Time Dependent Processing in a Parallel Pipeline Architecture

John Biddiscombe, Berk Geveci, Ken Martin, Kenneth Moreland, and David Thompson

Abstract— Pipeline architectures provide a versatile and efficient mechanism for constructing visualizations, and they have been implemented in numerous libraries and applications over the past two decades. In addition to allowing developers and users to freely combine algorithms, visualization pipelines have proven to work well when streaming data and scale well on parallel distributed-memory computers. However, current pipeline visualization frameworks have a critical flaw: they are unable to manage time varying data. As data flows through the pipeline, each algorithm has access to only a single snapshot in time of the data. This prevents the implementation of algorithms that do any temporal processing such as particle tracing; plotting over time; or interpolation, fitting, or smoothing of time series data. As data acquisition technology improves, as simulation time-integration techniques become more complex, and as simulations save less frequently and regularly, the ability to analyze the time-behavior of data becomes more important.

This paper describes a modification to the traditional pipeline architecture that allows it to accommodate temporal algorithms. Furthermore, the architecture allows temporal algorithms to be used in conjunction with algorithms expecting a single time snapshot, thus simplifying software design and allowing adoption into existing pipeline frameworks. Our architecture also continues to work well in parallel distributed-memory environments. We demonstrate our architecture by modifying the popular VTK framework and exposing the functionality to the ParaView application. We use this framework to apply time-dependent algorithms on large data with a parallel cluster computer and thereby exercise a functionality that previously did not exist.

Index Terms—data-parallel visualization pipeline, time-varying data.

1 Introduction

In order to develop a comprehensive understanding of the phenomena present in a dataset, scientists frequently use a variety of different visualization techniques which may involve a sequence of several post-processing operations or visualization algorithms. Pipeline architectures are well suited to this type of work flow and we review existing implementations in the next section. However, existing visualization pipelines do not accommodate visualization algorithms that perform temporal processing of time-varying data. Although they are able to perform simple tasks such as animation, existing pipelines typically make only a single time snapshot of a dataset available to each algorithm.

One reason pipelines do not accommodate temporal processing is that there is not a canonical representation for time-varying data, or indeed a consistent set of representations. The following list illustrates some of the issues raised by different treatments of time:

- Data acquired from experiment/measurement may be collected from sensors at different sampling rates. Sometimes sensors fail to report, or generate spurious values.
- Simulations advance time differently. Some treat time as any other spatial dimension. Some produce uniformly distributed time series, while others do not. Some solutions do not have a unique temporal interpolant and those that do may have an interpolant that is nonlinear so that poor or misleading visualizations result when a linear approximation is rendered.
- *hp*-adaptive [23] and AMR techniques define meshes which change topology over time, leading to representations which may not be directly interpolated or have well defined solutions at intermediate time steps.
- In large simulations, limitations in hard disk bandwidth restrict the writing of state to disk (for later visualization and analysis) to less frequent and possibly irregular intervals which can lead to the same misleading and unappealing visualizations as large time step integrators.

Beyond representational problems for time-varying data, many visualization algorithms such as thresholding, isocontouring, cutting, clipping, etc. do not require information about the time-behavior of a dataset. Requiring implementations of these algorithms to be aware of time introduces an undesirable overhead. Different visualization algorithms may require a value to be present for all times while others can deal with missing or uncertain data. As we consider requirements for incorporating temporal processing into pipeline architectures, we must keep these caveats in mind.

The next section reviews existing pipeline architectures. After that, we present algorithms that define the requirements for temporal processing, consider several alternate pipeline designs, select one design and discuss its implementation and performance.

2 Related Work

The process of visualization is often modeled as a pipeline that transforms raw data through various stages that eventually result in a human-observable image [6, 11]. Thus, it is no great surprise that so many systems reflect this pipeline model by applying the pipes-and-filters architectural pattern [4] that allows a user to connect multiple independent components together [5].

Pipeline architectures are straightforward to implement, simple to use, and powerful in their expressiveness. Individual algorithms are encapsulated in components called "filters" that accept inputs and provide outputs. Filters can accept any number of prescribed inputs and outputs. Components that provide outputs but take no inputs are usually called "sources," and components that accept inputs but do not provide outputs are called "sinks." The output of one filter can be attached to the input of another filter.

Pipelines can be data driven where processing and data flows downstream from a modified filter or source or they can be demand driven where a request starts at the end of the pipeline (by the render system, writer, or specific user request) and flows upstream. Demand driven pipelines are far more powerful in that an iterative upstream and downstream process can query and negotiate requests while incorporating input from all filters in the pipeline. This architecture can be used to perform subset queries, memory constrained streaming, scalar subset

- *John Biddiscombe is with the Swiss National Supercomputing Centre, E-mail: biddisco@cscs.ch.*
- *Berk Geveci and Ken Martin are with Kitware, Inc., E-mail: {berk.geveci,ken.martin}@kitware.com.*
- *Kenneth Moreland and David Thompson are with Sandia National Laboratories, E-mail: {kmorel,dcthomp}@sandia.gov.*

Manuscript received 31 March 2007; accepted 1 August 2007; posted online 2 November 2007.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

queries, spatial subset queries, proper ghost or partition cell requests all with each filter in the pipeline properly incorporating its requirements and how it impacts the request.

Temporal data has been used in pipeline architectures such as VTK [22], OpenDX [1, 18], SciRun [20], and AVS [17, 25] by visualizing one time step at a time, independent from other time steps. But such an approach does not allow complex time dependent operations such as temporal interpolation, caching, and the computation of pathlines. Intelligent fetching, caching, and rendering of temporal data has been addressed by systems that directly render the data, such as volume rendering or direct isosurfacing [14, 19] but again these lack the ability to add any complex temporal operations into a user defined pipeline.

Pipeline systems such as Visit have extended the pipeline execution model of VTK by the use of *contracts* which allow the flow of data to be optimized by modules based upon their data requirements [7]. This is similar to the flow of information within the VTK 5 pipeline architecture [3, 15], which provides a mechanism allowing for the arbitrary addition of information keys by modules into the data flow, which in turn enables modules to react to and change their data requests in response to meta-information about the data.

