Computational Steering

Nate Woody

Lab Materials

- I’ve placed some sample code in ~train100 that performs the operations that I’ll demonstrate during this talk. We’ll walk through how to use it later on, but let’s go ahead and grab it first.
  - ssh trainxx@ranger.tacc.utexas.edu
- Move the zip file into your directory
  - cp ~train100/compsteer.zip ~/compsteer.zip
- Unpack the zip
  - unzip compsteer.zip
- You should then have the example that we’ll go through:
  - py_steer/simple/rev0/
  - Py_steer/gauss_steer/

What is computational steering?

- Generally, computational steering can be thought of as a method (or set of methods) for providing interactivity with an HPC program that is running remotely.
- This is used to:
  - Alter parameters in a running program
  - Receive real-time information from an application
  - Visualize the progress of an application
- The principal idea is to save cycles by directing an application toward more productive areas of work or at a minimum stopping unproductive work as early as possible.

Types of Computational Steering

- We can think of a computer program as a map between parameter space and a solution space. Given a set of inputs the application moves through a trajectory in the solution space, and we are able to interrogate and visualize this trajectory.
- Given this framework, we can group computational steering efforts into two types:
  - Type 1: We are primarily interested in exploring the parameter space. By monitoring the trajectory, we can short-circuit the examination of input parameters that appear to be leading away from our desired state or revise our search strategy to more finely search an interesting area of space.
  - Type 2: We are looking for a particular solution and want to be able to adjust parameters on the fly, to ensure that we arrive at the desired result in the most efficient manner.

Basic examples of steering

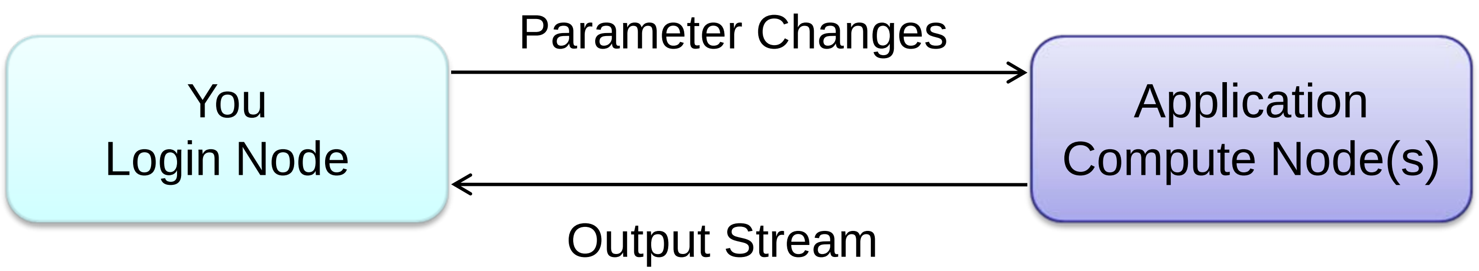
- Fail Fast (but gracefully)
  - If you're application produces periodic output, that output can probably be examined to determine how the application is doing. If you determine that you're simulation has run astray, you could just cancel the job to prevent further usage of resource.
  - Executing a cancel job command will likely leave your application in an unknown state, unless you have included some sort of a handler to handle system-specific interupts that the system may pass. The periodic output your application is producing may be left in a strange state and you aren't likely to get final output from your application.
  - Implementing the steering that handles a stop command, will allow you to terminate a job but still produce any desired output. Data output can still be preserved for later examination of what went wrong w/o waiting for the job to completely finish.

Basic examples of steering

- Fancy checkpointing
  - Steering can easily be implemented around a checkpoint iteration and can be safely thought of as an upgraded form of checkpointing.
  - Checkpointing provides output that can be used to re-start your application given some sort of extreme failure.
  - Steering provides the ability to do more than just serialize current data structures at each of iteration.   You can request a stop, or adjust parameters, so that the checkpointing involves a read as well as simply a dump.
  - Steering can also be used to tweak the amount or type of data that the application produces at each iteration. This allows you to filter the data that is produced so that you can examine the application in as much detail as you would like without producing all output all the time.

