Overcoming Barriers to Restructuring in a Modular Visualisation Environment

Olav Beckmann[†], Anthony J. Field[†], Gerard Gorman[††], Andrew Huff[†],
Marc Hull[†], and Paul H. J. Kelly[†]
[†] Department of Computing,
[††] Department of Earth Sciences Engineering,
Imperial College London, SW7 2AZ, UK
p.kelly@imperial.ac.uk

ABSTRACT

This paper explores the potential for automatic cross-component optimisation in the Python / VTK-based MayaVi modular visualisation environment. The idea is to delay execution of the VTK components called from the MayaVi tool, which requires no signi cant structural change to the MayaVi code base, but which opens up the possibility for dynamic performance optimisations such as tiling, fusion, memoisation and shared-memory parallelisation. The paper concludes with experimental results on an unstructured mesh hierarchy model from an adaptive three-dimensional gravity current simulation.

1. INTRODUCTION

Modular visualisation environments (MVEs) present end-users with a graphical interface for composing data analysis and rendering components. Such a dynamic-assembly architecture forms the core of many software frameworks, and is essential for their exibility. Unfortunately it also presents a barrier to conventional compile-time optimisation. Software environments for visualising large unstructured datasets, such as the Python / VTK-based open-source MayaVi tool [10–12], provide a high-level graphical language that allows computational scientists to program a visualisation pipeline: before the rendering step various feature extraction or data ltering computations may be executed, such as iso-surface calculation, interpolation of a regular mesh or ow-lines integration.

In this paper, we describe a technique that allows us to apply restructuring optimisations, speci cally tiling, to visualisation pipelines speci ed from the MayaVi tool. Our approach requires minimal changes to the underlying MayaVi code. We achieve this by intercepting and delaying calls made from the Python frontend at the Python-VTK binding interface. This allows us to build up a "visualisation plan" of the underlying VTK routines applied to the dataset without actually executing those routines. We partition the dataset o -line using a tool such as METIS, and then apply the captured visualisation plan partition-by-partition.

The work described in this paper was motivated by the visualisation requirements arising from the simulation of ocean currents using adaptive, unstructured (*i.e.* tetrahedral) meshes. Even small runs generate multi-gigabyte datasets. Each stage of such a visualisation pipeline can be a computationally very expensive operation which is typically applied to the entire dataset. This can lead to very signi - cant delays before any visual feedback is o ered to an application scientist who is trying to compose and parameterise a visualisation pipeline.

1.1 Contributions of this Paper

- We present our experience of performing cross-component optimisation in a challenging, dynamic, multi-language context (Sections 3–4).
- We show how delaying execution at the Python / C++ interface allows a description of the required computation to be extracted at runtime while avoiding many complex dependence issues (Section 4).
- We present results from performance experiments illustrating some of the fruits of the restructuring. In particular we show how demand-driven execution of the MayaVi pipeline allows interactive exploration of hierarchical, partitioned, unstructured meshes of unlimited size (Section 5).

The framework presented here provides the basis for a programme of research aimed at extending aggressive optimisation techniques across component-based scienti c software environments and deploying the results in large-scale industrial systems.

1.2 Structure of this Paper

In Section 2, we place this work in the context of ongoing research into optimisation techniques for dynamically composed assemblies of high-level software components. Much of the paper is devoted to an analysis of the software architecture of the MayaVi MVE in order to pinpoint barriers to optimisation (Section 3) and how we have overcome these

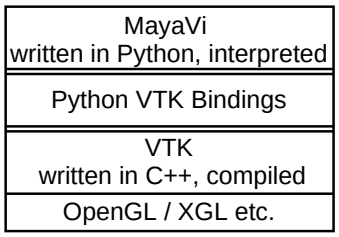


Figure 2: Software architecture of the MayaVi visualisation environment in terms of languages and libraries.

(Section 4). We present preliminary experimental results in Section 5. We discuss future work in Section 6 and conclude in Section 7 with a discussion of the potential for the work, and the challenges that remain.