Another framework, designed for the visualization of unsteady flow data (and also built on top of VTK) is Vista FlowLib [21], which uses a parallelization framework Viracocha [10] to handle the passage of data between modules on a distributed system. Data requests are handled by an additional layer of logic on top of the pipeline, which makes it possible to compute multiple time steps in parallel by launching multiple pipeline processes. VIRACOCHA differs from a traditional pipeline model by using CORBA to request resources from a centralized controller, it also has the novel ability to send the algorithm to the data, rather than the data to the algorithm. Although it uses a pipeline model to execute individual algorithms, the connections between them are managed by a separate system completely.

The work described in this paper complements those packages using VTK by incorporating a mechanism for handling time based data, permitting the creation of new a class of temporal-parallel visualization algorithms. In particular, the ParaView application [12], which is a data-parallel visualization application implemented on top of the VTK library, has been extended to support our new features.

3 Pipeline-Time Requirements

Algorithms are generally classified by either the type of operation they perform or by the type of data upon which they operate. In this section, we present representative time-dependent data visualization and post processing tasks categorized by the structure of the time input that they require and by the way that the algorithms are driven (or iterated). By classifying tasks in this way, we show that a number of broad classes of temporal operations can be grouped together – and that once the pipeline-time requirements for each category have been met – the implementation of each of the presented methods is in principle possible. Note that the categorization in Table 1 is not strict, but rather, serves as a guideline for how an implementation of the given task would be broken down. Certain of the algorithms could be placed in one or more groups since they are defined in the broadest terms.

The algorithms of group 1 from Table 1, are characterized by the fact that they produce output continuously in response to input from either a GUI or another task. A single request for output (such as a frame rendered) requires an algorithm to request multiple (generally contiguous) time slices of data as input upon which it then operates to produce the requested output. The process is then repeated for the next frame with the output being usually part of a sequence making up an animation, or part of another process which eventually results (by virtue of further post-processing) in an animation. The group 1 tasks are further defined by noting that the time steps required will in general be a small number – examples such as interpolation (1b) (including polynomial N^{th} order), particle tracing (1e) or smoothing (1c) will generally be satisfied by a subset of the time steps from the original data and not the dataset in its entirety – this allows the implementation to request all the required data in one go. Note that mode shape animation (1d) differs slightly from the other tasks because time represents a parameter which is used to interpolate between datasets which represent displacements. However, the implementation requires only that data for particular modes (which can be represented as time steps) be available for an interpolation to take place.

Table 1. Tasks grouped by time requirements and implementation.

Iterative tasks using contiguous chunks (two or more steps) of time	
1a	Animation (using linear or nonlinear time) of data with non-linear, non-uniform or missing time steps. Including interpolation (1b) of values.
1b	Linear (2 steps) and polynomial interpolation (N steps) of data over time.
1c	Moving average (or other smoothing) of data over time.
1d	Animation of mode shapes (vibrational displacements).
1e	Particle tracing/advection (pathlines/streaklines) in unsteady vector fields.
Tasks requiring (subsets of) data between times t_0 and t_1	
2a	Plot fields (and compute derived fields) over time.
2b	Space-time height-field plots.
2c	Integrate over a cutting surface and over time.
2d	Extract time window of data values for elements between times.
2e	Compute envelopes (convex hull) over time.
Tasks requiring one step progressively	
3a	Motion trails/pathlines (*e.g.* see the path of a bouncing ball).
3b	Display of sensor data as it is received.
Tasks requiring random or semi-random access to time	
4a	Data-mining, searching, descriptive statistics and correlations over space and/or time.
4b	Retrieve non-contiguous ranges of time-varying data.
4c	Comparative visualization of data using time shifting (and/or scaling).
4d	Comparative visualization of measurement and simulation data.
Miscellaneous additional requirements	
5a	Caching of data to avoid wasted or unnecessary IO.
5b	No dependence on time, or overhead, for algorithms that are ignorant of it.

Group 2 differs from group 1 in that a single request for an output will (in general) be enough to satisfy the plotting/viewing requirements. A graph of some value (either derived or original data) extracted over times t_0 to t_1 is usually plotted in a single pass, rather than updated on a frame by frame basis. Clearly there are times, such as when plotting sensor acquired data in real time, when this rule is broken (c.f. group 3 item b), but the tasks of group 2 are usually executed once to produce a single numerical or graphical result. We note that if these operations are to be performed as part of a continuously updating process (such as the change of a convex hull over a windowed region of time), then they can be treated as a group 2 task wrapped inside a group 1 layer. Whilst plotting of a value from one or more probed locations (2a) usually requires a small set of data points, space-time plots (2b) and integration over surfaces (2c) may require time data from larger regions of space – this is generalized as item 2d since it may be a requirement of a great many custom visualization/analysis methods. These algorithms also differ from group 1 because they may operate on the whole range of times present in the dataset (i.e. t_0 and t_1 cover all available times) – this places a restriction on the implementation that not all data can necessarily be requested in a single pass if memory allocation is likely to exceed that available – this implies that multiple data requests may need to be issued for each output data value.

The example of motion trails in group 3 represents a type of visualization which is possible using a conventional pipeline since only one time step of data is actually required at any time. If it is assumed that a pipeline is already capable of looping over time steps, then all that is required is for a filter to 'listen' to, or 'watch' data passing through and capture positions (or other desired values) which can be used to

build up a trajectory on the fly. However, when time steps are irregular or acquired out of order, an implementation may need access to additional information about the time values associated with each step of data to produce an acceptable result. This places the requirement that time information must be present and flow through the pipeline even when not specifically requested by a filter.

Group 4, featuring random access to time steps, encompasses algorithms which include searching, data mining, topology-tracking [24], and statistical operations (4a), all of which may require an arbitrary number of time steps (generalized as 4b) in arbitrary order. Additionally, comparative visualization (4c) of the same data at two different times requires the ability to handle quite different times on different branches of a pipeline – whilst comparing measurement data and simulation data (4d) may involve displaying the nearest available time step from one or other datasets. The examples of group 4 demonstrate that we require the ability to perform operations on time values themselves within pipeline requests, so that a request for time t_x may be modified during processing and result in a request for $\{t_a, t_b, t_c, \text{etc.}\}$ where the relationship between the original and final times may not be obvious.

