Cornell University
Center for Advanced Computing

# Computational Steering

Nate Woody
Drew Dolgert

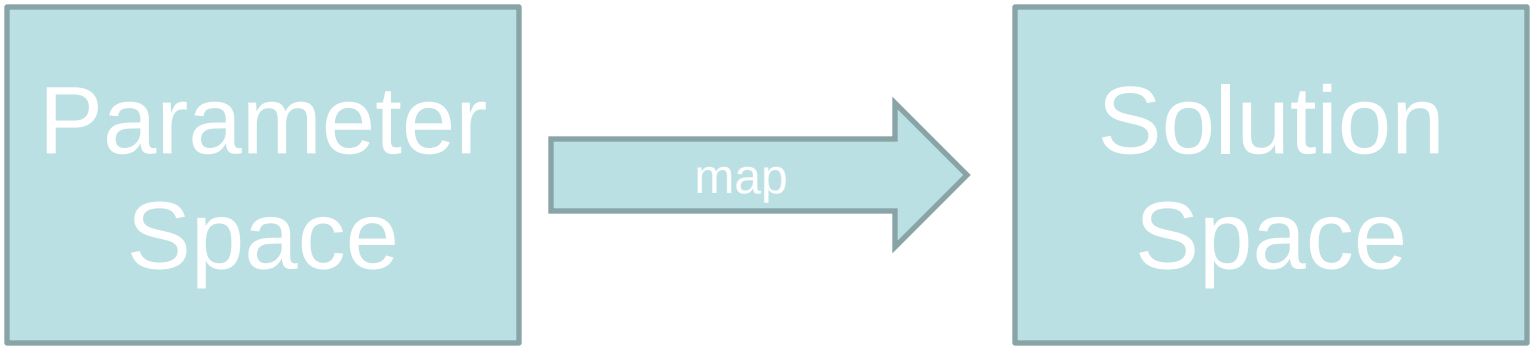
# Lab Materials

- In with the other labs.
- compsteer/simple
- compsteer/gauss_steer

# What is Computational Steering?

- Interactivity with remote HPC program.
- Save cycles by redirecting program or stopping unproductive work.

# Programs with Parameters




1. Monitor trajectory.
2. Adjust parameters for finding solution.

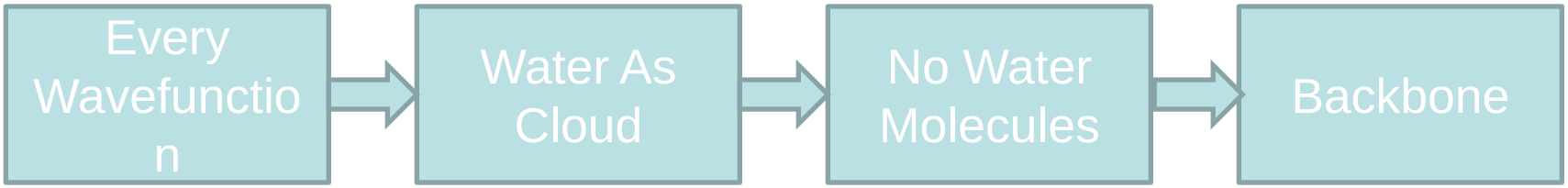
# Controlled Failures – the How

- Examine periodic output.
- Canceling a job leave output amiss.
- Steering can request a nice shutdown.

Other ways to cancel a batch job nicely?

# Fancy Checkpointing

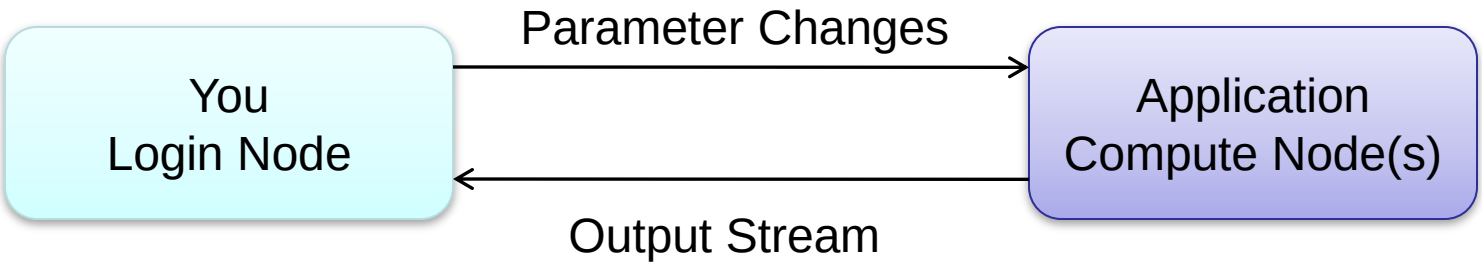
- *Checkpoint iteration* is a good place for steering.
- (Checkpointing isn't bad, either.)
- As simple as adding a read during checkpoint loop.
- Can choose what data to save.




