

# Building Blocks for the Rapid Development of Parallel Simulation

Peter Stoltz, Tech-X Corporation



# We are working to provide NASA researchers with tools for interactive parallel computing

- Project is an SBIR project (Phase I finishing this month)
- We are extending an existing tool for interactive computing (iPython) to work with parallel computers (multicore and distributed)
- In the Phase I, we automated the use of arrays and demonstrated parallel FFT (both multicore and distributed) within iPython
- In Phase II, we plan a suite of interactive tools, including tools for using legacy C/Fortran code with iPython, a library of mathematical functions, and tools for performance analysis

# Traditional Tools for Scientific Computing



# Compiled Languages

- C/C++/Fortran are FAST for computers, SLOW for you.
- Assumes that CPU time is more expensive than human time.
- Everything is low-level, you get nothing for free:
  - Only primitive data types.
  - Few built-in libraries.
  - Manual memory management: bugs and more bugs
- No interactive capabilities:
  - Endless edit/compile/execute cycles.
  - Any change means recompilation.
- Awkward access to plotting, 3D visualization, system shell

# Problems

- Lots of time spent in debugging/compiling.
- Keeps us away from the science.
- Optimization/parallelization can mean (nearly) starting over.
- Parallel code:
  - Difficult to debug, even less interactive.
  - Extremely time intensive to develop good parallel code.
  - Even more awkward access to visualization facilities.

# Interactive Computing Environments

- Matlab, Mathematica, IDL, Maple
- Extremely popular with working scientists:
  - Interactive: matches the exploratory nature of science.
  - Seamless access to data, algorithms, visualization, etc.
  - Great for algorithm development, testing, prototyping, data analysis.
- Poor performance
  - No easy route to optimization.
  - No easy route to parallelization.
- Expensive \$\$\$

# Python



# Why Python?

- Freely available (BSD license).
- Highly portable: OS X, Windows, Linux, supercomputers.
- Can be used interactively (like Matlab, Mathematica, IDL)
- Can wrap existing C/C++/Fortran codes

# Libraries: Numpy

- High level Matlab-like arrays/vector/matrices
- Linear algebra, FFTs, Random #s, Fortran Integration
- <http://numpy.scipy.org>
- Good documentation

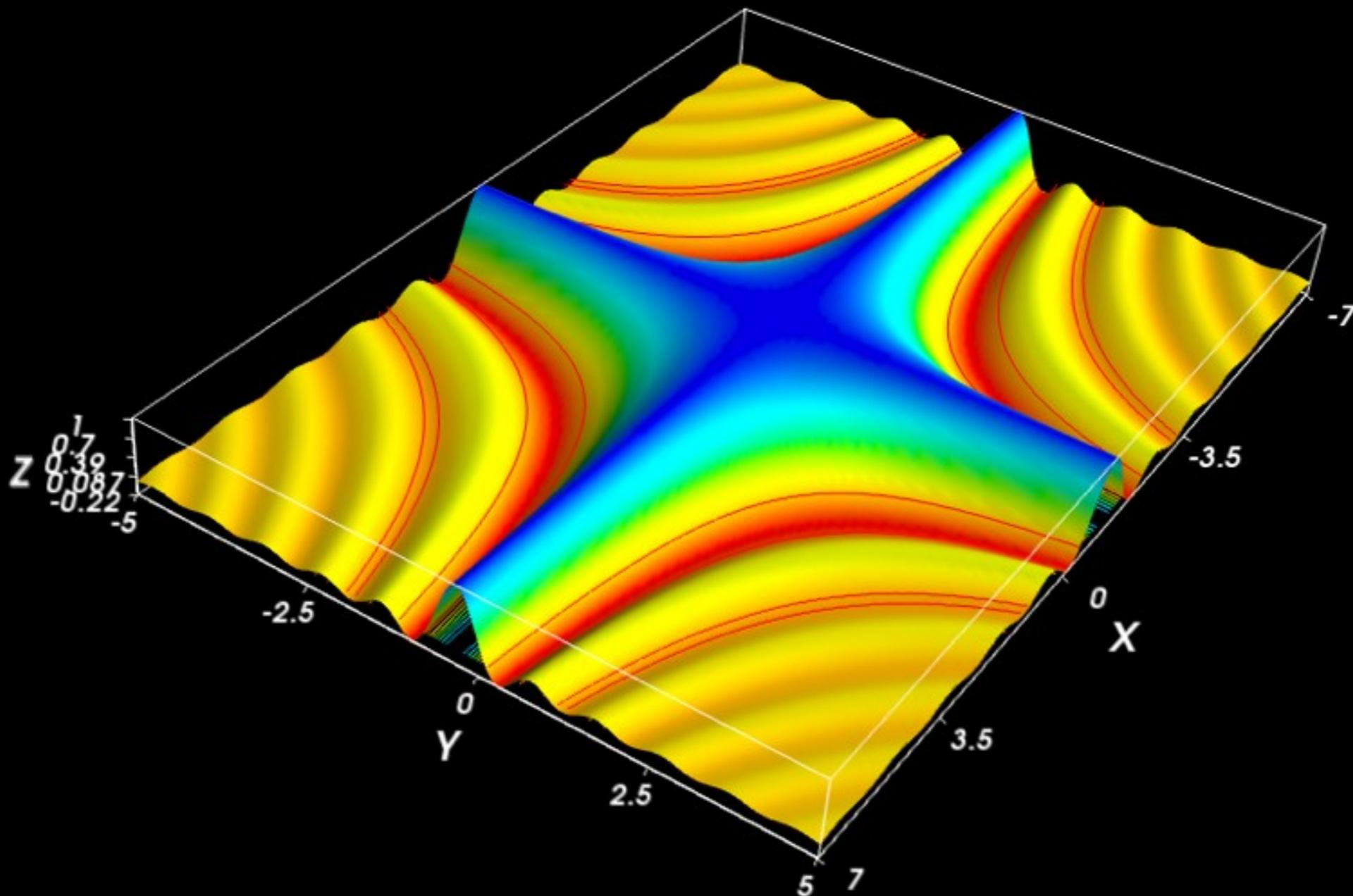
```
In [1]: a = random.rand(10,10)           # 10x10 random matrix
In [2]: evals = linalg.eigvals(a)       # compute eigenvalues
In [3]: evals.sort(); evals[0]
Out[3]: (-0.90544548812727244+0j)
In [4]: sum(evals)
Out[4]: (4.3298947969780439+0j)
```

