Python for CFD: A case study

Prabhu Ramachandran

Department of Aerospace Engineering

IIT-Madras, Chennai, INDIA

Introduction

- Python: ideally suited for scientific computing
  - Interfaces with Fortran/C/C++
  - Standard library, `Numeric`, `scipy`
  - Extremely versatile
  - Interactive interpreter: prototyping, testing, debugging, analysis (post-processing)
  - Makes doing ordinarily mundane tasks fun!
  - Is not just a “glue” language

Outline

- Vortex methods
- Building with SCons
- Miscellaneous scripts
- Job scheduler
- Wrapping C++: SWIG
- Interactive data analysis
- Parallel application
- Conclusion

Vortex methods

- Lagrangian and grid-free numerical scheme
- 2D, incompressible, Navier-Stokes fluid
- Vorticity is the curl of the velocity field
- Represent the flow in terms of vorticity
- Discretize vorticity into particles (blobs and sheets)
- Track vorticity as per: $\frac{D\omega}{Dt} = \nu\nabla^2\omega$
- $\omega = \text{curl}\vec{V} \cdot \hat{k}$, ν kinematic viscosity
- BC: no-penetration, no-slip

Vortex Methods: Advantages

- No grid generation
- Self adaptive
- Ideal for unsteady flows
- Intuitive solution procedure

Numerical procedure

- Create vorticity to satisfy no-slip
- Two-step procedure to track vorticity
- Advection: $\frac{D\omega}{Dt} = 0$
  - Vortex particles move with the flow
- Diffusion: $\frac{\partial \omega}{\partial t} = \nu \nabla^2 \omega$
  - Solution using non-deterministic and deterministic schemes
  - Random Vortex Method (RVM)

Salient features

- Computationally intensive
- Complex algorithms for efficiency (Adaptive Fast Multipole Methods)
- RVM is stochastic, requires ensemble averaging
- High-resolution simulations of flow past an impulsively started cylinder
- VEBTIFS – Vortex Element Based Two-dimensional Incompressible Flow Solver: library in C++

Building with SCons

- Fairly large code base: ≈ 80 classes
- Build variants: graphics, debug, profile, shared and static
- SCons scripts are Python scripts: no new sytax to learn
- Does more with less (1000 line Makefile - 200 line SConscript)
- Parallel from ground up
- Works well with distcc

Miscellaneous scripts

- Parse text data files with a known structure
- Trivial to write with Python and very useful

```
key = value # comment
# or
key1, key2 = value1, value2 (garbage text)
# or just
value1, value2
```

Misc. scripts

- Output/input file types and versions change
- ASCII, XDR, optionally zipped files
- Many versions and formats: inter-conversions
- A few day's work resulted in code that can
  - Interactively explore data files
  - Change formats
  - Documents the formats
  - Useful command line interface

Job Scheduler

- Schedule ≈ 1000 runs in limited time
- CPU intensive (0.5 to 24 hours per run)
- Cluster (running Linux) used by other users
- Manually running/managing: inefficient and error prone
- Limited time: eliminates other clustering tools
- Not suitable for shell scripts or C/C++
- Python: One day's coding, 500 lines, runs as a daemon, command line interface
- Fun to write!

Managing runs and data

- Scripts generate the data files
- File associating directories to machines
- Scripts to transfer files and submit the jobs
- Similar scripts to get data from machines
- IPython was used as a shell to drive everything
- Easy to write: a few days of effort

SWIG: Wrapping C++

- VEBTIFS is written in C++: ≈ 80 classes
- Wrapped to Python using SWIG
- About 500 lines of SWIG interface code
- Took around a week's effort to get working wrappers
- Lets us script runs, inspect and analyse data
- Scripting VEBTIFS is a huge plus

