Python for Scientific and High Performance Computing

SC10
New Orleans, Louisiana, United States

Monday, November 16, 2009 8:30AM - 5:00PM
http://www.mcs.anl.gov/~wscullin/python/tut/sc10
or
http://bit.ly/pytutsc10

disc image at: http://tinyurl.com/pyiso
tutorial files at: http://tinyurl.com/pytutorialsc10

Introductions

Your presenters:

- William R. Scullin
  - wscullin@alcf.anl.gov
- James B. Snyder
  - jbsnyder@northwestern.edu
- Massimo Di Pierro
  - mdipierro@cs.depaul.edu
- Jussi Enkovaara
  - jussi.enkovaara@csc.fi

NORTHWESTERN
UNIVERSITY

DEPAUL
UNIVERSITY

CSC – IT Center for Science

Overview

We seek to cover:

- Python language and interpreter basics
- Popular modules and packages for scientific applications
- How to improve performance in Python programs
- How to visualize and share data using Python
- Where to find documentation and resources

Do:

- Feel free to interrupt
  - the slides are a guide - we're only successful if you learn what you came for; we can go anywhere you'd like
- Ask questions
- Find us after the tutorial
- http://www.web2py.com/sc10survey

About the Tutorial Environment

Updated materials and code samples are available at the url on the title page. We suggest you retrieve them before proceeding. They should remain posted for at least a calendar year.

Along with the exercise packets at your seat, you will find instructions on using a bootable DVD with the full tutorial materials and all software pre-installed. You are welcome to use your own linux environment, though we cannot help you with setup issues. You will need:

- Python 2.6.5 or later
- NumPy 1.3.1 or later
- mpi4py 1.0.0 or later
- matplotlib 1.0.0 or later
- iPython (recommended)
- web2py 1.85 or later

Outline

1. Introduction
   - Introductions
   - Tutorial overview
   - Why Python and why in scientific and high performance computing?
   - Setting up for this tutorial
2. Python basics
   - Interpreters
   - data types, keywords, and functions
   - Control Structures
   - Exception Handling
   - I/O
   - Modules, Classes and OO
3. SciPy and NumPy: fundamentals and core components
4. Parallel and distributed programming
5. Performance
   - Best practices for pure Python + NumPy
   - Using profilers
   - Optimizing when necessary
6. Real world experiences and techniques
7. Python for plotting, visualization, and data sharing
   - Overview of matplotlib
   - Example of MC analysis tool
8. Where to find other resources
   - There's a Python BOF!
9. Final exercise
10. Final questions
11. Acknowledgments

- Dynamic programming language
- Interpreted & interactive
- Object-oriented
- Strongly introspective
- Provides exception-based error handling
- Comes with "Batteries included" (extensive standard libraries)
- Easily extended with C, C++, Fortran, etc...
- Well documented (http://docs.python.org/)

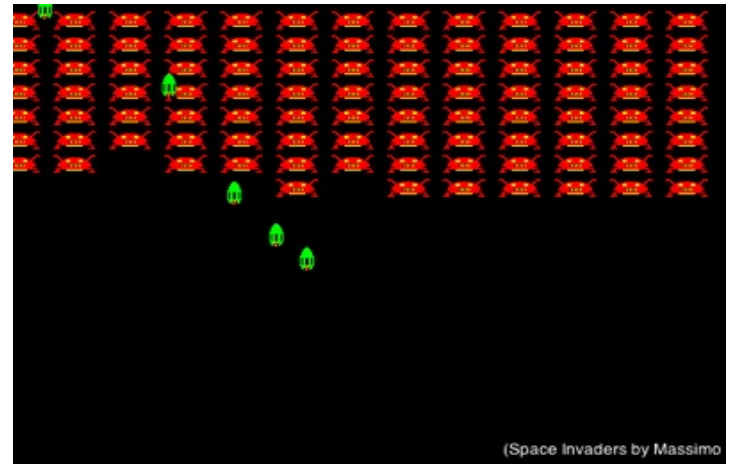
Easy to learn

```
#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
```

