A Domain-Specific Interpreter for Parallelising a Large Mixed-Language Visualisation Application

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

Department of Computing, Imperial College London,

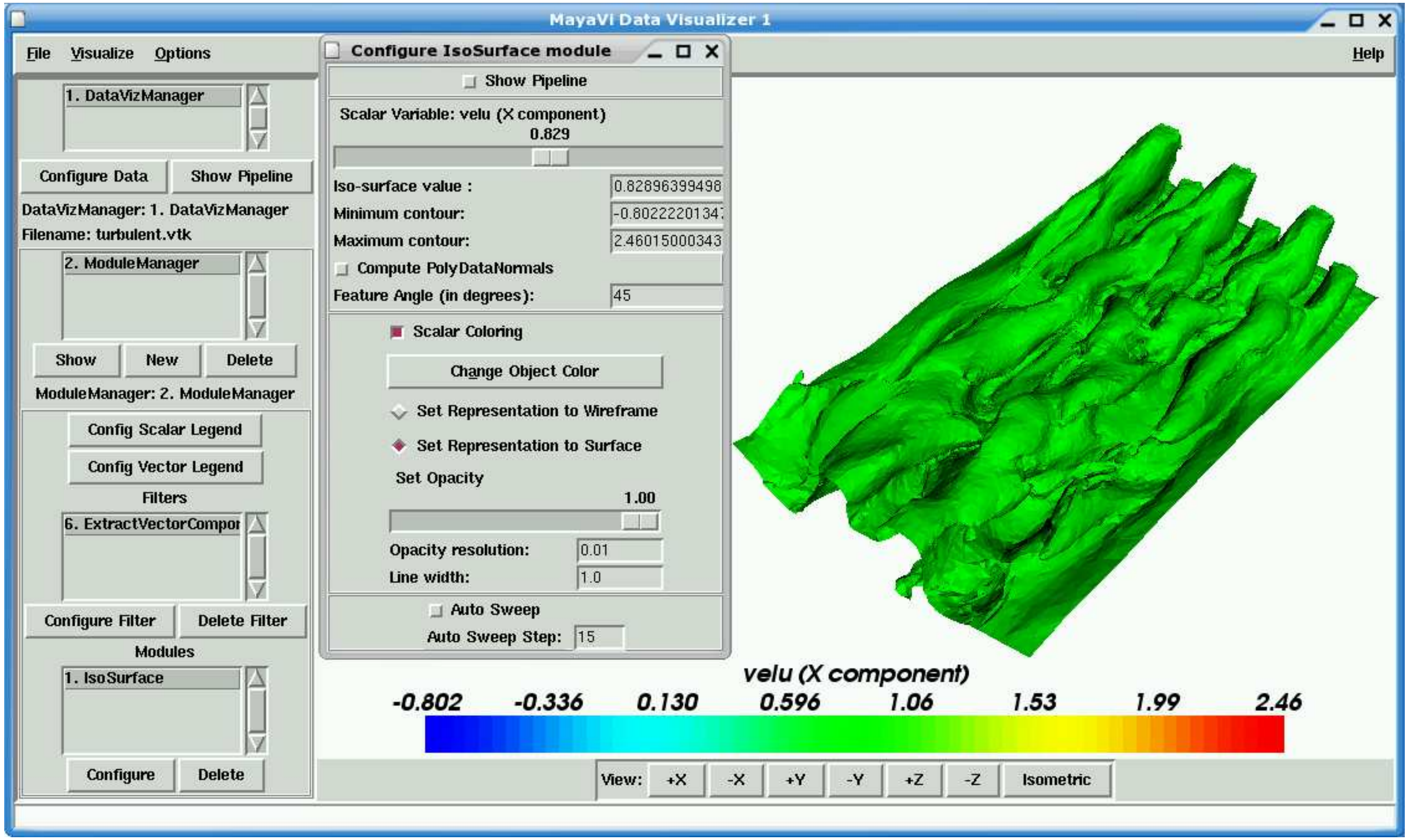
180 Queen's Gate, London SW7 2AZ, United Kingdom

`http://www.doc.ic.ac.uk/~ob3`

Visualising Large Ocean Current Simulations

MayaVi

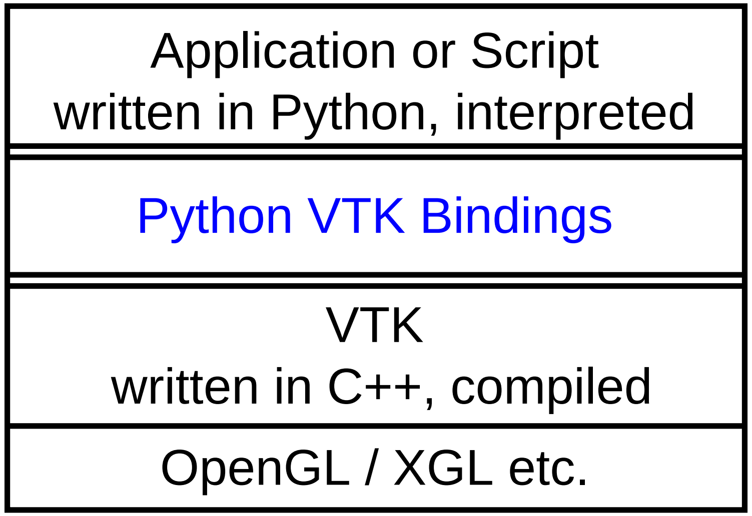
Modular Visualisation Environment



- Graphical interface for composing analysis and rendering components
- 22,000 LOC, Python + VTK
- Open source, active development
- Poor interactive performance limits usefulness

