

Distributed Data Structures, Parallel Computing and IPython

Brian Granger, Cal Poly San Luis Obispo
Fernando Perez, UC Berkeley



Funded in part by NASA

Motivation

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
- No interactive capabilities
- Awkward access to plotting, 3D visualization, system shell

Interactive Computing Environments

- Matlab, Mathematica, IDL, Maple, *Python*
- 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 relative to compiled languages
 - No easy route to optimization
 - No easy route to parallelization
- Some are very expensive (\$\$\$)

Trends in Hardware

- The thrilling ride of increasing clock speed is over
- New emphasis on performance per watt
- multicpu & multicore & manycore
- A recent Berkeley white paper suggests that 1000 cores per chip is optimal (google for “view from berkeley”)
- Faster = parallel. Power efficient = parallel

Challenges

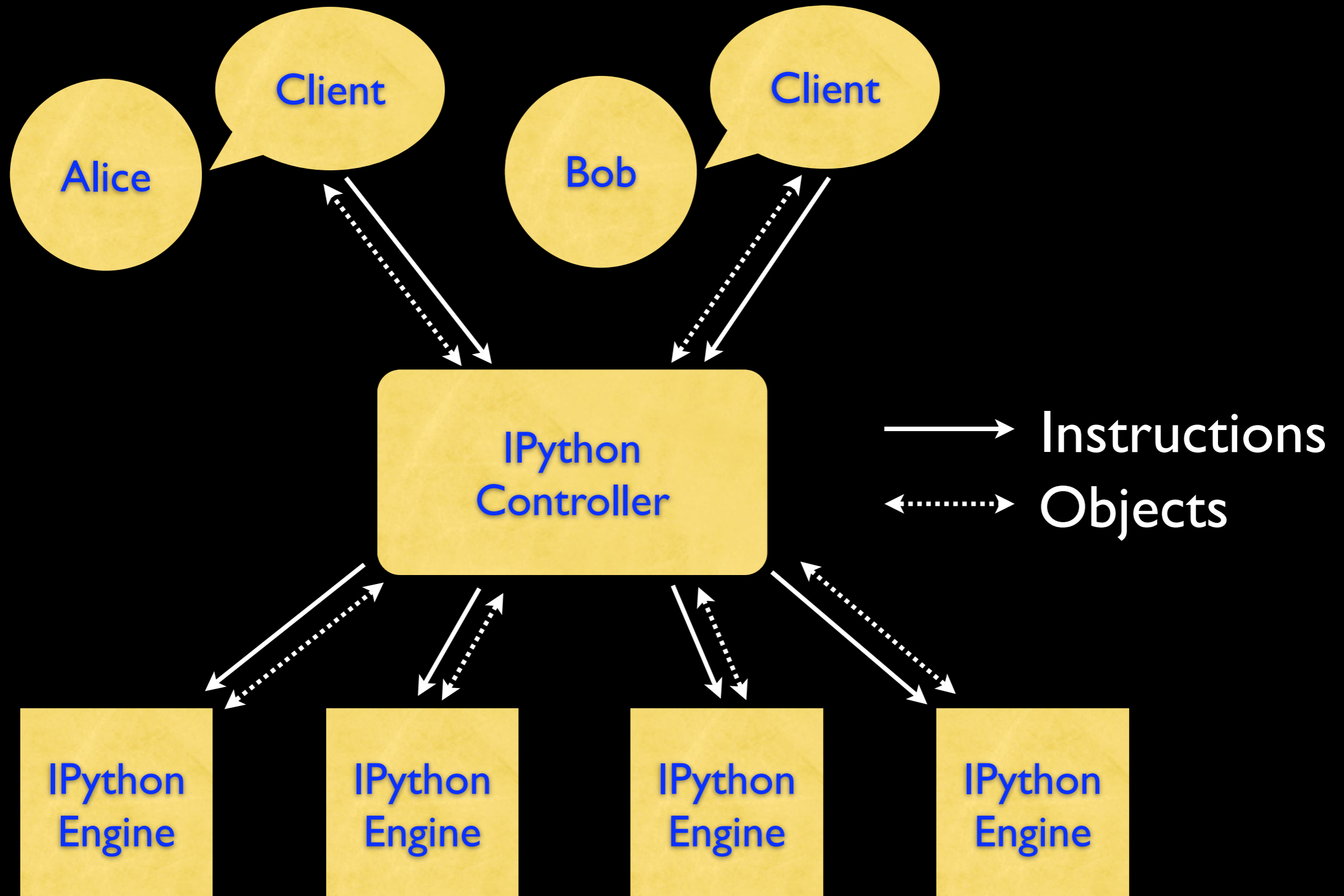
- We want performance and productivity...
- ...but performance = parallelism...
- ...so, we really need parallelism and productivity
- Our existing tools only give us one or the other:
 - Compiled languages
 - MPI
 - High-level languages