Think *filters.*

# Interactive Processing

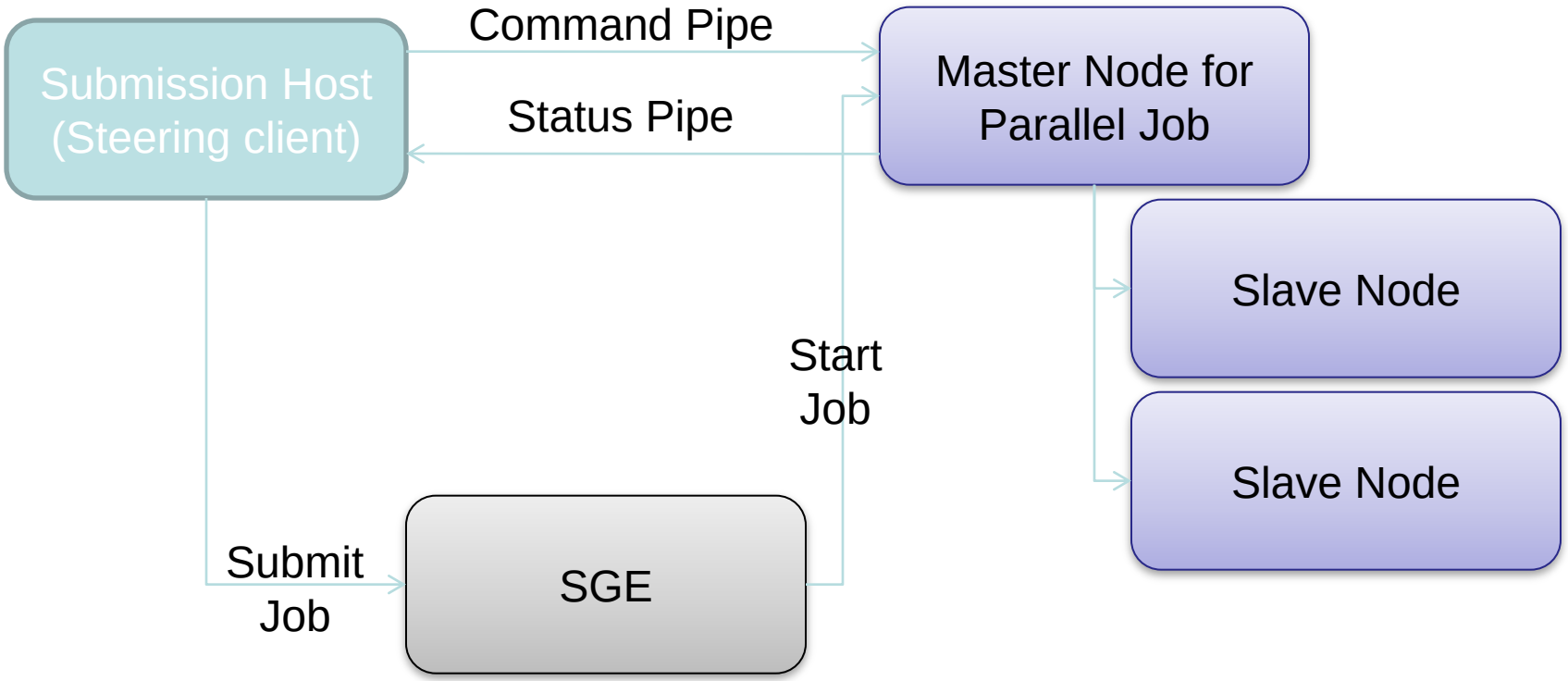
- The application changes behavior depending on parameters.
- Output streams of application become visualization on-the-fly.



# Architectural Approaches

- DIY – files, sockets, redis
- General purpose packages – Reality Grid, Cactus
- Application-specific – within VisIt, VisTrails

# Basic Architecture



# DIY Pieces

- Some protocol to talk to the job on the cluster.
- A way for the application to advertise what's changeable.
- Something in the app's mainloop to accept new parameters.