Python/VTK Visualisation Software Architecture

- Visualisation typically involves a pipeline of feature-extraction operations
- When working on extremely large datasets, response time for interactive parameterisation of the visualisation pipeline is poor.
- The challenge is to make visualisation of large datasets interactive by improving use of memory hierarchy and parallelisation



- Multi-language: Python, C++, C
- Component-based
- Actively changing code base, maintained by people who have no time for parallelisation
- Mixed dynamic / static
- Domain-specific semantics in DSL (VTK)

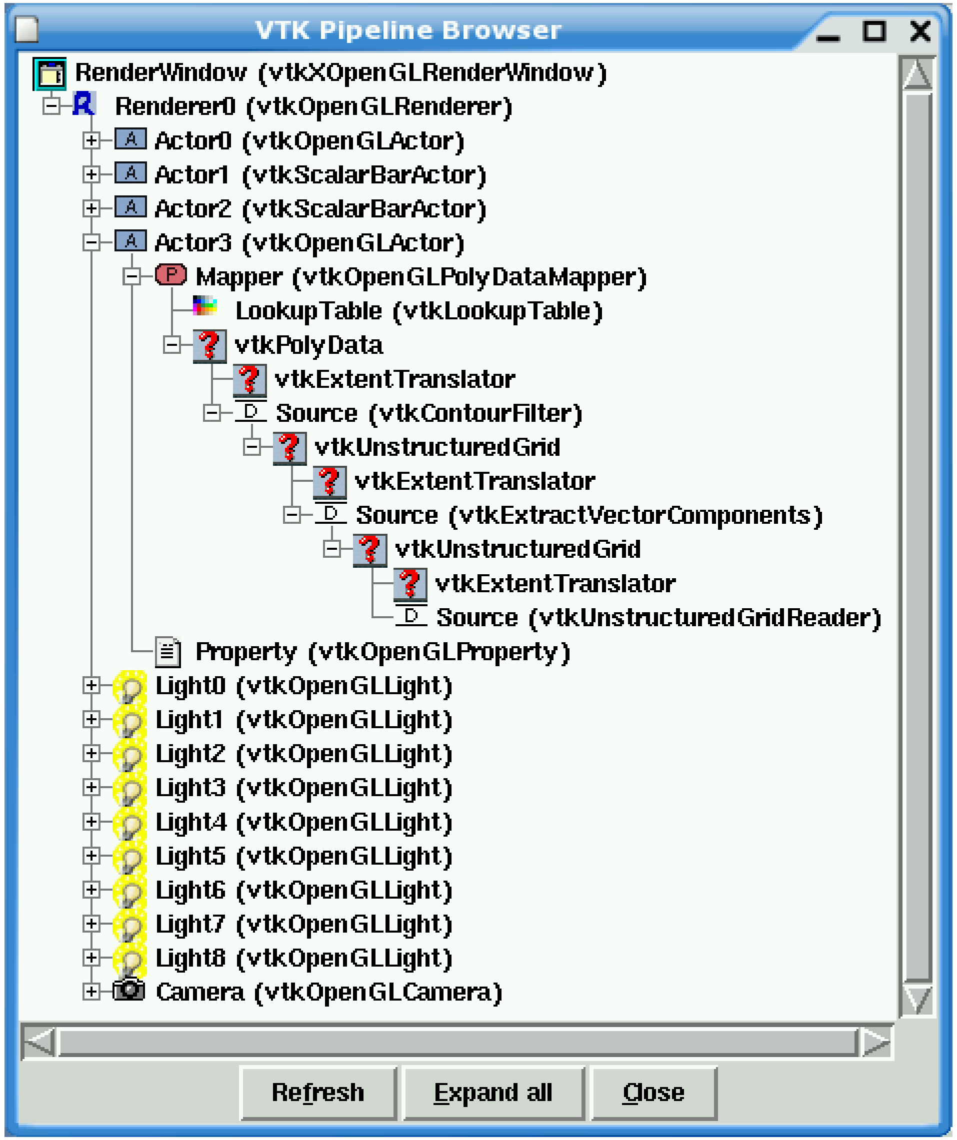
Object-Oriented Visualisation in VTK

