

Parallel Visualization using the Domain-Specific Interpreter Pattern

Karen Osmond, Olav Beckmann, Anthony J. Field, and Paul H. J. Kelly

Department of Computing, Imperial College London,
180 Queen's Gate, London SW7 2AZ, United Kingdom
`p.kelly@imperial.ac.uk`

Abstract. This paper concerns the software engineering problem of deploying domain-specific optimisation. The work is motivated by experience with parallelization and tiling in MayaVi, a 44,000-line visualisation application written in Python and VTK.

To minimize disruption to MayaVi's open-source codebase, we do this by interposing a proxy between the Python client code, and the VTK library. The proxy delays execution of the library code, and captures a recipe for the computation required. This creates the opportunity for a "domain-specific interpreter" to select an optimised execution plan.

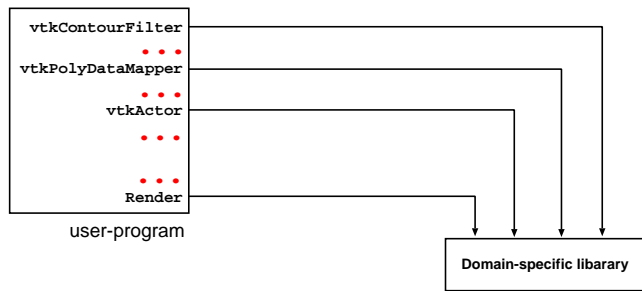
The "domain-specific interpreter" is a design pattern applicable in many contexts, in particular where relatively heavyweight operations are involved to outweigh interpretive overheads. The key issue is whether the captured recipe fully accounts for dependences between client and library code.

We present a generic mechanism for interposing a domain-specific interpreter in a Python application, together with experimental results demonstrating the technique's effectiveness. In an application involving visualization of isosurfaces in an unstructured mesh fluid flow simulation, we show good speedups from improved memory hierarchy performance, and through both SMP and distributed-memory parallelization.

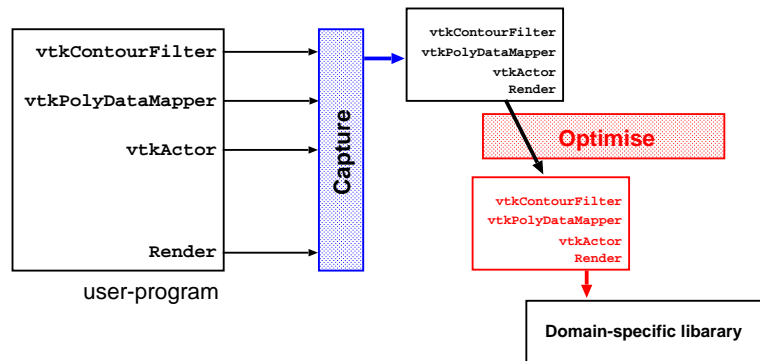
1 Introduction

Key objectives in engineering high-quality software are the need for high performance and protecting existing investment. The work we present in this paper illustrates how the use of domain-specific libraries can make it difficult to bridge these requirements. In response to this we propose domain-specific interpreters as a design pattern for addressing this problem. We show an example of a domain-specific interpreter implemented in Python and demonstrate that this can be used to achieve transparent parallelisation of large-scale visualisation tasks.

Software systems are being built from increasingly large and complex domain-specific libraries. Using such domain-specific libraries (DSLs) often dominates and constrains the way a software system is built just as much as a programming language. To illustrate the increasing size and complexity of DSLs, consider the following three examples:



(a) User program is processed by a standard compiler or interpreter. DSL code is mixed with other code. No domain-specific optimisation is performed.



(b) User program is compiled or interpreted by an unmodified language compiler or interpreter. All calls to the DSL are captured and recorded in an execution plan. Domain-specific optimisations are applied to the execution plan before it is executed.

Fig. 1. Domain-specific library use: (a) typical use and (b) domain-specific interpreter.

- The Legacy BLAS 1, 2 and 3 libraries [1] are very successful libraries, domain-specific to dense linear algebra, with a total number of around 150 functions.
- MPI is a slightly later, but equally successful library which is domain-specific to message-passing communication. MPI-1 [2] included over 100 functions, MPI-2 over 200 functions.
- VTK (Visualisation Toolkit) [3,4] is a large C++ visualisation library. The total number of classes and methods is hard to count; the user’s guide is 325 pages long, and additional documentation is found in a 500-page book.