Basic examples of steering

- Interactive processing
  - The actual power of steering comes from applications that process data according to a set of parameters that are set at input. Steering allows you to alter those parameters to affect the ongoing simulation.
  - Output data streams from the simulation are processed into "real-time" visualization of the applications working. This real-time view of the operation of the application allows rapid intervention by the user to affect the application.



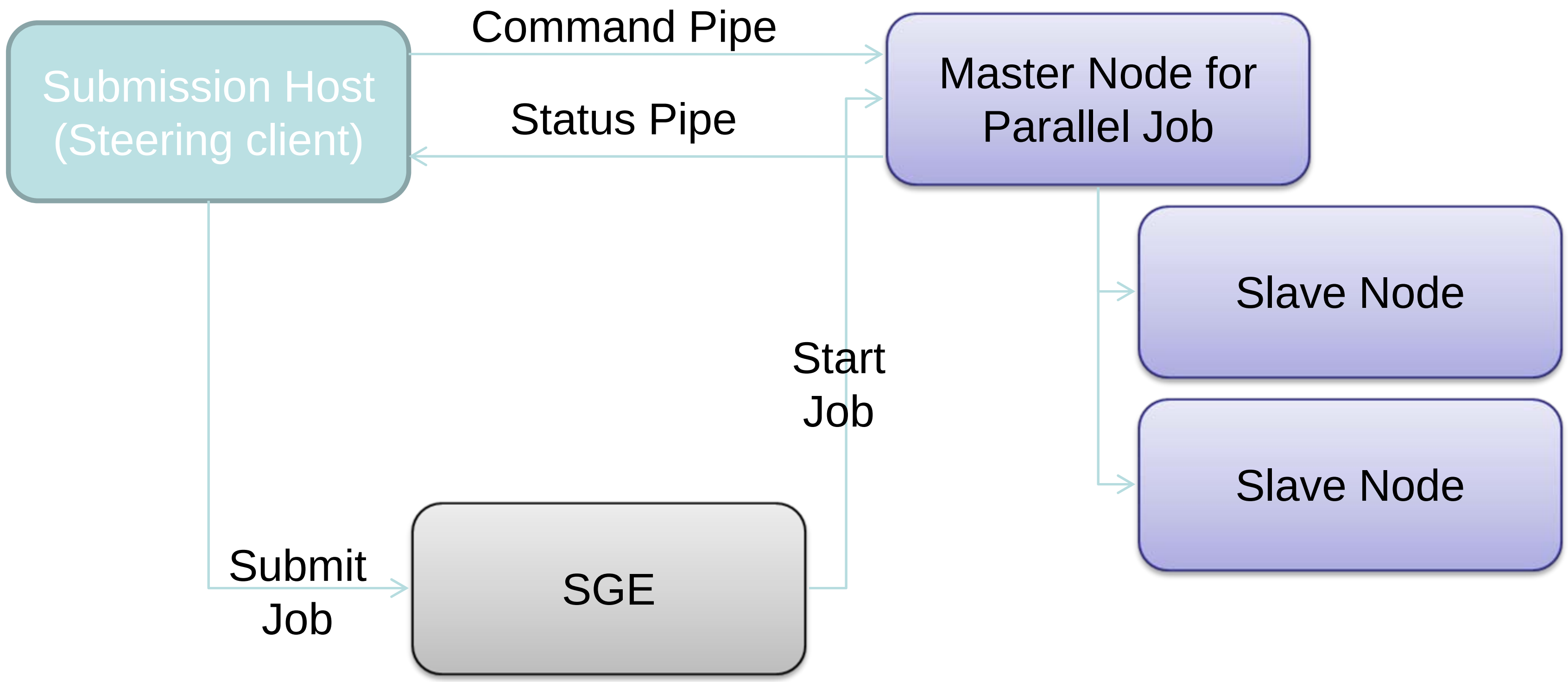
Approaches to steering

- DIY – it's simple to add basic computational steering to an application by taking advantage of a shared network file system.