Graphics Model:
Object-oriented representation of 3D computer graphics

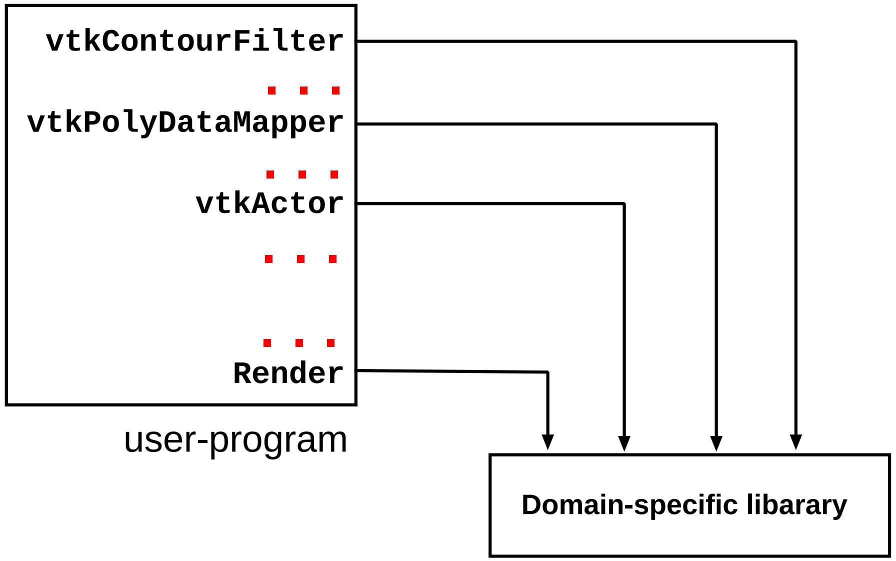
Visualisation Model

- Model of data flow.
- Capable of representing complex data-flow graphs: "visualisation pipelines"
- Data-flow graphs can be executed in a demand-driven or data-driven manner.
- Surprisingly similar to high-level compositional programming models.

VTK Visualisation Pipeline



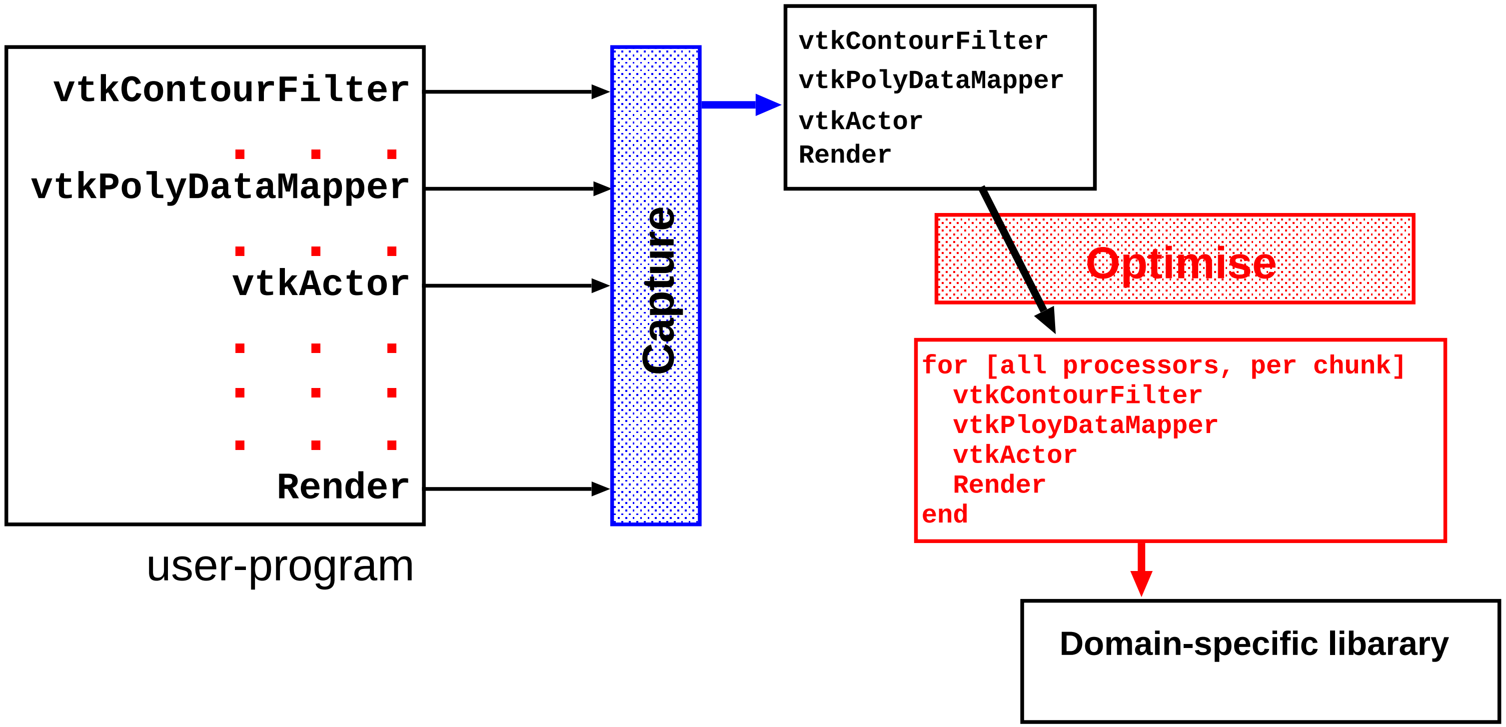
Domain-Specific Libraries: Typical Use



- Program compiled with standard compiler (gcc, icc, . . .) or interpreted with standard interpreter (*e.g.* python).
- DSL code mixed with other code.
- No domain-specific optimisation.

