

Interactive Parallel and Distributed Computing with IPython



Brian Granger
Research Scientist
Tech-X Corporation, Boulder CO

Collaborators: Fernando Perez (CU Boulder), Benjamin Ragan-Kelley
(Undergraduate Student, SCU)

PyCon 2007, TX

Motivation

We Have a Problem

- We = developers, scientists, users, etc.
- Problem = There are computational things that we can't do today: CPU bound, large amounts of data, algorithms that scale poorly ($n!$, a^n)
- Traditional solution = just wait, the hardware folks will make us happy.

The Reality

- The free ride is over.
- Hardware is getting faster, ... but only because of parallelism.
- Multi-core, multi-CPU, clusters, supercomputers.
- Unless we do something, Python will run at roughly the same speed a few years from now :(
- Performance is now a software problem.
- Traditional software tools for expressing parallelism (threads, message passing) are time consuming.
- Software development time is often the bottleneck (not CPU speed).

Case Study: Parallel Jobs at NERSC in 2006

- NERSC = DOE Supercomputing center at Lawrence Berkeley National Laboratory
- Seaborg = IBM SP RS/6000 with 6080 CPUs
 - 90% of jobs used less than 113 CPUs
 - Only 0.26% of jobs used more than 2048 CPUs
- Jacquard = 712 CPU Opteron system
 - 50% of jobs used fewer than 15 CPUs
 - Only 0.39% of jobs used more than 256 CPUs

* Statistics (used with permission) from NERSC users site (<http://www.nersc.gov/nusers>)

Our Goals with IPython

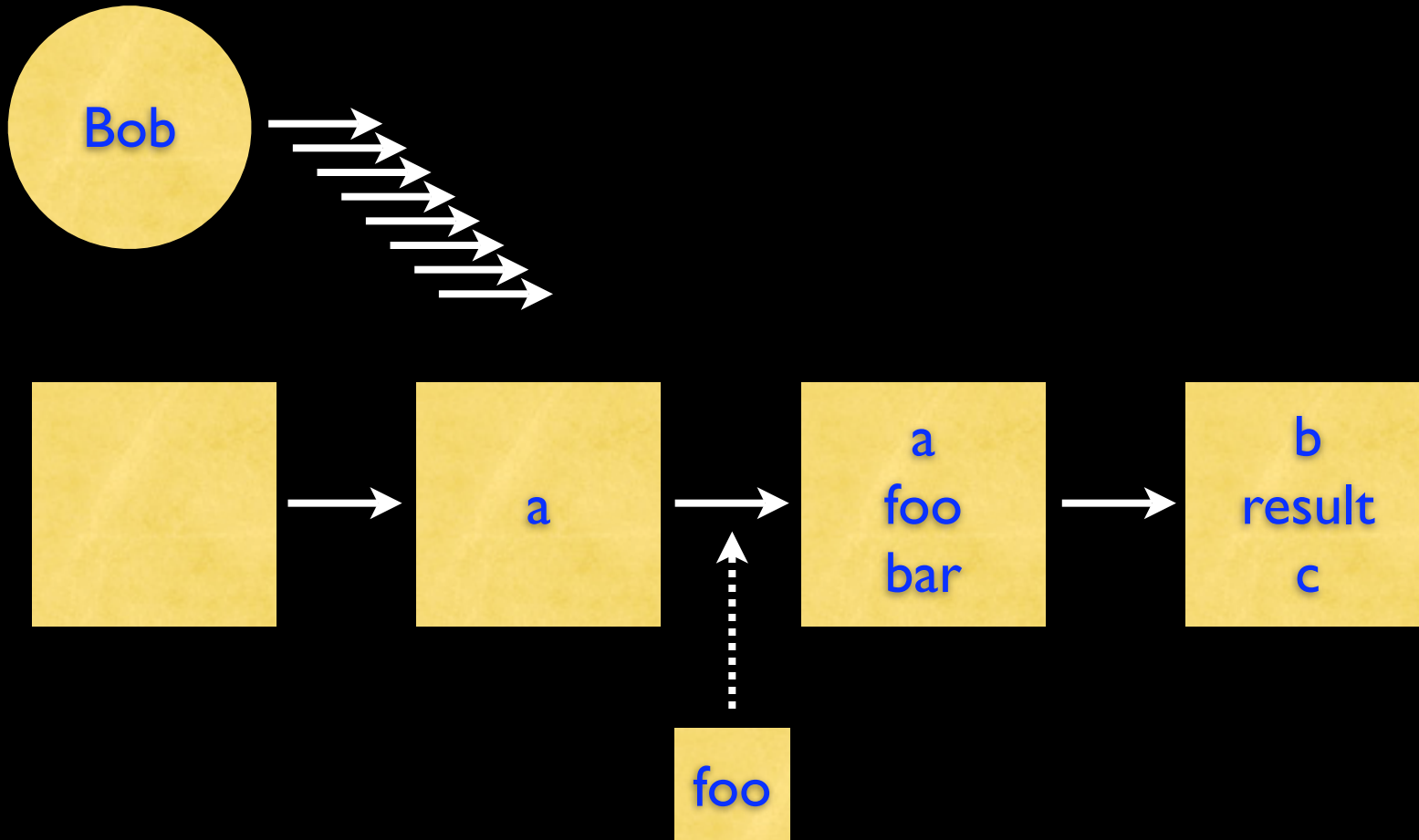
- Trivial parallel things should be trivial.
- Difficult parallel things should be possible.
- Make all stages of parallel computing fully interactive: development, debugging, testing, execution, monitoring,...
- Make parallel computing collaborative.
- More dynamic model for load balancing and fault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

