Argonne NATIONAL LABORATORY

Python for High Performance Computing

William Scullin
wscullin@alcf.anl.gov
Leadership Computing Facility
Argonne National Laboratory

U.S. DEPARTMENT OF ENERGY

Why this talk?

"People are doing high performance computing with Python…

How do we stop them?"

- Senior Performance Engineer

Why Python?

What's Python?

- Created by Guido van Rossum in 1989
- Originally a scripting language for the Amoeba OS
- Highly influenced by Modula-3, ABC, Algol, and C
- It refers to both the language and to the reference implementation CPython
- Two major versions of the language:
  - Python 2
  - Python 3

Why Use Python?

- If you like a programming paradigm, it's supported
- Most functions map to what you know already
- Easy to combine with other languages
- Easy to keep code readable and maintainable
- Lets you do just about anything without changing languages
- The price is right!
  - No license management
  - Code portability
  - Fully Open Source
  - Very low learning curb
- Comes with a highly enthusiastic and helpful community

Easy to learn

```
#include "iostream"
#include "math"
int main(int argc,char** argv)
{
    int n = atoi(argv[1]);
    for(int i=2;
        i<(int) sqrt(n);
        i++)
    {
       p=0;
       while(n % i)
       {
          p+=1;
          n/=i;
       }
       if (p)
          cout << i << "^"
               << p << endl;
    }
    return 0;
}
```

```
import math, sys

n = int(sys.argv[1])
for i in range(2,math.sqrt(n)):

     p=0
     while n % i:

     (p,n) = (p+1,n/i)

     if p:
        print i,'^',p
                    sys.exit(0)
```


Why Use Python for Scientific Computing?

- "Batteries included" + rich scientific computing ecosystem
- Good balance between computational performance and time investment
  - Similar performance to expensive commercial solutions
  - Many ways to optimize critical components
  - Only spend time on speed if really needed
- Tools are mostly open source and free
- Strong community and commercial support options.
- No license management for the modules that keep people productive

Science Tools for Python

General
NumPy
SciPy

GPGPU Computing
PyCUDA
PyOpenCL

Parallel Computing
PETSc
PyMPI
Pypar
mpi4py

Wrapping C/C++/ Fortran
SWIG
Cython
ctypes

Plotting & Visualization
matplotlib
VisIt
Chaco
MayaVi

AI & Machine Learning
pyem
ffnet
pymorph
Monte
hcluster

Biology (inc. neuro)
Brian
SloppyCell
NIPY
PySAT

Molecular & Atomic Modeling
PyMOL
Biskit
GPAW

Geosciences
GIS Python
PyClimate
ClimPy
CDAT

Bayesian Stats
PyMC

Optimization
OpenOpt

Symbolic Math
SymPy

Electromagnetics
PyFemax

Astronomy
AstroLib
PySolar

Dynamic Systems
Simpy
PyDSTool

Finite Elements
SfePy

Other Languages
R
MATLAB

For a more complete list: http://www.scipy.org/Topical_Software

Why Not Use Python? - The Language

- Low learning curve
  - It's easy to write Fortran / C / C++ in Python
  - PEP 8 isn't the law, just a really good idea
    http://www.python.org/dev/peps/pep-0008/
  - Reimplementation of existing solutions is way too easy
    - if it's important, there's already a solution out there

- Easy to combine with C/C++/Fortran
  - there are communities around most major packages
  - really important packages have Python bindings
  - not all bindings are "Pythonic"

Why Not Use Python? - The Language

- There's constant revision through the PEP process

- Language maintainers strive for philosophical consistency
  - Backwards compatibility is seldom guaranteed
  - They're not kidding when the goal is to have only one way to do something
    - features have been known to vanish e.g.: lambda
  - Future features are often available in older versions to ease transitions

- Tim Peter's *The Zen of Python* notes:

  In the face of ambiguity, refuse the temptation to guess.
  There should be one-- and preferably only one - obvious way to do it.
  Although that way may not be obvious at first unless you're Dutch.
  Now is better than never.
  Although never is often better than *right* now.