# Libraries: Plotting

- Python bindings for major GUI toolkits (wx, qt, cocoa, tk).
- Matplotlib
  - Publication quality 2D plots.
  - Builtin LaTeX support.
  - Works with Numpy arrays.
- 3D plotting: Visit, TVTK

# Example:

```
from enthought.tvtk.tools.mlab import *
fig = figure()
def f(x, y):
    return scipy.sin(x*y)/(x*y)
x = scipy.arange(-7., 7.05, 0.1)
y = scipy.arange(-5., 5.05, 0.05)
s = SurfRegularC(x, y, f)
fig.add(s)
```



# Wrapping Compiled Code

- Scientific codes spend most of their time in a few lines of code.
- Only optimize those parts using C/C++/Fortran and write the rest in Python.
- Python has many tools for this:
  - ctypes: directly call a C shared library.
  - Pyrex/Cython: write C code in Python.
  - f2py: autogenerate python wrappers for Fortran/C.
  - Weave: Write C++ inline in your Python code.

# Examples: Weave and ctypes

Weave: write/compile/call C/C++ code inline

```
In [17]: code = 'return_val = a + b;'      # This is C code
In [18]: a = 5.0
In [19]: b = 10.0
In [20]: inline(code, ['a', 'b']) # Compile/call on the fly
<weave: compiling>
Out[20]: 15.0
```

ctypes: call existing C libraries with no wrapping

```
In [17]: libm = cdll.LoadLibrary('libm.so')
In [18]: sqrt = libm.sqrt
In [19]: sqrt.restype = c_double      # Set the return type
In [20]: print sqrt(c_double(4.0))
2.0
```

It is straightforward to pass Numpy arrays to these tools

# IPython: An Environment for Interactive Computing

<http://ipython.scipy.org>



# Overview of IPython

- Freely available (BSD license).
- Comes with every major Linux distribution.
- Goal: provide an efficient environment for exploratory and interactive scientific computing.
- All network connections are fully authenticated and encrypted
- Now iPython has distributed memory arrays! (funded through this NASA Phase I SBIR)

# Trends in Hardware

- multicpu and multicore
- Cluster and supercomputers built out of such components.
- Heterogeneous architectures with complex memory hierarchies.
- A recent Berkeley white paper suggests that 1000 cores per chip is optimal (google for “view from berkeley”).

# Trends in Software

- For the last decade, the Message Passing Interface (MPI) and C/C++/Fortran have dominated parallel computing. High performance, low productivity.
- Threads work well on multicore architectures. Low-level, error prone.
- Emerging approaches: OpenMP, Fortress, Co-Array-Fortran, MapReduce.