- General Purpose Packages – Packages/libraries implement basic communication structure that allows a client to communicate with a running program and for the running program to provide intermediate results to a (graphical or otherwise) client.

- Application specific packages – Packages that provide interactive components that a user may stitch together to perform the desired work.

Basic Steering architecture



Making an application steerable

- In the next few slides, we'll demonstrate the functions neccesary to add simple steering functionality to an existing app. We'll do this using simple Python functionality, but you can use any language you feel comfortable with.
- We need the following functionality:
  - A means of communicating between the running application and you (or at least the steering client).
  - A means for the running application to identify steerable parameters that a steering client can change.
  - Added application functionality to parse and respond to steering commands from the steering client.

Lab Materials

- I encourage you to follow along and play with the provided code as it's a little more informative about what is actually happening.
- ~/py_steer/simple/rev0 is the starting point and contains a fully functional example that will allow you to run the steered application.
- There are two ways to run this code,
  - Locally – which we'll be doing, just create to login windows to ranger, one will be the "compute-node" and one will be the "login-node". The compute node will be running the application, which you will be able to control with the login-node.
  - Batch – Modify the provided steer.sh file to create a Ranger batch file that will submit a job that kicks off the application (app.py). This is how you would actually do this in practice, but means we have to wait in the queue.

Steering an application

- Let’s look at how easy it is to add a simple steering mechanism to an existing application using a shared file system. A means of communcation then is simple a file.
- We’ll start out by the application setting up 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)
```

Steering an application

- Next we create functions to check the modify time of the file and to read in the file:

```
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)
```

Steering an application

- Finally, we add these to our main loop and add handling the steering parameters

```
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."
```

Interacting with an application

- This example is a trivial version of being able to feed input into an application.
- It clearly extends to any parameters that you could wish to provide and requires a minimal amount of work.
- However, the demonstrated application doesn't provide any output that the user could use to determine HOW to steer the client. We can use the same principle of just supplying output to a file from the app and managing that from the client.
- Some care should be taken to output the data into a format that is easy to examine/display back at the headnode.

Interacting with an application

- Add a couple of functions to create and write to an output file