Computing With Namespaces

Namespaces

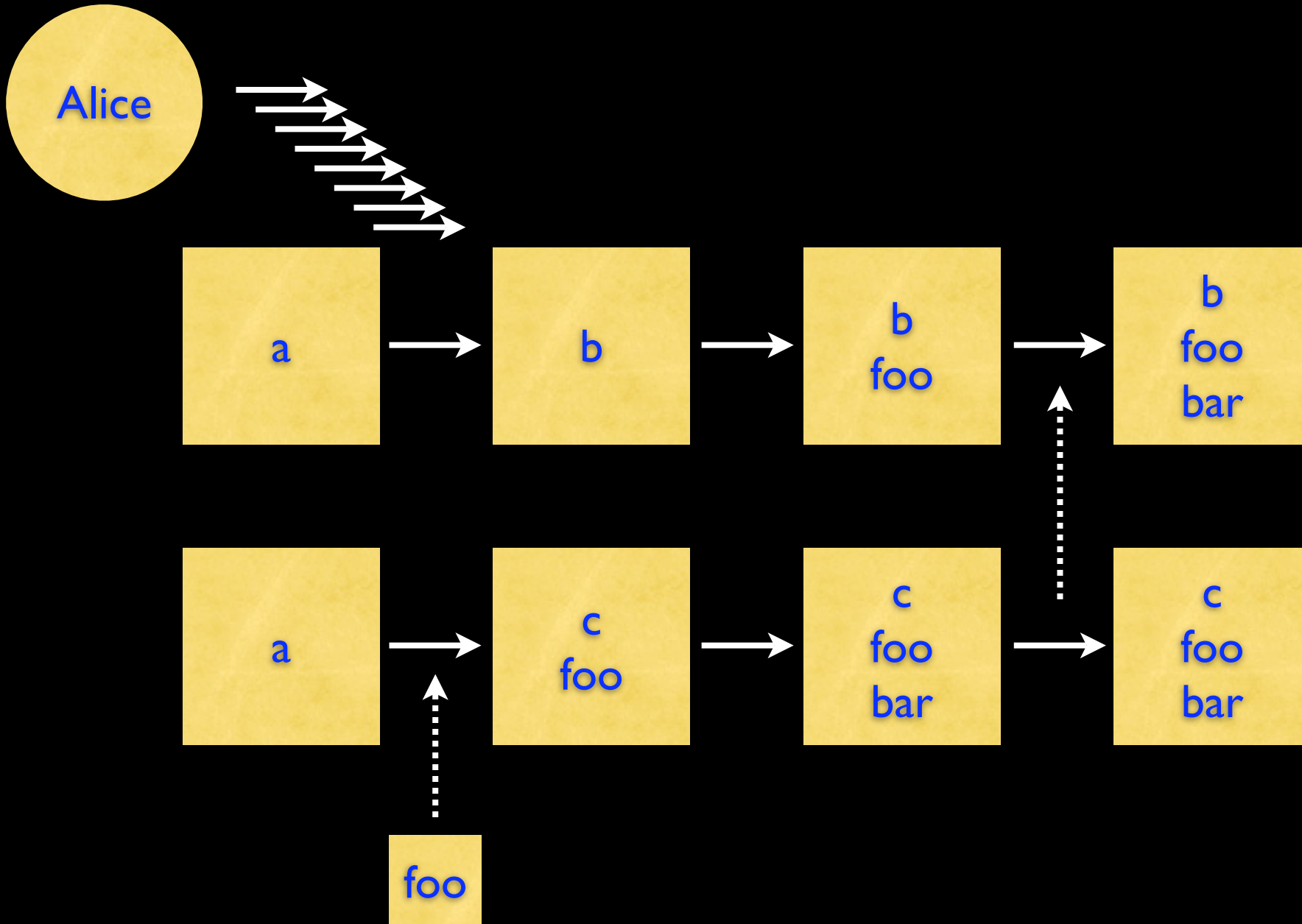
- Namespace = a container for objects and their unique identifiers.
- An instruction stream causes a namespace to evolve with time.
- Interactive computing: the instruction stream has a human agent as its runtime source at some level.
- A (namespace, instruction stream) is a higher level abstraction than a process or thread.
- Data in a namespace can be created in-place (by instructions) or by external I/O (disk, network).
- Thinking about namespaces allows us to abstract parallelism and interactivity in a useful way.