- Language maintainers strive for "principle of least surprise"
  - Web folks are fighting for decimal numerics by default

Why Not Use Python? - CPython

- It's inefficient
  - Python 2.x is a true interpreter
  - Pure Python is interpreted line-by-line
  - "If you want your code to run faster, you should probably just use PyPy."
    — Guido van Rossum

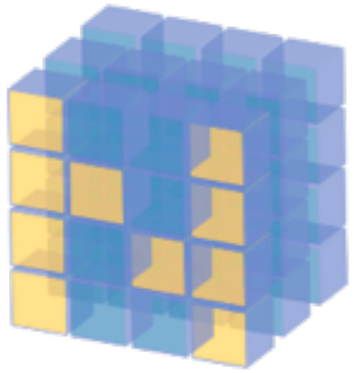
- The GIL
  - David Beazley covers it better than anyone:
    http://www.dabeaz.com/python/GIL.pdf

- Distutils
  - Conceived of as a way to make it easy to build and install Python modules
  - Really a way of thwarting custom linking and cross-compiling

- Lots of small file I/O as part of runs

- Debuggers and performance tools hate mixing languages

How About A Quick Demo?



NumPy

- N-dimensional homogeneous arrays (ndarray)
- Universal functions (ufunc)
  - basic math, linear algebra, FFT, PRNGs
- Simple data file I/O
  - text, raw binary, native binary
- Tools for integrating with C/C++/Fortran
- Heavy lifting done by optimized C / Fortran libraries
  - ATLAS or MKL, UMFPACK, FFTW, etc...

Creating NumPy Arrays

```
# Initialize with lists: array with 2 rows, 4 cols
>>> import numpy as np
>>> np.array([[1,2,3,4],[8,7,6,5]])
array([[1, 2, 3, 4],
       [8, 7, 6, 5]])


# Make array of evenly spaced numbers over an interval
>>> np.linspace(1,100,10)
array([   1.,   12.,   23.,   34.,   45.,   56.,   67.,   78.,
89.,  100.])


# Create and prepopulate with zeros
>>> np.zeros((2,5))
```


Slicing Arrays

```
>>> a = np.array([[1,2,3,4],[9,8,7,6],[1,6,5,4]])
>>> arow = a[0,:] # get slice referencing row zero
>>> arow
array([1, 2, 3, 4])

>>> cols = a[:,[0,2]] # get slice referencing columns 0 and 2
>>> cols
array([[1, 3],
       [9, 7],
       [1, 5]])

# NOTE: arow & cols are NOT copies, they point to the original data
>>> arow[:] = 0
>>> arow
array([0, 0, 0, 0])

>>> a
array([[0, 0, 0, 0],
       [9, 8, 7, 6],
       [1, 6, 5, 4]])

# Copy data
>>> copyrow = arow.copy()
```


Broadcasting with ufuncs

apply operations to many elements with a single call

```
>>> a = np.array(([1,2,3,4],[8,7,6,5]))
>>> a
array([[1, 2, 3, 4],
       [8, 7, 6, 5]])


# Rule 1: Dimensions of one may be prepended to either array to match the array with the
greatest number of dimensions
>>> a + 1 # add 1 to each element in array
array([[2, 3, 4, 5],
       [9, 8, 7, 6]])

# Rule 2: Arrays may be repeated along dimensions of length 1 to match the size of a larger
array
>>> a + np.array(([1],[10])) # add 1 to 1st row, 10 to 2nd row
array([[ 2, 3, 4, 5],
       [18, 17, 16, 15]])

>>> a**([2],[3]) # raise 1st row to power 2, 2nd to 3
array([[ 1,    4,    9, 16],
       [512, 343, 216, 125]])
```


SciPy

- Extends NumPy with common scientific computing tools
  - optimization
  - additional linear algebra
  - integration
  - interpolation
  - FFT
  - signal and image processing
  - ODE solvers
- Heavy lifting done by C/Fortran code

mpi4py - MPI for Python

- wraps a native mpi
- provides all MPI2 features
- well maintained
- requires NumPy
- insanely portable and scalable
- http://mpi4py.scipy.org/

How mpi4py works...