```
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()
```

- Add a hook in the working loop to call the write Function

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

Interacting with an application

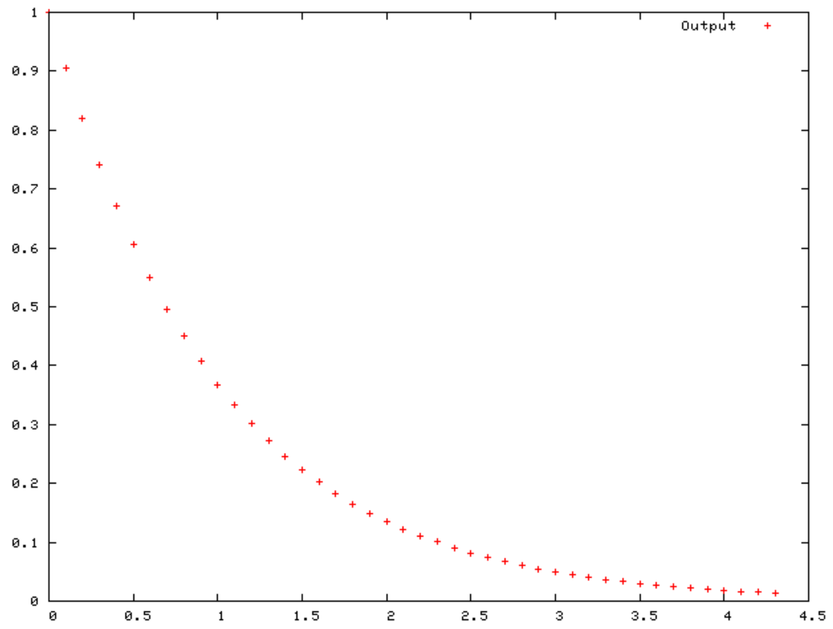
- In this trivial example, we outputted into something gnu plot likes. Then we can just periodically “replot” on the headnode to watch the app proceed and save the graph if we would like:

gnuplot> set terminal png

gnuplot> set output ‘decay.png’

gnuplot> plot “App_nojob.out” using 3:4 title ‘Output’

gnuplot> exit



Interacting with an application

- Now we have all the pieces to actually do something interesting. Let’s add a steerable parameters called “step”.

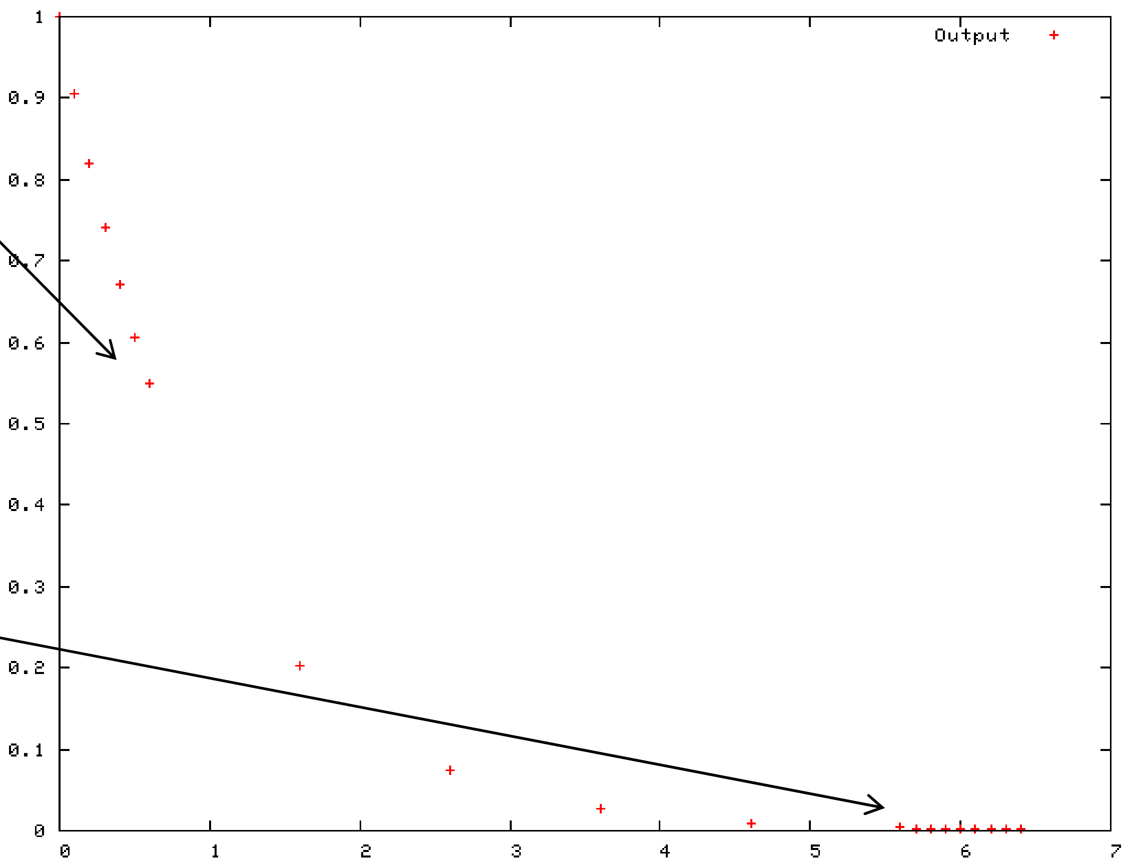
```
def createSteerFile():
    fid = open(STEER_FILE,'w')
    steerFile = Properties()
    steerFile['stop'] = "0"
    steerFile[‘step’] = “0.1”
    steerFile.store(fid)
    fid.close()
    return os.path.getmtime(STEER_FILE)
```