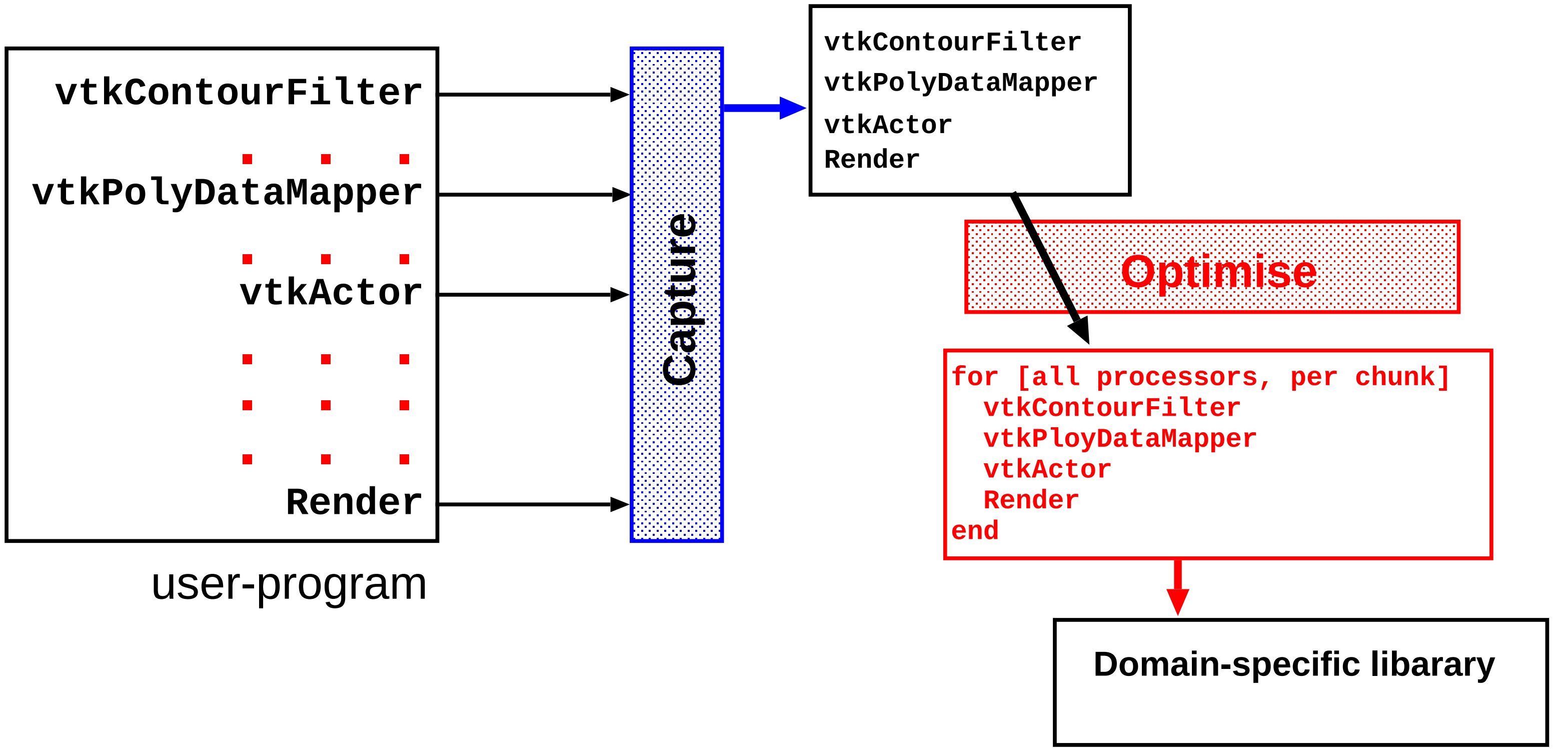
- Using such DSLs often dominates and constrains the way a software system is built just as much as a programming language.
- Compiling a quasi domain-specific language without a domain-specific compiler or optimiser.
- Typically miss out on cross-component optimisation opportunities that exploit the domain-specific semantics of the library.

Domain-Specific Interpreter Pattern



- User program is unmodified and is compiled with or interpreted by unmodified language compiler or interpreter.
- Capture all calls to methods from a DSL.
- Apply domain-specific optimisation, then call the underlying library.

Domain-Specific Interpreter Pattern



Applicability (Requirements)

- Reliable capture
  VTK/Python bindings
- Reliable capture of data-flow through DSL routines.
  Opaque VTK data structures

Profitability

- Domain-specific semantics
  Piecewise evaluation valid
- Opportunities for optimisations across method calls
  Size of intermediate data

Domain-Specific Interpreter for VTK in Python

`mv vtkpython.py vtkpython_real.py` then `vtkpython.py`:

```python
if ("vtkdsi" in os.environ): # Control DS Interpreter via Environment
   import vtkpython_real # Original vtkpython.py re−named
   from vtkdsi import proxyObject
   for className in dir(vtkpython_real): # For all classes in this module
      exec "class " + className + "(proxyObject): pass" # class with no methods (yet)
else:
   from vtkpython_real import * # fall−through to original VTK Python
```

- For all classes from `vtkpython_real.py`, create a class by the same name, with no methods, derived from proxyObject.
- Explicit hooks for capturing all field and method accesses (cf. AOP)

```python
class proxyObject:
   def __getattr__(self, callName): # Catch−all method
      return lambda *callArgs: self.proxyCall(callName, callArgs) # lambda call
```

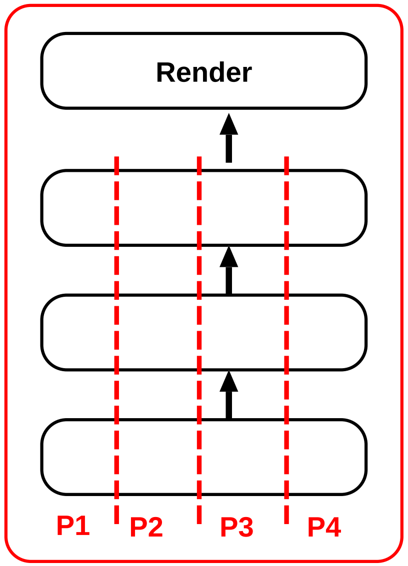
Visualisation Recipes

- The scheme we showed on the previous slide works lazily for all calls through VTK Python interface
  - We need to identify *force points* (i.e. Render()).
  - Lazy indirection causes Python’s reflection mechanism to break; therefore we actually use a more eager scheme.
- The proxy stores all calls made to VTK in a visualisation *recipe*.
- When a force point is reached, the recipes are evaluated.