Using domain-specific libraries and abstractions often introduces domain-specific semantics into a software system in a manner similar to a programming language. The problem is that the base-language compilers or interpreters have no knowledge of these domain-specific semantics, and in particular, of domain-specific optimisations that might be possible. Furthermore, calls to DSLs are typically mixed with other, non-domain-specific code, which might make it hard

for a conventional compiler to infer the exact sequence of calls that will be made to the DSL. This is illustrated in Figure 1(a).

1.1 Domain-Specific Interpreter Design Pattern

We propose a “domain-specific interpreter” as a design pattern for overcoming the problem described above. The idea is illustrated in Figure 1(b): The application program is still processed by a standard compiler or interpreter. However, calls to the DSL are captured by a proxy layer which records an execution plan consisting of the operations to be performed. We then have the opportunity to apply restructuring optimisations to the execution plan before it is executed.

Applicability. The applicability of domain-specific interpreters depends on being able to capture reliably all calls that are made to the DSL, and, on having accurate data-flow information available. The latter means knowing whether the data which is processed by the DSL can also be modified by the intervening non-domain-specific code, and being able to derive an accurate data-flow graph by inspection of the execution plan. As discussed in Section 3, both these requirements are met for visualisation applications built on VTK and Python.

Profitability. The likely benefit of using a domain-specific interpreter depends on whether we have domain-specific semantic information available, and on whether opportunities for cross-component optimisation exist. For our case-study, the optimisations we discuss are parallelisation and a form of tiling. Semantic information is, for the time being, supplied by hand.

1.2 Visualisation of Large Scientific Datasets

Visualising large scientific datasets is a computationally expensive operation, which typically involves processing a “visualisation pipeline” of domain-specific data analysis and rendering components: before the rendering step various feature extraction or data filtering computations may be executed, such as iso-surface calculation, interpolation of a regular mesh or flow-lines integration. Modular visualisation environments (MVEs), such as the Python/VTK-based open-source MayaVi tool [4–6], present end-users with an interface for assembling such components. This effectively defines a high-level graphical programming language for visualisation pipelines. Such a dynamic-assembly architecture forms the core of many software frameworks, and is essential for their flexibility. As discussed in Section 1, it unfortunately also presents a barrier to conventional compile-time optimisation.

We describe the implementation of a domain-specific interpreter that allows us to apply restructuring optimisations, specifically parallelisation, to visualisation pipelines specified from MayaVi. Our approach requires no changes to the MayaVi code. We achieve this by intercepting DSL calls 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 off-line using a tool such as METIS [7], and then apply the captured visualisation plan in parallel on each partition where that is consistent with the semantics of the data analysis components.

The work described in this paper was motivated by the visualisation requirements of ocean current simulations 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 significant delays before any visual feedback is offered to an application scientist who is trying to compose and parameterise a visualisation pipeline.

1.3 Contributions

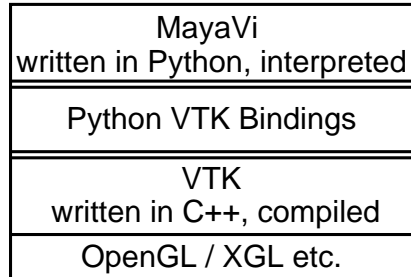
The main contributions of this paper are as follows.

- We present our experience of performing cross-component optimisation in a challenging, dynamic, multi-language context.
- We present a domain-specific interpreter which intercepts DSL calls at the Python/C++ interface, and we show how this allows a data-flow graph for the required computation to be extracted at runtime while avoiding many complex dependence issues (Section 3).
- We discuss how applying visualisation pipelines one partition at a time, even on a uniprocessor, can lead to performance improvements due to better use of the memory hierarchy. We refer to this optimisation as “tiling”¹ (Section 4).
- We present parallelisation strategies for visualisation pipelines captured by the domain-specific interpreter, for both shared- and distributed-memory architectures (Sections 5–6).
- We present results from performance experiments that show encouraging speedups for tiling and both types of parallelisation (Sections 5–6).

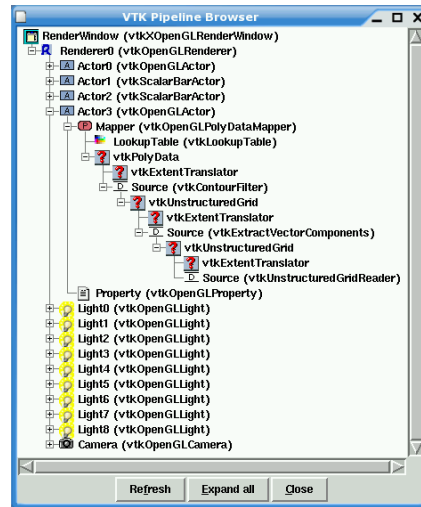
This paper builds on our earlier work [8] where we restructured the MayaVi source code by hand to improve response time. Apart from changing the MayaVi source code, this earlier work also did not achieve parallelisation.