# Steering an Application

- Begin!
- Application writes a *properties file.*

```
def createSteerFile():
    fid = open(STEER_FILE, 'w')
    steerFile = Properties()
    steerFile['stop'] = "0"
    steerFile.store(fid)
    fid.close()
    return os.path.getmtime(STEER_FILE)
```

# Read the Properties

- Check modify time and read it.

```
def checkSteerFile(steer_mod_time):
    modTime = os.path.getmtime(STEER_FILE)
    if (modTime != steer_mod_time):
            return True
    else:
            return False
```

```
def readSteerFile():
    fid = open(STEER_FILE,'r')
    steerFile = Properties()
    steerFile.load(fid)
    fid.close()
    return steerFile,os.path.getmtime(STEER_FILE)
```

```
def run():
        modtime = createSteerFile()
        while 1:
        #do work
        time.sleep(5)
        if checkSteerFile(modtime):
                        print "Got New Steerage!"
                        p,modtime = readSteerFile()
                        if p['stop'] == "1":
                        sys.exit(0)
        else:
                        print "No Steerage."
```

# Interaction

- We have feed in.
- Easily extensible.
- No output, though.
- What format for output?
  - Charts and graphs already made.
  - Preprocessed, to png, jpeg.
  - Movies.

# Interacting with an Application

- Add a couple of output functions.

```
def createOutFile():
    fid = open(OUT_FILE,'w')
    fid.write("date\tx\ty")
    fid.close()
```

```
def writeOutFile(vals):
    n= datetime.datetime.today()
    fid = open(OUT_FILE,'a')
    fid.write("%s\t%s" % (n, vals) )
    fid.close()
```

- And a hook in main loop to write.

```
def run():
while 1:
        #do work
        writeOutFile("%s\t%s\n" % (count, val) )
        if checkSteerFile(modtime)
            ...
```

# Gnuplot Goodness

- In this trivial example, we outputted into something gnu plot likes. Then we can just periodically "replot" on the headnode to watch the app proceed and save the graph if we would like:

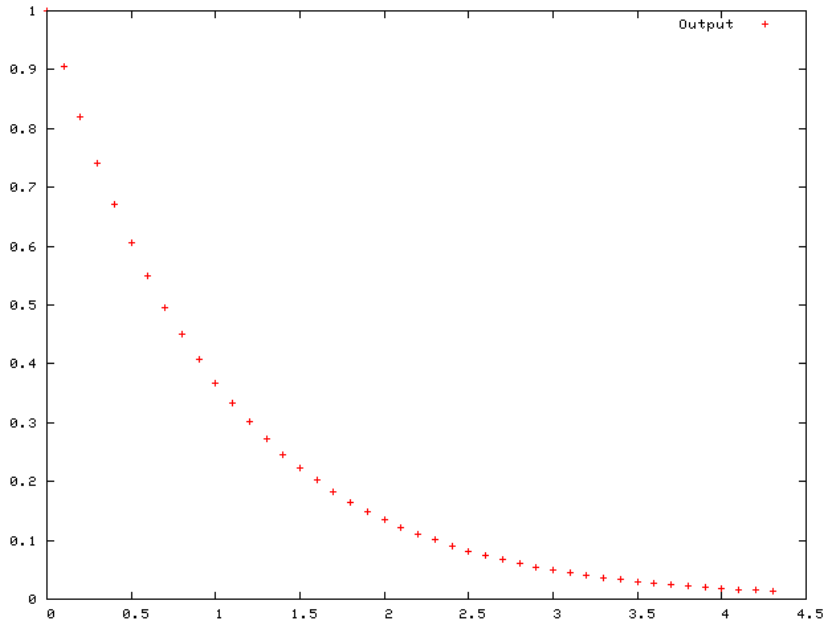
**gnuplot> set terminal png**

**gnuplot> set output 'decay.png'**

**gnuplot> plot "App_nojob.out"**

**using 3:4 title 'Output'**

**gnuplot> exit**




Other examples of fast and stupid?