2. BACKGROUND

Modular visualisation environments present end-users with a GUI representing the analysis and rendering pipeline [2]. MayaVi is one of many MVEs implementing this general model. Other examples from image processing include Adobe Photoshop, Adobe Premiere or the Gimp, via its scripting mechanism. The MVE architecture o ers the potential to integrate visualisation with simulation and computational steering [9, 13] and this is nding broader application in the Grid [4, 6]. To make MVEs work interactively on very large datasets, execution needs to be demand-driven, starting from a volume-of-interest (VOI) control, which speci es the 3-dimensional region where high resolution is required [3].

However, this paper is not about visualisation itself, but rather about the performance optimisation challenge raised by MVE-like software structures: how can we extend optimising and restructuring compiler technologies to operate on dynamically-composed assemblies of high-level components? This work is part of a wider programme of research into cross-component optimisation issues: our DESO (delayed evaluation, self-optimising) parallel linear algebra library [1, 8] uses carefully constructed metadata to perform cross-component parallel data placement optimisation at runtime, and the ongoing DÉSORMI project has resulted in a generalised framework for deploying runtime optimisation and instrumentation in Java programs [14]. Optimising component-based applications is also one of the research challenges addressed by the Grid e ort [5].

3. MAYAVI'S MVE ARCHITECTURE

MayaVi presents application scientists with a high-level graphical environment for constructing visualisations of scienti c data sets. In Figure 2, we illustrate the software architecture of MayaVi in terms of programming languages and libraries used: MayaVi is written in the interpreted language Python. The core of VTK (visualisation tool kit) is written in C++ and is compiled; however, VTK has a complete set of bindings for Python, Tcl/Tk and Java. MayaVi uses the VTK Python bindings to construct a VTK "visualisation pipeline". VTK in turn uses di erent graphics libraries such as OpenGL for 3D rendering.

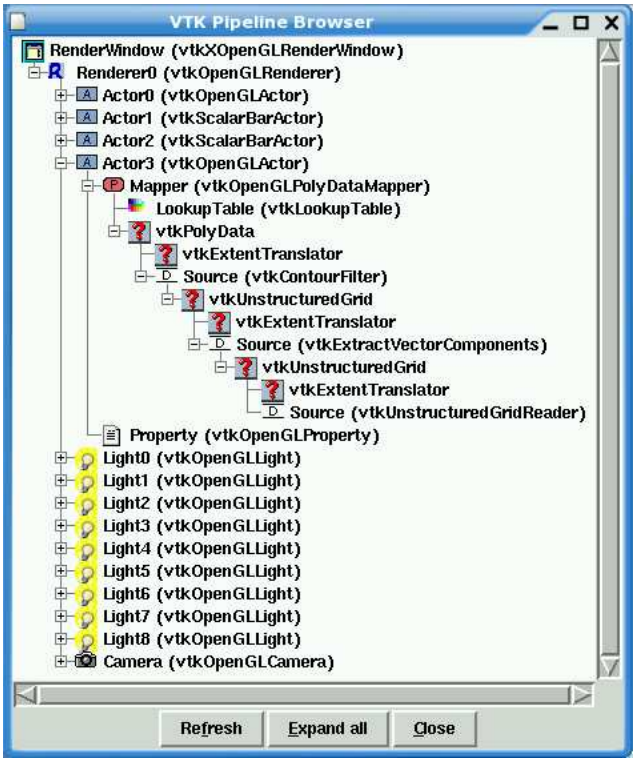


Figure 3: VTK visualisation pipeline, as represented by the MayaVi pipeline browser for the visualisation in Figure 1.

3.1 Object-Oriented Visualisation in VTK

The VTK design distinguishes between the graphics model, an object-oriented representation of 3D computer graphics and the visualisation model, which is essentially a model of data- ow.

3.1.1 The VTK Graphics Model

The VTK graphics model is described in detail in [11]. The key concepts that are relevant to this paper are the following. A RenderWindow represents a window on the display. A Renderer is an object that is responsible for rendering a region of such a window. Lights and a Camera characterise the illumination of the scene and the viewpoint. Actors are objects that are rendered within the scene. In Figure 1, we show an isosurface visualisation of a turbulent ow. Such an isosurface corresponds to one Actor in the VTK graphics model. Actors consist of Mappers, representing the geometric structure of the Actor (in the case of the isosurface in Figure 1, this is a set of polygons), Properties, representing the object colour, texture etc., and Transforms, which are 4×4 matrices that describe the usual object transformations on homogeneous coordinates used in 3D computer graphics.