The remainder of this paper is structured as follows. 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. In Section 3, we present an implementation of the domain-specific interpreter pattern in Python for optimising the use of VTK. In Section 4, we present results for the tiling optimisation mentioned above. In Section 5 we discuss parallelisation of visualisation pipelines for shared memory, and in Section 6, we discuss parallelisation for distributed memory platforms. Section 7 concludes and describes our plans for future work.

¹ This is not tiling in the classical sense of non-linear transformations of loops over dense arrays — however, it does represent a re-ordering of the iteration space over a large dataset for the purpose of improving memory hierarchy utilisation.



(a) Software architecture of MayaVi in terms of languages and libraries.



(b) VTK visualisation pipeline in MayaVi's pipeline browser for the visualisation in Figure 3.

Fig. 2. MayaVi software architecture (a) and pipeline browser (b).

2 Background and Related Work

Modular visualisation environments (MVEs) present end-users with a GUI representing an analysis and rendering pipeline [9]. MayaVi is one of many MVEs implementing this general model. Other examples from image processing include Adobe Photoshop or the Gimp, via its scripting mechanism. The MVE architecture offers the potential to integrate visualisation with simulation and computational steering [10,11] and this is finding broader application in the Grid [12,13]. To make MVEs work interactively on very large datasets, execution needs to be demand-driven, starting from a volume-of-interest (VOI) control, which specifies the 3-dimensional region where high resolution is required [14].

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 issue is part of a wider programme of research into cross-component optimisation issues: our DESO (delayed evaluation, self-optimising) parallel linear algebra library [15,16] uses carefully constructed metadata to perform cross-component parallel data placement optimisation at runtime, and the DÉSORMI project has resulted in a generalised framework for deploying runtime optimisation and instrumentation in Java programs [17]. Optimising component-based applications is also one of the research challenges addressed by the Grid effort [18].

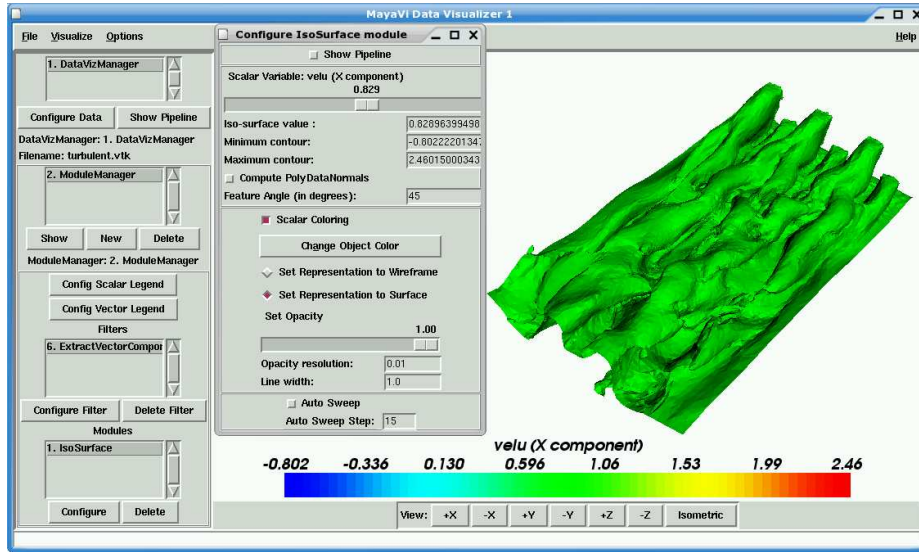


Fig. 3. MayaVi screenshot, showing the main MayaVi GUI, the GUI for configuring a specific visualisation module (IsoSurface) and the render window, showing an isosurface of the x component of the velocity vectors in a turbulent flow simulation.

2.1 MayaVi's Modular Visualisation Environment Architecture

In Figure 2(a), 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 [3, 4] is written in C++ and is compiled; however, VTK has a complete set of bindings for Python, Tcl/Tk and Java. VTK in turn uses different graphics libraries such as OpenGL for 3D rendering.

2.2 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-flow.

The VTK graphics model is described in detail in [4]. 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. *Actors* are objects that are rendered within the scene. In Figure 3, we show an isosurface visualisation of a turbulent flow. Such an isosurface corresponds to one *Actor*. *Actors* consist of *Mappers*, representing geometric structure (in the case of the isosurface in Figure 3, this is a set of polygons), *Properties*, representing colour, texture etc., and *Transforms*, which are 4×4 matrices that describe the usual transformations on homogeneous coordinates used in 3D computer graphics.