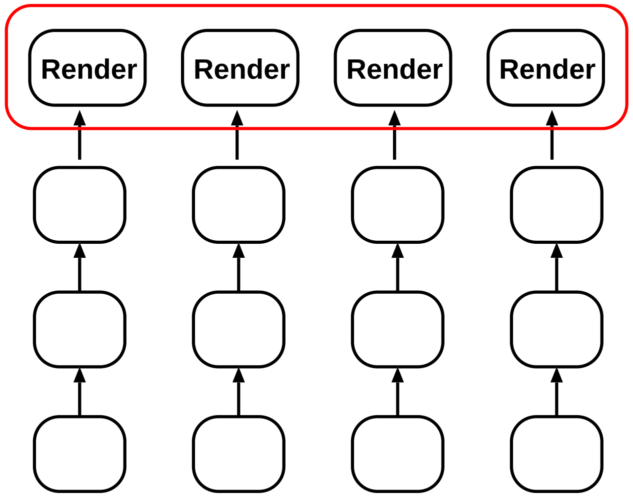
```
[’construct’, ’vtkConeSource’, ’vtkConeSource_913’]
[’callMeth’, ’vtkConeSource_913’, ’return_926’, ’SetRadius’, ’0.2’]
[’callMeth’, ’vtkConeSource_913’, ’return_927’, ’GetOutput’, ’’]
[’callMeth’, ’vtkTransformFilter_918’, ’return_928’, ’SetInput’, "self.ids[’return_927’]"]
[’callMeth’, ’vtkTransformFilter_918’, ’return_929’, ’GetTransform’, ’’]
[’callMeth’, ’return_929’, ’return_930’, ’Identity’, ’’]
```

Optimising VTK Visualisation Pipelines

- Simulations generating the datasets we are visualising are run in parallel, resulting in a parallel tetrahedral VTK data set.
  - This means: XML file giving locations of partitions
  - Normally, VTK fuses the partitions into one whole dataset.
  - If a dataset has not been generated as a collection of partitions, we can use METIS to create a partitioned version.
- VTK does have parallel routines — data-parallel using MPI.
- We are interested in a more dynamic scenario, steered from a client.