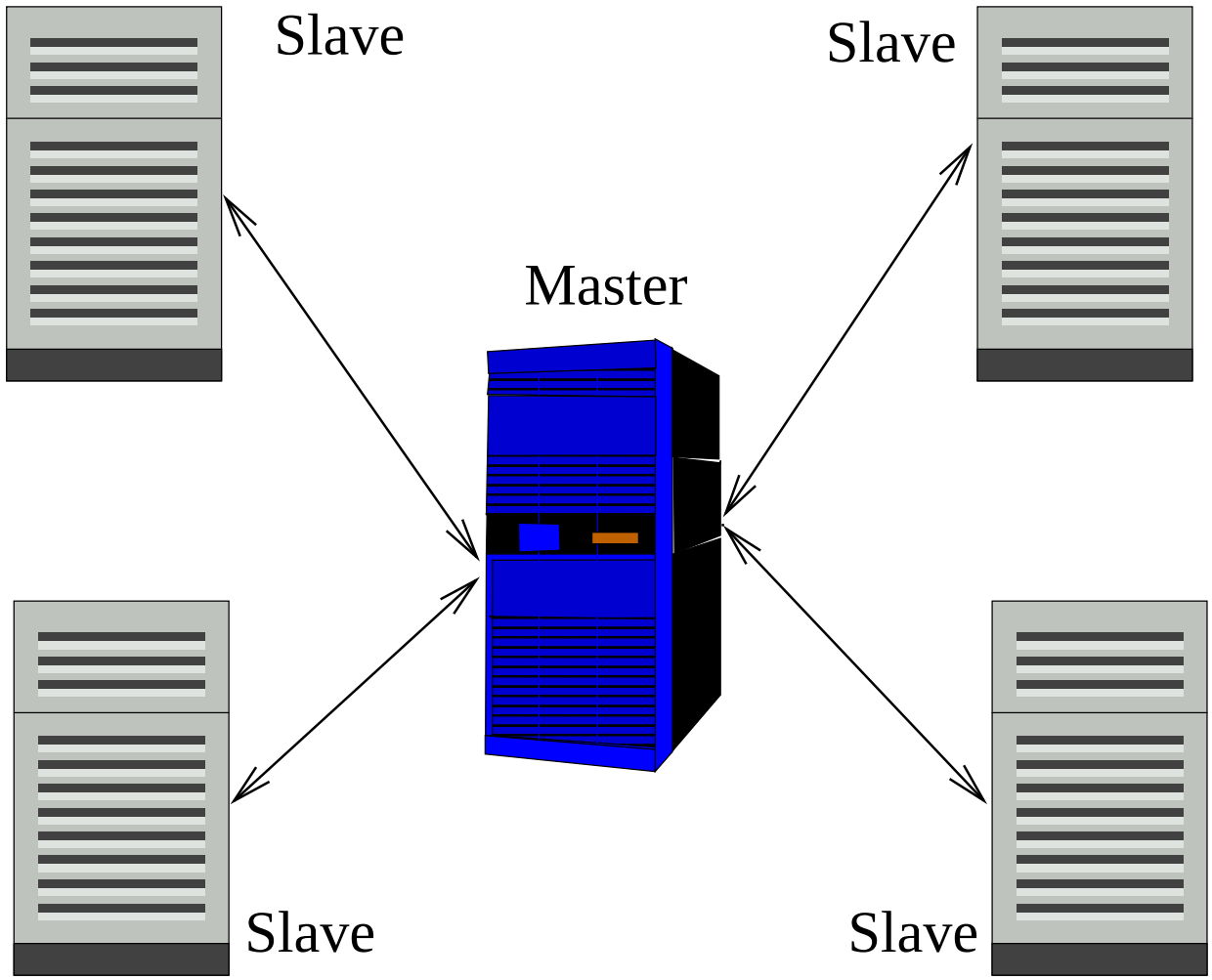
Interactive data analysis

- IPython is used exclusively for the shell
- Abstract common tasks into useful functions and classes
- Post-processing of collected data: statistics, de-noising, derivatives, plotting etcetra
- Use `Numeric` and `scipy` for various tasks
- 2D plots using Grace and `gracePlot.py`
- 2D surface and 3D plots using MayaVi

Parallel application

- Uses a cluster of Linux machines
  1. Slaves and master perform computations
  2. Every n_{sync} time steps, slaves communicate particle data to master
  3. Master processes data and sends it back to slaves
  4. Repeat from step 1
- Embarassingly parallel
- Small amount of communication (3MB per slave)

Machine configuration



Implementation in Python

- Ideal to prototype using Python
- CPU intensive work uses SWIG wrapped VEBTIFS
- Communicate data using `Pypar`: Numeric arrays representing particle data sent using MPI
- Particle data stored in internal C++ objects
- Protyping the idea took half a day
- Debugging took the other half

Salient points

- Easy to write
- ≈ 500 lines of code in all
- Pypar initializes MPI and makes it extremely transparent to use
- Easy ability to debug and fix problems
- Display graphical progress on all slaves
- Problem: Conversion of data from C++ to Numeric and back is *slow*

Slow code

```
def data2Blob(blb_data, bm):
    # Given numeric arrays containing blob data and a  BlobManager,
    # populate the BlobManager with blobs.
    z, gam, core = blb_data['z'], blb_data['strength'], \
                   blb_data['core']
    bf = vebtifs.BlobFactory()
    for i in xrange(len(z)):
        bm.addElement(bf.create(z[i], gam[i], core[i]))
```