- This will now appear in the .steer file, and we can adjust it from there. This parameter affects how fast we move along the x-axis.

Interacting with an application

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

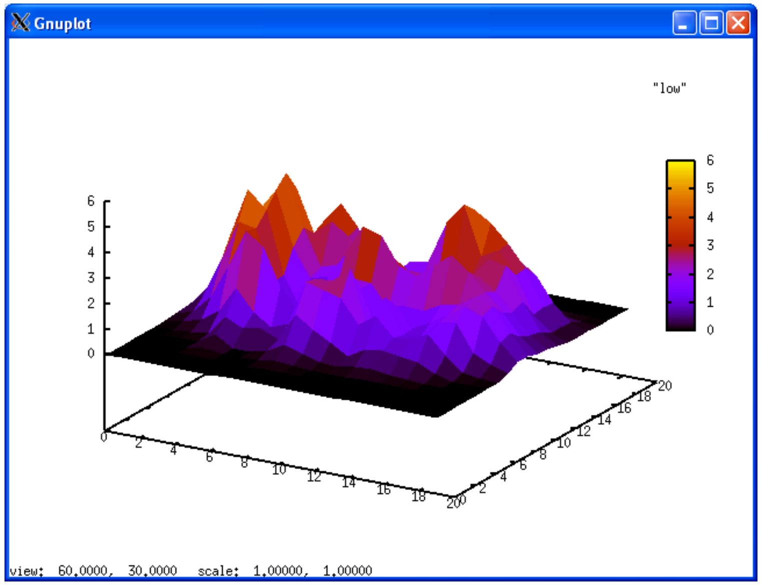
- This toy example demonstrates the principle of steering an application and the last example hints at some powerful things that can be done.
- A key example of this is to control the actual output of the program. The toy example showed how to affect the program which was reflected in the output. Another 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.

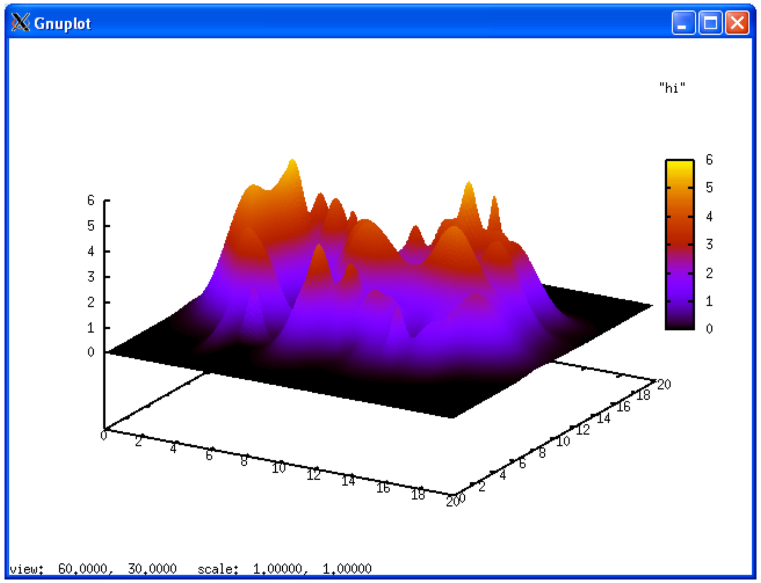
Resolution/Drill Down Example

- In order to play, we can make a toy problem, where we'll try and find the maximum of a 2D surface.
- We can simulate a surface with a mixture of gaussians, which makes a nice bumpy random surface to look at and is easily amenable to a number of different algorithms.