3.1.2 The VTK Visualisation Pipeline

The VTK visualisation pipeline is an object-oriented representation of a directed data- ow graph, consisting of data and processes, which are operations performed on the data. Process objects can be sources, representing inputs, lters, which can be many-to-many operations on data, and mappers, representing outputs from the data- ow graph that

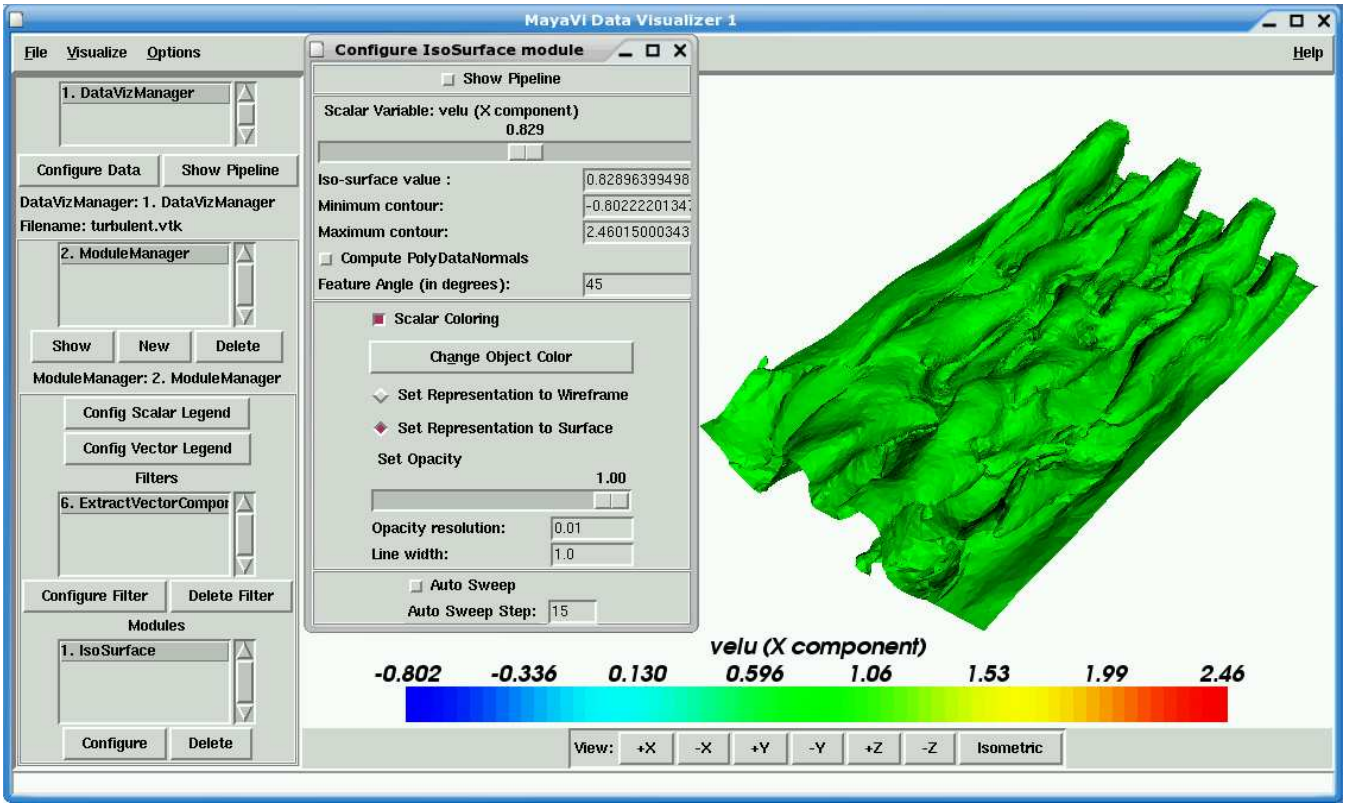


Figure 1: MayaVi screenshot, showing the main MayaVi GUI, the GUI for con guring a speci c visualisation module (IsoSurface) and the render window, showing an isosurface of the x component of the velocity vectors in a turbulent flow simulation.

are then used by the graphics model for rendering. The VTK visualisation pipeline can represent complex data- ow graphs, including graphs with cycles. Time-stamps are used to make sure that such cycles execute only once per invocation of the pipeline. The VTK design provides for data- ow graphs to be executed in either a *demand-driven* or a *data-driven* manner.