Interactive Parallel Computing with IPython

Goals

- Make all stages of parallel computing fully interactive: development, debugging, testing, execution, monitoring,...
- Easy things should be easy, difficult things possible
- Make parallel computing collaborative
- More dynamic model for load balancing and fault tolerance
- Seamless integration with other tools: plotting/ visualization, system shell
- Also want to keep the benefits of traditional approaches:
 - Should integrate with threads/MPI if appropriate
 - Should be easy to integrate compiled code and libraries
- **Support many types of parallelism**

IPython's Architecture



Architecture Details

- The IPython Engine is a Python interpreter that executes code received over the network
- The IPython Controller maintains a registry of engines and a queue for code to be run on each engine
- The Client connects to the Controller to submit user code. Client = user controlled Python session
- The Controller and Engines are *fully asynchronous*. Can't hack this on as an afterthought
- Robust error handling. All exceptions are collected and brought back to the client
- Everything is interactive - even on a supercomputer!

Usage Case: Multicore

- Run the Controller, Engines and a Client on a multicore laptop or desktop = ipcluster
- Zero setup: go from serial to parallel in seconds in many cases
- Don't even have to quit IPython
- Same code will run on a cluster or supercomputer

```
$ ipcluster local -n 4
```

Usage Case: Cluster+MPI

- You have existing C/C++/Fortran code that uses MPI
- You want to use it interactively from Python/IPython
- First, wrap the code into Python (mpi4py helps!)
- Start a controller and engines using ipcluster.
- Connect to the Controller from an IPython session on your laptop, import your code and use it interactively

```
$ ipcluster mpirun -n 64 --mpi=mpi4py
```

```
$ ipcluster pbs -n 64 --pbs-script=myscript.sh
```

Models of Parallelism

- Most models of parallel/distributed computing can be mapped onto IPython's architecture:
 - Message passing, task farming, tuple spaces
 - Bulk synchronous parallel (BSP), MapReduce
- We use interfaces/adapters to map these models into IPython's architecture
- With IPython all of these types of parallel computations can be done *interactively* and *collaboratively*

Example

```
In [1]: import sympy
```

```
In [2]: def factorit(n):  
...:     x = sympy.var('x')  
...:     return sympy.factor(x**n-1,x)  
...:
```

```
In [3]: factorit(5)
```

```
Out[3]:  $-(1 - x)(1 + x + x^2 + x^3 + x^4)$ 
```

```
In [4]: from IPython.kernel import client
```

```
In [5]: mec = client.MultiEngineClient()
```

```
In [6]: mec.execute('import sympy')
```

```
In [7]: f = mec.map(factorit, range(100, 110))
```

```
In [11]: f[0]
```

```
Out[11]:  $-(1 + x)(1 + x^2)(1 - x)(1 + x + x^2 + x^3 + x^4)(1 - x + x^2 - x^3 + x^4)(1 + x^5 + x^{10} + x^{15} + x^{20})(1 - x^5 + x^{10} - x^{15} + x^{20})(1 - x^{10} + x^{20} - x^{30} + x^{40})(1 - x^2 + x^4 - x^6 + x^8)$ 
```