- We'll demonstrate a simple drive-able grid search, where we can specify the location and detail of the area that we will search.
- Parameters:
  - Xcenter and ycenter = the center of the area that we'll search
  - Extent = how far to extend our search in all directions
  - Step = the fineness of our grid

Example Grid Search Data

- Naïve algorithm that will repeatedly iterate through full grid with increasing resolution until the discovered maximum stop increasing (with some threshold).





Interacting with an application

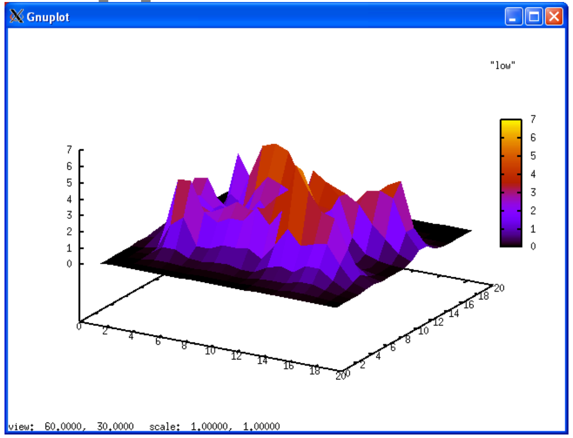
- Now we have all the pieces to actually do something interesting. Let’s add some steerable parameters.

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

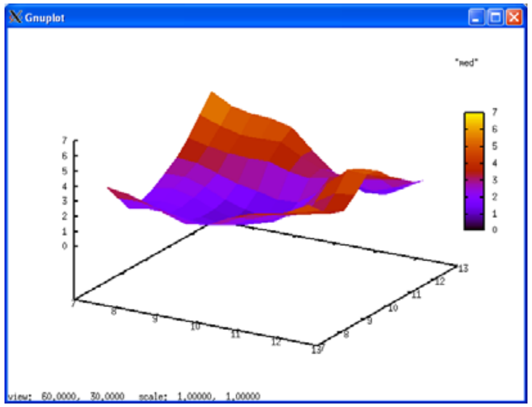
- This will now appear in the .steer file when the job starts and we will be able to drive how the application operates.

Interacting with the application

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

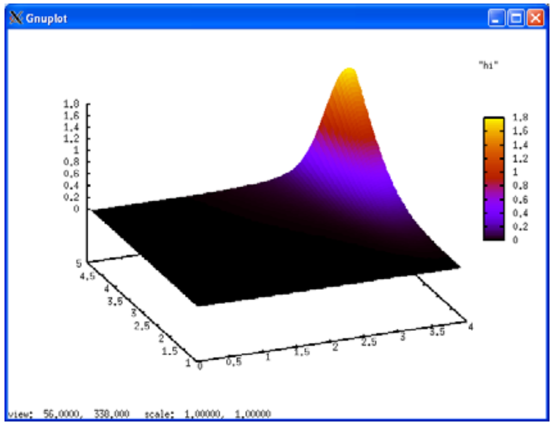