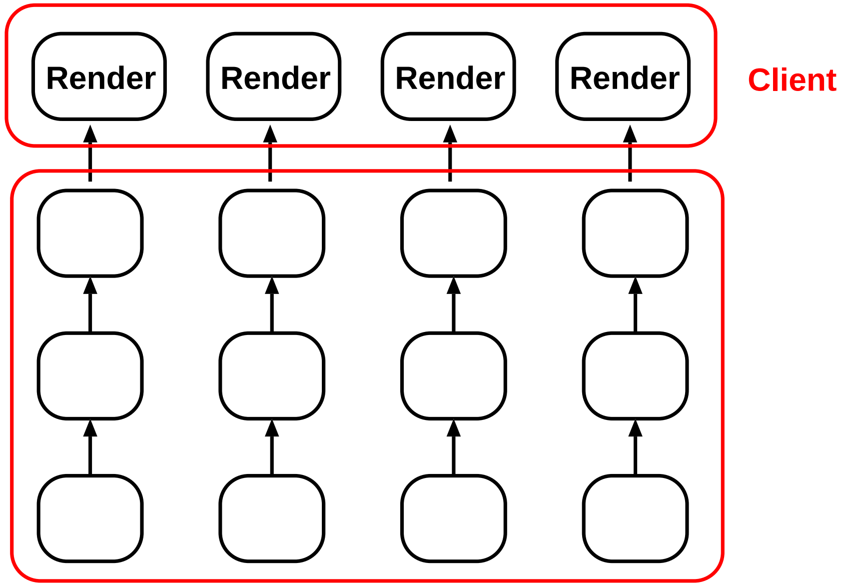


SPMD
Data-Parallel



Client

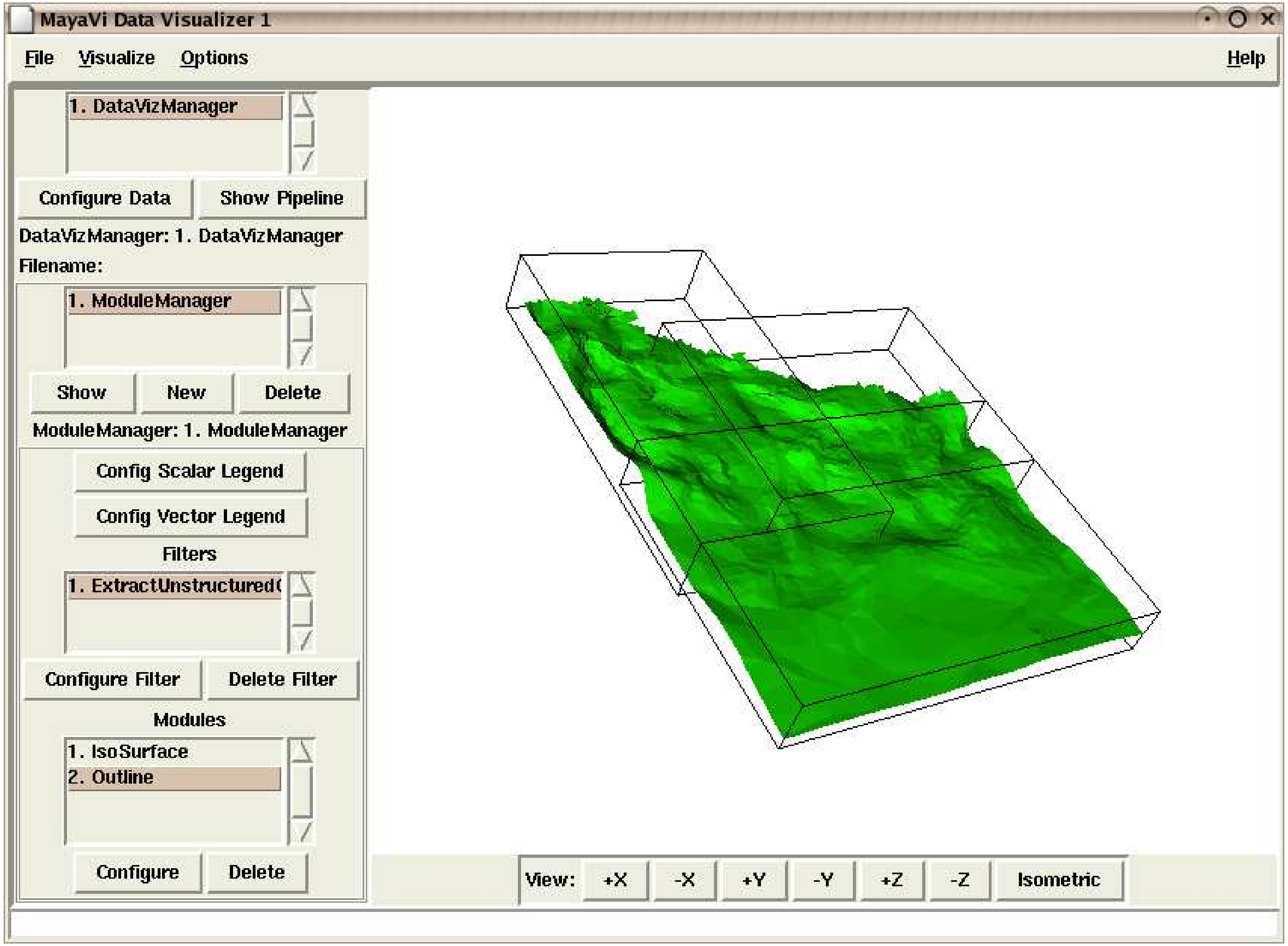
Coarse-Grained Tiling of VTK Visualisation Pipelines



- Large intermediate data means that multi-stage visualisation pipelines make poor use of memory hierarchy.

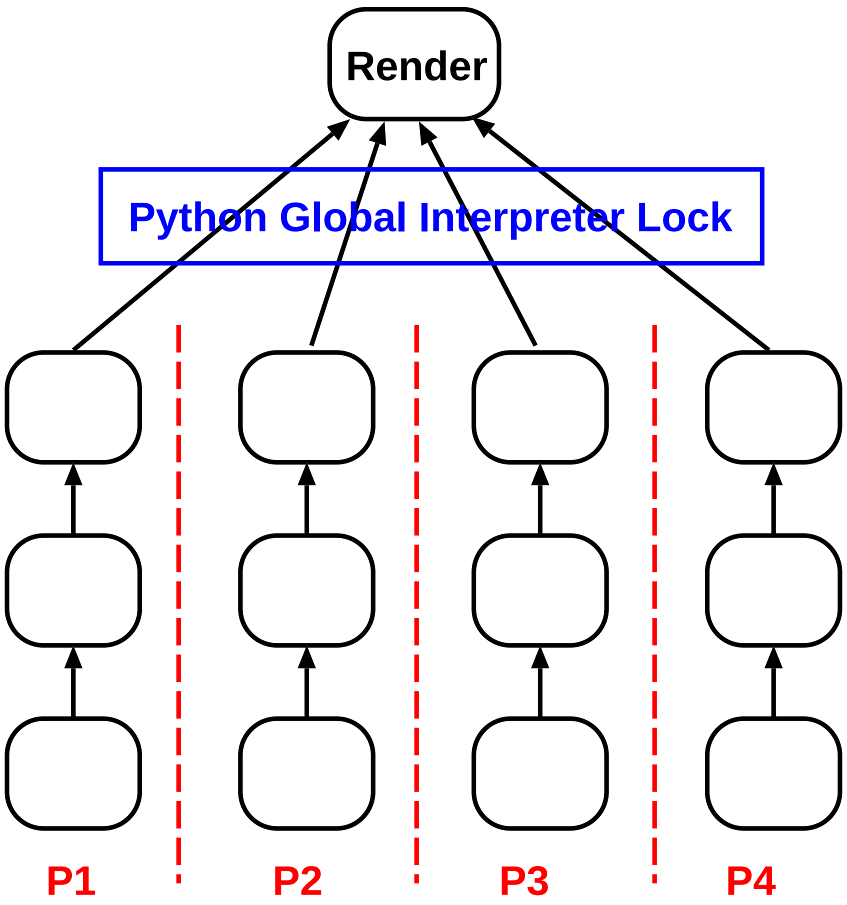
- Our domain-specific `vtkpython` interpreter builds a data-structure representing the sequence of operations performed.
- When the user-application calls `Render()`, we apply this partition-by-partition on the data-set.
- The only difference is an environment variable.
- Domain-specific semantics determine the validity of this transformation.

