration. Note, this abstract tandem design makes it also possible to nest loops. Moreover, in the previous example of feeding 100 `vtkImageData` objects into a linear pipeline with 10 filters now only requires keeping $100 + 100$ objects in machine memory (the input and output of the for loop) compared to the 100×10 objects without the for loop structure.

5.2. Dependent Iterations

Another common use case is to feed the output of the current iteration into the next iteration. For instance, consider a particle advection problem where the pipeline that performs the advection needs to be performed multiple times until the particles converge to a fixed position. Previously, such functionality was only possible within a single filter (which contains the entire loop logic), or by auxiliary code that interferes with the *VTK* pipeline execution.

For such use cases we provide two new classes called `ttkWhile` and `ttkEndWhile`. The `ttkWhile` filter consumes a single data object that is forwarded as the output of the first iteration (the first execution of the `ttkWhile` filter). This output is processed through the subsequent pipeline until it reaches the `ttkEndWhile` filter, which checks a flag on the incoming data object and then either forwards the output, or otherwise sends the output of the current iteration to the `ttkWhile` filter (which is provided to the `ttkEndWhile` filter as an input). The new output is then fed from the `ttkWhile` filter as its output into the next iteration.

Currently, the stop criterion needs to be computed by some filter between the `ttkWhile` and `ttkEndWhile` filters, and if the stop criterion is not present then the while loop will exit after the first iteration. We plan to add more pipeline logic filters such as `ttkIf` to further abstract and modularize the *VTK* pipeline.