In Figure 3, we show the VTK visualisation pipeline for the isosurface visualisation from Figure 1, as represented by MayaVi's pipeline browser tool. Note in particular the source (vtkUnstructuredGridReader), a lter that extracts one of the components of the velocity vector (vtkExtractVectorComponents) and the output of the pipeline that is passed to the mapper (vtkPolyData, representing a polygon collection). There are several instances of the vtkExtentTranslator process: this can be used to calculate a structured extent (*i.e.* a bounding box) for an unstructured dataset. Extents are calculated in order to facilitate demand-driven generation of only those data regions that are required. In the case of unstructured data, this functionality is very experimental in VTK at present [7].

3.2 MayaVi Construction of VTK Pipelines

In this section, we brie y describe how MayaVi interacts with the VTK Python interface to construct VTK visualisation pipelines.

In Figure 4, we use a UML sequence diagram to illustrate

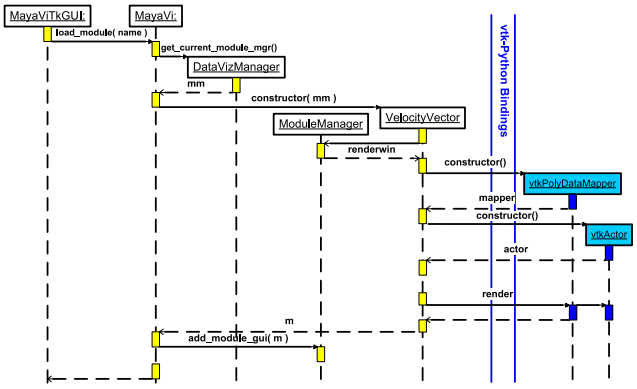


Figure 4: Original MayaVi: Calling structure when applying a visualisation module (VelocityVector) to a dataset that has been loaded into the MayaVi application.

the calling sequence that the original version of MayaVi performs to construct a visualisation of a dataset which has been loaded into the application. The reason for illustrating this sequence of operations by showing a UML diagram rather than Python source code is the large number of different classes and methods which such a calling sequence traverses. Figure 4 gives a high-level overview; a larger view of calls across the Python VTK binding interface is shown in Figure 5 and discussed in Section 3.2.1.

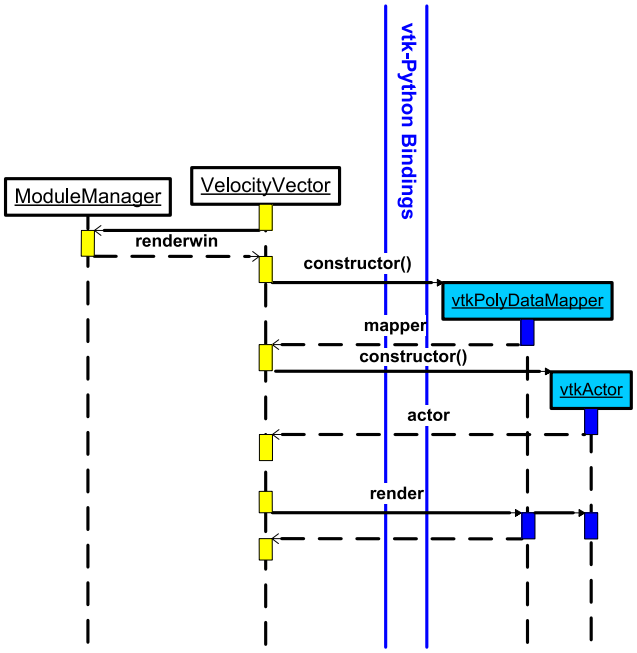


Figure 5: Original MayaVi: Construction and execution (via `render` call) of a VTK visualisation pipeline from the MayaVi VelocityVector module.

When the user selects a particular visualisation module from the GUI, the MayaViTkGUI class calls the `load_module` method from the main MayaVi class. This method in turn obtains a reference for the current ModuleManager[1] from the DataVizManager[2] and then calls the constructor for the "visualisation class"[3] which the user has selected, passing the ModuleManager reference as a parameter. The visualisation class constructs a VTK visualisation pipeline via calls through the VTK Python interface and obtains references, such as a handle for the data being visualised (the source of the VTK visualisation pipeline), as well as the RenderWindow from the ModuleManager. The visualisation is actually computed when the visualisation class calls the `Render()` method of RenderWindow, forcing evaluation of the demand-driven VTK pipeline.