Try to do this in C++

```
from Tkinter import Tk, Label, Canvas, PhotoImage
import math, time
root = Tk()
canvas, aliens, missiles = Canvas(root,width=800,height=400,bg='white'), {}, {}
canvas.pack()
i1, i2 = PhotoImage(format='gif',file="alien.gif"), PhotoImage(format='gif',file="missile.gif")
for x,y,p in [(100+40*j,160-20*i,100*i) for i in range(8) for j in range(15)]:
    aliens[canvas.create_image(x,y,image=i1)]=p
canvas.bind('<Button-1>', lambda e: missiles.update({canvas.create_image(e.x,390,image=i2):10}))
while aliens:
    try:
        for m in missiles:
            canvas.move(m,0,-5)
            if canvas.coords(m)[1]<0:
                score -= missiles[m];
                canvas.delete(m); del missiles[m]
        for a in aliens:
            canvas.move(a,2.0*math.sin(time.time()),0)
            p = canvas.coords(a)
            items = canvas.find_overlapping(p[0]-5,p[1]-5,p[0]+5,p[1]+5)
            for m in items[1:2]:
                canvas.delete(a); del aliens[a]; canvas.delete(m); del missiles[m]
        time.sleep(0.02); root.update()
    except: pass
```



- only 24 lines of python code
- uses standard Python libraries.

- "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 (many are MIT/BSD license)
- Strong community and commercial support options.
- No license management

Science Tools for Python

Large number of science-related modules:

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

Please start the Tutorial Environment

Let the presenters know if you have any issues.

Start an iPython session:

```
santaka:~> wscullin$ ipython
Python 2.6.2 (r262:71600, Sep 30 2009, 00:28:07)
[GCC 3.3.3 (SuSE Linux)] on linux2
Type "help", "copyright", "credits" or "license" for more
information.

IPython 0.9.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ??
prints more.

In [1]:
```

Python Basics

- Interpreter
- Built-in Types, keywords, functions
- Control Structures
- Exception Handling
- I/O
- Modules, Classes & OO

Interpreters

- CPython Standard python distribution
  - What most people think of as "python"
  - highly portable
  - http://www.python.org/download/
    - We're going to use 2.6.x for this tutorial
    - The future is 3.x, the future isn't here yet
- iPython
  - A user friendly interface for testing and debugging
  - http://ipython.scipy.org/moin/

Other Interpreters You Might See...

- Unladen Swallow
  - Blazing fast, uses llvm and in turn may compile!
  - x86/x86_64 only really
  - Sponsored by Google
  - http://code.google.com/p/unladen-swallow/
- Jython
  - Python written in Java and running on the JVM
  - http://www.jython.org/
  - performance is about what you expect
- IronPython
  - Python running under .NET
  - http://www.codeplex.com/IronPython
- PyPy
  - Python in... Python
  - A serious contender that can use CPython modules
  - http://codespeak.net/pypy/dist/pypy/doc/

CPython Interpreter Notes

- Compilation affects interpreter performance
  - Precompiled distributions aim for compatibility and as few irritations as possible, not performance
    - compile your own or have your systems admin do it
    - same note goes for most modules
  - Regardless of compilation, you'll have the same bytecode and the same number of instructions
  - Bytecode is portable, binaries are not
  - ESSL, MKL and other libraries can effect

- Portability is actually pretty good these days - most modules work on most platforms - but always run validation tests and read the release notes.

A note about distutils and building modules

Unless your environment is very generic (ie: a major linux distribution under x86/x86_64) with everything installed in default locations, manual compilation and installation of modules is a very good idea.

Distutils and setuptools often make incorrect assumptions about your environment in HPC settings. Your presenters generally regard distutils as evil as they cross-compile a lot.