The final category, group 5, refers to miscellaneous requirements that arise from general observations of time-dependent data. We do not wish existing algorithms which do not support time (5b) to be dependent on new time related features or to be encumbered by them. Caching of time steps (5a) will be essential for iterative tasks that reuse the same time values on consecutive passes. Additionally, within branching pipelines where the same data may be used, but the order of operations may be different, there should be caching of data to prevent retrievals from one branch invalidating data in another.

We can summarize this section by observing that a pipeline which is capable of passing only a single time step from filter to filter will not meet our needs and we must therefore extend the pipeline to handle the following

- Data requests for multiple time steps must be possible in a single pass.
- Time requests need not be sequential or contiguous.
- The capability of filters to impose their own requirements for time in addition to those directly requested from downstream must exist, this implies the ability to augment time requests, which may be to add, remove or change values as they pass through the pipeline.
- A mechanism that permits a single request for data to spawn multiple further requests must exist.
- Information about time must be available to filters – but it must *not be required* for operation if it is not needed.
- Existing sources/filters which have not been designed to support multiple time requests should still operate within the new framework. Additionally, components that have no concept of time should continue to work well with those that do.

In the next section we discuss architectures that can meet these requirements by passing data for multiple time steps between filters in the pipeline.

4 Pipeline Architecture

Modern implementations of visualization pipelines require many pieces of "meta" information to be communicated through the pipeline. For example, when streaming data or distributing data across parallel machines, it becomes necessary to add extent, piece, and ghost information to the update requests [2]. Recent work has proven it beneficial to extend this mechanism to also allow general metadata to be passed up and down the pipeline [3, 7]. Within the VTK framework this metadata is stored in "information" objects, which hold data in key/value pairs where keys are arbitrary tokens with symbolic names helping to identify the role of the data they hold [15]. The controller

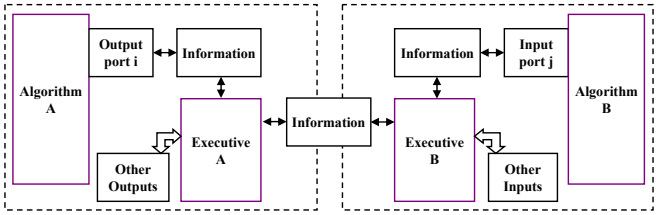

Fig. 1. Connections in a pipeline using executives to manage information passing between the algorithm modules

responsible for sending/receiving information and data between algorithmic modules broadly follows the strategy design pattern [9], with the pipeline split into two units: an executive object and an algorithm object. The executive unit controls the operation of the pipeline and maintains the state of the filter. The algorithm object encapsulates the code that transforms input data into output data. Pipeline objects are connected as illustrated in Figure 1 and the executives are described more fully in section 4.2.

4.1 Design Alternatives and Selection

Within the context of a parallel processing/visualization framework, several possible pipeline designs were considered as candidates for temporal support

- Allow each time step to be a single piece of data so that parallel execution could take place on different time steps on separate processes. In principle, simultaneous requests for data could be made to different processes allowing multiple time steps to be retrieved on request.
- Add a TemporalAlgorithm base class which would instruct the executive to loop the upstream portion of the pipeline N times in order to fetch N time steps.
- Add support for a TemporalDataSet collection and modify the executive to loop the upstream pipeline, gathering the results into the collection before passing them on to the temporal algorithm. In addition to adding a temporal collection data type, add support for multiple time requests by enhancing the request type itself.

The first solution is straightforward and elegant, however it suffers from the serious flaw that if each time step of the data is too large to fit into memory, the system will fail to work. Existing pipelines allow datasets to be broken into pieces and this functionality must be maintained if large data is to be processed in parallel.

The second solution was proven to be workable, but suffers from the drawback that it is not possible to fulfill the requirement of supplying multiple time steps simultaneously. Only algorithms operating within group 2 of Table 1 could be implemented with this strategy reliably. Whilst this design has not been selected for implementation, the basic idea of allowing an algorithm to loop the executive has been used and is described in section 4.5.

The TemporalDataSet solution allows multiple time steps to exist inside a container collection object and thus allows multiple steps to be simultaneously requested from a source. To support parallelization, a temporal dataset may consist of N pieces of data split across processes, however the datasets within each time step are split, rather than the time steps being separated as pieces. This solves the problem of the size of each step exceeding memory requirements and allows for arbitrary parallel expansion. Information about pieces is passed up and down the pipeline as before, but time based information is added.

4.2 The Role of Executives

Separating each functional unit into executive and algorithm provides features we utilize to support time requests. A careful look at Figure 1

shows that the algorithm objects do not directly communicate with each other, this allows two key operations:

- The executives can insert additional time related metadata into the information stream. This data is used to provide the control mechanism that satisfies our time requests without affecting the algorithms themselves.
- All traffic whether it be data or metadata, passing to or from an algorithm, is in fact passed from executive to executive via information objects before entering the algorithm. This allows an executive to change the data that is produced by one algorithm before passing it on to another. The executive uses this feature to collect data from multiple time steps before passing it on to a filter requiring temporal data.

4.3 Temporal Pipeline Extensions

We have enhanced the capabilities of the executives to recognize and act upon new information keys which have been introduced to support temporal data. The following key/value pairs have been defined in our pipeline.

TIME_RANGE A 2-vector containing the minimum and maximum times at which data produced by a source (or filter) are defined. This range may be queried by downstream filters of user interface components to limit the range of allowable operations.

TIME_STEPS A vector containing all the possible discrete time steps of data that are defined in the output produced by a source. A source capable of delivering data for any time (i.e. continuous time) supplies only the above TIME_RANGE key and does not need to provide any TIME_STEPS. Typical data readers will expose those time steps that they are capable of delivering in this key, which will usually be those generated during a simulation and stored on disk. There is no requirement that the steps are at regular intervals and during pipeline requests if a source is not capable of delivering a particular time, the requested time can be snapped to the nearest value present.

UPDATE_TIME_STEPS A vector containing the list of times required by a filter before it attempts to update itself. The majority of filters do not operate on any specific time value and do not set this key, however a filter which specifically requires N time steps will set this key with the values it requires. The values themselves will usually be a subset of the TIME_STEPS exported by a filter further upstream. Where an upstream source is providing continuous time in the form of a TIME_RANGE key, the update request may contain any values within the range.