# The Challenge:

- Parallelism in hardware is coming
- The critical issue for us (developers/users of numerical software) is how parallelism will be expressed in software.
- Can we have both performance and productivity?
- Can we keep legacy C/C++/Fortran MPI codes?

# IPython's Architecture for Parallel Computing

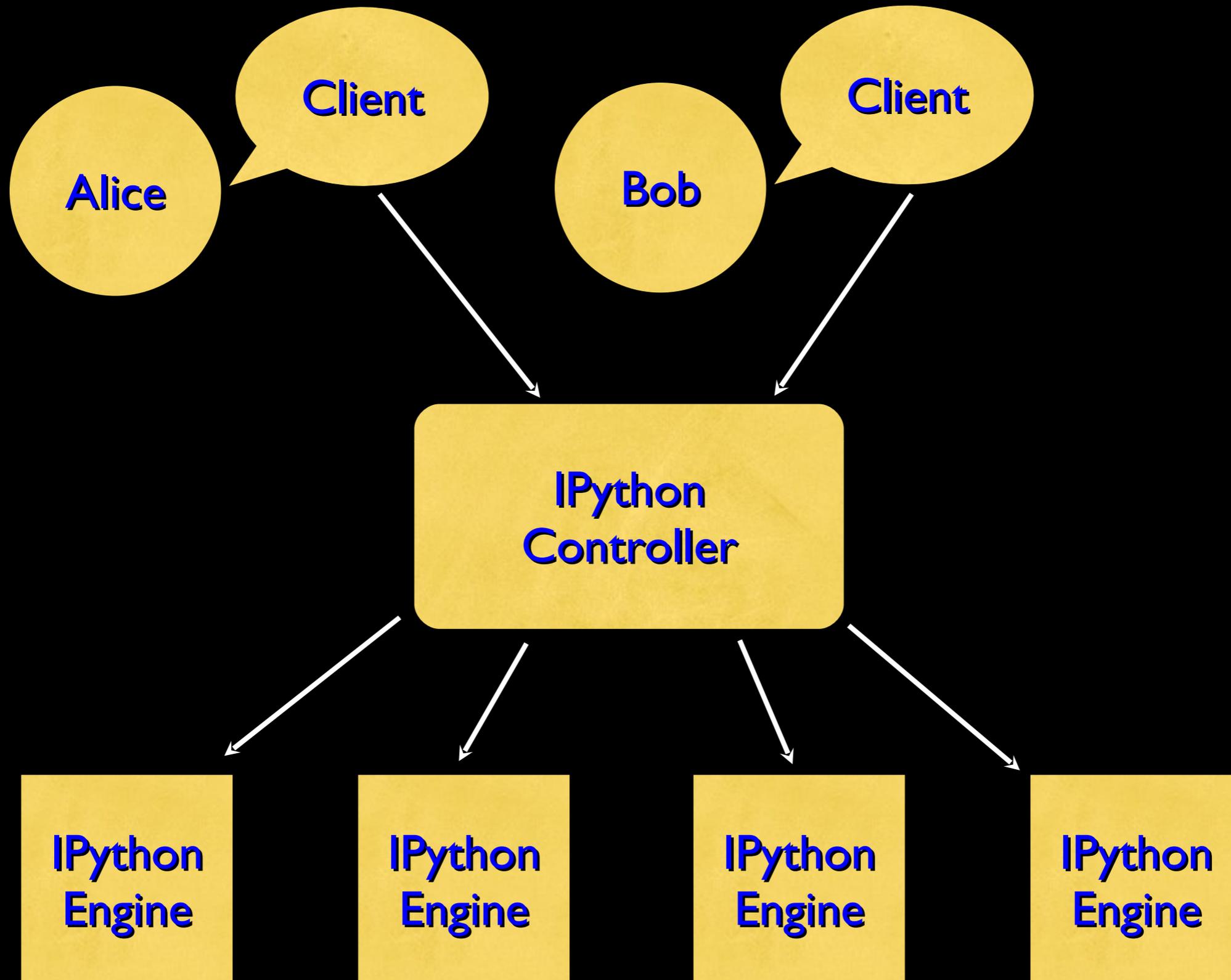
<http://ipython.scipy.org>



# Goals

- Make all stages of parallel computing *interactive*: development, debugging, testing, execution, monitoring,...
- Make parallel computing *collaborative*.
- Make use of existing tools and code
  - Integrate with threads/MPI/GPUs if appropriate.
  - Integrate legacy/compiled code and libraries.