3.2.1 *Cross-Component Optimisation in MVEs*

From the point-of-view of software architecture, MayaVi's modular structure, which we have illustrated in the above example, has many advantages, in particular ease of adding new visualisation modules. However, from the point-of-view of cross-component optimisation, this is a uniquely challenging software environment.

[1]A ModuleManager is a Python class which MayaVi uses to manage one or many visualisations which are applied to a dataset.

[2]For each dataset that is loaded into MayaVi, a DataVizManager class is created which manages all visualisations of that dataset.

[3]It may seem odd that a visualisation — e ectively a computation — is a class, but this is in line with the VTK pipeline model where both data and processes are represented as objects.

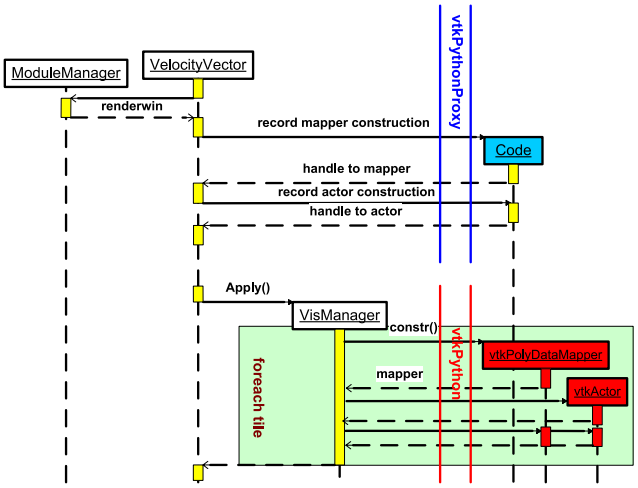


Figure 6: Modi ed MayaVi Visualisation: Capturing the code for constructing a VTK visualisation pipeline by intercepting calls through the VTK Python interface, followed by execution of the the captured code from the VizManager class.

- The user's actions via the GUI result in dynamic construction of a data- ow pipeline, which is then executed. Furthermore, this construction happens in stages, and the pipeline can be dynamically modi ed. In order to facilitate this, "manager" objects hold references for parts of the pipeline as it is being constructed. These include data sources that result from opening a le or from adding a lter to the pipeline. Static optimisation is clearly not possible since it would require prior knowledge of the user's actions via the GUI.
- However, the visualisation pipeline which is eventually executed is a computational structure which has been extensively studied by high-performance compiler architects: it comprises a graph were the edges denote data ow and the nodes denote (possibly very expensive) computations.

In the following section, we outline our approach to enabling restructuring optimisations on the VTK visualisation pipelines constructed by MayaVi.

4. RESTRUCTURING OPTIMISATION

Our initial objective was to improve the response time of MayaVi visualisations when applied to very large datasets. The strategy for achieving this was to partition the dataset using a tool such as METIS and then apply the visualisation partition-by-partition, resulting in much faster response times for smaller regions of the mesh. The problem with this strategy is that it does not seem possible to implement such optimisations without interfering with the modular architecture of MayaVi: Inspecting the sequence of calls in Figure 5 shows that the following changes would have to be implemented:

- Every visualisation module (there are currently more than 20) and every data lter (currently more than

15) would have to be changed to loop over partition collections rather than just invoking the underlying VTK pipeline.

- Every call to `Render()` would have to be replaced with a loop iterating over partitions.

Not only would such re-writing require a substantial re-coding e ort, but it would also seriously deteriorate the clarity of MayaVi's Python code. Implementors of data visualisations should not be forced to implement code optimisations such as tiling by hand.

4.1 Capturing Visualisation Plans

The key observation is that when a visualisation is rendered, the data ow happens entirely on the C++ side of VTK. Furthermore, this implies that all nodes in the data ow graph have to be inserted via calls through the VTK Python bindings. Therefore, we are able to capture an accurate representation of the pipeline which is being constructed if we intercept any calls made though this interface.