If you are running on PowerPC, IA-64, Sparc, or in an uncommon environment, let module authors know you're using their software and report problems. It's just being a good citizen in the community.

Understanding CPython

CPython was written to be easy to port and maintain. It was started in an era when people hadn't really thought about running across thousands of processors and using distributed storage...

It was designed from the beginning to provide an interactive interpreter. This is both a blessing and a curse.
Likewise, portability and expandability was taken to be very important - and once again this is has been a blessing and a curse.

So let's look at the impact...

Starting up Python

When Python starts there are a handful of environment variables that are particularly useful:

Shell Variable	Flag	Effect
PYTHONPATH		add a colon separated list of directories to the search path
PYTHONHOME		change the prefix of the library and module search path
PYTHONSTARTUP		the location of a readable file with python commands that will be executed before you get a prompt in an interactive session
PYTHONDEBUG	-d	output python parser debugging information
PYTHONNOUSERSITE	-s	disable user specific module search paths
	-S	don't perform import site at startup
PYTHONUNBUFFERED	-u	use unbuffered output for stderr and stdout
PYTHONDONTWRITEBYTECODE	-B	don't produce .pyc and .pyo files
PYTHONVERBOSE	-v	be verbose about importing modules, if the variable is set to an integer or the flag is repeated it raises the verbosity of the import.

Starting up

The interpreter calls:

```
zipimport  #allows for the importation of zipped module files
site.py       # holds the site specific information regarding
```

Everything is an object

and all Objects are the PyObject type in CPython's internals...

Just a record of size and a pointer to a datatype.

```
PyObject *
_PyObject_New(PyTypeObject *tp)
{
    PyObject *op;
    op = (PyObject *) PyObject_MALLOC(_PyObject_SIZE(tp));
    if (op == NULL)
        return PyErr_NoMemory();
    return PyObject_INIT(op, tp);
}

PyVarObject *
_PyObject_NewVar(PyTypeObject *tp, Py_ssize_t nitems)
{
    PyVarObject *op;
    const size_t size = _PyObject_VAR_SIZE(tp, nitems);
    op = (PyVarObject *) PyObject_MALLOC(size);
    if (op == NULL)
        return (PyVarObject *)PyErr_NoMemory();
    return PyObject_INIT_VAR(op, tp, nitems);
}
```

This has many implications...

Built-in Numeric Types

int, **float**, **long**, **complex** - different types of numeric data

```
>>> a = 1.2  # set a to floating point number
>>> type(a)
<type 'float'>

>>> a = 1  # redefine a as an integer
>>> type(a)
<type 'int'>

>>> a = 1e-10  # redefine a as a float with scientific notation
>>> type(a)
<type 'float'>

>>> a = 1L  # redefine a as a long
>>> type(a)
<type 'long'>

>>> a = 1+5j  # redefine a as complex
>>> type(a)
<type 'complex'>
```

Gotchas with Built-in Numeric Types

Python's int and float can become as large in size as your memory will permit, but ints will be automatically typed as long. The built-in long datatype is very slow and best avoided.

```
>>> a=2.0**999
>>> a
5.3575430359313366e+300

>>> import sys
>>> sys.maxint
2147483647
>>> a>sys.maxint
True
>>> a=2.0**9999
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Result too large')
>>> a=2**9999
>>> a-((2**9999)-1)
1L
```

Gotchas with Built-in Numeric Types

Python's int and float are not decimal types

- IEEE 754 compliant (http://docs.python.org/tutorial/floatingpoint.html)
- math with two integers always results in an integer

```
>>> a=1/3   # no type coercion, integer division
>>> a
0

>>> a=1/3.0  # binary op with int and float -> int coerced to float
>>> a
0.3333333333333331
>>> a=1.0/3.0  # float division
>>> a
0.3333333333333331

>>> 0.3
0.2999999999999999 # thanks binary fractions!
>>> a=1.0/10
>>> a
0.10000000000000001
```