- mpi4py jobs must be launched with mpirun/mpiexec
- each rank launches its own independent python interpreter
  - no GIL!
- each interpreter only has access to files and libraries available locally to it, unless distributed to the ranks
- communication is handled by MPI layer
- any function outside of an if block specifying a rank is assumed to be global
- any limitations of your local MPI are present in mpi4py

mpi4py basics - datatype caveats

- mpi4py can ship *any* serializable objects
- Python objects, with the exception of strings and integers are pickled
  - Pickling and unpickling have significant overhead
  - overhead impacts both senders and receivers
  - use the lowercase methods, eg: recv(),send()
- MPI datatypes are sent without pickling
  - near the speed of C
  - NumPy datatypes are converted to MPI datatypes
  - custom MPI datatypes are still possible
  - use the capitalized methods, eg: Recv(), Send()
- When in doubt, ask if what is being processed is a memory buffer or a collection of pointers!

Calculating pi with mpi4py

```
from mpi4py import MPI
import random

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
mpisize = comm.Get_size()
nsamples = int(12e6/mpisize)

inside = 0
random.seed(rank)
for i in range(nsamples):
    x = random.random()
    y = random.random()
    if (x*x)+(y*y)<1:
      inside += 1

mypi = (4.0 * inside)/nsamples
pi = comm.reduce(mypi, op=MPI.SUM, root=0)

if rank==0:
    print (1.0 / mpisize)*pi
```


Calculating pi with mpi4py and NumPy

```
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
mpisize = comm.Get_size()
nsamples = int(12e6/mpisize)

np.random.seed(rank)
xy=np.random.random((nsamples,2))
mypi=4.0*np.sum(np.sum(xy**2,1)<1)/nsamples

pi = comm.reduce(mypi, op=MPI.SUM, root=0)

if rank==0:
    print (1.0 / mpisize)*pi
```


Anyone do this in production?

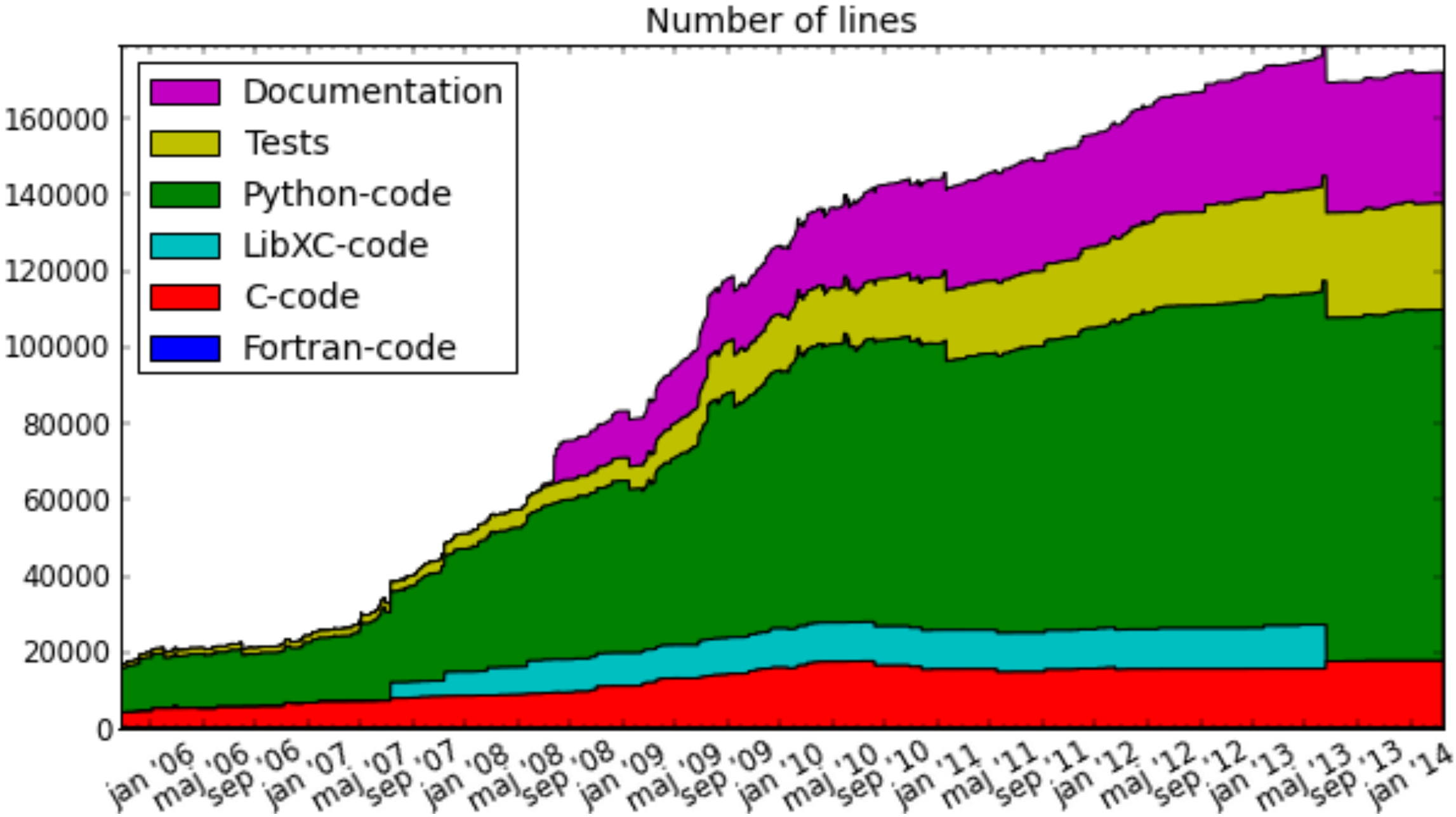
a massively parallel Python-C code
for electronic structure calculations