Our approach is to intercept and delay all calls though the VTK Python interface. This allows us to build a visualisation plan — a representation of the calls that have been performed in order to construct the VTK pipeline. When visualisation is "forced", *i.e.* when an image has to be rendered, we can apply the visualisation plan partition-by-partition.

We illustrate this in Figure 6. Rather than making calls through the vtkpython module, we now make calls to the vtkPythonProxy module, which constructs a visualisation plan instead of executing VTK methods directly. Also, instead of directly calling `Render()` on the RenderWindow, the visualisation module now calls a method called `Apply()` on a new VisualisationManager class. This class then loops over the partitions of the dataset, constructing a visualisation pipeline for each partition and rendering these pipelines incrementally.

4.2 Remaining Changes to MayaVi

One advantage of our approach, as illustrated in Figure 6 and described in the previous section, is that the remaining changes which are required to the MayaVi source code can now be automated.

4.2.1 Building the vtkPythonProxy Module

The vtkPythonProxy module has to implement all method calls that are made through the VTK Python interface by MayaVi modules. Rather than executing on the C++ side of MayaVi, calls to the vtkPythonProxy module construct a data structure (the visualisation plan) representing the operations which have been performed. We have implemented a Python script which parses a MayaVi visualisation module and automatically builds the required vtkPythonProxy module capable of intercepting all calls to VTK methods made form that MayaVi module.

4.2.2 Adapting MayaVi Modules

Calls through the VTK Python interface normally have the following structure:

```
res = vtkpython.Constructor()    .
```

We have to replace all such calls with calls that instead go through the special vtkPythonProxy module:

```
res = vtkPythonProxy.Constructor(self.code)
```

where `self.code` is a reference for the visualisation plan being constructed. In addition, methods that de ne the source of a visualisation pipeline (`SetInput()`) and force evaluation (`Render()`) are also substituted, the latter with a call to the `Apply()` method from a new VisualisationManager class.

The full story is somewhat more involved: methods called on objects that are returned from calls through the VTK Python interface also have to be intercepted. We implement this by returning special proxy objects from the vtkPythonProxy class. Furthermore, some method calls in MayaVi are made using the Python `apply()` construct, which means that without re-writing such calls to a more conventional syntax, these would be missed by our script for automatically constructing the vtkPythonProxy class.

5. EXPERIMENTAL EVALUATION

As stated in Section 4, the original motivation for this work was to make the visualisation of very large unstructured datasets more interactive. Our preliminary experimental evaluation has therefore focused on response time.

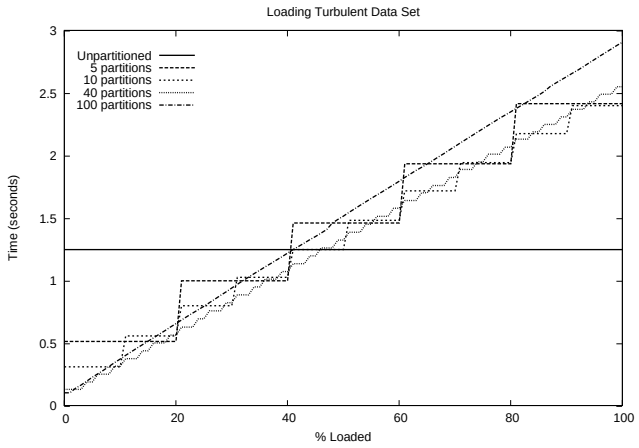
5.0.3 Preliminary Results

In Figure 7, we show the response time achieved for loading an unstructured mesh into MayaVi for two di erent datasets. We show the time taken to load a given percentage of the mesh when the mesh is partitioned into 5, 10, 40 and 100 partitions, as well as the time to load the unpartitioned mesh. The experimental platform was a Pentium 4 2.6 GHz with 1GB RAM, reading the datasets from an NFS-mounted server over fast ethernet. When the volume of interest (percentage) loaded is small, the partitioned version is much faster; however, when the volume of interest exceeds about 30–50% of the entire mesh, overheads dominate. Some of these overheads are inherent in our approach because some nodes belonging to tetrahedral elements on the surface of partitions have to be replicated.

In Figure 8, we show the response time when performing an isosurface visualisation of two di erent datasets. This again shows that the response time when visualising a small percentage of the partitioned dataset is good; however, when a larger percentage is visualised overheads dominate.