Serial Namespace Computing



- Instructions
-→ Data from Network/Disk

Parallel Namespace Computing

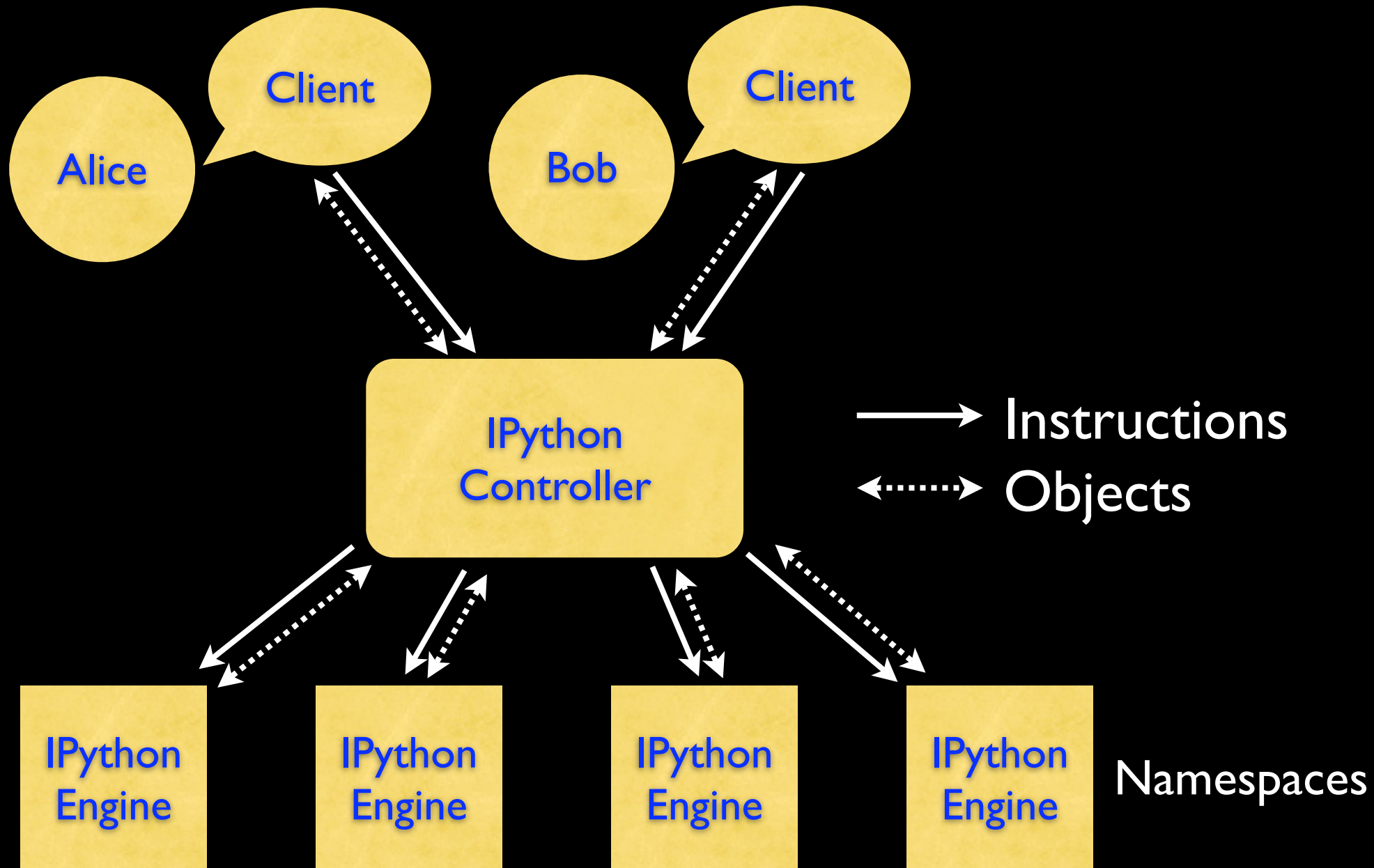


Important Points

- Requirements for Interactive Computation:
 - Alice/Bob must be able to send instruction stream to a namespace.
 - Alice/Bob must be able to push/pull objects to/from the namespace (disk, network).
- Requirements for Parallel Computation:
 - Multiple namespaces and instruction streams (for general MIMD parallelism).
 - Send data between namespaces (MPI is really good at this)
- Requirements for Interactive Parallel Computation:
 - Alice/Bob must be able to send multiple instruction streams to multiple namespaces.
 - Alice/Bob must be able to push/pull objects to/from the namespaces .

* These requirements hold for any type of parallelism

IPython's Architecture



Architecture Details

- The IPython Engine/Controller/Client are typically different processes. Why not threads? Scalability, philosophy and the GIL.
- Can be run in arbitrary configurations on laptops, clusters, supercomputers.
- The Controller and Engines are fully asynchronous. Can't hack this on as an afterthought. We use Twisted :)
- Must deal with long running commands that block all network traffic.
- Dynamic process model. Engines and Clients can come and go at will at any time*.
- We use the interface/adaptor pattern extensively. Everything (network protocols, Engine, Controller, Client) is pluggable (by you).

*Unless you are using MPI

Mapping Namespaces To Various Models of Parallel Computation

Key Points

- Most models of parallel/distributed computing can be mapped onto this architecture.
 - Message Passing
 - Task Farming