- *Ab initio* atomistic simulation for predicting material properties
  - density functional theory (DFT) and time-dependent density functional theory (TD-DFT)
    - Nobel prize in Chemistry to Walter Kohn (1998) for DFT
- Finite difference stencils on uniform real-space grid
- Non-linear sparse eigenvalue problem
  - ~106 grid points, ~103 eigenvalues
- Written in Python and C using the NumPy library
- Massively parallel using MPI
- Open source (GPL)

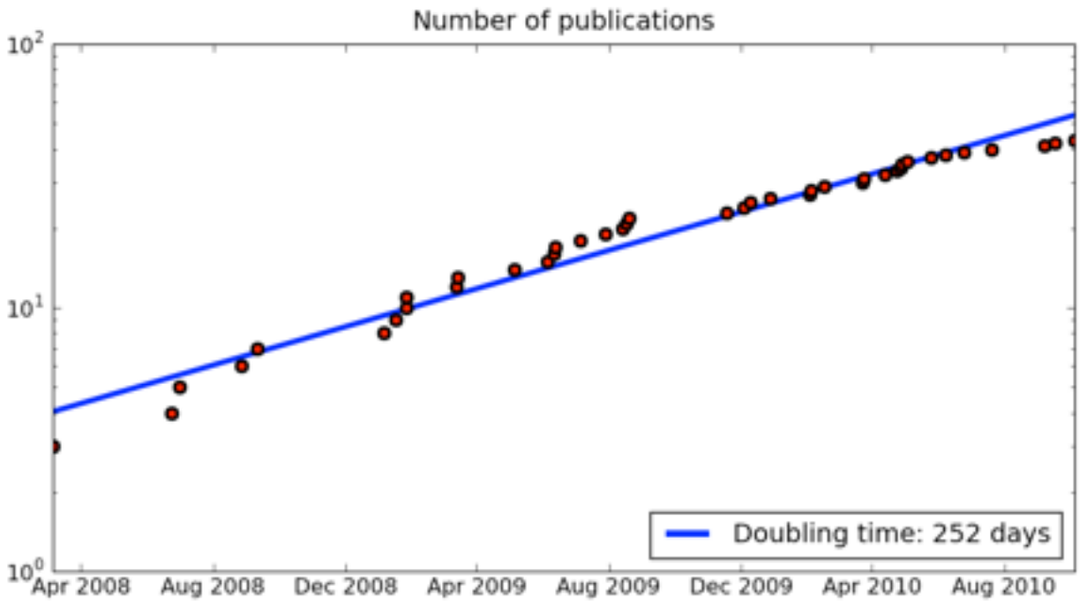
http://wiki.fysik.dtu.dk/gpaw

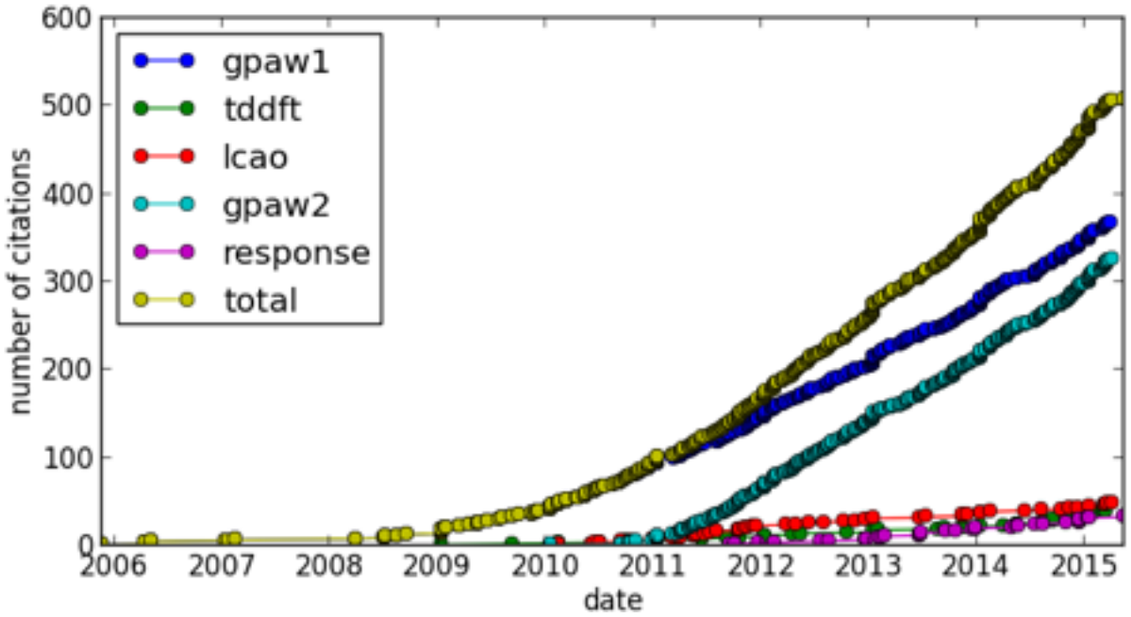
J. Enkovaara *et al.* J. Phys.: Condens. Matter **22**, 253202 (2010)