Replace MultiEngineClient by TaskClient to get load balancing

Distributed Data Structures: Arrays

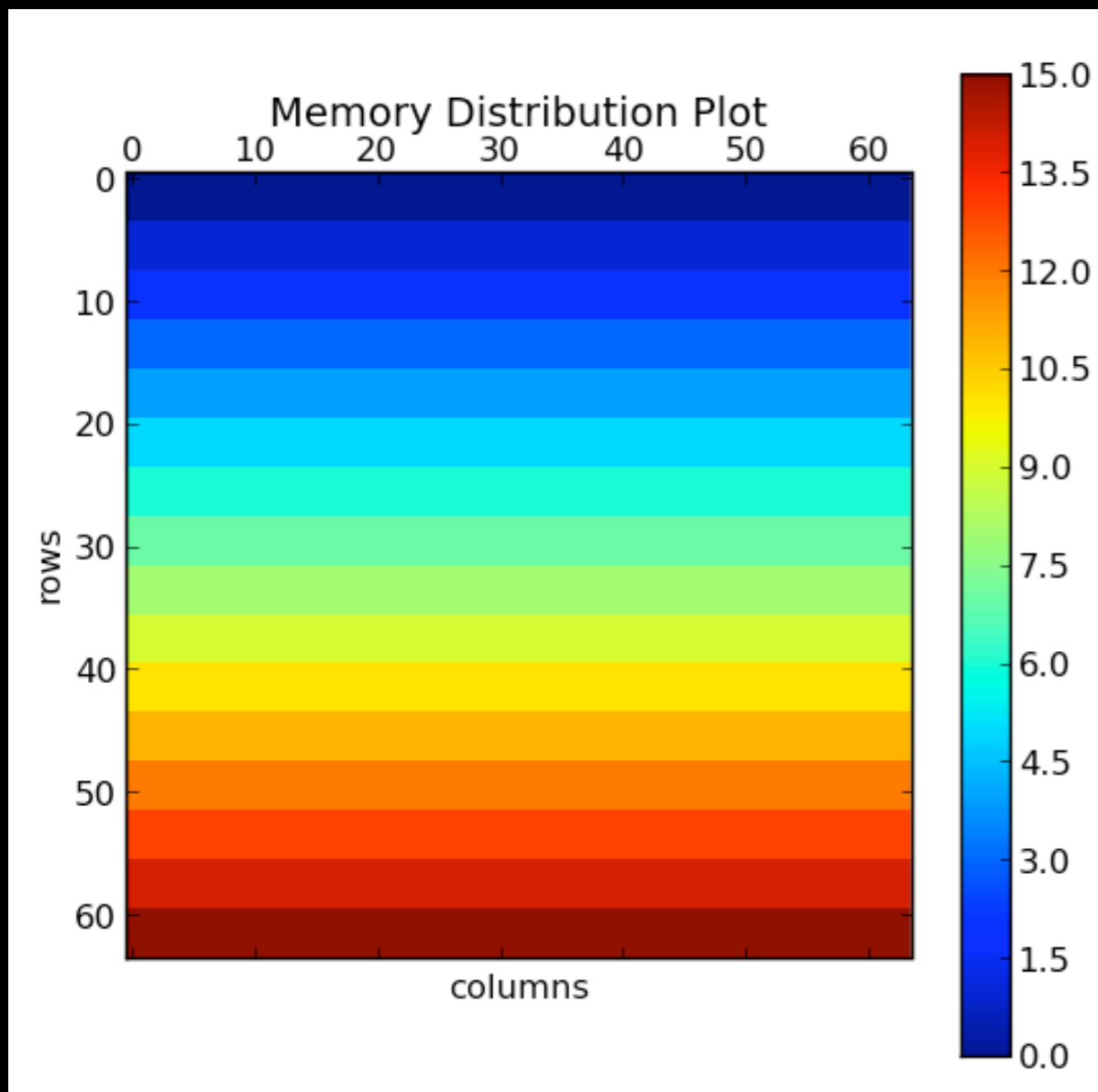
Overview

- Based on a single program multiple data + message passing (MPI) model
- Use mpi4py for MPI calls - thin wrapper around standard MPI implementations
- High level interface that mirrors NumPy
 - Array/tensors of arbitrary rank/dimension
 - Arbitrary data types (not just double, float, int)
- Two main parts
 - DistArray Python class
 - Algorithms that work with DistArrays