DATA_TIME_STEPS A vector containing the actual time steps that were generated by a source. Note that this differs from TIME_STEPS in that the former represents all steps that could be produced, whereas the latter is those that were actually produced during a pass of the algorithm. The DATA_TIME_STEPS is attached to the data itself, rather than the information about the input or output connection port of the algorithm.

REQUIRES_TIME_DOWNSTREAM There is no value associated with this key, it is a flag which is set by one executive to inform another executive upstream that specific time requests have been made.

CONTINUE_EXECUTING There is no value associated with this key. It is used internally within a pipeline component to signal that it needs to repeat execution.

In addition to these keys, we have added the TEMPORALDATASET data type which stores multiple instances of other data sets, each defined at a different time. The TEMPORALDATASET type can be generated directly by a source if it supports this capability, but most will not.

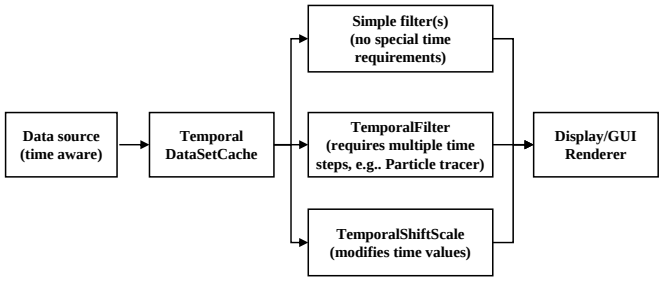

Fig. 2. In cases where a pipeline branches, different times may be required on different paths. In this case, the TEMPORALDATASETCACHE filter acts as a buffer separating non-temporal and temporal portions of the pipeline.

4.4 Basic Temporal Pipeline Operation

In its simplest operation, a pipeline can update its output in one pass: Upstream components process their inputs and feed the result to their outputs. However, when we introduce time to the pipeline, we require multiple passes to communicate information about time. The following keys (listed in the order in which they are performed) are used to designate the passes we require to update the pipeline.

REQUEST_DATA_OBJECT This request is used by the executive to generate the correct data type for a filter to use when generating its output. By default each filter declares itself as a producer of a particular kind of data (structured/unstructured/etc). However, when a filter requiring temporal data requests data from a simple filter operating on a stream of time dependent data the executive can replace the data object generated by the simple filter after each pass with a TEMPORALDATASET collection before giving it to the consumer. This pass plays a crucial role in cases where the pipeline branches between temporal and non-temporal portions (see Figure 2 and section 4.6).

REQUEST_INFORMATION In this pass, sources report the TIME_RANGE and any discrete TIME_STEPS for which they can produce data. By default, filters pass this information from up to downstream, but they can also alter the values for the downstream components if their operation performs some change to the time semantics.

REQUEST_UPDATE_EXTENT In this pass, the user-provided UPDATE_TIME_STEPS is passed upstream from sinks toward sources. Again, filters pass this information by default, but can modify them if they change the time semantics or require further information to complete their request.

REQUEST_DATA This final pass is where filters are actually executed and generate their data. When a filter begins executing, its output information will hold the UPDATE_TIME_STEPS that are required. A filter that requested temporal data will have time steps present as a temporal input and can access them directly.

These pipeline passes dovetail well with parallel pipeline execution. The REQUEST_INFORMATION pass can also be used to report ways in which data can be distributed and the REQUEST_UPDATE_EXTENT pass can be used to propagate piece requests [16]. These passes also mirror other pipeline systems that optimize the data processing in the pipeline [3, 7].

It is often desirable to directly connect a filter that requests a temporal data set input to an algorithm that does not directly produce this data type. Our pipeline implementation supports this connection. In this case, the downstream executive places REQUIRES_TIME_DOWNSTREAM in the upstream executive's information object. When the upstream filter receives the request, its executive will loop the upstream portion of the pipeline over each time

value in UPDATE_TIME_STEPS. The resulting data is collected in a temporal data set and passed during REQUEST_DATA_OBJECT instead of the original simple data. This achieves a primary goal of allowing non-temporal filters to co-exist with temporal ones. Since UPDATE_TIME_STEPS is a vector key capable of holding any number of time step values, in any order, and algorithms that operate with time can be connected to algorithms not supporting time, we are able to support all the tasks in group 1 and group 4 of Table 1. Whilst we have not listed any algorithms that run time backwards (such as finding the position of a particle at time t in the past the ability to request or modify any time values as they pass through the pipeline means that algorithms that run time backwards are just as straightforward to implement as those that run forwards.

The looping behavior is similar to how some current systems handle multi-block data [15]. However, is worth noting that iteration over time co-exists with looping over blocks of a multi-block dataset – with one subtle difference. When looping over time, it is sufficient to place a new time value on the output of the filter, trigger an update (which will in turn update anything upstream) and collect the results after N passes. When looping over blocks, it is necessary to replace the input of each filter with each successive block prior to the update. We combine both these loops in our executive by nesting block looping inside of time looping. In this way we are capable of implementing multi-block temporal algorithms in parallel. Figure 3 summarizes the overall flow of data and information within our parallel visualization architecture.

4.5 Iterative Temporal Pipeline Operation

Our pipeline also supports the ability for an algorithm to loop itself over multiple time steps rather than request the data for all time steps at once. This is required for the implementation of tasks/algorithms from group 2 of Table 1 and is particularly useful for algorithms that require many time steps to complete their operation but require only a small window of data for any particular calculation. In this situation, requesting all of the data is unnecessary and prohibitive for large data. An algorithm loops itself by passing CONTINUE_EXECUTING to its own executive as it finishes the REQUEST_DATA process. This signals the executive to re-update itself and gives the algorithm a chance to continue with data for a new time step. Portions of data are specified by placing a specified piece or extent in the REQUEST_UPDATE_EXTENT in the usual manner.

4.6 Branching Pipeline Considerations

Figure 2 shows an example of a visualization pipeline where one path has no need or knowledge of time, whereas another path has explicitly requested multiple time steps. In this situation we introduce a TEMPORALDATASETCACHE object which can hold several steps (or pieces of steps when operating in parallel) in memory and supply them on request rather than forcing the upstream pipeline to re-execute. When the simple filter requires updating, it requests data from the cache, which is controlled by its own executive. The executive triggers the 4 passes mentioned previously and passes the generated data onto the simple filter. When the TemporalFilter updates itself, the cache's executive detects that multiple time steps are required and triggers the 4 passes (which may be redundant if the data is already cached) and collects the data after each iteration. The executive passes a TEMPORALDATASET to the temporal filter, but a simple one to the simple filter.