# IPython's Architecture



# Possibilities: Multicore

- Run the Controller, Engines and a Client on a multicore laptop/desktop
- Makes it easy to take advantage of multicore CPUs
- Simple path from serial to parallel
- Same code will run on cluster/supercomputers

# Possibilities: Clusters

- Run the Controller on a head node
- Use batch system to start Engines on compute nodes. The Engines can call `MPI_Init` if desired
- Engines can run any C/C++/Fortran code (including MPI calls) that has been wrapped into Python
- Connect to the Controller from your laptop and execute parallel code on a cluster interactively

# Possibilities: Collaboration

- The system is designed to allow multiple users to simultaneously connect to a Controller and share a set of Engines for collaboration
  - I start an MPI parallel simulation
  - An exception is raised
  - You connect, look at the data, debug a function and set the simulation to continue
  - When it finishes I connect and plot the data

# IPython's Distributed Memory Arrays

(work performed under this Phase I)

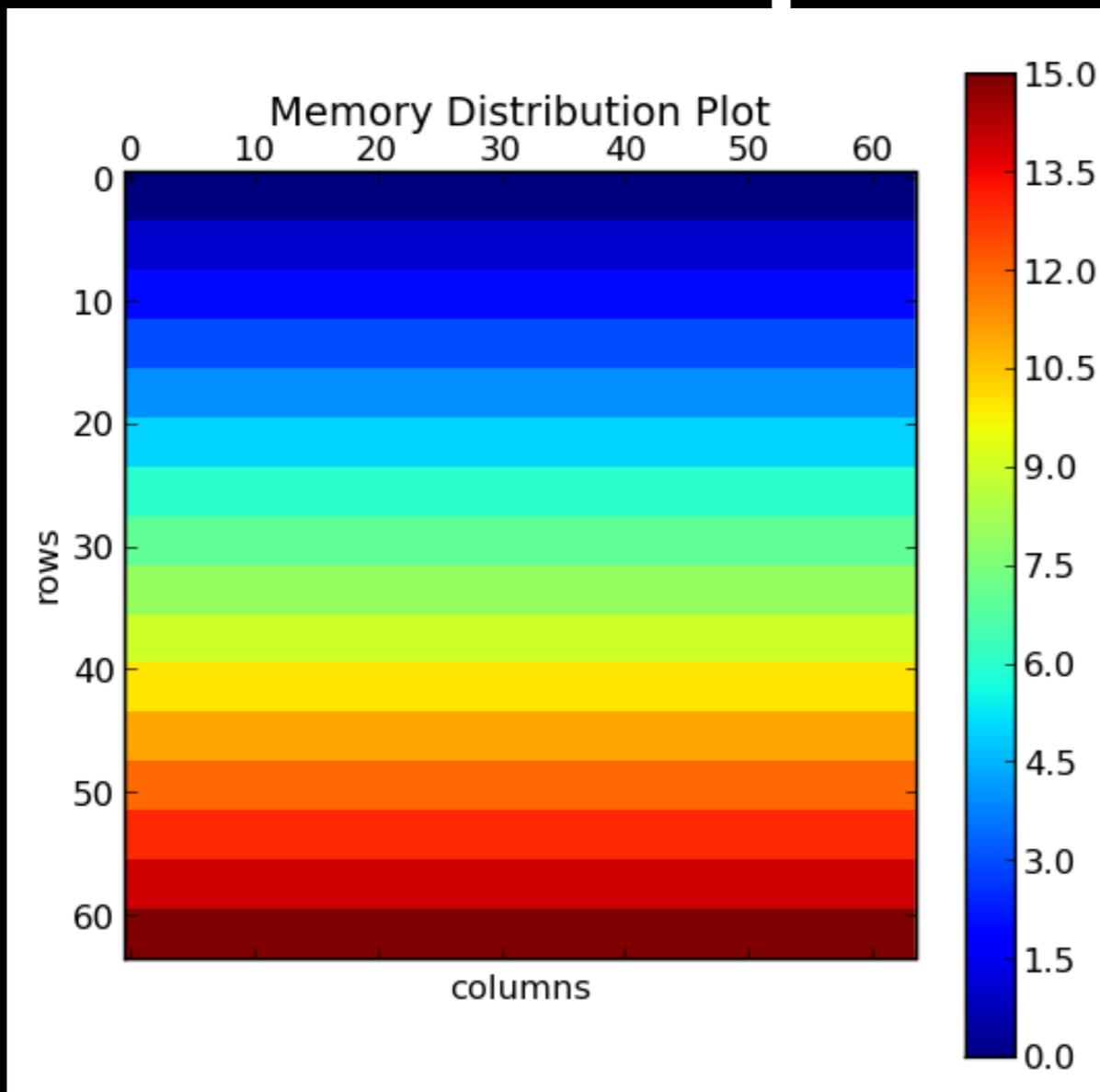
# Overview

- Based on an single program multiple data, message passing model
- Two main parts developed in Phase I:
  - DistArray Python class
  - Algorithms that work with DistArrays