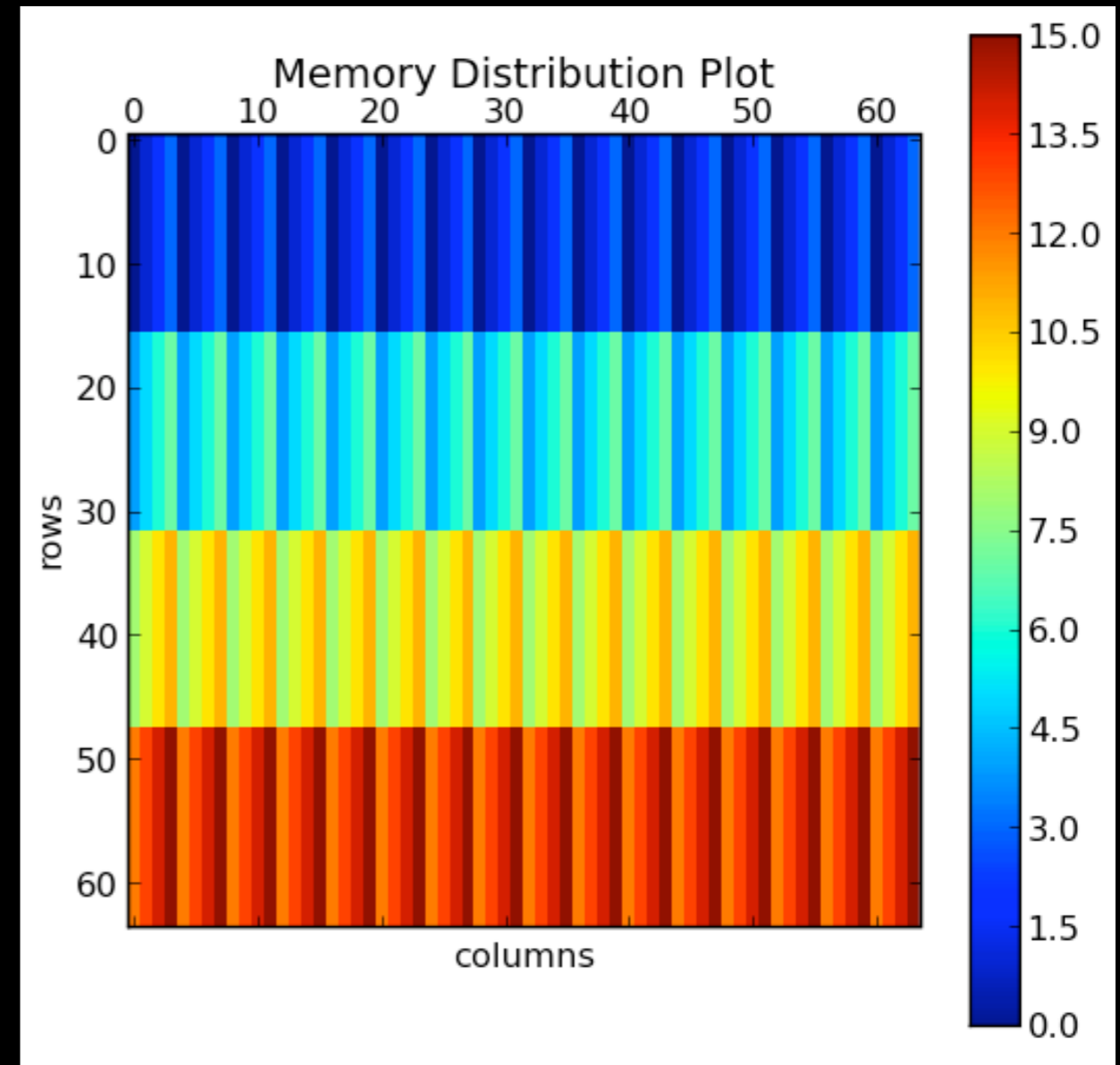
Data Decomposition

- Each array dimension can be distributed or not
- Each distributed dimension can be either block or cyclic
- Default is block distributed along first dimension
- Simple syntax for controlling decomposition