NumPy Numeric Data Types

NumPy covers all the same numeric data types available in C/C++ and Fortran as variants of int, float, and complex

- all available signed and unsigned as applicable
- available in standard lengths
- floats are double precision by default
- generally available with names similar to C or Fortran
  - ie: long double is longdouble
- generally compatible with Python data types
- follow endianness of the platform

Built-in Sequence Types

str, **unicode** - string types

```
>>> s = 'asd'
>>> u = u'fgh'  # prepend u, gives unicode string
>>> s[1]
's'
```

list - mutable sequence

```
>>> l = [1,2,'three']  # make list
>>> type(l[2])
<type 'str'>

>>> l[2] = 3;   # set 3rd element to 3
>>> l.append(4)  # append 4 to the list
```

tuple - immutable sequence

```
>>> t = (1,2,'four')
```

Built-in Mapping Type

dict - match any immutable value to an object

```
>>> d = {'a' : 1, 'b' : 'two'}
>>> d['b']   # use key 'b' to get object 'two'
'two'

# redefine b as a dict with two keys
>>> d['b'] = {'foo' : 128.2, 'bar' : 67.3}
>>> d
{'a': 1, 'b': {'bar': 67.299999999999997, 'foo':
128.19999999999999}}

# index nested dict within dict
>>> d['b']['foo']
128.19999999999999

# any immutable type can be an index
>>> d['b'][(1,2,3)]='numbers'
```

Built-in Sequence & Mapping Type Gotchas

Python lacks C/C++ or Fortran style arrays.

- Best that can be done is nested lists or dictionaries
  - Tuples, being immutable are a bad idea
- You have to be very careful on how you create them
- Growing these types will cost performance (minimal pre-allocation)
- NumPy provides real n-dimensional arrays with low overhead

Python requires that your correctly: indent your code.

- Only applies to indentation
- Will help keep your code readable
- Use 4 spaces for tabs, and you won't have any problems (if you indent correctly)

If you have further questions, see PEP 8:
http://www.python.org/dev/peps/pep-0008/

Control Structures

if - compound conditional statement

```
if (a and b) or (not c):
    do_something()
elif d:
    do_something_else()
else:
    print "didn't do anything"
```

while - conditional loop statement

```
i = 0
while i < 100:
    i += 1
```

Control Structures

for - iterative loop statement

```
for item in list:
    do_something_to_item(item)
```

```
# start = 0, stop = 10
>>> for element in range(0,10):
...     print element,
0 1 2 3 4 5 6 7 8 9
```

```
# start = 0, stop = 20, step size = 2
>>> for element in range(0,20,2):
...     print element,
0 2 4 6 8 10 12 14 16 18
```

Generators

Python makes it very easy to write funtions you can iterate over- just use yield instead of return at the end of functions

```
def squares(lastterm):
    for n in range(lastterm):
        yield n**2

>>> for i in squares(4): print i
...
0
1
4
9
16
```

List Comprehensions

List Comprehensions are powerful tool, replacing Python's lambda function for functional programming

- syntax: `[f(x) for x in generator]`
- you can add a conditional `if` to a list comprehension

```
>>> [i for i in squares(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> [i for i in squares(10) if i%2==0]
[0, 4, 16, 36, 64]

>>> [i for i in squares(10) if i%2==0 and i%3==1]
[4, 16, 64]
```

Exception Handling

try - compound error handling statement

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by
zero

>>> try:
...     1/0
... except ZeroDivisionError:
...     print "Oops! divide by zero!"
... except:
...     print "some other exception!"