The VTK visualisation pipeline is an object-oriented representation of a directed data-flow graph, consisting of data and processes, which are operations performed on the data. Process objects can be *sources*, representing inputs, *filters*, which can be many-to-many operations on data, and *mappers*, representing outputs from the data-flow graph that are then used by the graphics model for rendering. The VTK visualisation pipeline can represent complex data-flow graphs, including graphs with cycles (time-stamps are used to control looping). The VTK design provides for data-flow graphs to be executed in either a *demand-driven* or a *data-driven* manner.

In Figure 2(b), we show the VTK visualisation pipeline for the isosurface visualisation from Figure 3, as represented by MayaVi’s pipeline browser tool. Note in particular the source (`vtkUnstructuredGridReader`), a filter 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.

3 A Domain-Specific Interpreter for VTK in Python

In Section 1, we outlined our proposal for a domain-specific interpreter as a means of overcoming barriers to restructuring optimisation in the use of domain-specific libraries. In this section, we present an implementation of this design pattern in Python, using VTK as a domain-specific library.

The key observation is that when a MayaVi visualisation is rendered, the data flow happens entirely on the C++ side of the Python/VTK interface. This implies that all nodes in the data flow graph have to be created via calls through the VTK Python bindings. Therefore, we are able to capture an accurate representation of the visualisation pipeline, which represents the data-flow graph of the computation to be performed, if we intercept calls made through this interface.

When visualisation is “forced”, *i.e.* when an image has to be rendered, we can perform optimisations on this pipeline before it is executed. The next four sections explain how this is done.

3.1 Building the Proxy Layer

We rename the original `vtkpython.py` file which implements VTK’s Python bindings to `vtkpython_real.py`. We implement a new file `vtkpython.py`, which is shown in Listing 1.1. This file dynamically creates a new, initially empty class definition for every class in the original `vtkpython` interface. The key point is that these empty classes are all derived from a new class `ProxyObject`; thus, the absence of methods in the dynamically created empty classes means that all method calls will be deferred to the superclass.

Listing 1.1. Implementation of a proxy layer for intercepting calls through the VTK Python binding interface

```

1 import os
2 if ("new_vtk" in os.environ): # Control the DS interpreter via the environment
3     import vtkpython_real # Import the original VTK Python bindings
4     from parallel import proxyObject
5     from parallel import setPartitionInfo
6     from parallel import setParameters
7     from parallel import setScalar
8     for className in dir(vtkpython_real): # For all classes in vtkpython_real
9         # Create a class with the same name and no methods (yet),
10        # derived from "ProxyObject".
11        exec "class " + className + "(proxyObject): pass"
12 else:
13     # default behaviour: fall-through to the original VTK Python bindings
14     from vtkpython_real import *
```

Listing 1.2. A sample portion of a visualisation plan or recipe

```

1 ['construct', 'vtkConeSource', 'vtkConeSource_913']
2 ['callMeth', 'vtkConeSource_913', 'return_926', 'SetRadius', '0.2']
3 ['callMeth', 'vtkConeSource_913', 'return_927', 'GetOutput', '']
4 ['callMeth', 'vtkTransformFilter_918', 'return_928', 'SetInput', "self.ids['return_927']"]
5 ['callMeth', 'vtkTransformFilter_918', 'return_929', 'GetTransform', '']
6 ['callMeth', 'return_929', 'return_930', 'Identity', '']
```

3.2 Creating Skeleton Classes

MayaVi uses a dynamic lookup of method signatures in Python VTK classes as well as `__doc__` strings to create some GUI components on the fly, including for example the Pipeline Browser tool shown in Figure 2(b). We have to make sure therefore that the interface of the classes in the proxy layer matches the original VTK Python classes in terms of method signatures and `__doc__` strings. This is done by adding skeleton methods and `__doc__` strings on the fly following a dynamic lookup of class interfaces (using Python's built-in reflection mechanism). This adds a few seconds to program startup time when the tool is launched.

The only action performed by the skeleton methods is to call the `proxyCall` method from the `proxyObject` superclass, passing the name of the class, the name of the method and the list of actual parameters as arguments (more on this below).