```
view: 60.0000, 30.0000   scale: 1.00000, 1.00000
```



```
view: 60.0000, 30.0000   scale: 1.00000, 1.00000
```

Drill down to specific regions



```
view: 56.0000, 338.000   scale: 1.00000, 1.00000
```

Interacting with an application

- Consider a simulation that produces particle tracking simulation data through time steps. Most any sort of dynamics code will do.
- Invariably, this code already has the functionality to dump out coordinate/property data for any given timestep. We can utilize this in a couple of different ways.
- Add a steerable “steptimeoutput” parameter that would change how often the entire coordinate data would be outputted to a file. You would then need to visualize this data (again, code which may already exist).
- Another alternative is too look at a subset of that data. Add a steerable parameter that affects how many particles, our the edge of a boundary box that you would like output. You can grow or shrink the boundary box to ensure that conditions are satisfactory w/o needing to save all the data.

Steering Toolkits

- The previous examples were meant to demonstrate how steering can help you and to make sure that it was clear that there is no magic involved. Steering does not need to complicated and can easily be engineered into existing code with a potentially high level of benefit.
- However, toolkits do exist that provide a lot of functionality that would be more difficult to code on your own. These toolkits range from libraries that implement nice versions of what I’ve just shown here to application-area specific codes that are designed to be run as shown.
- I’ll highlight two of these applications that are well-developed and may serve you’re needs.

Reality Grid Steering Toolkit

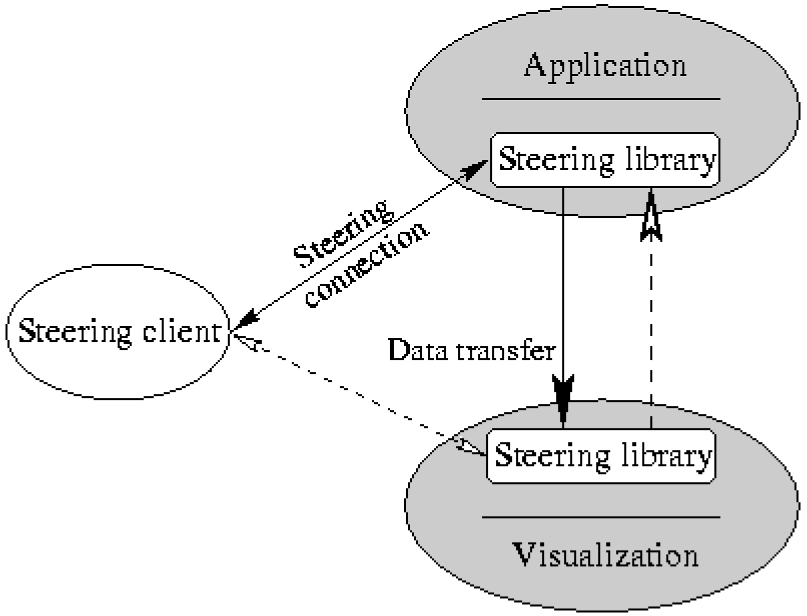
- RealityGrid is a large-ish EU project for developing grid middleware and applications to ease the use of HPC resource.

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 Steering Toolkit

- 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.



RealityGrid Steering Toolkit

- Adding RealityGrid to an existing project
  - The idea of this toolkit is to integrate the steering library into an existing application to gain steering ability.
  - The toolkit is a library providing communication and I/O functionality accessible by adding the appropriate function calls to your application and re-compiling against the appropriate libs.
  - The application (the MPI code you run on the compute node), will need to register the steerable parameters at start-up and then periodically check to see if a client has registered and passed commands.
  - The client (a new piece of code you'll have to write), needs to look for steerable applications and then pass commands to it.
  - The visualization component (which could be the client), needs to accept/read I/O streams from the application.

RealityGrid Steering Toolkit - App

Download at:
http://www.rcs.manchester.ac.uk/research/realitygrid/downloads?action=AttachFile&do=get&target=steer_lib-2.0.tgz

See mini_app.c

RealityGrid Steering Toolkit - client

Download at:

http://www.rcs.manchester.ac.uk/research/realitygrid/downloads?action=AttachFile&do=get&target=steer_lib-2.0.tgz

See mini_steerer.c

Cactus Code

- Cactus describes itself as a problem solving environment and is a fully-developed set of code.
- Cactus is organized where functionality is arranged into objects called ‘thorns’. The idea is that you can assemble the thorns that you need into an application (as well as write new thorns for functionality not provided). Cactus then manages the communication between the thorns and automatically exposes steerable parameters and I/O pipes for visualization.
- Cactus is principally written for solving Numerical Relativity problems and the available thorns reflect this. For people doing this kind of research, Cactus is absolutely the way to go.

Cactus Code

- Another nice feature of Cactus is that integrates seamlessly with a number of visualization utilities. Most toolkits provide an I/O stream that you can code to, or at most utilities to create common file formats (HDF, etc). Cactus provides much better integration directly into a number of visualization tools, and allows interactivity all the way back into visualization.
- The learning curve for Cactus is rather steep. If the functionality that you need is not provided by available thorns, you should expect to spend some time learning the Cactus interface and how things tie together. It may also not be trivial to identify and assemble the thorns that you do need.
- Cactus is much more of a package and less of a toolkit to add to existing code. This is good and bad. Vastly more functionality is provided by Cactus than just about anything else there, but there is a complexity price to be paid.

Steering Summary

- The goal of this talk was to introduce and discuss what computational steering is and how it can be used.
- It can be trivial to introduce very lightweight steering components into your application that can provide real benefits for understanding what your application is doing. It is simply not acceptable to have a day+ long job produce bad results that COULD be detected in the first 4 hours of the job.
- The key aspect of steering for data analysis is to be able to observe what your application is doing in near real-time instead of operating completely as a batch operation.