# An Actual Parameter

- Add something called *step.*

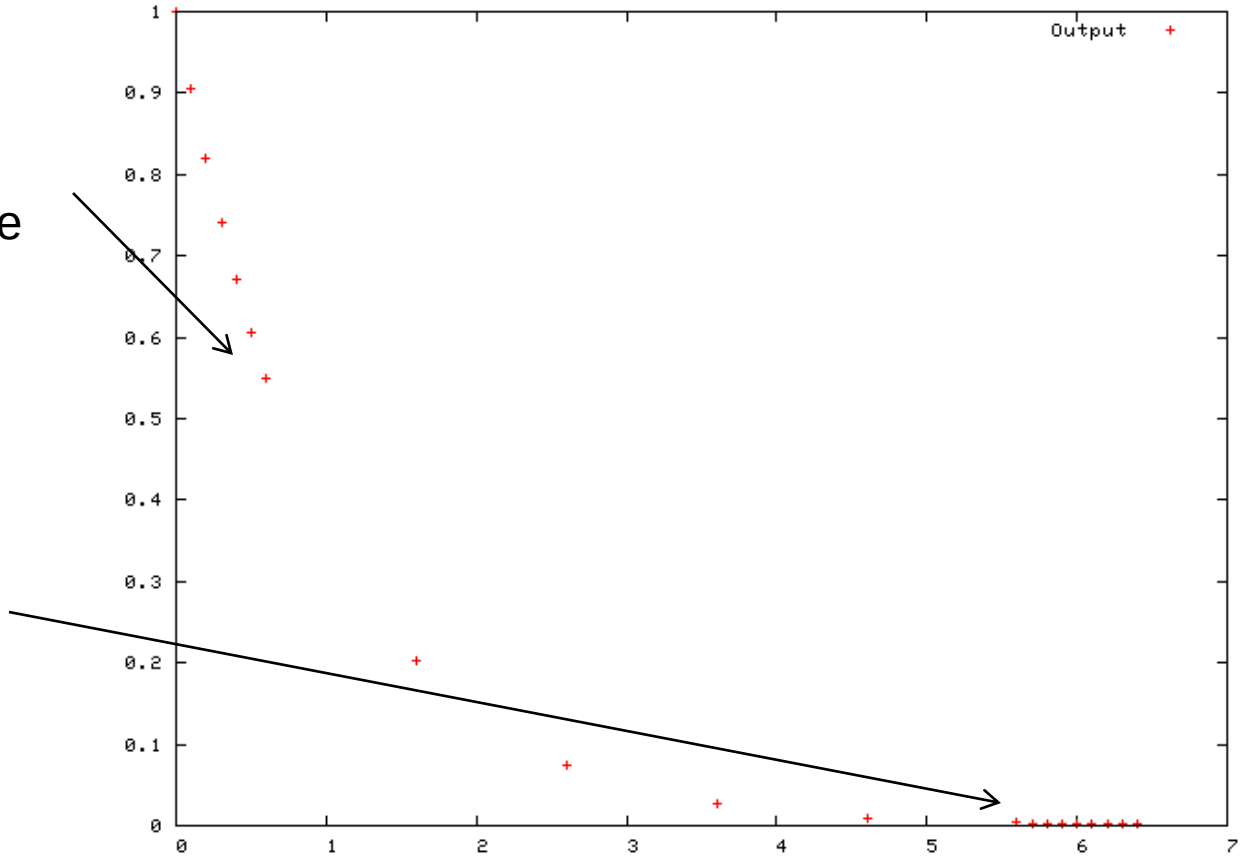
```
def createSteerFile():
    fid = open(STEER_FILE,'w')
    steerFile = Properties()
    steerFile['stop'] = "0"
    steerFile['step'] = "0.1"
    steerFile.store(fid)
    fid.close()
    return os.path.getmtime(STEER_FILE)
```

- This will be in .steer file. Affects x-axis movement.

# Step Size Problem

The step size is too short in the beginning, we notice and increase it.

At some point, we decrease the step size again.