We also introduce a TEMPORALSHIFTSCALE object which manipulates time but does not modify data. This filter can be used to arbitrarily adjust the time values between some desired range, effectively 'normalizing' the time between different paths such that the GUI can iterate from T1 to T2, but the paths can each 'see' their own time values. In this way (when combined with a cache) data may be visualized concurrently at different times, or data from two sources with different time ranges can be displayed together. Note that if the cache were not present, the fetching of data from the source at different times would set a 'modified' state in the other branch and cause unnecessary updates – this would be particularly serious problem if multiple branches requested the same or similar time steps, but in different orders, the cache can be set to hold enough time step (or time step pieces) to satisfy all branches without causing re-execution for a particular step value.

5 ENABLED TECHNOLOGIES

Many visualization algorithms are difficult or impossible to establish within a pipeline that does not support temporal requests. In this section we present our implementations of 6 visualization methods covering groups 1-4 of Table 1. By covering each of the groups introduced earlier, we show that all of the visualization types introduced earlier are possible in our framework.

5.1 Temporal Interpolation

Of the data that contains a temporal component to it, the vast majority that we and our collaborators visualize is defined at discrete moments in time. This discrete nature of the data is simply due to the nature of data acquisition devices, the iteration of simulations, and the constraints of digital storage. Occasionally we wish to perform operations on the data as if they were defined over a continuous time range, allowing a consumer of the data to make requests for time steps which do not exist in the original, perhaps regenerating missing data values or extracting regular sequences (of higher or lower resolution) from irregular data and vice versa.

We can achieve the conversion from discrete time to continuous time by interpolating the data between two adjacent time steps. Application of this interpolation needs to be explicit: It is an approximation and makes assumptions about the data that may not always be valid. We can do this by providing a TEMPORALINTERPOLATOR filter.

The TEMPORALINTERPOLATOR filter participates in three update passes of the pipeline. In REQUEST_INFORMATION, the filter passes the TIME_RANGE information unchanged, but hides the TIME_STEPS information from the downstream components. In essence, this reports that data exists in a continuous time range.

In REQUEST_UPDATE_EXTENT, TEMPORALINTERPOLATOR changes the UPDATE_TIME_STEPS to the list of discrete values reported by the upstream TIME_STEPS that are required to interpolate datasets matching the downstream request. (In the case where the input has no TIME_STEPS or a requested time step coincides with an existing value in TIME_STEPS, the requested time step is passed upstream, and TEMPORALINTERPOLATOR does nothing).

In REQUEST_DATA, the filter performs a linear interpolation of the fields in the data of the two time steps requested of the upstream component. The TEMPORALINTERPOLATOR filter verifies that the topology is the same in both time steps; the results are invalid otherwise. In the case of multi-block or hierarchical data, each level is traversed and interpolation occurs between equivalent blocks at each time.

Although we do not yet support it, it is straightforward to expand the operation of TEMPORALINTERPOLATOR to non-linear interpolation. It simply needs to request more time steps in REQUEST_UPDATE_EXTENT and perform a polynomial interpolation in REQUEST_DATA. Our implementation supports interpolation of any data type including multi-block and hierarchical datasets providing the topology of the data within each block does not change between time steps and the hierarchy of blocks is invariant. Interpolation of AMR-like datasets would require the provision of some specialized interpolant (or re-gridding algorithm) to define the structure of the data at intermediate times.

5.2 Animating Mode Shapes

We usually think of temporal data as being a sequence of data values over time, however, it is possible to characterize how data changes over time without storing a time-sequence of values. Such is the case for data from mode shape simulations, where the harmonic frequencies and vibrational modes of an object are computed by assuming each vertex v has a deflection

$$d_v(t) = A_v \sin(\omega t + \phi).$$

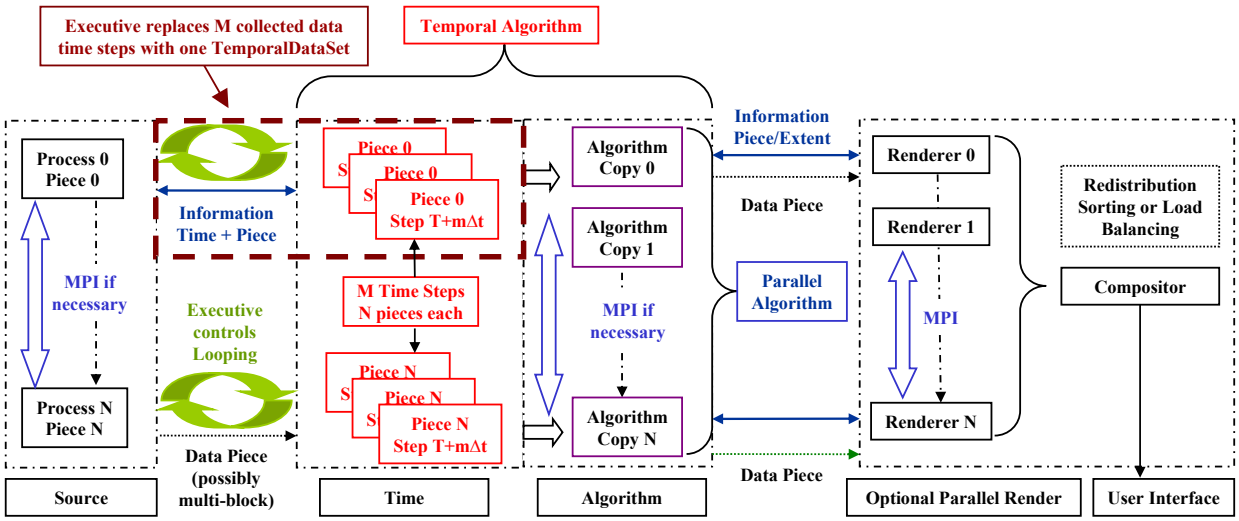

Fig. 3. A symbolic representation of data and information flow within a parallel pipeline architecture. Data requests originate from the application, which, if capable of parallel rendering or data processing, triggers update requests in parallel data pipelines. These requests propagate upstream as information, holding piece numbers (corresponding to MPI process Ids) and extents of data requested. When these requests reach temporal filters, the executives initiate looping and collecting of data, with pieces of multiple time steps held in each TEMPORALDATASET. If a source of data is itself capable of producing temporal data (i.e. multiple steps simultaneously), then the executive does not need to loop and simply passes a collection downstream directly. The splitting of data within each time step across processes allows temporal processing of very large datasets to take place even when the memory requirements of a single node do not permit the loading of a single complete time step.