3.3 Representing Execution Plans

We have implemented a Python data structure called `Code` which holds the information representing one call through the Python VTK interface. A `CodeList`

Listing 1.3. Code which creates an entry in the visualisation plan.

```
218 def proxyCall(self, callName, callArgs): # Add an entry to a recipe
219
220     # Check whether we have reached a "force point"
221     if(globals()["codeList"].numPartitions > 0 and callName == "Render"):
222         return forcePointReached()
223
224     result = proxyObject_return() # Create an identifier for the result
225     code = Code(result, self, callName, callArgs)
226         # Construct "Code" object which represents one method call
227     globals()["codeList"].add(code) # Add to the visualisation plan
```

Listing 1.4. Code snippet of recipe application for a method call

```
1 def callMeth(self, objId, retId, methName, argString):
2     object = self.ids[objId]
3     retobj = None
4     retobj = eval('object.' + methName + '(' + argString + ')')
5     self.ids[retId] = retobj
6     return retId
```

maintains the whole visualisation plan (or “recipe”). A symbol table for looking up identifiers is also maintained.

Listing 1.2 gives an example of what a part of a recipe may look like. The first item in the list is always `callMeth` or `construct` and signifies whether the recipe component is a constructor or an ordinary method call. If it is a constructor, the following items give the name of the class of the object to be constructed, the name of the identifier for the returned object, and (optionally) any arguments. If it is an ordinary method call, the following items give the object the method is to be called on, the name of the identifier for the returned object or value, the name of the method to be called and finally, the argument list. In the argument list, any names of identifiers for objects are converted into the a symbol table lookup of the form `self.ids[identifier]`.

Listing 1.3 shows part of the implementation of the `proxyCall` method, which creates entries in the visualisation plan, and which is called for every method invocation, via the skeleton methods in the proxy layer.

3.4 Forcing Execution

Listing 1.3 shows that when we call `Render`, we reach a force point, *i.e.* we force evaluation of the visualisation plan.

Listing 1.4 gives a code snippet (slightly simplified for clarity) of the function which applies a method call when a visualisation plan is executed. Again, the symbol table `ids` is used to map names of identifiers to real objects.

In Sections 4–6, we now present the performance benefits that accrue from applying two kinds of optimisation (tiling and parallelisation) to a VTK execution plan.

4 Coarse Grained Tiling of Visualisation Pipelines

Our case study is an ocean circulation model developed by our collaborators in the Department of Earth Science and Engineering at Imperial College. The datasets that result from such simulations are multi-gigabyte unstructured meshes. We use the METIS tool [7] to partition these datasets. The results presented in this paper are for a sample dataset representing a fluid flow around a heated sphere. This dataset is 16MB in size, and we have used a 2-,4-,8- and 16-way partitioning of this dataset. Note that VTK does have a built-in mechanism for handling partitioned datasets (“parallel unstructured grid storage format”); however, for the unmodified MayaVi, using such a dataset does not change the way the data is processed — the partitions are simply ‘glued together’ after loading from disk, resulting in a single monolithic dataset.

The first optimisation we study is coarse grained tiling. By coarse grained tiling, we mean that we apply the visualisation plan to one partition of the dataset at a time, rather than following the default behaviour where the partitions would be merged to form one monolithic dataset which is then processed. Note that this has strong similarities with classical tiling optimisations in that we are effectively restructuring the execution order (iteration space) in order to make better use of the memory hierarchy.

Experimental Setup. The MayaVi GUI has the capability of being run from a Python script, and this was of immense benefit for performance evaluation. Python’s built-in `time` module was used to take wall-clock time measurements. Each test was repeated three times (error bars are shown in all graphs). Two hardware platforms were used in testing:

- Intel Pentium 4 2.8GHz with hyperthreading (one physical processor), cache size 512KB, 1GB physical memory. We used this architecture both as a uniprocessor and in a cluster of four such processors.
- Athlon MP 1600+, cache size 256KB, 1GB physical memory. We used this architecture both as a uniprocessor and as a 2-way SMP.

In each case, the benchmarks were applied to a version of MayaVi that uses the unmodified Python/VTK library and to “MayaVi+DSI”, the version that uses our domain-specific interpreter for Python/VTK.

Use Case. The following usage scenario was used for evaluation: The dataset is loaded, and the IsoSurface module added to the visualisation pipeline. The contour value is changed seven times (0.00, 0.15, 0.30, 0.45, 0.60, 0.75 and 0.90), and the time taken for each change is recorded. When calculating an IsoSurface, the number of polygons generated, as well as computation time, varies widely with contour value.

Results for Tiling. Table 1 includes results for the tiling optimisation on uniprocessors. This indicates that in some cases, we can achieve a speedup of nearly a factor of 3 by this optimisation. The likely explanation for this speedup is that the isosurface algorithm produces intermediate data which is traversed repeatedly, and which fits more favourably into the memory hierarchy when processing one partition at a time.