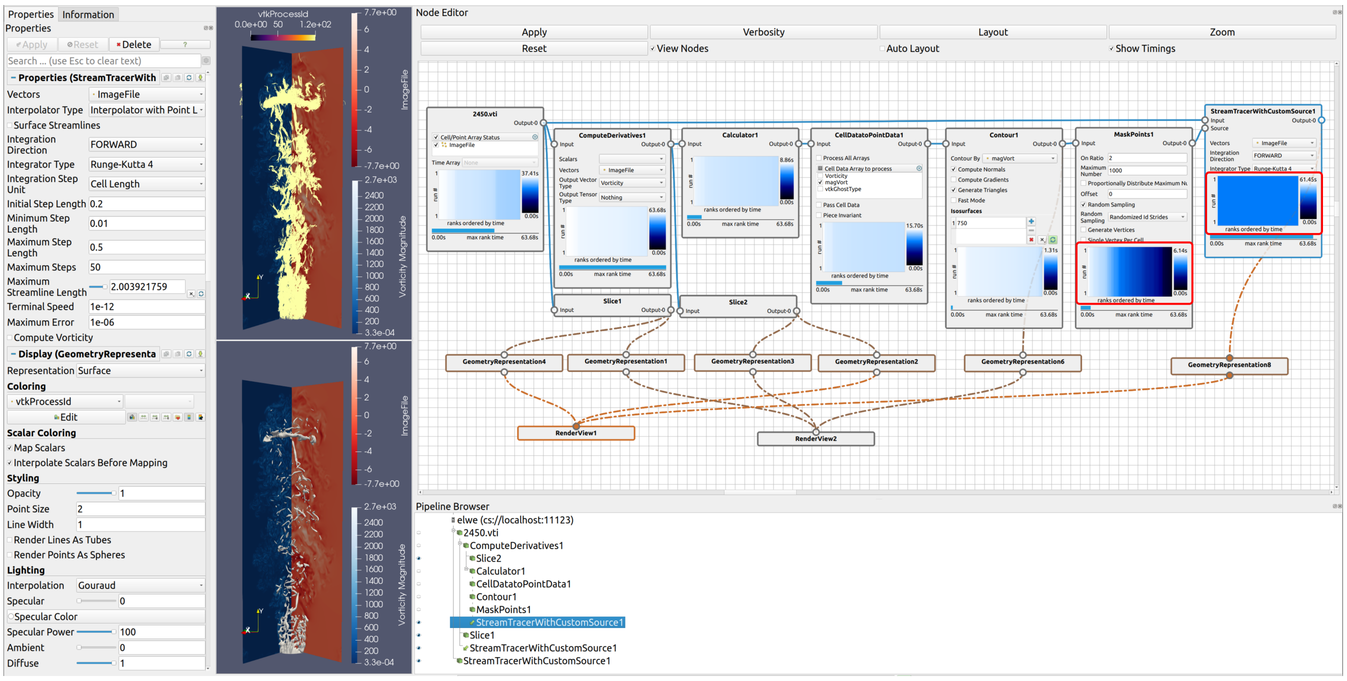
6. Case Studies

6.1. Multiview Flow Visualization

The first pipeline (c.f. Figure 7)—which exemplifies some of the features of the node editor—is a flow visualization of a vector field from a jet flow data set. It consists of a reader source, a filter computing vorticity, two slices showing vorticity and velocity, respectively, a contour filter of vorticity, a streamline filter with a custom source for the seed points (retrieved from the contour), and some helper filters. For comparison, the tree view is included on the screenshot at the bottom right. We first focus on the differences of the presented interface to the tree view, while exemplifying possible advantages and new opportunities in the workflow afterwards.

The first advantage over the tree view is that the view edges from nodes to the different render views make it directly obvious which nodes constitute to each view, whereas the pipeline browser only indicates the visible objects of the active view via an eye icon. This realization of the visualized output of filters via links is closer in its representation to the data flow that has to occur in order for the output to appear on screen.

Furthermore, filters that have multiple input ports—such as the `StreamTracerWithCustomSource` in this example or a `GroupDatasets` filter—appear in the tree view as a new root entry, and all their inputs are followed by a special reference entry



Properties
Search ... (use Esc to clear text)
Properties (StreamTracerWith

Vectors	ImageFile
Interpolator Type	Interpolator with Point L
Surface Streamlines	
Integration Direction	FORWARD
Integrator Type	Runge-Kutta 4
Integration Step Unit	Cell Length
Initial Step Length	0.2
Minimum Step Length	0.01
Maximum Step Length	0.5
Maximum Steps	50
Maximum Streamline Length	2.003921759
Terminal Speed	1e-12
Maximum Error	1e-06
Compute Vorticity	

Display (GeometryRepresenta

Representation	Surface

Coloring
vtkProcessId
Edit

Scalar Coloring
✓ Map Scalars
✓ Interpolate Scalars Before Mapping

Styling

Opacity	1
Point Size	2
Line Width	1
Render Lines As Tubes	
Render Points As Spheres	

Lighting

Interpolation	Gouraud
Specular	0
Specular Color	
Specular Power	100
Ambient	0
Diffuse	1

Pipeline Browser
elwe (cs://localhost:11123)
2450.vti
ComputeDerivatives1
Slice2
Calculator1
CellDatatoPointData1
Contour1
MaskPoints1
StreamTracerWithCustomSource1
Slice1
StreamTracerWithCustomSource1
StreamTracerWithCustomSource1

Figure 7: A visualization pipeline in *ParaView* showing a time step from a jet flow dataset, with two slices showing vorticity and velocity and an iso-contour of vorticity in the bottom view and streamlines seeded on the contour in the top view. One can observe that the `MaskPoints` and `StreamTracer` filter show a large difference in execution time between ranks (see the heat maps in the red boxes). Furthermore, the streamlines in the top view all have the same process id, this coincides with a higher execution time for one rank in the streamline filter, as shown by the white line in the heat map of the streamline filter on the left side. Although the load imbalance for the `MaskPoints` seems larger in the heat map, comparing the blue bars below the heat maps of the two filters reveals the larger impact of the `StreamTracer` filter and its load imbalance.

with an arrow icon. This multiplication of entries is unnecessary in the node editor. Therefore, the structure of the pipeline and the number of inputs in a filter becomes more readily apparent.

However, due to the occupied screen space and the size of the nodes in the editor zooming and panning becomes necessary if the pipeline consists of more modules. Especially, if parameters of the nodes have to be changed. For example, changing the slice view's and the stream tracer's parameters often could be laborious, although this can be mitigated by arranging the often manipulated nodes near each other. As already discussed, having the nodes next to each other—or at least inside the view—also allows inspecting and changing parameters without the parameters of the other filters disappearing, which creates a more thorough overview in only one image. Furthermore, it is still possible to use the properties panel outside the node editor, since the displayed property widgets are the same, which can be seen in Figure 7 with the `StreamTracer` filter's properties being shown on the left and in the node editor.