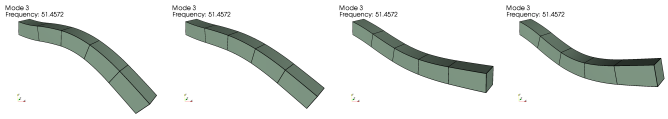

Fig. 4. Animation of one mode shape, which shows a bar's motion from one extreme of a vibration to the other and back.

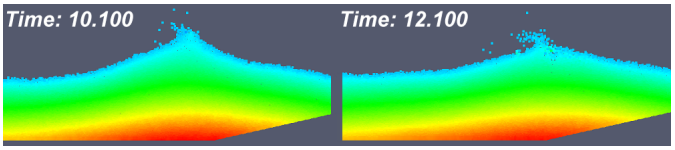

Fig. 5. Side-by-side comparison of wave breaking data at two times separated by one wave period.

Substituting this assumed solution into a governing differential equation of the form $[K]d + [M]\ddot{d} = [0]$ yields an eigenvalue problem of the form $\left([K] - \omega^2 [M]\right) A \sin(\omega t + \phi) = 0$ where $[K]$ and $[M]$ are "stiffness" and "mass" matrices from the differential equation [13]. This yields a set of resonant frequencies ω_j and peak deflections $A_{v,j}$. Rather than record a series of displacements $d_{j,v}$ at times t_j, a series of peak displacements $A_{v,j}$ are stored for each frequency ω_j. A mode shape is the deflection over time associated with a single resonant frequency: $d_{v,j}(t) = A_{v,j} \sin\left(\omega_j t + \phi\right)$.

Given a data source capable of supplying mode shape data, it is only necessary to modify it to provide a TIME_RANGE from 0 to 1 and arrange that when the reader receives a request for data at time t, it displaces each vertex in the dataset by $d_{v,j}$, as shown in Figure 4. Prior to implementing our pipeline, it was necessary to set up a sequence of keyframes to produce a mode shape animation – with the new pipeline, a single click of a 'play' button (in the ParaView GUI) delivers the data.

5.3 Temporal Comparative Visualization

The analysis of periodic data can be enhanced by comparing results from one period with the equivalent results from subsequent or previous periods, this allows subtle changes that may not be visible directly to be detected.

To create a visualization of the same data at two different times, we make use of a branching pipeline such as shown in Figure 2. Along one branch, we display the original data, but along another we apply a TEMPORALSHIFTSCALE filter with, (in this example) a shift equal to the period of the phenomena of interest and a unity scale factor. The filter intercepts the time passed during REQUEST_UPDATE_EXTENT and replaces it with the modified value (which may be positively or negatively offset). The GUI makes a single request for time, but the time shift results in two distinct and different datasets being displayed. The visualization can be enhanced by providing additional filters to perform subtraction of one dataset from another to display a quantitative difference between the two time snapshots. Figure 5 shows an example of a wave breaking simulation where two time values are displayed side by side.

In the case of comparative visualization of measured versus simulated data where the time steps or time resolutions of the datasets are not identical, we have provided a TEMPORALSNAPTOTIMESTEP filter similar to TEMPORALSHIFTSCALE which forces the pipeline to request a time which is guaranteed to exist upstream. This enables one branch of a pipeline to operate on known time steps, whilst the other will adapt itself to the nearest or next available step (either above or below according to user selection) on the other branch.

5.4 Plotting over Time

A basic but important query of data is plotting values over time. That is, given a point, cell, or location in space, extract the value of one or more fields over time. Our temporal pipeline extensions allow us to embed this functionality within a filter.

Because it provides a single result for data that ranges over time, the EXTRACTOVERTIME filter reports no time support to downstream components even though it expects time support from upstream components. Thus, it removes any TIME_RANGES or TIME_STEPS information from its output and ignores any values in UPDATE_TIME_STEPS attached to the output.

A naïve implementation of EXTRACTOVERTIME might simply request all time steps at once and then extract the appropriate data. However, this could easily overrun memory if either the mesh or the time

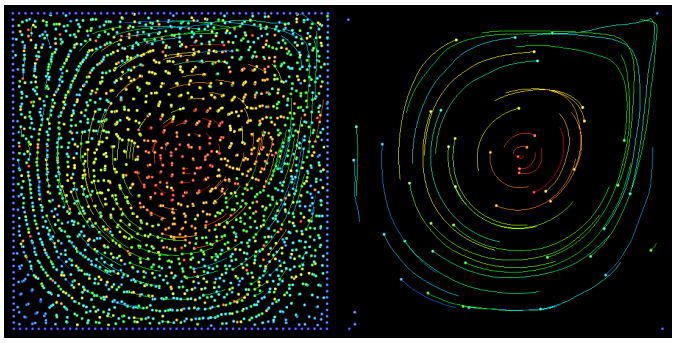

Fig. 6. Motion trails: Many particles with short trails added to every N^{th}(left), a smaller sub set of particles with much longer trails (right)

range is large. Instead, the filter iteratively processes one value at a time.

EXTRACTOVERTIME iteratively processes time steps by working in conjunction with the executive. Initially, EXTRACTOVERTIME requests the first time step. As it exits the REQUEST_DATA portion of the update request, it sends a CONTINUE_EXECUTING information key to its executive. This signals the executive to repeat the update for this filter and those upstream in the pipeline. On the second iteration, EXTRACTOVERTIME requests the next time step and processes it accordingly. The iteration continues until the last time step is reached, at which time EXTRACTOVERTIME clears the CONTINUE_EXECUTING key from its executive, and processing continues downstream.