Coarse-Grained Tiling of VTK Visualisation Pipelines



Calculating isosurfaces one partition at a time, showing outlines of partitions.

Shared-Memory Parallelisation



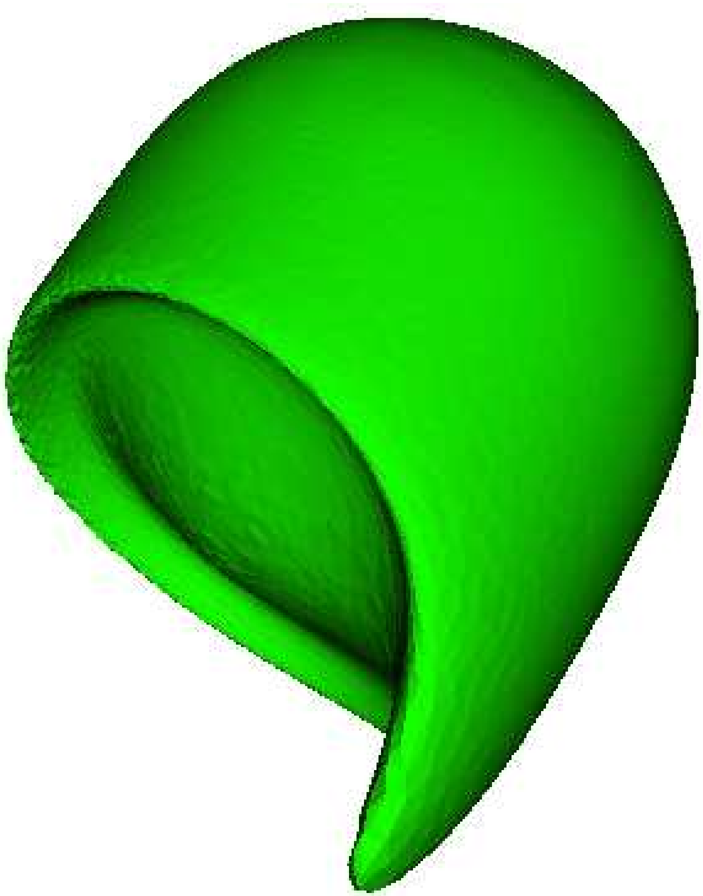
- Plan: execute the visualisation pipelines for each tile in parallel on an SMP.

- The first obstacle is that Python interpreter is not thread-safe!
  - This can be overcome by manually lifting the GIL (global interpreter lock) on the C++ side.
- Some VTK routines are also not thread-safe, or do not have parallel semantics.
- Rendering via OpenGL is not thread-safe.
  - So we do *not* lift the GIL when calling C++-side rendering from Python.

Distributed Memory Parallelisation

- Use a cluster of machines to perform the calculation in parallel and then render on one client machine.
- Used Python library `Pyro` to provide RMI-like features for Python.
  - `Pyro` allows 'pickleable' (serialisable) objects to be transferred over the network.
  - Our recipes can be transferred to servers in the cluster in this way.
  - Unfortunately, VTK objects cannot be serialised using the 'pickle' mechanism.
  - Therefore use a shared filesystem to transfer VTK objects.
- This is a dynamic, client-server model of distributed memory parallelisation, not data-parallel.