# Interacting with an Application

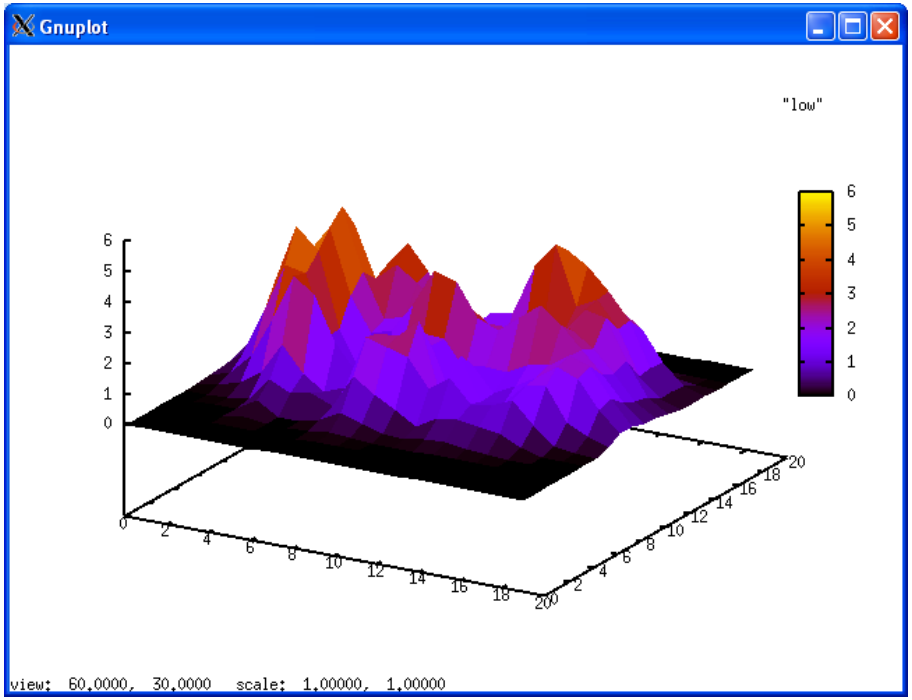
- This toy example demonstrates the principle of steering an application and the last example hints at some powerful things that can be done.
- A key example of this is to control the actual output of the program. The toy example showed how to affect the program which was reflected in the output. Another thing to do is to increase or decrease the amount of output at each step.
- Collecting all the data for all the timestep in a simulation, may not always be important, but it may be important for understanding problems or unexpected results. Steering allows you the ability to toggle how to the output of your application, so you don't have "drink from the firehose" in order to look at your simulation.

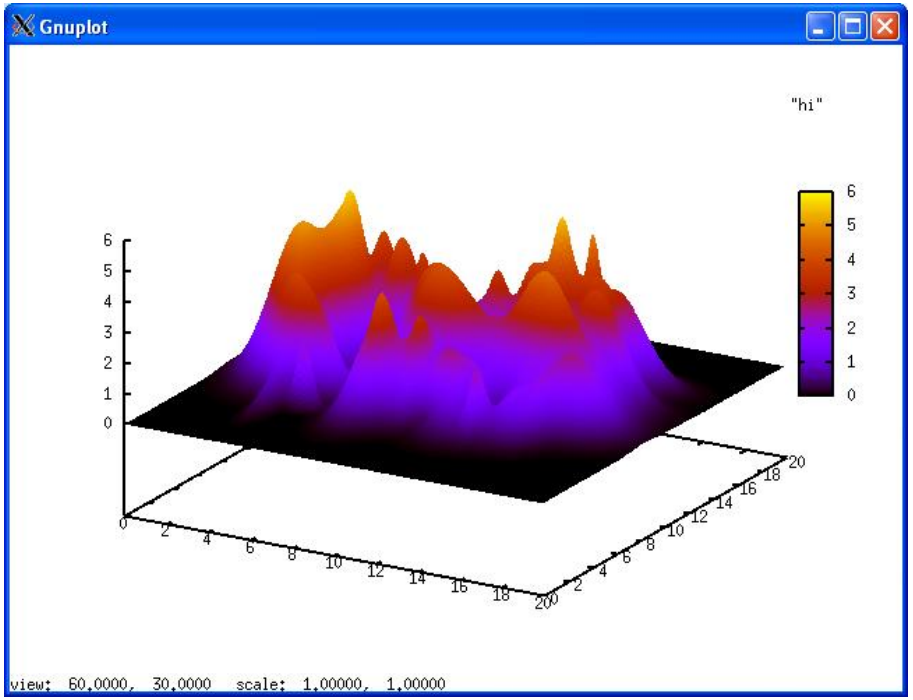
# Resolution / Drill Down Example

- Find the maximum of a 2D surface.
- Simulate surface with mixture of Gaussians.
- Implement drivable grid search.
- Parameters
  - Xcenter and ycenter – of the area we search
  - Extent – how far in all directions
  - Step – fineness of our grid

# Example Grid Search Data

- Naïve algorithm will repeatedly iterate through full grid with increasing resolution.