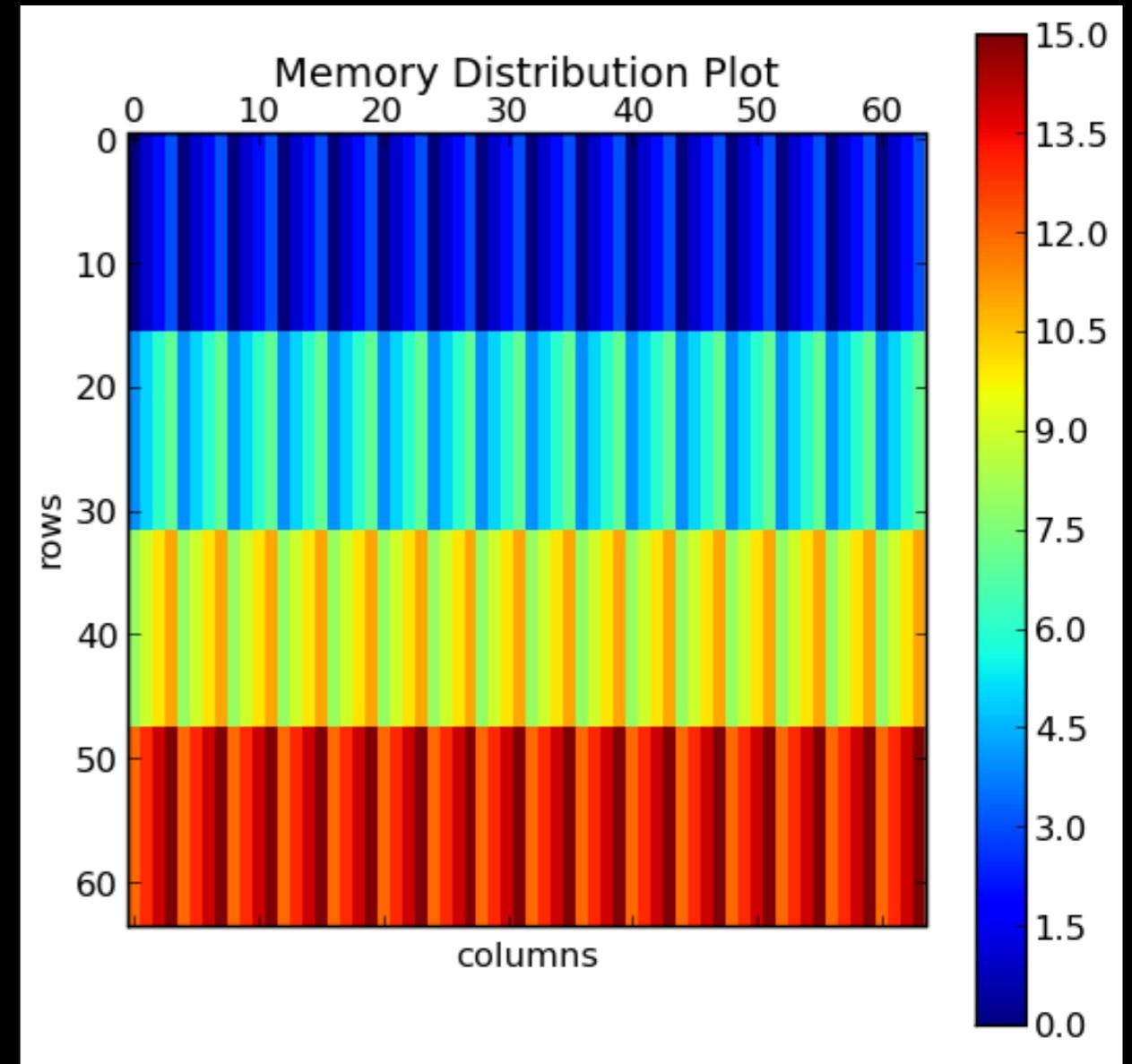
# Data Decomposition

- Each array dimension can be distributed or not
- Each distributed dimension can be either block or cyclic
- Default is block distributed along first dimension