GPAW Source Code Timeline



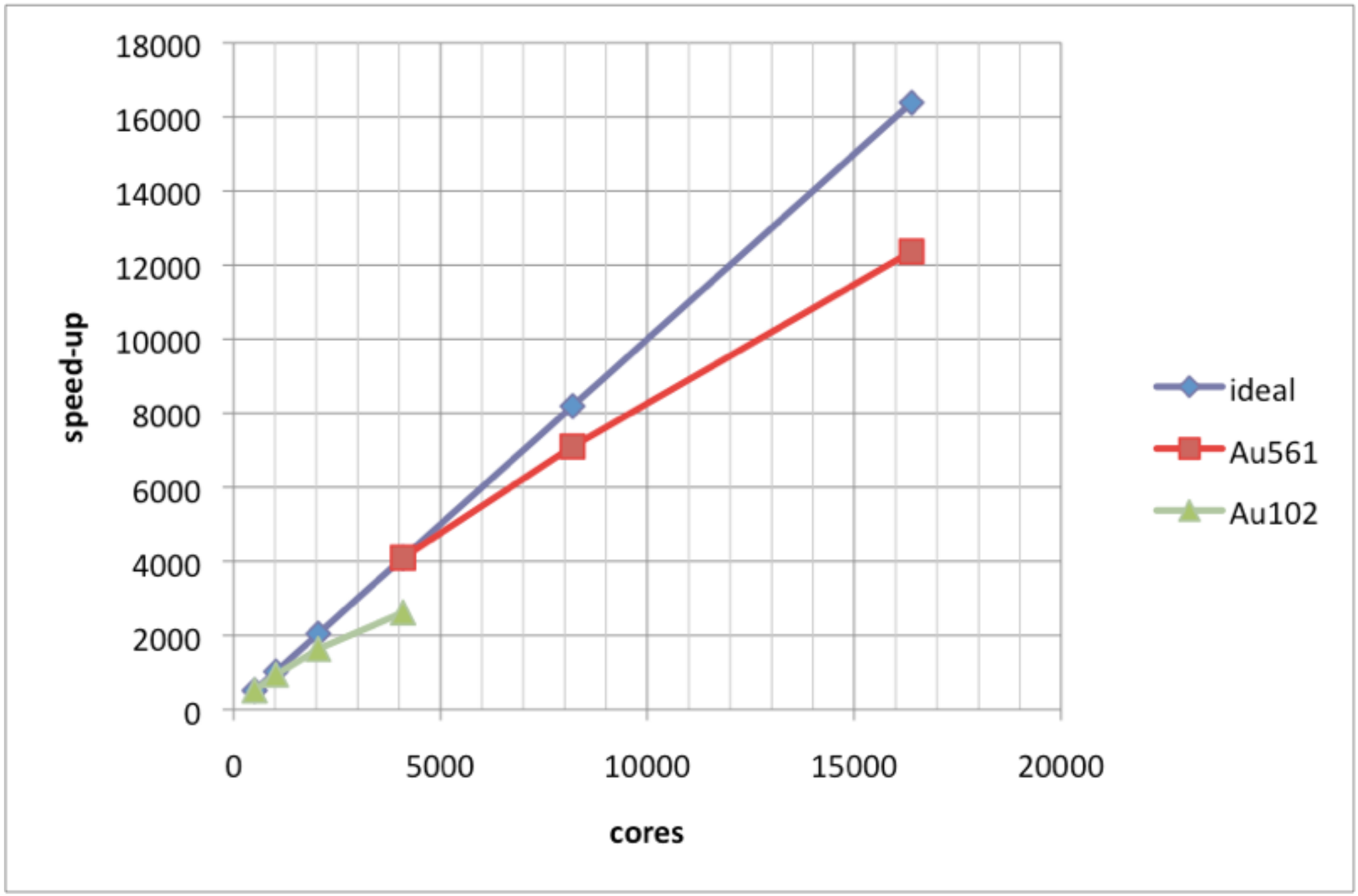
Science done with GPAW