With the basic performance information displayed, there are several behaviors of interest that we can observe. The `ComputeDerivatives` filter is easily identified as the filter with the overall longest runtime, the contour's influence on the runtime is negligible and the measurements of the `MaskPoints` and `StreamTracer` filter show a large difference in execution time between ranks, indicating a load imbalance. In the top view, streamlines colored by process id can be seen. All of them are bright yellow, meaning all were assigned the same process id, which in this case is 119 and matches with the increased runtime on this rank.

6.2. Viscous Fingers

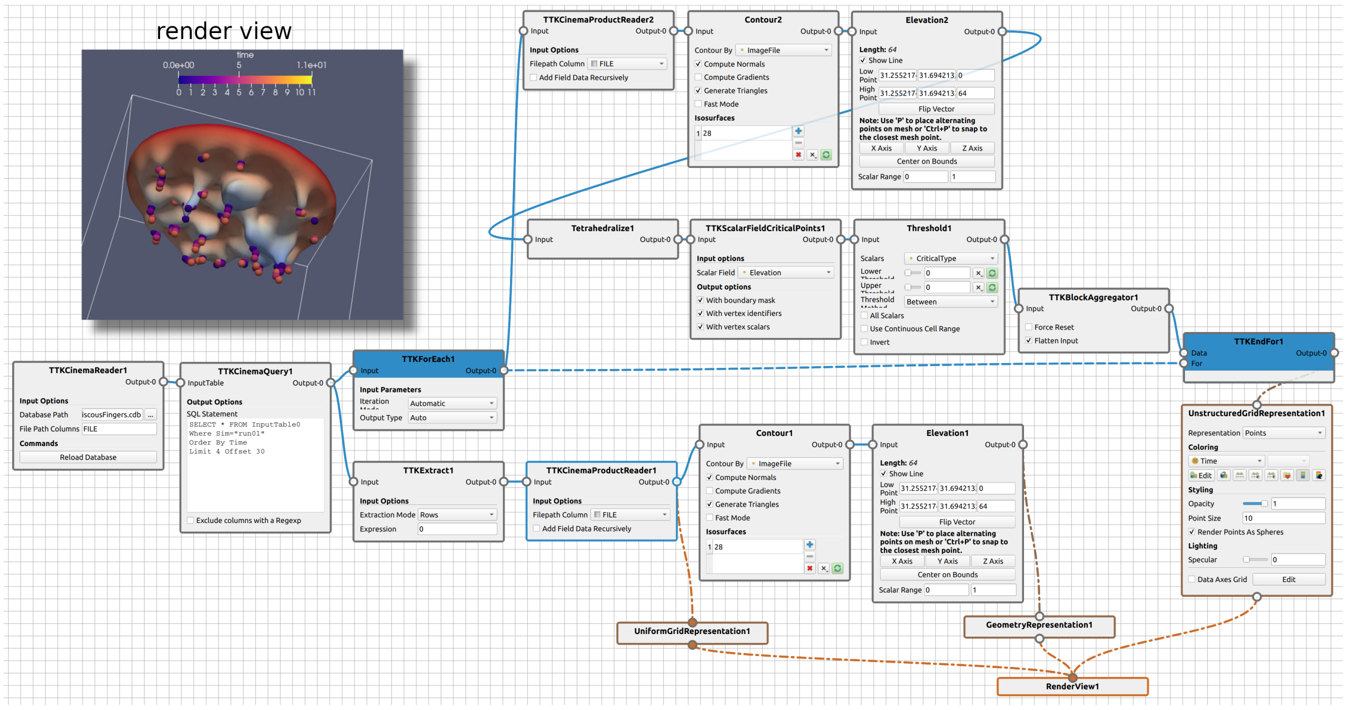
In this second example we show the structure of a pipeline that contains a for-loop. The pipeline at hand reads and queries a cinema database—the viscous finger database provided by *TTK* [TTK21]—and loops over the extracted time steps in order to extract minima in the elevation of an iso-contour created from the image data of the current time step. These minima are aggregated via the `TTKBlockAggregator` and colored by the time step in which they were extracted. Furthermore, the first time step is extracted and its iso-contour is visualized with the elevation w.r.t the z-axis, analogously to the procedure inside the for loop. The structure and output of this pipeline is shown in Figure 8.

As already mentioned in Section 5, a pipeline working on a `vtkMultiBlockDataSet` is able to achieve similar results, computing the contours and minima for each of its blocks. However, this approach does not expose the iterative behavior. Using the advanced control flow filters, this can be easily spotted in the node editor. Furthermore, performing each loop iteration sequentially potentially lessens memory requirements.