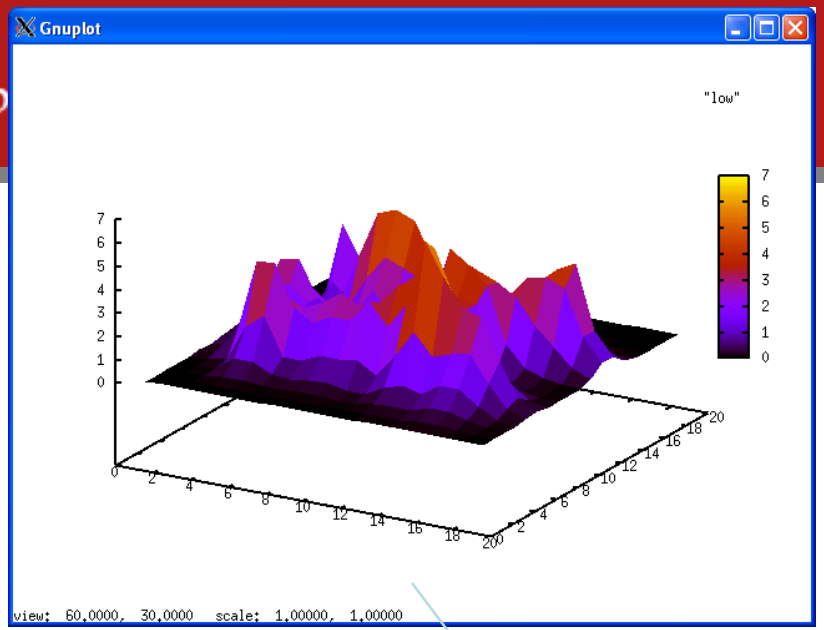



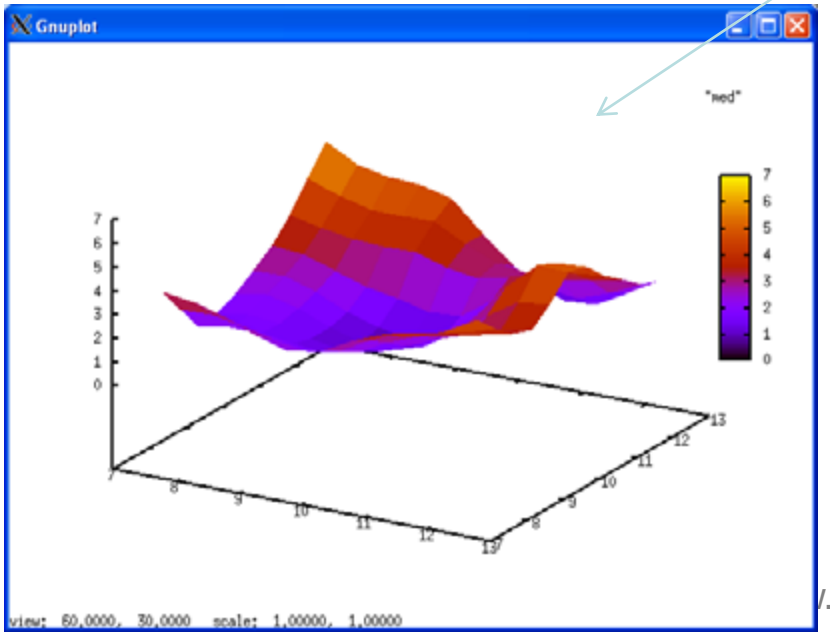
# Add to Steer File

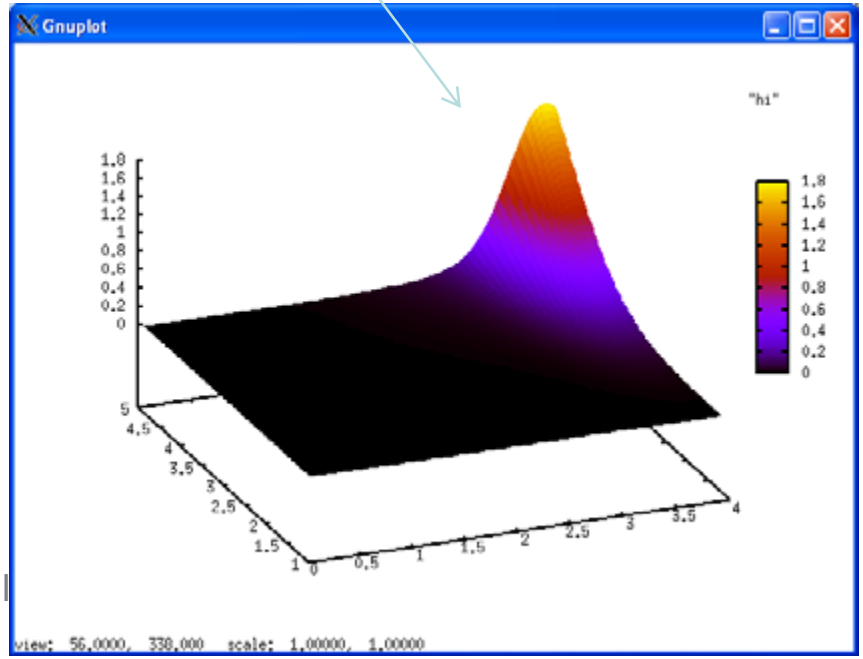
```
def createSteerFile():
    fid = open(STEER_FILE,'w')
    steerFile = Properties()
    steerFile['stop'] = "0“
    steerFile[‘xcenter’] = “10”
    steerFile[‘ycenter’] = “10”
    steerFile[‘res’] = “0.1”
    steerFile[‘step’] = “0.1”
    steerFile.store(fid)
    fid.close()
    return os.path.getmtime(STEER_FILE)
```

# Active Steering

Starting with the initial low resolution search of the entire surface, we can look at areas of interest.