Oops! divide by zero!
```

File I/O Basics

Most I/O in Python follows the model laid out for file I/O and should be familiar to C/C++ programmers.

- basic built-in file i/o calls include
  - `open(), close()`
  - `write(), writeline(), writelines()`
  - `read(), readline(), readlines()`
  - `flush()`
  - `seek() and tell()`
  - `fileno()`
- basic i/o supports both text and binary files
- POSIX like features are available via `fctl` and `os` modules
- be a good citizen
  - if you open, close your descriptors
  - if you lock, unlock when done

Basic I/O examples

```
# open a text file for reading with default buffering
>>> f=file.open('myfile.txt','r')
```

```
# opens a text file for reading and writing with no buffering.
>>> f=file.open('myfile.txt','w+',0)
```

Let's write ten integers to disk without buffering, then read them back:

```
>>> f=open('frogs.dat','w+',0) # open for unbuffered reading and writing
>>> f.writelines([str(my_int) for my_int in range(10)])
>>> f.tell() # we're about to see we've made a mistake
10L # hmm... we seem short on stuff
>>> f.seek(0) # go back to the start of the file
>>> f.tell() # make sure we're there
0L
>>> f.readlines()  # Let's see what's written on each line
['0123456789'] # we've written 10 chars, no line returns... oops
>>> f.seek(0) # jumping back to start, let's add line returns
>>> f.writelines([str(my_int)+'\n' for my_int in range(10)])
>>> f.tell() # jumping back to start, let's add line returns
20L
>>> f.seek(0) # return to start of the file
>>> f.readline() # grab one line
'0\n'
>>>f.next()  # grab what ever comes next
'1\n'
>>> f.readlines() # read all remaining lines in the file
['2\n', '3\n', '4\n', '5\n', '6\n', '7\n', '8\n', '9\n']
>>> f.close()  # always clean up after yourself - no need other than courtesy!
```

Pickling

a.k.a.: serializing Python objects

```
# make a list w/ numeric values, a string, and a dict
>>> a = [1, 3, 5, 'hello', {'key':'value', 'otherkey':'othervalue'}]

# use pickle to serialize and dump to a file
>>> import pickle
>>> pickle.dump(a,open('filename.pickle','wb'))

# unpickle serialized data
>>> b=pickle.load(open('filename.pickle','rb'))
>>> b
[1, 3, 5, 'hello', {'otherkey': 'othervalue', 'key': 'value'}]
```

I/O for scientific formats

I/O is fairly basic out of the box - but there are modules to support many formats:

- h5py
  - Python bindings for HDF5
  - http://code.google.com/p/h5py/
- netCDF4
  - Python bindings for NetCDF
  - http://netcdf4-python.googlecode.com/svn/trunk/docs/netCDF4-module.html
- mpi4py allows for classic MPI-IO via MPI.File

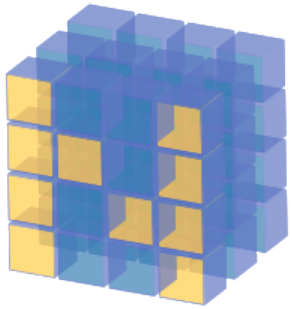
Modules

import - load module, define in namespace

```
>>> import random # import module
>>> random.random()  # execute module method
0.8258545378964787

>>> import random as rd # import and rename
>>> rd.random()
0.2271554216424868 1

# bring randint into namespace from random
>>> from random import randint
>>> randint(0,10)
4
```

Classes & Object Orientation

```
>>> class SomeClass:
...       """A simple example class""" # docstring
...       pi = 3.14159 # attribute
...       def __init__(self, ival=89): # init w/ default
...           self.i = ival
...       def f(self): # class method
...           return 'Hello'
>>> c = SomeClass(42)  # instantiate
>>> c.f() # call class method
'hello'