# Examples: 16 processes



```
import distarray as da  
a = da.DistArray((64,64))
```



```
import distarray as da  
a = da.DistArray((64,64), dist=('b','c'))
```

# Algorithms

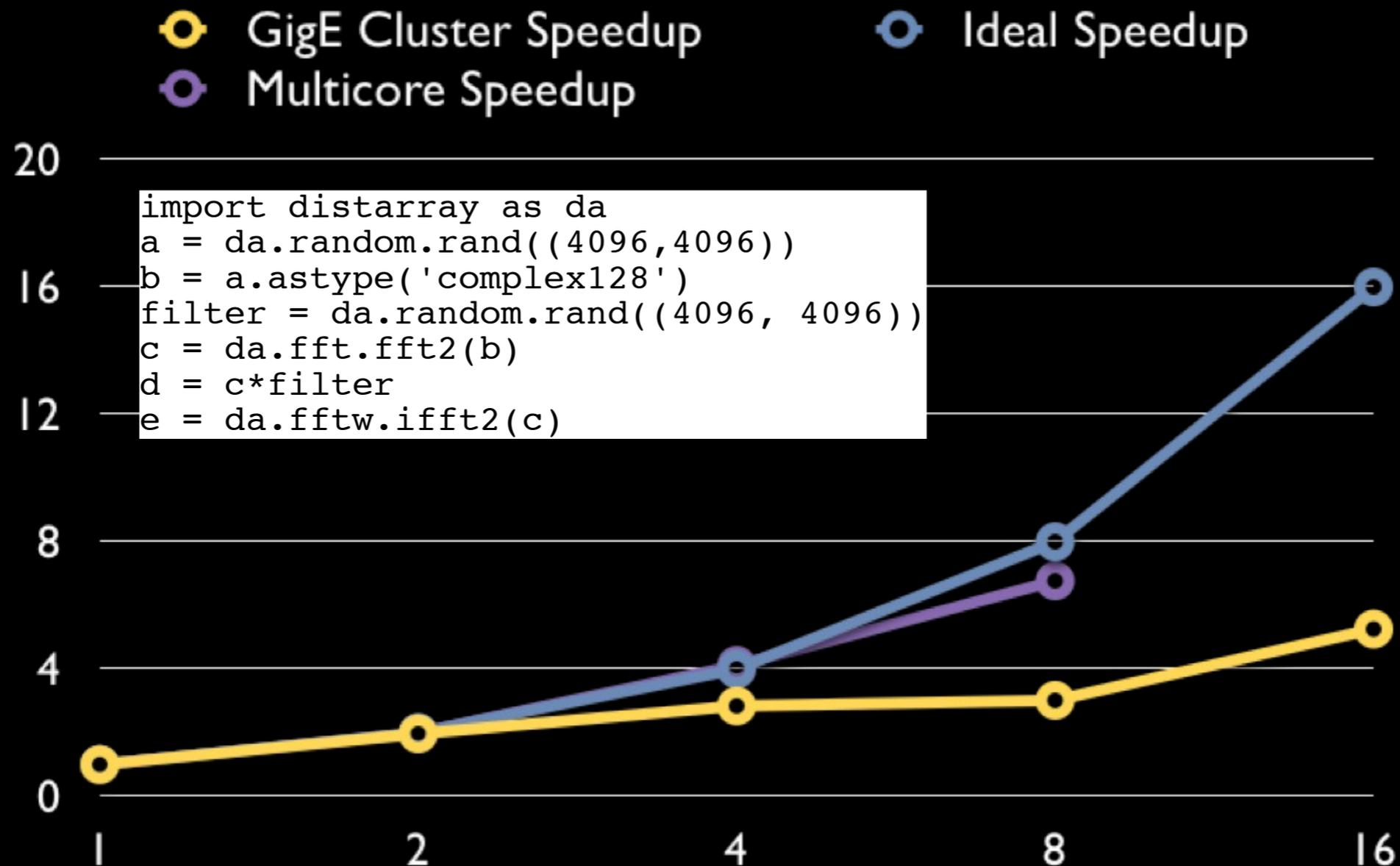
- Basic element-wise operations
  - +/\*/-//, cos, sin, etc.
- Simple reductions
  - Sum, avg, std, etc.
- Random arrays
- FFT's

# Trigonometric Example



Multicore scaling likely limited by memory access

# FFT Example



A parallel FFT has significant communication and thus benefits from multicore

# FFT Example

Nasa  
Researcher

```
import distarray as da
a = da.random.rand((4096,4096))
b = a.astype('complex128')
filter = da.random.rand((4096, 4096))
c = da.fft.fft2(b)
d = c*filter
e = da.fftw.ifft2(c)
```