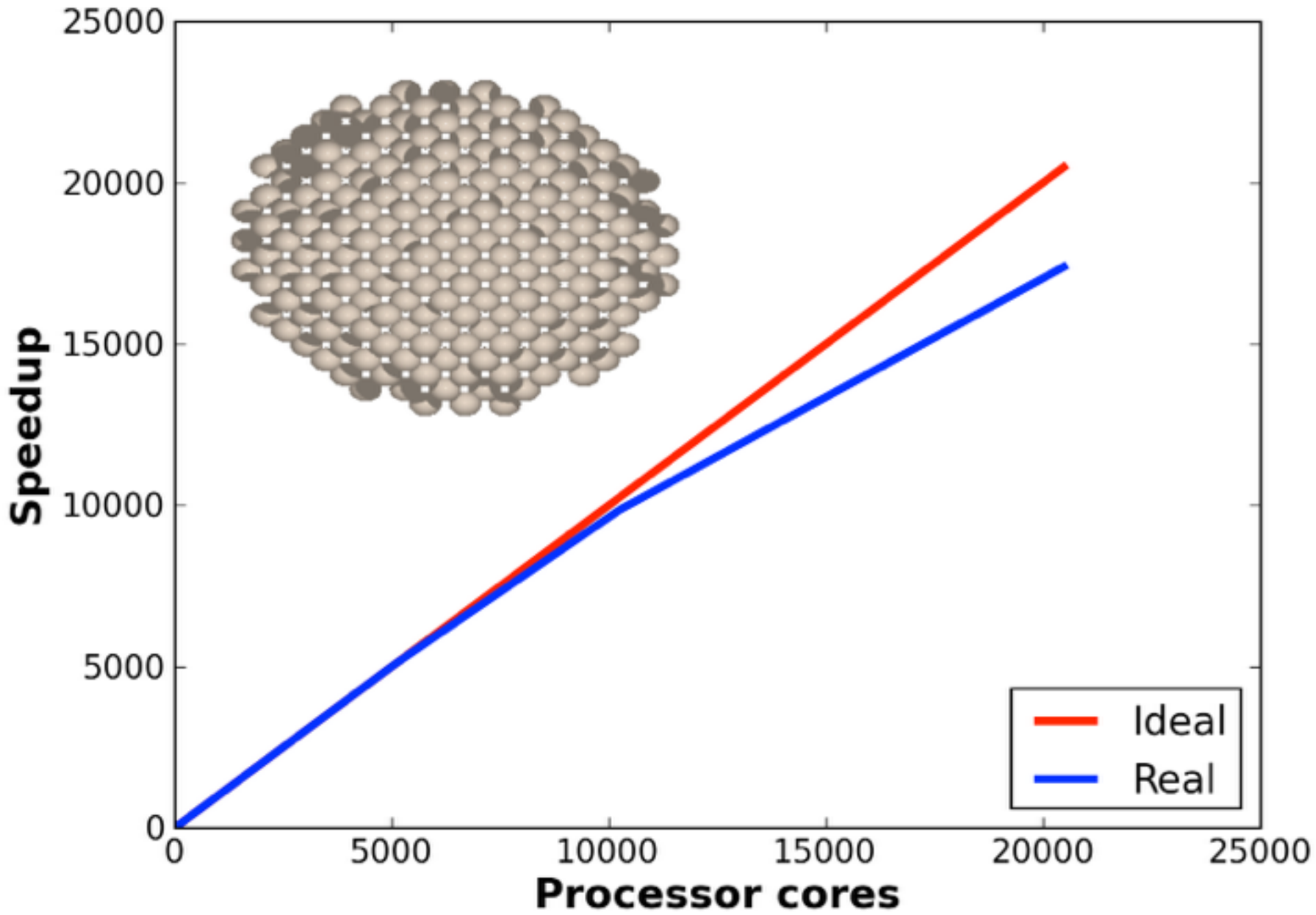
Nature Chemistry, PRL, JACS, PNAS, PRB, ...