Similar to the `vtkStreamTracerWithCustomSource` node from the previous case, the `TTKEndFor` node has two inputs, but the second input is used to connect it to the `TTKForEach` node that has an additional output port for closing the loop. While *ParaView*'s tree view pipeline browser struggles to represent multiple input nodes, it becomes even harder to indicate this cyclical behavior. Especially for more complex pipelines with multiple for-loops and several multi-input filters.

6.3. Subset Extraction

Another use case is the iterative process of creating a visualization pipeline. We exemplify this again with a simple iso-contour pipeline (see Figure 9). After extracting contours over the whole data set (.pvti reader and *Contour1*), one identifies a region of interest and extracts only a subset of the whole data to compute the contours on (*ExtractSubset1* and *Contour2*). After inserting the subset extraction, we can observe a load imbalance inside the subset extraction and in the contour filter that is recognizable as a steep gradient on the left in both heat maps. To check whether a redistribution of data is beneficial, a redistribution filter is added after the subset extraction and before the iso-contour filter. Investigating the execution time of the contour filter behind the redistribution filter, one can still observe a gradient, but none of the ranks show black. While in this case the load is more balanced the overall execution time for the redistribution filter now dominates the runtime of the pipeline. Accepting the small load imbalance leads to lower execution times in this case.



TTKCinemaReader1 — Database Path: iscousFingers.cdb; File Path Columns: FILE

TTKCinemaQuery1 — SQL Statement:

```
SELECT * FROM InputTable0
Where Sim="run01"
Order By Time
Limit 4 Offset 30
```

TTKForEach1 — Iteration Mode: Automatic; Output Type: Auto

TTKExtract1 — Extraction Mode: Rows; Expression: 0

TTKCinemaProductReader1/2 — Filepath Column: FILE

Contour1/Contour2 — Contour By: ImageFile; Isosurfaces: 28

Elevation1/Elevation2 — Length: 64; Low Point: 31.255217 31.694213 0; High Point: 31.255217 31.694213 64

TTKScalarFieldCriticalPoints1 — Scalar Field: Elevation

Threshold1 — Scalars: CriticalType; Lower Threshold: 0; Upper Threshold: 0; Threshold Method: Between

TTKBlockAggregator1 — Flatten Input ✓

UnstructuredGridRepresentation1 — Representation: Points; Coloring: Time; Opacity: 1; Point Size: 10; Render Points As Spheres ✓; Specular: 0

Figure 8: Node editor view of a visualization pipeline in *ParaView* that aggregates minima of multiple iso-contours stemming from time steps of the viscous finger data set. The resulting visualization can be seen in the top left.

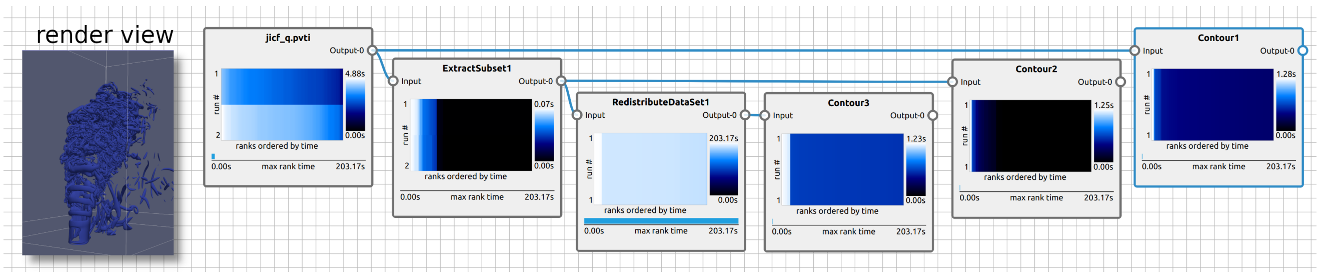


Figure 9: Multiple steps of creating an iso-contour pipeline visualizing the Q-criterion of a jet in cross flow [GGK*12]. The provided performance information helps to make decisions about the structure of the pipeline. The three steps, each ending in a contour filter, are ordered vertically.