 - TupleSpaces
 - BSP (Bulk Synchronous Parallel)
 - Google's MapReduce
 - ???
- With IPython's architecture all of these types of parallel computations can be done **interactively** and **collaboratively**.
- The mapping of these models onto our architecture is done using interfaces/adapters and requires very little code.

Demos

Conclusions

- Namespaces provide a useful starting point for thinking about interactive parallel computing.
- Interactivity is a separate question from the model of parallel computation.
- IPython provides interactivity for many kinds of parallelism.
- Lots of interesting questions to think about and explore. Come to the concurrency/parallelism BOF tonight.
- Still BSD. Today the parallel stuff is in a separate branch. Watch for new release soon.

Error Propagation

- Building a distributed system is easy...
- Unless you want to handle errors well.
- We have spent a lot of time working/thinking about this issue. Not always obvious what should happen.
- Our goal: error handling/propagation in a parallel/distributed context should be a nice analytic continuation of what happens in a serial context.
- Remote exceptions should propagate to the user in a meaningful manner.
- Policy of safety: don't ever let errors pass silently.

MPI

- Without full and robust MPI support, these tools would be a no-go for many applications
- Engines can be started using `mpiexec` and call `MPI_Init`. From then on, instruction streams sent to engines can contain arbitrary MPI calls.
- Can use MPI through:
 - Low-level C/C++/Fortran bindings
 - Python bindings (see <http://mpi4py.scipy.org>)
- Remains fully interactive/collaborative.
- Fully supported today!