GPAW Strong-scaling Results



Ground state DFT on Blue Gene P

GPAW Strong-scaling Results



TD-DFT on Cray XT5

Special operating systems

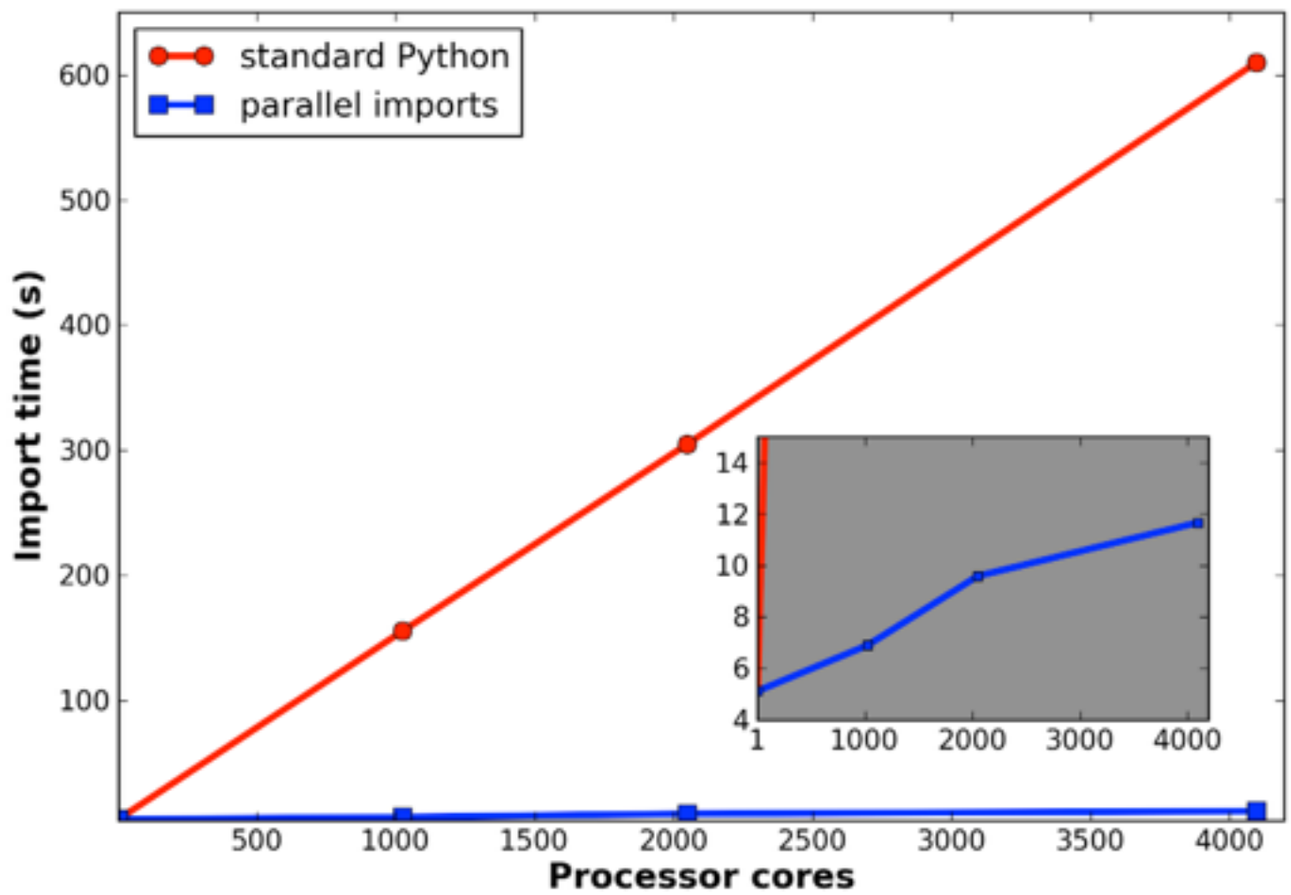
- Some supercomputing systems (BG, Cray XT) have special light-weight kernels on compute nodes
- Lack of "standard" features
  - dynamic libraries
  - lots of missing system calls
  - did we mention all I/O is forwarded?
- Python relies heavily on dynamic loading
  - static build of Python (including all needed C-extensions) is possible
  - modification of CPython is needed for correct namespace resolution
  - See wiki.fysik.dtu.dk/gpaw/install/Cray/jaguar.html for some details
- Cross-compilation can be challenging - disttools is evil

Python's import mechanism and parallel scalability

- import statement triggers lots of metadata traffic
  - directory accesses, opening and closing files
- parallel filesystems deal well only with large files/data
- There is considerably amount of imports already during Python initialization (and yes, we trim site.py and the module search path)
  - Initialization overheads do not show up in the Python timers
- With > 1000 processes problem can be severe even in production calculations
  - with 8 racks (~32 000 cores) on Blue Gene /P Python start-up time can be 45 minutes!

Python's import mechanism and parallel scalability

- Possible solutions (all are sort of ugly)
  - Put all the Python modules on a ramdisk
  - Hack CPython - only single process reads (module) files and broadcasts data to others with MPI
  - develop extreme patience



Questions?

Acknowledgments

This work is supported in part by the resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

Extended thanks to

- CSC
- Northwestern University
- De Paul University
- Sameer Shende, ParaTools, Inc.
- NumFocus for their continued support and sponsorship of SciPy and NumPy
- Lisandro Dalcin for his work on mpi4py and petsc4py
- ChiPy