5.0.4 Future Experimental Work

The above examples have been solely concerned with response time. We believe, however, that our methodology will also facilitate an overall speedup in generating visualisations for whole datasets: As we showed in Section 3, MayaVi constructs a VTK visualisation pipeline, which in e ect is a data ow graph consisting of data sources and multiple operations applied to the dataset. Given that the datasets can easily be tens of gigabytes in size, we believe that partitioning and "streaming" data through the visualisation pipeline



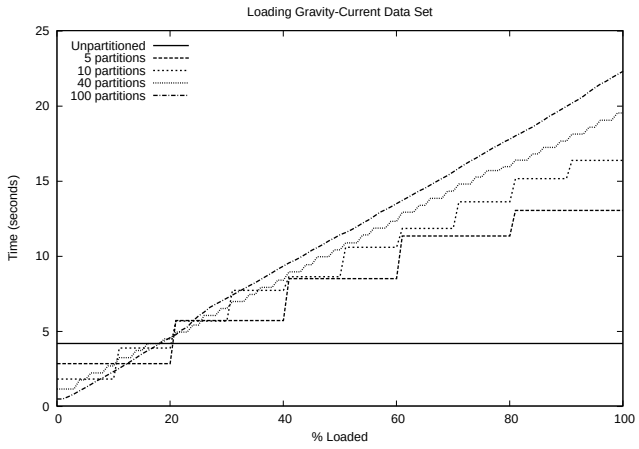
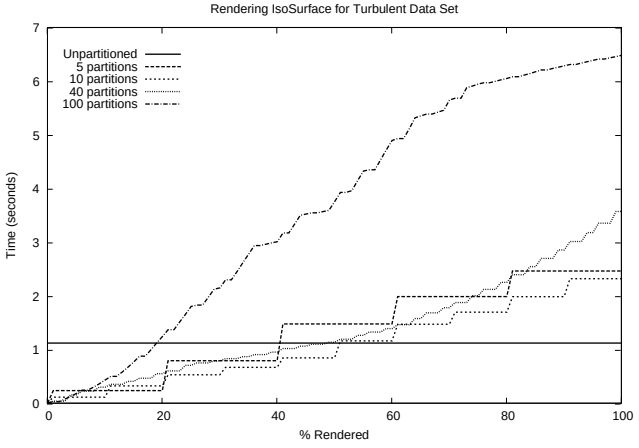


Figure 7: Time to load a speci ed volume of interest (VOI) of two di erent unstructured meshes. Top: Turbulent flow (15 MB unpartitioned), below: Gravity current (54 MB unpartitioned). The graph shows that response time is good when the VOI loaded is a small percentage of the whole mesh, but when the VOI exceeds 20-50% of the mesh, overheads dominate.



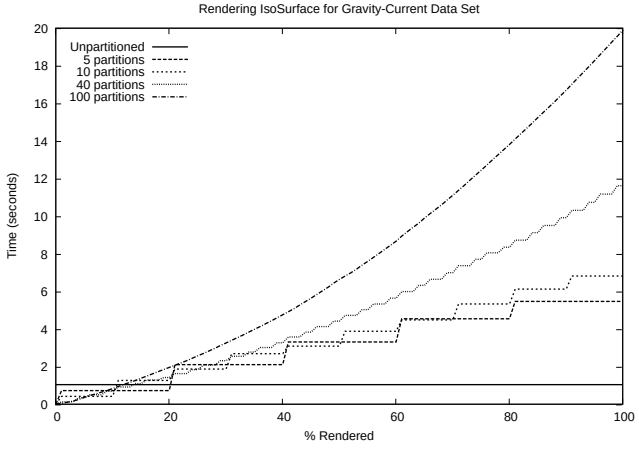


Figure 8: Time to load a render an isosurface visualisation on a speci ed volume of interest (VOI) of two di erent unstructured meshes. Datasets are as in Figure 7. The graph again shows that response time is good when the VOI visualised is a small percentage of the whole mesh, but when the VOI exceeds 20-45% of the mesh, overheads dominate.

one partition at a time will achieve signi cant end-to-end speedups. We plan to evaluate this assumption using larger datasets.

6. FUTURE WORK

We are currently exploring how we can build on this basic infrastructure.