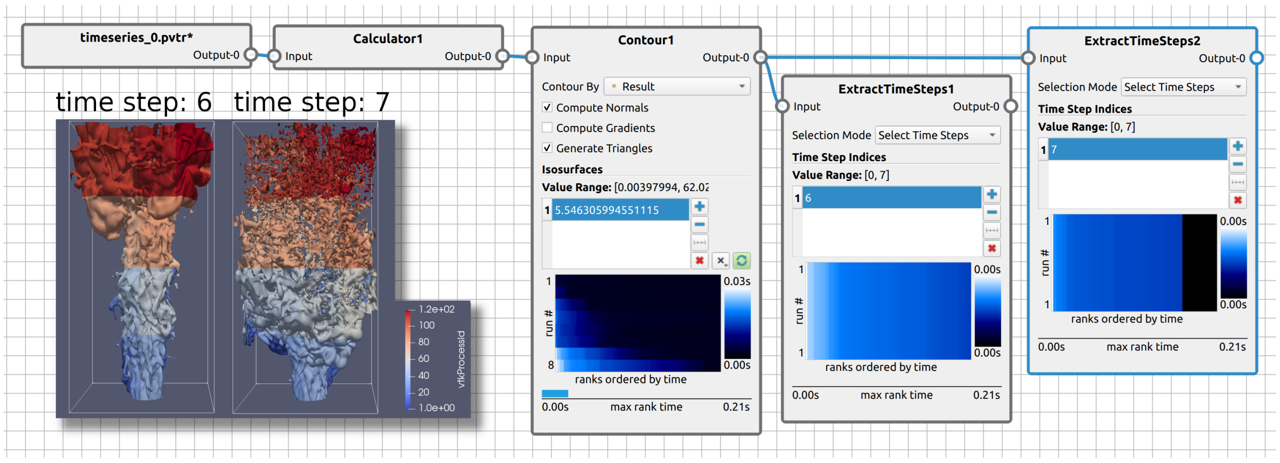


Figure 10: Iso-contour pipeline showing two time steps of a flow simulation, colored by process id. The heat map reveals an increasing execution time over the eight time steps and a higher usage of the 120 MPI ranks with each iteration in the contour filter. The last iteration displayed in the contour filter heat map results from the extraction of the second to last time step for rendering.

6.4. Runtime Inspection for Time Series

Consider the iso-contour pipeline for the velocity magnitude for multiple time steps of a flow simulation shown in Figure 10. Here, the performance information is used to identify trends and interesting points in time. The execution times of the contour filter show a small increase in load and execution time with each processed time step for the first six time steps and then increasing rapidly over the last two. Rendering the contours of the last two time steps, one can observe more geometry being created for the contour as the flow reaches the boundary and propagates back. Due to the increase in geometry and the more even spread across the domain, the execution time increases and more of the allocated resources are used.

7. Limitations and Possible Extensions

A limitation of VTK—which becomes apparent in the node editor—is that parameters cannot function as ports. However, in many pipelines it is desirable to use computed values of other filters as parameter values of the current filter, which is for example possible in *Blender* and *Inviwo*. The implementation of this feature would require a serious modification of the *VTK* architecture, but it is the opinion of the authors that such a modification will be very beneficial in the long run.

Currently, it is possible in the node editor to connect modules in a way that creates loops without the use of the mentioned for- and while filters, since it is also possible in ParaView without the node editor. This is not supported by VTK's pipeline design and will likely lead to errors. However, these cyclical dependencies are easier to spot in the node editor than the tree view.

A limitation in the extraction of runtimes is given by the dependency on *VTK* logging. Currently, multiple render views cannot be distinguished inside the log. Furthermore, there are executions of additional filters, e.g., the transformation of data into the right format for rendering, that are not part of the node graph and which runtimes are therefore not visible.

Including the runtime annotation on very large pipelines reveals a scalabilty issue, since the runtime annotation widget increases the size of each module drastically if it was on the lowest verbosity level beforehand. While grouping multiple filters as pipeline parts into custom filters works and reduces the number of modules visible in the node editor, these custom filters do currently not display performance information. Further future improvements to the interface include the following:

- The ability to snapshot the state of the pipeline and the runtimes of its filters for a comparative view.
- Grouping nodes and changing the visibility of whole groups.
- Exclude individual filters from maximum computation in the performance annotation.
- Providing information about the output data at each output port.