Our current implementation of EXTRACTOVERTIME has one major drawback. In general, EXTRACTOVERTIME requests a large portion of data at each time step but passes only a small portion of that data downstream. If the input to EXTRACTOVERTIME happens to be one of the structured data types and the extraction is of a single point or cell, then EXTRACTOVERTIME mitigates this problem by requesting a very small extent of data from the upstream components. In the future, we will be looking at ways to cull upstream data through extents for other types of data. We plan to use design patterns already in the literature [3, 7] – which propagate metadata about a request upstream and use it to cull data further up the pipeline – but also expect to define new techniques specific to requesting a small set of data over a large range of time.

5.5 Motion Trails

Motion trails could be implemented by fetching N time steps of data and building a list of visited positions for each point/particle, and then repeating this process as time is incremented. This would be unnecessarily expensive since in the majority of cases the underlying data is going to be animated anyway and the trails are to be added as a visual aid. If particles are animated on one branch of a pipeline, time information will be flowing along it and all we need do is add a second data path where the the DATA_TIME_STEPS information key (generated in response to the UPDATE_TIME_STEPS request from downstream) is monitored. We then accumulate newer positions and allow older ones to be dropped according to how long we wish our trails to be. Being aware of time step values, we can tolerate instances where the user temporarily halts animation or scrolls back and forth through time erratically. In Figure 6 we show examples of the output of our filter with long or short trails enabled for subsets of the data.

5.6 Particle Tracing

Tracking particles through unsteady flows is computationally expensive and practically difficult when the number of cells becomes large. Parallelization is desirable to solve both problems. Currently there are no off the shelf solutions to parallel particle tracing that can be plugged into a general purpose visualization tool and used on arbitrarily large vector field data. Our implementation of the particle tracer is similar

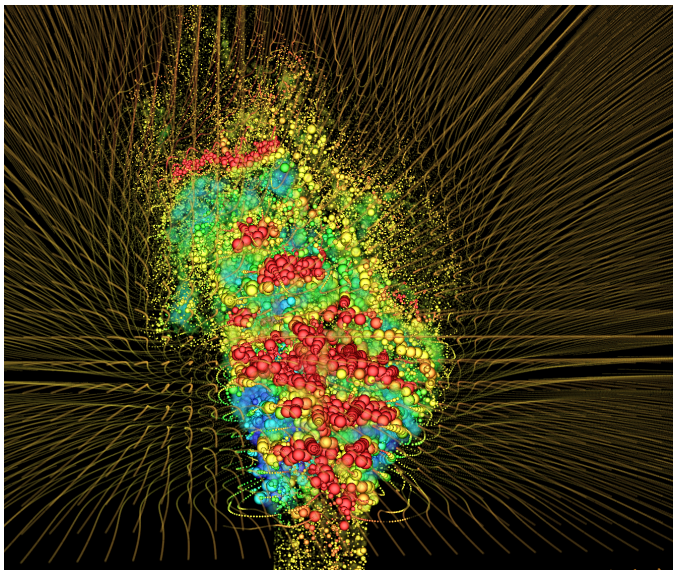

Fig. 7. Particles flowing through a turbulent boundary layer. Radius and opacity are proportional to vortex identification measure λ_2. Color represents pressure.

to that described in [8] with data broken into pieces and distributed on a generic cluster by the existing parallel framework built into the pipeline. The central path of Figure 2 shows the pipeline used for the generation of data (with the addition of a writer which saves particle data to disk for later generation of animations). The Particle tracer requests 2 time steps per iteration, fed to it by the TEMPORALDATASETCACHE, which is set to hold just 2 values. The particle tracer accepts a primary input containing the vector field dataset and secondary inputs which represent the seed locations. Since the seed points are also connected to their own pipelines, they may be animated using key frames linked to the primary time loop, or generated from analysis of the input data to find optimum positions for seed point placement. Particle injection may happen on a single processor (the default), however if the seed points are generated from some operation on the primary input (such as a slice or a region selection), then the seeds will be generated on the processor already holding the piece of data of interest. Thus seeds can be injected on the processor already owning the block in question. By default, particles are tracked using Runge-Kutta 2nd order integration of the vector field; other integrators (such as Runge-Kutta 4th order) may be plugged in. Adaptive time integrators are not supported in the initial implementation, but could be added without significant modification to the code. When leaving a processor, particles are sent via MPI to the other processors where they are picked up and continued. Potentially more efficient parallelization by could be achieved by duplicating data on all processors and integrating particles in parallel without the need for MPI communication, but we wish the algorithm to run on arbitrarily large datasets which may be not fit into memory in a single piece, particularly problematic as 2 time steps are required in memory for processing at a time.

No special preprocessing or optimization is made to the initial data other than to enable or disable the caching of cell index information depending upon whether the mesh is static or dynamic over time. Dynamic meshes are handled by the particle tracer but there is a performance penalty associated with the need to re-evaluate parametric cell coordinates for each particle at each time step.

Figure 7 shows an image generated from particle data produced by our TEMPORALSTREAMTRACER module. The example shows a snapshot from a sequence where 400,000 particles were injected over 1000 time steps using 32 CPUs. The dataset consists of 10 million hexahedral cells stored in multi-block form with 58 blocks - consuming a total memory (mesh plus transient data) of 700MB per step (0.7TB in total). The performance of our implementation is highly dependent on

the seeding strategy of particles. In the example shown in Figure 7 particles were injected into the inflow of the boundary layer, causing the processors holding blocks of data around the inflow to become fully loaded whilst others in the unoccupied voids had no work to do. The processing time for 400,000 particles using this seeding strategy was around 15 hours. Using a randomized seeding pattern for particle injection to evenly balance the processor load improved the processing time to just 3 hours for 400,000 particles on 32 CPUs and 6 hours for 16 CPUs.

The implementation is however extremely flexible, handling any 3D cell type available in the library, catering for dynamic meshes, and optionally interpolating any scalars present in the data for each particle generated. Particles passing from one domain to another where the meshes do not align perfectly (rotating turbine domains for example) cause problems as particles can be lost in the gaps between meshes. Future improvements to the implementation should improve both the tolerance for dynamic meshes and the computation time required.