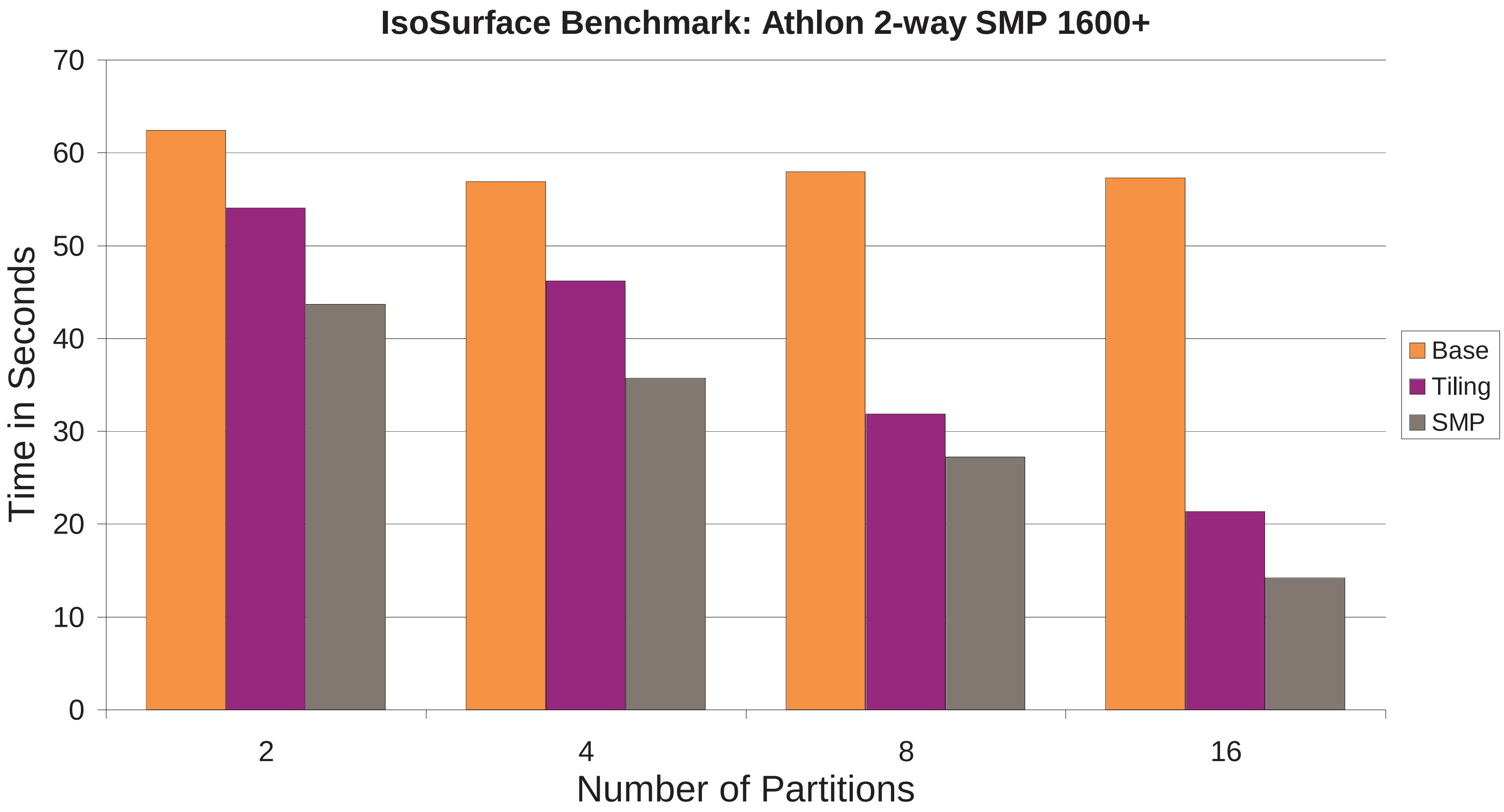
Evaluation



- Size: Approx 325 MB

- Benchmark Scenario
  - Open a dataset representing flow over heated sphere
  - Plot seven isosurfaces at different values
- Platforms
  - Athlon 1600+, dual SMP, 256 KB L2, 1GB RAM, Linux 2.4
  - Cluster of 4 Pentium 4 2.8 GHz, 512 KB L2, 1GB RAM, Linux 2.4

Results for IsoSurface Benchmark



- Benchmark consists of loading a dataset representing flow over heated sphere and calculating 7 isosurfaces at different values.

Results for IsoSurface Benchmark

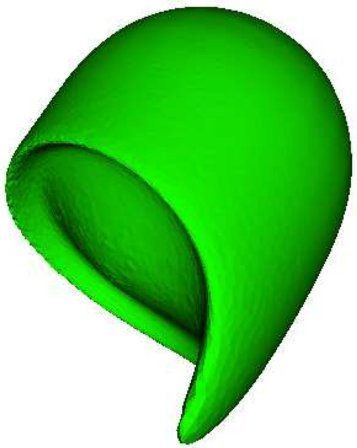
- Benchmark consists of loading a dataset representing flow over heated sphere and calculating 7 isosurfaces at different values.

Related Work

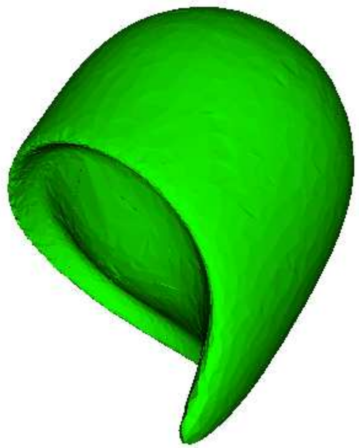
- Dynamic component assembly software architectures
  - SciRun / BioPSE / Uintah
  - “Virtual data-grid” projects
- Dynamic cross-component optimisation
  - Telescoping languages (Rice, LLNL)
  - Code generation approaches
- Kitware tool: Paraview
  - This makes use of VTK’s data-parallel routines, relies on MPI
- Grid workflow engines
  - Related in that we assemble a workflow at runtime, then execute
  - Our work illustrates an interesting pathway for facilitating “legacy code” to operate in such an environment.

What's Next

- Use of metadata to carry additional domain-specific semantics
  - Parallel semantics, thread-safe on C++ side, use-def equations
- Reducing the size of the polygon set before rendering
  - Cache full sets on servers, return decimated sets to client



full



80%

85%

- Level of detail (LOD) and region-of-interest (ROI) selection
- Multiple time steps
  - Speculatively applying the recipe to future timesteps
  - Aim to achieve smooth rendering of a series of timesteps

Conclusion

- We have parallelised a large, open-source visualisation application without changing a single line of code.
- Entirely transparent to application program, controlled via an environment variable.
- Works for any Python visualisation script using VTK.
- Use Python to implement a domain-specific interpreter for a domain-specific library
  - Facilitated by reliable capture of DSL calls and known data-flow due to opaque objects on the C++ side.
- Optimisations
  - Coarse-grained tiling
  - SMP parallelisation
  - Distributed memory parallelisation
- Dynamic, runtime parallelisation