8. Conclusion

We described the implementation of a visual programming interface for *ParaView* and demonstrated its advantages over the existing pipeline browser. The node editor portion of this work is already integrated into ParaView 5.11. The design of the node editor follows other prominent interfaces such as the ones of *Blender*, *Inviwo*, *VisTrails*, and so forth. We further enhanced the visual programming interface with a performance annotation for each module, allowing all users to obtain a basic understanding of their pipeline's performance and make more informed decisions about potential changes. In addition to the node editor, we also contributed new *VTK* filters to support dependent and independent iterations of pipeline segments. This enables explicit modeling of loops in the pipeline; circumventing the need to break pipelines apart and to run batch processes. Although our implementation makes heavy use of existing *ParaView* source code, it remains completely optional and non-invasive to existing code.

Acknowledgments

This research was supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under project 398122172.

References

[AGL05] Ahrens J., Geveci B., Law C.: Paraview: An end-user tool for large data visualization. *The Visualization Handbook* (2005). 1, 3

[Aut] Autodesk, Inc.: Maya. https://autodesk.com/maya. [Online; accessed 24-April-2023]. 1, 2

[Ava] Avanced Visual Systems Inc.: AVS/Express. https://www.avs.com/avs-express/. [Online; accessed 30-November-2022]. 2

[BCC*05] Bavoil L., Callahan S., Crossno P., Freire J., Scheidegger C., Silva C., Vo H.: Vistrails: Enabling interactive multiple-view visualizations. In *VIS 05. IEEE Visualization, 2005.* (2005), pp. 135–142. doi:10.1109/VISUAL.2005.1532788. 1, 2

[CBW*12] Childs H., Brugger E., Whitlock B., Meredith J. S., Ahern S., Pugmire D., Biagas K., Miller M. C., Harrison C., Weber G. H., Krishnan H., Fogal T., Sanderson A. R., Garth C., Bethel E. W., Camp D., Rübel O., Durant M., Favre J. M., Navrátil P. A.: Visit. In *High Performance Visualization - Enabling Extreme-Scale Scientific Insight*, Bethel E. W., Childs H., Hansen C. D., (Eds.), Chapman and Hall / CRC computational science series. CRC Press, 2012. URL: https://doi.org/10.1201/b12985-21, doi:10.1201/b12985-21. 3

[Com] Community B. O.: Blender - a 3d modelling and rendering package. http://www.blender.org. [Online; accessed 24-April-2023]. 1, 2

[DHJ07] DeRose L., Homer B., Johnson D.: Detecting application load imbalance on high end massively parallel systems. In *Euro-Par 2007 Parallel Processing* (Berlin, Heidelberg, 2007), Kermarrec A.-M., Bougé L., Priol T., (Eds.), Springer Berlin Heidelberg, pp. 150–159. doi:10.1007/978-3-540-74466-5_17. 3

[EGK*02] Ellson J., Gansner E., Koutsofios L., North S. C., Woodhull G.: Graphviz— open source graph drawing tools. In *Graph Drawing* (Berlin, Heidelberg, 2002), Mutzel P., Jünger M., Leipert S., (Eds.), Springer Berlin Heidelberg, pp. 483–484. doi:10.1007/3-540-45848-4_57. 4

[EHWS10] Eichelbaum S., Hlawitschka M., Wiebel A., Scheuermann G.: OpenWalnut - An Open-Source Visualization System. In *Proceedings of the 6th High-End Visualization Workshop* (2010), Benger W., Gerndt A., Su S., Schoor W., Koppitz M., Kapferer W., Bischof H.-P., Pierro M. D., (Eds.), pp. 67–78. 1, 2

[GGK*12] Grout R. W., Gruber A., Kolla H., Bremer P.-T., Bennett J. C., Gyulassy A., Chen J. H.: A direct numerical simulation study of turbulence and flame structure in transverse jets analysed in jet-trajectory based coordinates. *Journal of Fluid Mechanics 706* (2012), 351–383. doi:10.1017/jfm.2012.257. 9

[IGJ*14] Isaacs K. E., Giménez A., Jusufi I., Gamblin T., Bhatele A., Schulz M., Hamann B., Bremer P.-T.: State of the Art of Performance Visualization. In *EuroVis - STARs* (2014), Borgo R., Maciejewski R., Viola I., (Eds.), The Eurographics Association. doi:10.2312/eurovisstar.20141177. 3

[JJT*07] Jasak H., Jemcov A., Tukovic Z., et al.: Openfoam: A c++ library for complex physics simulations. In *International workshop on coupled methods in numerical dynamics* (2007), vol. 1000, IUC Dubrovnik Croatia, pp. 1–20. 2