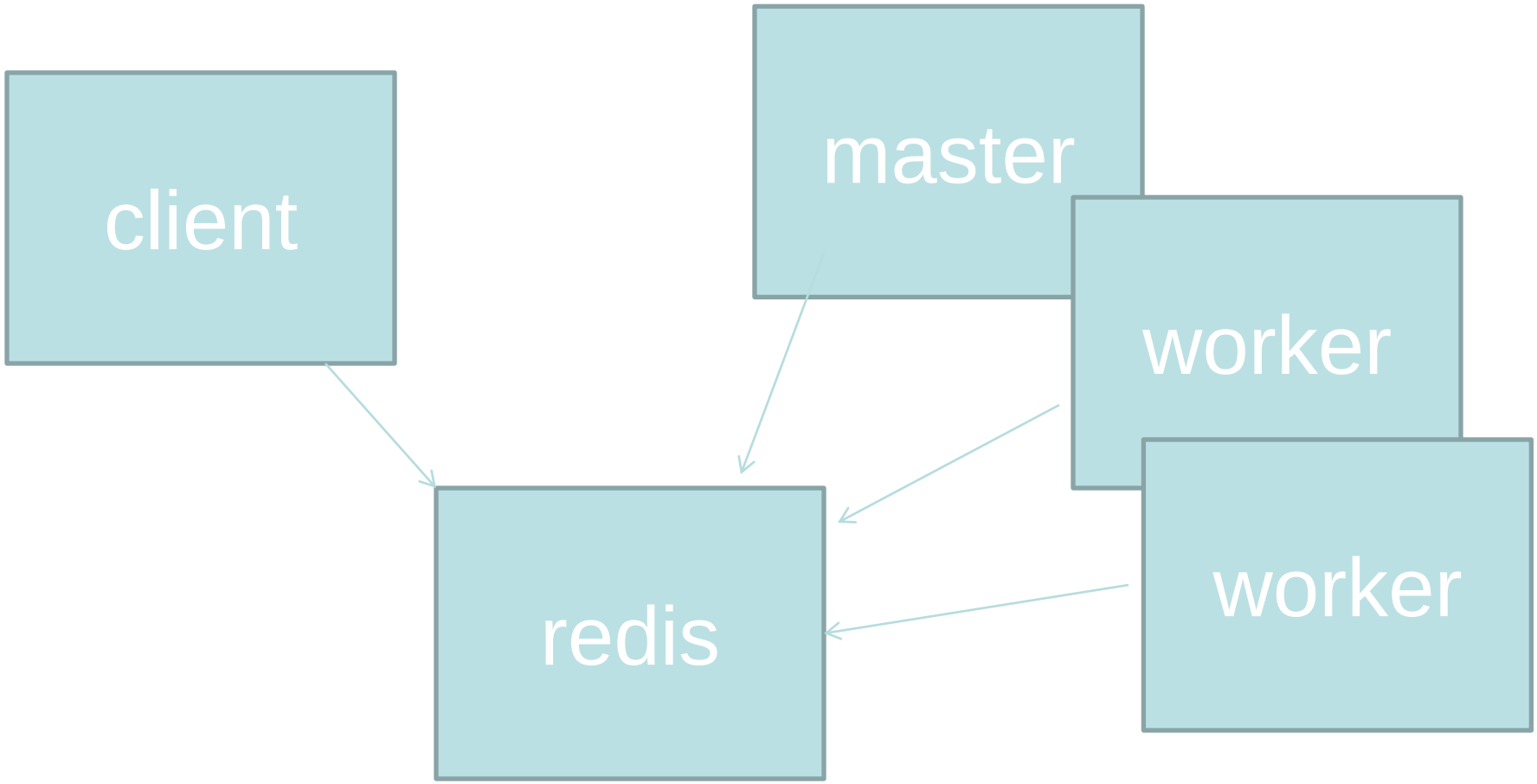
# Things to Change about Output

- Particle tracking simulation data
- Change
  - How often
  - Which subset of the data
  - What properties we write
- A Filter Pattern, or an Observer Pattern, or both.

# Redis

- Key-value store with some *atomic* operations.
- Compiles quickly and easily.
- Can run one for each job. Lives in-memory.
- API in just about every language (sorry Fortran).
- We could *totally* make Fortran happen.

# Redis as Place to Stick Messages



# Reality Grid Steering Toolkit

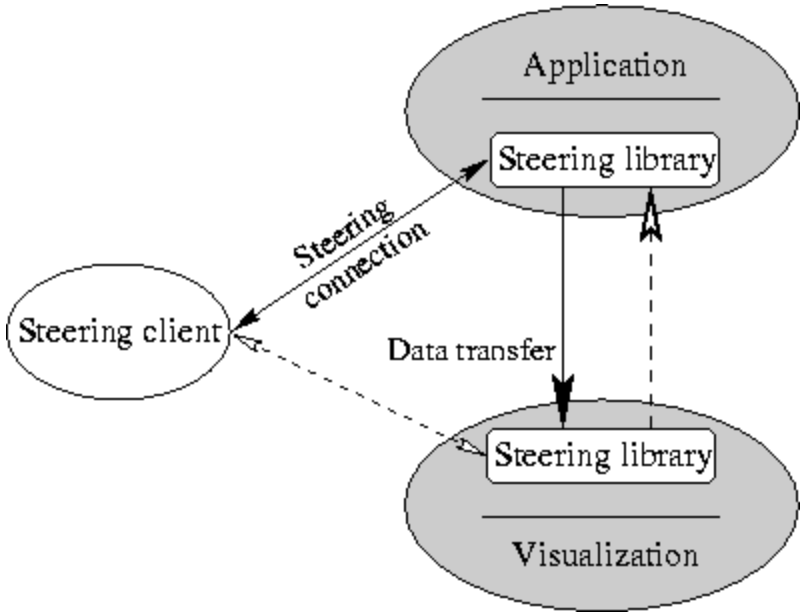
- RealityGrid is a large-ish EU project for developing grid middleware and applications to ease the use of HPC resource.

*This refers to an ambitious and exciting global effort to develop an environment in which individual users can access computers, databases and experimental facilities simply and transparently, without having to consider where those facilities are located. Using grid technology to closely couple high throughput experimentation and visualisation, RealityGrid has led the way in showing how close we are to realising this new computing paradigm today.*

*[http://www.realitygrid.org]*

# RealityGrid Architecture

- C, C++, and Fortran wrappers to communication and I/O functionality.
- Allows the steering connection via file or sockets (SOAP).
- Visualization is basically a data-sink that must display the data appropriately.
- Is based on the process of having existing code that you would like to "instrument" to add steering capability to.



# Adding RealityGrid

- Library with an API, so add calls and link
- Toolkit provides I/O
- App registers steerable parameters at start-up.
- App checks for client commands.
- Client searches for apps.
- Visualization part reads I/O streams from app.
- See mini_steerer.c

# Cactus Code

- “Problem solving environment”
- Functionality in *Thorns.*
- You assemble thorns, and it manages communication. Data arrays are registered with Cactus, so it knows where to look.
- Numerical Relativity is original domain.

# Experience with Cactus

- Seamless with HDF and other file formats, web browser visualization, lots else.
- Steep learning curve. Non-trivial to identify and assemble thorns.
- Not to add to existing code, maybe to start a new code.