- *Shared-memory Parallelisation.* The tiling transformation which we have described in this paper does not yet make use of the task-parallelism which is exposed. We are planning to investigate using Python threads to execute the visualisation plans which we capture in a task-parallel manner.
- *Using VTK Streaming Constructs.* As stated in Section 3.1, VTK itself provides various constructs for data streaming and parallel execution using MPI that could be exploited from within our framework; we are planning to investigate this.
- *Interaction with the Underlying Simulation.* We are interested in investigating the possibility of pushing the scope for cross-component restructuring optimisations further back into the simulation software that generates the datasets which are visualised by MVEs such as MayaVi. In particular, we are interested in extending demand-driven data generation into the simulation model: if a higher level of detail is required for a small VOI of the dataset, how much of the simulation has to be re-run?

We see this work as part of a wider programme of research into optimising component-based scienti c software frameworks.

7. CONCLUSION

We have presented an overview of a project which is aimed at applying traditional restructuring compiler optimisations in the highly dynamic context of modular visualisation environments. The challenge is that a computationally expensive pipeline of operations is constructed by the user via interactions with the GUI and then executed. Our approach is based on intercepting the construction of the visualisation pipeline, assembling a visualisation plan, on which we can then perform optimisations such as tiling before it is executed. The MayaVi MVE enabled us to reliably capture the construction of the visualisation pipeline by intercepting all calls that pass through the Python/C++ VTK interface.

7.0.5 Acknowledgements

This project was funded by two grants from the United Kingdom Engineering and Physical Sciences Research Council which focus on cross-component optimisation: OSCAR (GR/R21486/01) and Désormi (GR/R15566/01).

8. REFERENCES

[1] O. Beckmann and P. H. J. Kelly. E cient interprocedural data placement optimisation in a parallel library. In *LCR98: Languages, Compilers and Run-time Systems for Scalable Computers*, number 1511 in LNCS, pages 123–138. Springer-Verlag, May 1998.

[2] G. Cameron. Modular visualization environments: Past, present, and future. *Computer Graphics*, 29(2):3–4, May 1995.

[3] P. Cignoni, C. Montani, and R. S. pig no. MagicSphere: An insight tool for 3D data visualization. *Computer Graphics Forum*, 13(3):C/317–C/328, 1994.

[4] I. Foster, J. Vöckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In 14^{th} *International Conference on Scienti c and Statistical Database Management (SSDBM'02)*, July 2002.

[5] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. Optimisation of component-based applications within a grid environment. In *Supercomputing 2001*, 2001.

[6] C. R. Johnson, S. G. Parker, and D. Weinstein. Large-Scale Computational Science Applications Using the SCIRun Problem Solving Environment. In *ISC 2000: International Supercomputer Conference*, Mannheim, Germany, 2000.

[7] Kitware, Inc. *The VTK User's Guide: VTK 4.2*, 2003.

[8] P. Liniker, O. Beckmann, and P. H. J. Kelly. Delayed evaluation self-optimising software components as a programming model. In *Euro-Par 2002: Proceedings of the* 8^{th} *International Euro-Par Conference*, number 2400 in LNCS, pages 666–673, Aug. 2002.

[9] S. G. Parker and C. R. Johnson. SCIRun: A scienti c programming environment for computational steering. In *Proceedings of Supercomputing 1995*, 1995.

[10] P. Ramachandran. MayaVi: A free tool for CFD data visualization. In *4th Annual CFD Symposium, Aeronautical Society of India*, Aug. 2001. `mayavi.sourceforge.net/docs.html`.

[11] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach To 3D Graphics*. Kitware, Inc., 3^{rd} edition edition, 2002.

[12] G. van Rossum and J. Fred L. Drake. *An Introduction to Python*. Network Theory Ltd, Sept. 2003.

[13] H. Wright, K. Brodlie, and M. Brown. The data ow visualization pipeline as a problem solving environment. In *Virtual Environments and Scienti c Visualization '96*, pages 267–276. Springer-Verlag, Vienna, Austria, Apr. 1996.

[14] K. C. Yeung and P. H. J. Kelly. Optimising Java RMI programs by communication restructuring. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference 2003, Rio De Janeiro, Brazil, 16–20 June 2003*, LNCS, June 2003.