[JSS*19] Jönsson D., Steneteg P., Sundén E., Englund R., Kottravel S., Falk M., Ynnerman A., Hotz I., Ropinski T.: Inviwo - a visualization system with usage abstraction levels. *IEEE Transactions on Visualization and Computer Graphics 26*, 11 (2019), 3241–3254. doi:10.1109/TVCG.2019.2920639. 1, 2

[LB08] Lingaraju G., Bagewadi C.: Animation of scientific data using vtk designer. *International Journal of Computer Science and Network Security* (2008), 318–325. 1

[Mes17] Messina P.: The exascale computing project. *Computing in Science Engineering 19*, 3 (2017), 63–67. doi:10.1109/MCSE.2017.57. 2

[MeV] MeVis Medical Solutions AG and Fraunhofer MEVIS, Bremen, Germany: Mevislab – development environment for medical image processing and visualization. http://www.mevislab.de. [Online; accessed 24-April-2023]. 2

[NAW*96] Nagel W. E., Arnold A., Weber M., Hoppe H.-C., Solchenbach K.: Vampir: Visualization and analysis of mpi resources. *Supercomputer 63 XII* (1996), 69–80. 3

[NBJ*21] Nguyen H. T., Bhatele A., Jain N., Kesavan S. P., Bhatia H., Gamblin T., Ma K.-L., Bremer P.-T.: Visualizing hierarchical performance profiles of parallel codes using callflow. *IEEE Transactions on Visualization and Computer Graphics 27*, 4 (2021), 2455–2468. doi:10.1109/TVCG.2019.2953746. 3

[PJ95] Parker S., Johnson C.: Scirun: A scientific programming environment for computational steering. In *Proc. ACM/IEEE Conference on Supercomputing* (1995), pp. 52–52. doi:10.1109/SUPERC.1995.241689. 2

[SBT*23] Sakin S. A., Bigelow A., Tohid R., Scully-Allison C., Scheidegger C., Brandt S. R., Taylor C., Huck K. A., Kaiser H., Isaacs K. E.: Traveler: Navigating task parallel traces for performance analysis. *IEEE Transactions on Visualization and Computer Graphics 29*, 1 (2023), 788–797. doi:10.1109/TVCG.2022.3209375. 3

[Sci16] Scientific Computing and Imaging Institute (SCI): Scirun: A scientific computing problem solving environment. http://www.scirun.org", 2016. [Online; accessed 24-April-2023]. 1, 2

[Sid] SideFX: Houdini. https://www.sidefx.com/products/houdini/. [Online; accessed 30-November-2022]. 1, 2

[SLB*11] Schulz M., Levine J. A., Bremer P.-T., Gamblin T., Pascucci V.: Interpreting performance data across intuitive domains. In *2011 International Conference on Parallel Processing* (2011), pp. 206–215. doi:10.1109/ICPP.2011.60. 3

[SML06] Schroeder W., Martin K., Lorensen B.: *The Visualization Toolkit*, 4 ed. Kitware, 2006. 1

[TFL*18] Tierny J., Favelier G., Levine J. A., Gueunet C., Michaux M.: The topology toolkit. *IEEE Trans. Vis. Comput. Graph. 24*, 1 (2018), 832–842. URL: https://doi.org/10.1109/TVCG.2017.2743938, doi:10.1109/TVCG.2017.2743938. 1, 6, 7

[TTK21] TTK Contributers: *TTK Data Repository*. https://github.com/topology-tool-kit/ttk-data/tree/dev, 2021. [Online; accessed 24-April-2023]. 8

[Udu15] Udupa P. N.: VTK Designer. https://vtkdesigner.sourceforge.io/, 2015. [Online; accessed 22-October-2021]. 2, 3

[VSLNMS20] Veroneze Solórzano A. L., Leandro Nesi L., Mello Schnorr L.: Using visualization of performance data to investigate load imbalance of a geophysics parallel application. In *Practice and Experience in Advanced Research Computing* (New York, NY, USA, 2020), PEARC '20, Association for Computing Machinery, p. 518–521. URL: https://doi.org/10.1145/3311790.3400844, doi:10.1145/3311790.3400844. 3

[Wik21] Wikipedia: IBM OpenDX — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=IBM%20OpenDX&oldid=1017374869, 2021. [Online; accessed 22-October-2021]. 2