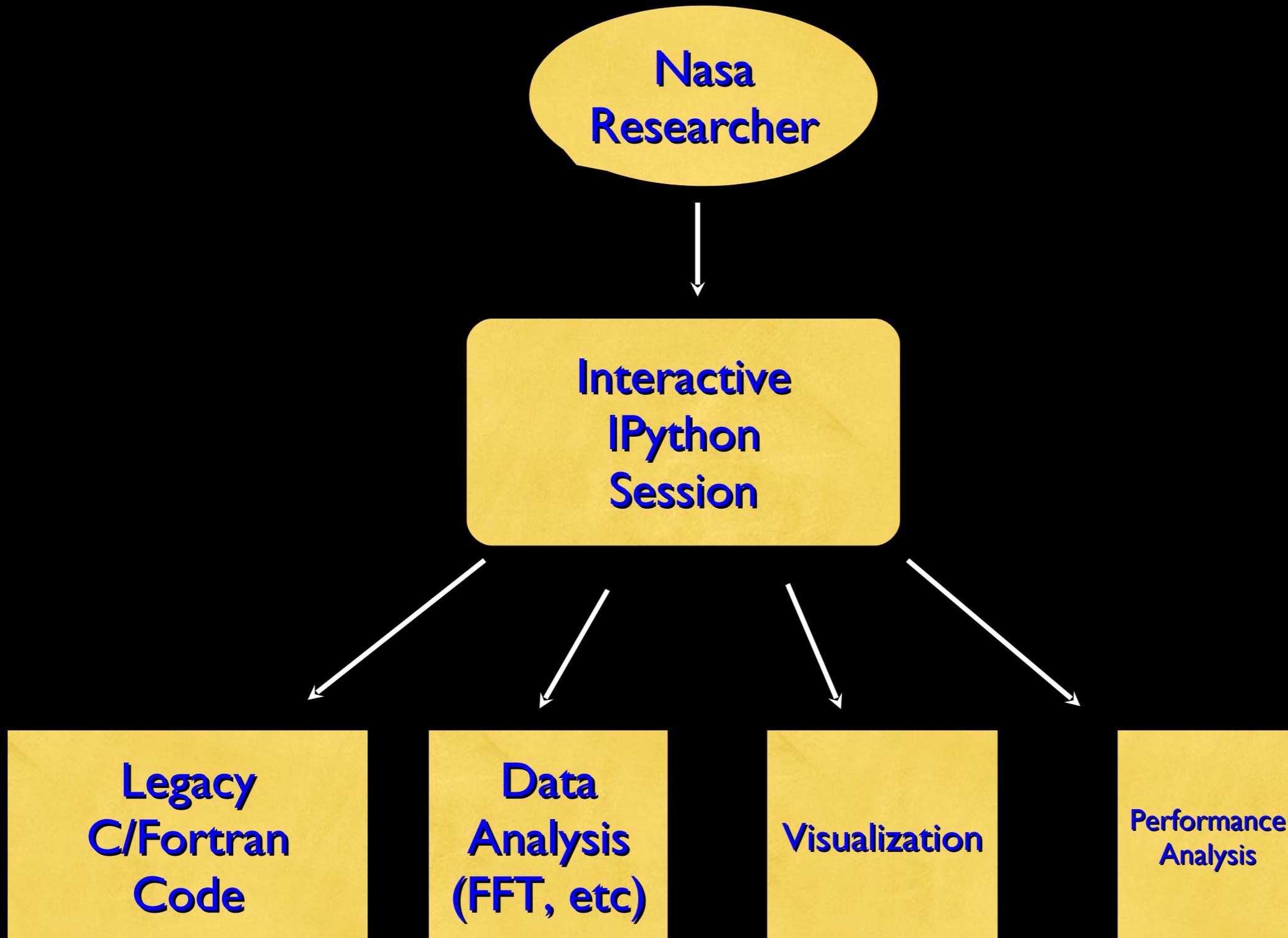
`fft(b[0:1027])`

`fft(b[1028:2047])`

`fft(b[2048:3071])`

`fft(b[3072:4095])`

# Our Phase II Vision



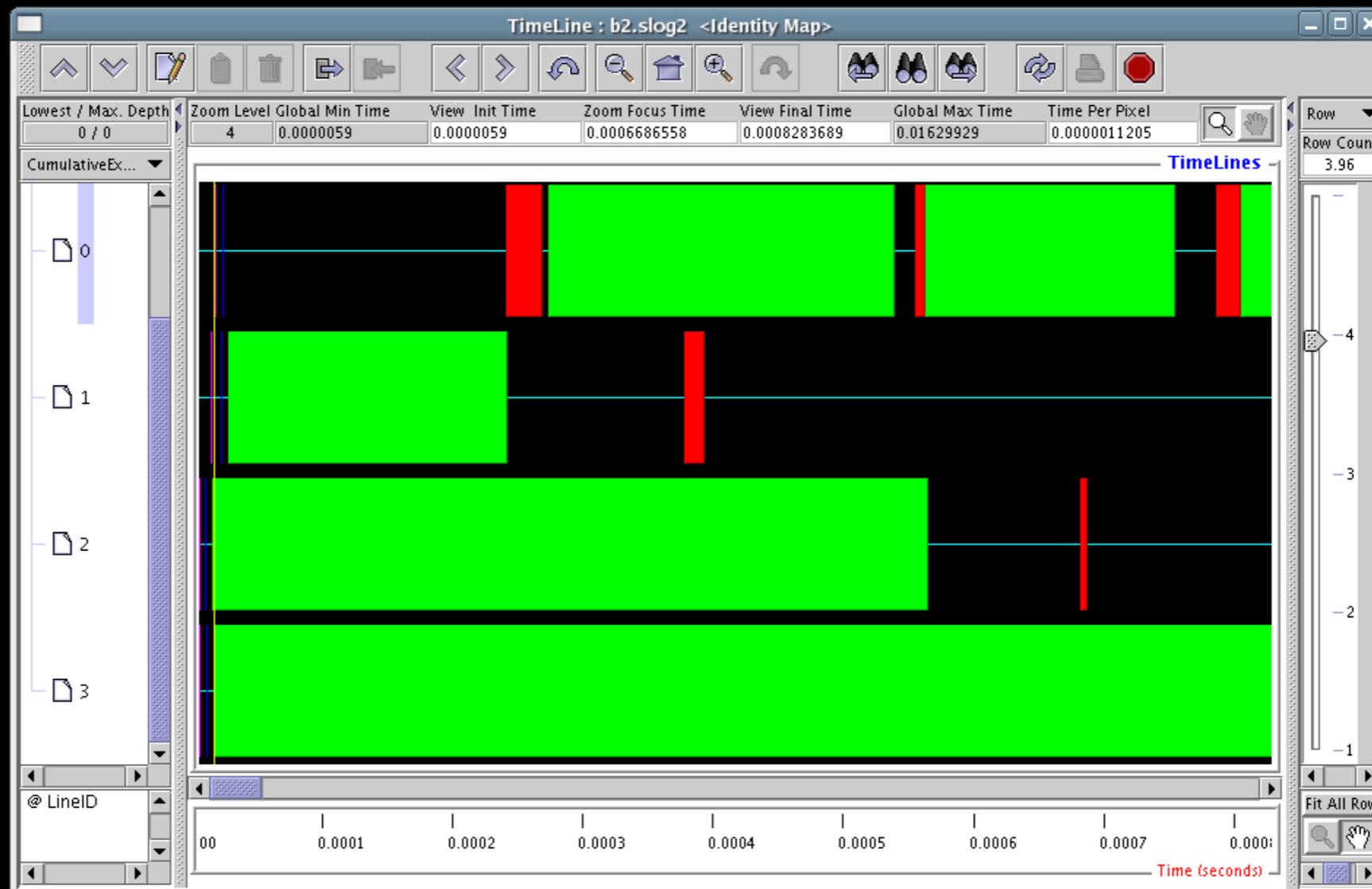
# Directions for Phase II?

- Develop a suite of intrinsic distributed routines (ScaLAPACK, FFTW, PETSc) optimized for various architectures (multicore, distributed)
- Develop implementations for OpenMP (better performance on multicore?)
- Develop tools for performance analysis (test suite, graphical display)
- Improve tools for wrapping legacy code (build tools for important platforms)
- Other directions?

# Partners for Phase II?

- Does someone want to try our tools for their project?
  - Someone with legacy C/Fortran code who would like to try Python (interactive data analysis, plotting)?
  - Someone who already uses Python, but would like to use multiple processors?

# Jumpshot is a tool for performance analysis



# Performance

- The `DistArray` class is written in Python
  - Lots of logic to create a new `DistArray`
  - Plan on implementing in C
- Overloaded operators and some functions create temporaries
- Can use inplace functions, but sacrifice syntax
- Balance between ease-of-use and performance