Enter `scipy.weave`

- Weave needed support for SWIG2: added in a day
- Weave'd code is similar and fairly easy to read
- Produces a 300-400 fold speed increase!
- Makes it possible to write efficient code
- Did take some effort and knowledge to get working

Weave code

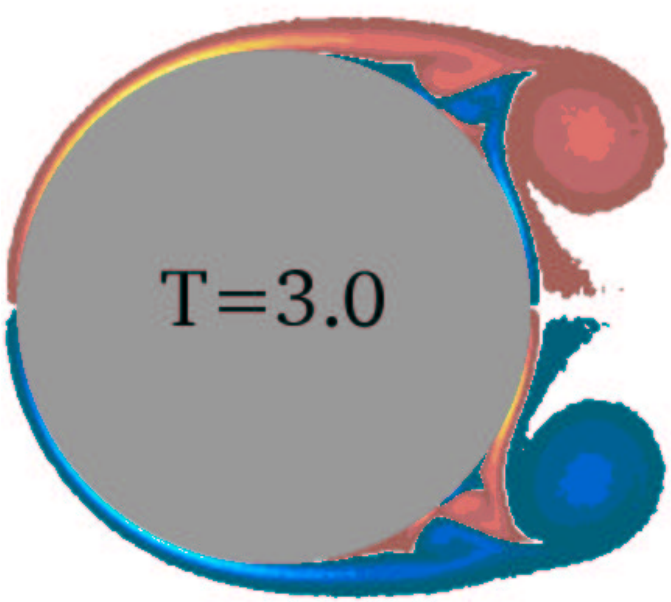
```
def data2Blob(blb_data, bm):
    z , gam, core = blb_data['z' ], blb_data['strength' ], \
                    blb_data['core']
    nb = len(z)
    bf = vebtifs .BlobFactory()
    code = """
    for (long i=0; i<nb; ++i) {
        bm->addElement(bf->create(z[i], gam[i], core[i]));
    }
    """
    weave.inline(code, ['bm', 'bf' , 'z' , 'gam', 'core' , 'nb'],
                 headers=['"blob.H"'])
```

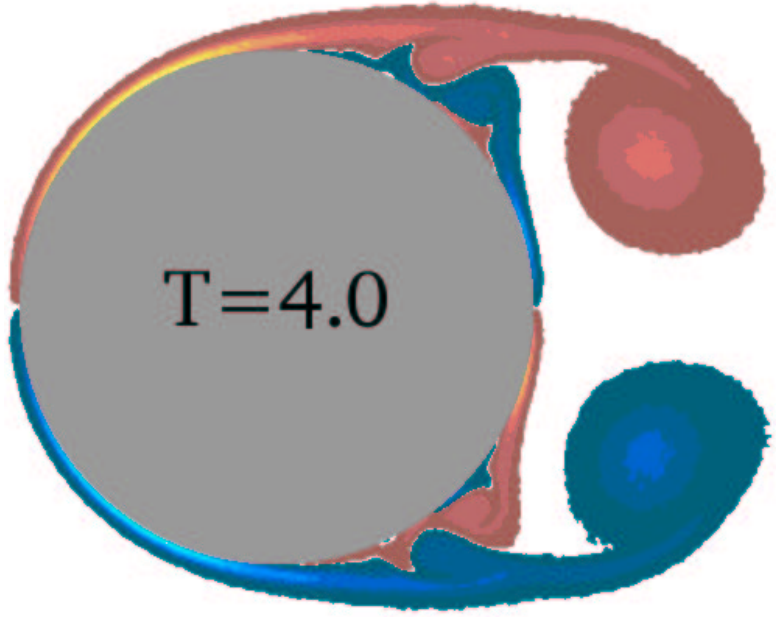
Results

- Data moves from C++ to Python and back seamlessly and efficiently
- Final code: ≈ 570 lines (≈ 500 without weave), 7 command line options
- All Python code
- Performance is excellent
- 3-4 days of effort from idea to production code
- This helped produce simulations of unprecedented resolution

Results ...

Results ...





Conclusion

- Python is of immense benefit to anyone pursuing scientific computing
- Easy to learn and very versatile; almost every aspect of scientific computing can be handled
- Relatively easy to interface to C/C++/Fortran
- IPython, Numeric, scipy, gracePlot, MayaVi make interactive, exploration and analysis a pleasure
- Weave helps accelerate slow code