The limit to the size of data which can be in principal be managed is determined by resources available. The current implementation scales linearly with increasing CPU count, providing the number of particles is large and the computation time significantly exceeds the IO time. Since every process must send particles that have left the domain to neighboring processes after each time step the overall speed is determined by the slowest process. Best results are achieved when large numbers of particles are seeded randomly or evenly throughout the data as opposed to when they are clumped in a single domain.

6 Conclusion

This paper has presented the design and implementation of a parallel visualization pipeline architecture for temporal data. The design requirements were obtained from a careful consideration of the problems that existing temporal support presented and the types of algorithms that need to be implemented. After several alternative architectures were developed, one was selected and implemented. Whereas previous architectures limit filters to dealing with datasets for a single time step at once, this new architecture allows an arbitrary number of time steps to be requested – including streaming pieces of datasets from multiple time steps at once so that the pipeline can continue to accommodate data parallelism. The new architecture does not require changes to filters that are unaware of the new temporal features. Furthermore, filters such as the TemporalDataSetCache were implemented to aid in its adoption by providing applications with a way to combine existing non-temporal pipelines with temporal processing. We demonstrate our design by implementing a number of different time dependent visualizations, including the processing of large time-dependent data. By extending a widely used visualization framework with these new capabilities we have placed a powerful tool in the hands of researchers and scientists, enabling the straightforward implementation of new time-dependent algorithms on large datasets that was not previously possible. Although other visualization systems provide some temporal processing capabilities, we are not aware of any that include user-configurable and extensible pipelines that enable the range of temporal processing functionality presented here.

Acknowledgements

Thanks to Jörg Ziefle, ETH Zürich; Theophane Foggia, Swiss National Supercomputing Centre; David Graham, Plymouth University, UK, for their assistance and test data. K. Moreland and D. Thompson were supported by the United States Department of Energy, Office of Defense Programs. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under contract DE-AC04-94-AL85000.

References

[1] G. Abram and L. Treinish. An extended data-flow architecture for data analysis and visualization. *SIGGRAPH Comput. Graph.*, 29(2):17–21, 1995.

[2] J. Ahrens, K. Brislawn, K. Martin, B. Geveci, C. C. Law, and M. Papka. Large scale data visualization using parallel data streaming. *IEEE Computer Graphics and Applications*, 21(4):34–41, 2001.

[3] J. P. Ahrens, N. Desai, P. S. McCormick, K. Martin, and J. Woodring. A modular, extensible visualization system architecture for culled, prioritized data streaming. In R. F. Erbacher, J. C. Roberts, M. T. Grohn, and K. Borner, editors, *Visualization and Data Analysis 2007*, volume 6495, pages 64950I–1 – 64950I–12. SPIE, 2007.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley, October 1996. ISBN 0-471-95869-7.

[5] G. Cameron. Modular visualization environments: Past, present, and future. *Computer Graphics*, 29(2):3–4, 1994.

[6] E. H. Chi. A taxonomy of visualization techniques using the data state reference model. In *IEEE Symposium on Information Visualization, 2000*, pages 69–75, October 2000.

[7] H. Childs, E. S. Brugger, K. S. Bonnell, J. S. Meredith, M. Miller, B. J. Whitlock, and N. Max. A contract-based system for large data visualization. In *Proceedings of IEEE Visualization 2005*, pages 190–198, 2005.

[8] D. Ellsworth, B. Green, and P. Moran. Interactive terascale particle visualization. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 353–360. IEEE Computer Society, 2004.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[10] A. Gerndt, B. Hentschel, M. Wolter, T. Kuhlen, and C. Bischof. Viracocha: An efficient parallelization framework for large-scale CFD post-processing in virtual environments. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, 2004.

[11] R. B. Haber and D. A. McNabb. Visualization idioms: A conceptual model for scientific visualization systems. In *Visualization in Scientific Computing*, pages 74–93, 1990.

[12] A. Henderson. *ParaView Guide, A Parallel Visualization Application*. Kitware Inc., 2005.

[13] D. V. Hutton. *Applied Mechanical Vibrations*. McGraw-Hill, 1981.

[14] R. Kaeler, S. Prohaska, A. Hutanu, and H. Hege. Visualization of time-dependent remote adaptive mesh refinement data. In *Proceedings IEEE Visualization conference 2005. VIS2005*, pages 175–182. IEEE Computer Society, 2005.

[15] B. King. VTK 5 pipeline architecture. Kitware Source Quarterly Newsletter 1, Kitware, Inc., July 2006.

[16] Kitware. *The VTK User's Guide*. Kitware, Inc., 2006.

[17] M. Krogh and C. Hansen. Visualization on Massively Parallel Computers using CM/AVS. In *AVS Users Conference*, Orlando, Fl., May 1993.

[18] B. Lucas, G. D. Abram, N. S. Collins, D. A. Epstein, D. L. Gresh, and K. P. McAuliffe. An architecture for a scientific visualization system. In *VIS '92: Proceedings of the 3rd conference on Visualization '92*, pages 107–114, Los Alamitos, CA, USA, 1992. IEEE Computer Society.

[19] K. Ma. Visualizing time varying volume data. *Computing in Science and Engineering*, pages 34–42, March 2003.

[20] M. Miller, C. D. Hansen, and C. R. Johnson. The SCIRun problem solving environment: Implementation within a distributed environment. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[21] M. Schirski, A. Gerndt, T. van Reimersdahl, T. Kuhlen, P. Adomeit, O. Lang, S. Pischinger, and C. Bischof. Vista flowlib - framework for interactive visualization and exploration of unsteady flows in virtual environments. In *EGVE '03: Proceedings of the workshop on virtual environments 2003*, pages 77–85, New York, NY, USA, 2003. ACM Press.

[22] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit, An Object Oriented Approach to 3D Graphics*. Kitware Inc., fourth edition, 2004.

[23] B. Szabo and I. Babuska. *Finite Element Analysis*. John Wiley & Sons, 1991.

[24] H. Theisel, T. Weinkauf, H.-C. Hege, and H.-P. Seidel. Topological methods for 2d time-dependent vector fields based on stream lines and path lines. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):383–394, 2005.

[25] C. Upson, J. Thomas Faulhaber, D. Kamins, D. H. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: A computational environment for scientific visualization. *IEEE Comput. Graph. Appl.*, 9(4):30–42, 1989.