Examples: 16 processes



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



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

Processor Grid

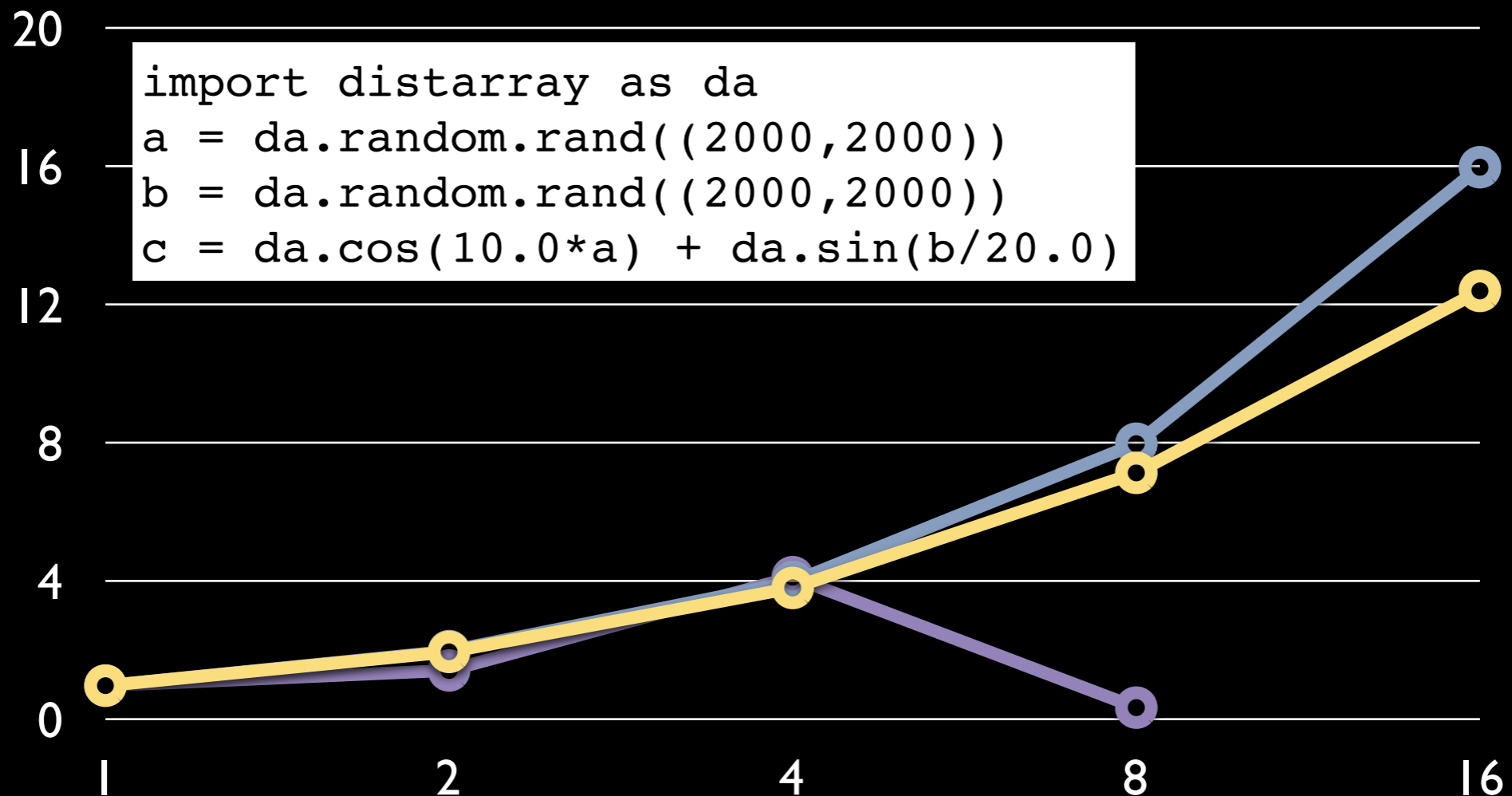
- A cartesian grid of processors is created to map the decomposed array onto processors
- The processor grid is created automatically using a sophisticated algorithm that attempts to load balance the array
- This algorithm is good for most purposes
- The processor grid can be customized if needed