5 Shared-Memory Parallelisation

It is apparently a simple extension of tiling to spawn one Python thread per partition. Unfortunately, with this basic approach, the threads always run sequentially due to Python’s Global Interpreter Lock (GIL), which prevents Python threads from running simultaneously. Any C code called from within the Python program is subject to this same limitation. To allow C code called from the VTK Python interface to be run in parallel, this lock must be explicitly claimed and dropped from within the C code.

Python wrapper classes for VTK are generated by a special-purpose C program (`vtkWrapPython.c`). Adding code to deal with the GIL within the wrapper generator requires minimal modification (2 lines): At every point that a VTK wrapped function is called, the function call is surrounded by code to drop and reclaim the lock. However, since not all of VTK is thread-safe, it is necessary to restrict parallel operation to certain parts of VTK only. In particular, no operations on the shared `Renderer` and `RenderWindow` occur in parallel.

Table 1 shows results for SMP parallelisation using the method outlined above on a dual-processor machine. This shows very encouraging results: a maximum speedup of around 6 for parallel execution of the visualisation pipeline over 16 partitions. We spawn one thread per partition, rather than per processor, these results show the combined effects of tiling and SMP parallelisation. This explains the superlinear speedup which is seen for some `IsoSurface` values. Our data indicate that on a 2-way SMP, a significant part of the overall speedup is already obtained by tiling alone.

Some readers may wonder where the “parallel loop” for our parallelisation is. The answer is that this is in the domain-specific interpreter. It is the task of this interpreter to execute the recipe that has been captured. This is done by applying the entire recipe, which could be a multi-stage computation, to each partition separately.

6 Distributed Memory Parallelisation

To allow for an easier implementation of distributed processing, a Python library called ‘Pyro’ was used, which provides Java RMI-like features. Pyro allows any pickleable² object to be transferred across the network. A class which needs to be

² A ‘pickleable’ object is one which can be serialised using Python’s built-in ‘pickle’ module

Athlon 1600+ SMP

IsoSurface	2 Partitions						4 Partitions					8 Partitions					16 Partitions				
	Base	Tiling		SMP		Base	Tiling		SMP		Base	Tiling		SMP		Base	Tiling		SMP		
	Time	Time	$S'up$	Time	$S'up$	Time	Time	$S'up$	Time	$S'up$	Time	Time	$S'up$	Time	$S'up$	Time	Time	$S'up$	Time	$S'up$	
0	4.73	5.12	0.92	4.9	0.97	3.32	3.96	0.84	3.63	0.92	3.16	5.21	0.61	4.92	0.64	3.01	3.84	0.78	2.53	1.19	
0.15	1.62	0.88	1.84	0.67	2.43	1.41	0.92	1.53	0.79	1.79	1.48	0.92	1.62	0.71	2.08	1.43	0.94	1.53	0.74	1.94	
0.3	1.77	1.66	1.07	1.02	1.74	1.64	1.58	1.04	1.06	1.55	1.67	1.35	1.24	1	1.67	1.74	1.33	1.31	1	1.74	
0.45	3.65	3.49	1.04	2.13	1.71	3.21	2.83	1.13	1.82	1.76	3.22	1.72	1.87	1.34	2.41	3.36	1.61	2.09	1.22	2.76	
0.6	7.7	6.91	1.11	4.78	1.61	6.98	5.42	1.29	3.6	1.94	7.46	3.27	2.28	2.65	2.81	7.28	2.6	2.8	1.84	3.96	
0.75	15.44	13.48	1.14	10.71	1.44	15.56	11	1.41	8.07	1.93	15.06	6.89	2.18	5.77	2.61	15.27	4.45	3.43	2.9	5.27	
0.9	27.5	22.53	1.22	19.47	1.41	24.78	20.48	1.21	16.82	1.47	25.92	12.55	2.07	10.88	2.38	25.24	6.58	3.83	4	6.31	