Algorithms

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

Scaling: Simple Example

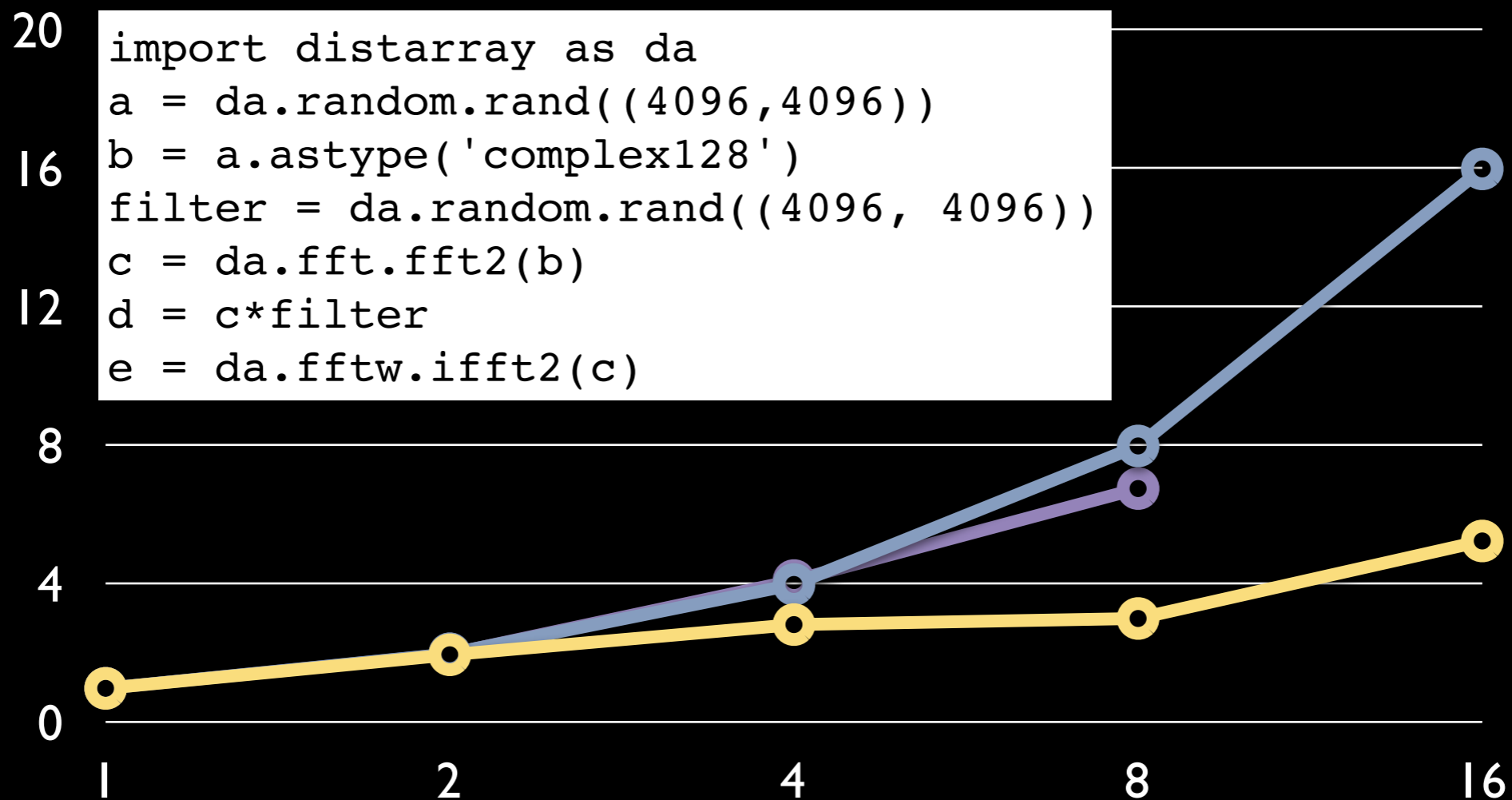
- GigE Cluster Speedup
- Ideal Speedup
- Multicore Speedup



Multicore scaling in this case limited by memory access

Scaling: FFT Example

- GigE Cluster Speedup
- Ideal Speedup
- Multicore Speedup



A parallel FFT has significant communication and thus benefits from multicore

Distribute Data Structures: Dictionaries

Dictionary/hash table

- Python has a built in dictionary/hash table
- We have implemented a distributed memory version using `mpi4py`
- Both SPMD and MPMD modes
- The core API is the same as that of Python's built-in dictionary.
- Extended API to allow efficient and asynchronous sets and gets.

Example

```
d = {}  
d['a'] = 2**14-1  
d['b'] = range(5)  
d['c'] = 'value'  
d['d'] = 5.32  
print d.keys(), d.values()
```

```
['a', 'c', 'b', 'd'] [16383, 'value', [0, 1, 2, 3, 4], 5.32]
```

```
from IPython.dist.distdict import DistDict  
dd = DistDict()  
dd['a'] = 2**14-1  
dd['b'] = range(5)  
dd['c'] = 'this is a value'  
dd['d'] = 5454.346563  
print dd.keys()  
print dd.rank, dd.size, dd.local_keys(), dd.local_values()
```

```
['a', 'c', 'b', 'd']  
0 2 ['a', 'c'] [16383, 'value']  
['a', 'c', 'b', 'd']  
1 2 ['b', 'd'] [[0, 1, 2, 3, 4], 5.32]
```

Concluding Thoughts

- IPython is actively working on parallelism + productivity
- Parallel computing can be dynamic, interactive, flexible, collaborative...
- ...and still integrate with legacy codes
- Lots of extremely interesting work to do - everything is open source, come join us