Pentium 4 2.8 GHz Cluster

IsoSurface	2 Partitions						4 Partitions					8 Partitions					16 Partitions				
	Base	Tiling		Parallel		Base	Tiling		Parallel		Base	Tiling		Parallel		Base	Tiling		Parallel		
	Time	Time	$S'up$	Time	$S'up$	Time	Time	$S'up$	Time	$S'up$	Time	Time	$S'up$	Time	$S'up$	Time	Time	$S'up$	Time	$S'up$	
0	4.91	4.56	1.08	6.6	0.74	3.55	2.96	1.2	4.18	0.85	3.23	3.28	0.99	4.18	0.77	3.14	3.29	0.95	1.9	1.65	
0.15	0.65	0.73	0.89	0.83	0.78	0.65	0.67	0.97	0.64	1.01	0.66	0.65	1.02	0.57	1.16	0.68	0.66	1.03	0.6	1.12	
0.3	1.33	1.47	0.9	1.32	1.01	1.25	1.29	0.97	0.92	1.36	1.27	1.02	1.24	0.81	1.57	1.32	0.97	1.36	0.72	1.82	
0.45	2.64	3.21	0.82	2.31	1.14	2.35	2.38	0.99	1.23	1.91	2.37	1.27	1.86	1.11	2.13	2.45	1.23	2	1.01	2.43	
0.6	5.35	5.9	0.91	4.3	1.24	5.02	4.5	1.12	2.57	1.95	5.08	2.41	2.1	2.01	2.53	5.18	2.12	2.45	1.45	3.57	
0.75	10.82	10.54	1.03	8.69	1.24	10.43	8.64	1.21	5.66	1.84	10.49	4.79	2.19	4.14	2.53	10.6	3.77	2.81	2.11	5.03	
0.9	18.28	17.24	1.06	14.39	1.27	18.6	14.72	1.26	10.21	1.82	16.92	8.48	2	7.38	2.29	17.04	5.83	2.92	2.79	6.1	

Table 1. IsoSurface Benchmark on Dual Athlon 1600+ SMP with 256 KB L2 cache and 1GB physical RAM (above) and cluster of four Pentium 4 2.8 GHz HT with 512 KB L2 cache and 1 GB physical RAM (below). Results for the Tiling columns only use one processor in both cases.

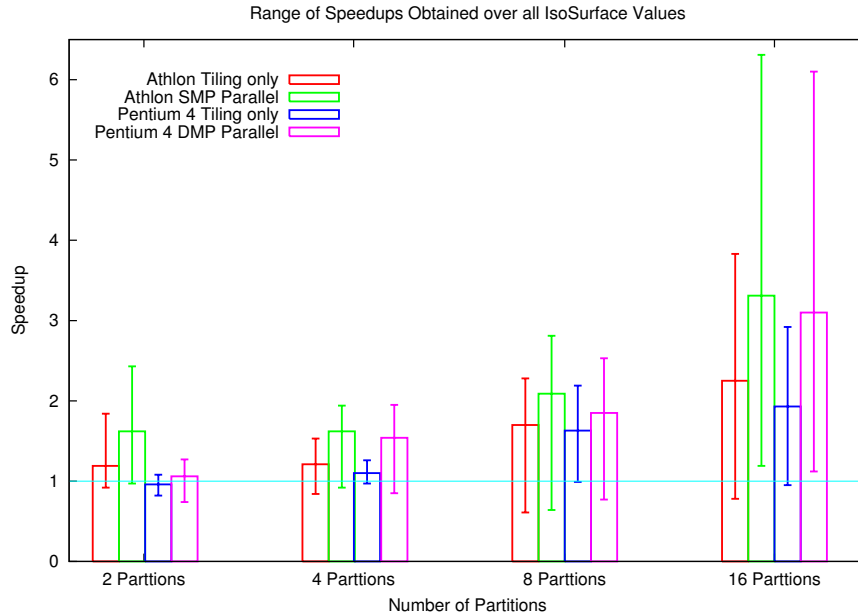


Fig. 4. Summary of speedups obtained through our optimisations: For each experiment, we show for all degrees of partitioning the average, smallest and largest speedup obtained over all IsoSurface values in our use case. The details for these figures are contained in Table 1.

remotely accessible is subclassed from `Pyro.core.ObjBase`. In addition, there is a `Pyro.core.SynchronizedObjBase` class, which automatically synchronises remote access to the class methods.

Unfortunately, VTK pipeline components and data objects are not pickleable, and, as such, cannot be transferred using Pyro. Data objects may, however, be read and written to temporary files using VTK’s data reading and writing classes, and these files are written by servers and read by clients to transfer input and output data. MayaVi includes some pickling capabilities for VTK pipeline components, although it is not complete enough as not all attributes are pickled. Therefore, in order to propagate the pipeline structure, the client needs to cache and transfer all calls and arguments needed to recreate the visualisation pipeline.

A limitation of the data transfer implementation is that files are not explicitly transferred between server and client, but rather to a shared file system. A direct transfer between local disks would be more flexible and may give better performance.

Table 1 shows the performance of our distributed memory parallelisation scheme. This shows that the speedup obtained from distributed memory parallelisation for most calculations (in particular, the slowest ones) outperforms the benefit of tiling on a uniprocessor. The key overhead involved with distributed

memory parallelisation is the saving and loading of resultant data (the implementation of returning results involves the server writing VTK data to a file, which the client then has to read in). So, it is indeed expected that the performance gains will be best when the decrease in computation time can outweigh this overhead (this, in turn, is smallest when the computation results are small).

In Figure 4, we summarise our experimental results. Since the computation time in our use case varies greatly with the IsoSurface value, we show speedups as box-and-whisker plots, giving the average, minimum and maximum speedup over all IsoSurface values. This shows that tiling is almost always beneficial: we obtain a speedup of around a factor 2 for larger numbers of partitions on both architectures. Furthermore, for the computationally more expensive operations, combining tiling with parallelisation can lead to significant performance improvements (up to a factor 6 for both a dual-processor Athlon SMP and a 4-processor Pentium 4 cluster).

7 Conclusions and Future work

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 modular visualisation environment enabled us to reliably capture the construction of the visualisation pipeline by intercepting all calls that pass through the Python/C++ VTK interface. We have presented results for tiling on a uniprocessor, as well as shared- and distributed-memory parallelisation.

We are currently exploring how we can build on the infrastructure we have described.

- *Using VTK Streaming Constructs.* As stated in Section 2.2, 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 scientific software frameworks.

References

1. BLAST Forum: Basic linear algebra subprograms technical forum standard. (2001) Available via www.netlib.org/blas/blas-forum.
2. Message Passing Interface Forum: MPI: A Message Passing Interface Standard. University of Tennessee, Knoxville, Tennessee. (1995) Version 1.1.
3. Kitware, Inc.: The VTK User's Guide: VTK 4.2. (2003)
4. Schroeder, W., Martin, K., Lorensen, B.: The Visualization Toolkit: An Object-Oriented Approach To 3D Graphics. 3rd edn. Kitware, Inc. (2002)
5. Ramachandran, P.: MayaVi: A free tool for CFD data visualization. In: 4th Annual CFD Symposium, Aeronautical Society of India. (2001) mayavi.sourceforge.net.
6. van Rossum, G., Fred L. Drake, J.: An Introduction to Python. Network Theory Ltd (2003)
7. Karypis, G., Kumar, V.: Multilevel algorithms for multi-constraint graph partitioning. In: Supercomputing '98, IEEE Computer Society (1998) 1–13
8. Beckmann, O., Field, A.J., Gorman, G., Huff, A., Hull, M., Kelly, P.H.J.: Overcoming barriers to restructuring in a modular visualisation environment. In Cox, A., Subhlok, J., eds.: LCR '04: Languages, Compilers and Runtime Support for Scalable Systems. (2004) ACM Digital Library.
9. Cameron, G.: Modular visualization environments: Past, present, and future. *Computer Graphics* **29** (1995) 3–4
10. Parker, S.G., Johnson, C.R.: SCIRun: A scientific programming environment for computational steering. In: Proceedings of Supercomputing 1995. (1995)
11. Wright, H., Brodlie, K., Brown, M.: The dataflow visualization pipeline as a problem solving environment. In: Virtual Environments and Scientific Visualization '96. Springer-Verlag, Vienna, Austria (1996) 267–276
12. Johnson, C.R., Parker, S.G., Weinstein, D.: Large-Scale Computational Science Applications Using the SCIRun Problem Solving Environment. In: ISC 2000: International Supercomputer Conference, Mannheim, Germany (2000)
13. Foster, I., Vöckler, J., Wilde, M., Zhao, Y.: Chimera: A virtual data system for representing, querying, and automating data derivation. In: 14th International Conference on Scientific and Statistical Database Management (SSDBM'02). (2002)
14. Cignoni, P., Montani, C., Scopigno, R.: MagicSphere: An insight tool for 3D data visualization. *Computer Graphics Forum* **13** (1994) C/317–C/328
15. Beckmann, O., Kelly, P.H.J.: Efficient interprocedural data placement optimisation in a parallel library. In: LCR98: Languages, Compilers and Run-time Systems for Scalable Computers. Number 1511 in LNCS, Springer-Verlag (1998) 123–138
16. Liniker, P., Beckmann, O., Kelly, P.H.J.: Delayed evaluation self-optimising software components as a programming model. In: Euro-Par 2002: Proceedings of the 8th International Euro-Par Conference. Number 2400 in LNCS (2002) 666–673
17. Yeung, K.C., Kelly, P.H.J.: 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 (2003)
18. Furmento, N., Mayer, A., McGough, S., Newhouse, S., Field, T., Darlington, J.: Optimisation of component-based applications within a grid environment. In: Supercomputing 2001. (2001)