>>> c.pi = 3 # change attribute

>>> print c.i # print attribute
42
```



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))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

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

Parallel & Distributed Programming

threading

- useful for certain concurrency issues, not usable for parallel computing due to Global Interpreter Lock (GIL)

subprocess

- relatively low level control for spawning and managing processes

multiprocessing - multiple Python instances (processes)

- basic, clean multiple process parallelism

MPI

- mpi4py exposes your full local MPI API within Python
- as scalable as your local MPI

GPU (OpenCL & CUDA)

- PyOpenCL and PyCUDA provide low and high level abstraction for highly parallel computations on GPUs

Python Threading

Python threads

- real POSIX threads
- share memory and state with their parent processes
- do not use IPC or message passing
- light weight
- generally improve latency and throughput
- there's a heck of a catch, one that kills performance...

The Infamous GIL

To keep memory coherent, Python only allows a single thread to run in the interpreter's space at once. This is enforced by the Global Interpreter Lock, or GIL. It also kills performance for most serious workloads.

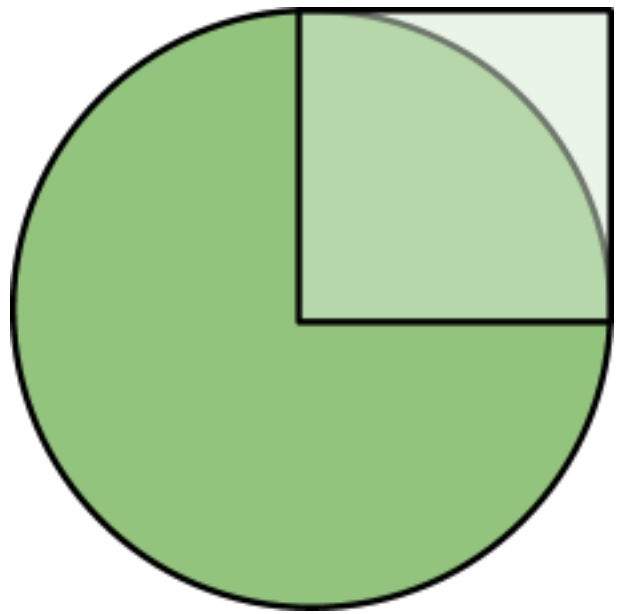
It's not all bad. The GIL:

- Is mostly sidestepped for I/O (files and sockets)
- Makes writing modules in C much easier
- Makes maintaining the interpreter much easier
- Makes for any easy target of abuse
- Gives people an excuse to write competing threading modules (please don't)

For the gory details See David Beazley's talk on the GIL: http://blip.tv/file/2232410

Implementation Example: Calculating Pi

- Generate random points inside a square
- Identify fraction (f) that fall inside a circle with radius equal to box width
  - $x^2 + y^2 < r$
- Area of quarter of circle (A) = $pi*r^2 / 4$
- Area of square (B) = r^2
- A/B = f = pi/4
- pi = 4f



Calculating pi with threads

```
from threading import Thread, Lock
import random

lock = Lock() # lock for making operations atomic

def calcInside(nsamples,rank):
    global inside # we need something everyone can share
    random.seed(rank)
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x*x)+(y*y)<1:
            lock.acquire() # GIL doesn't always save you
            inside += 1
            lock.release()

if __name__ == '__main__':
    nt=4 # thread count
    inside = 0 # you need to initialize this
    samples=int(12e6/nt)
    threads=[Thread(target=calcInside, args=(samples,i)) for i in range(nt)]

    for t in threads: t.start()
    for t in threads: t.join()

    print (4.0*inside)/(1.0*samples*nt)
```

Execution Time
nt=1: 15.45±0.22 sec
nt=2: 55.38±0.46 sec

Mac OS X, Python 2.6
Core 2 2.53 GHz

Subprocess

The `subprocess` module allows the Python interpreter to spawn and control processes. It is unaffected by the GIL. Using the `subprocess.Popen()` call, one may start any process you'd like.

```
>>> pi=subprocess.Popen('python -c "import math; print
math.pi"',shell=True,stdout=subprocess.PIPE)
>>> pi.stdout.read()
'3.14159265359\n'
>>> pi.pid
1797
>>> me.wait()
0
```

It goes without saying, there's better ways to do subprocesses...

Multiprocessing

- Added in Python 2.6
- Faster than threads as the GIL is sidestepped
- uses subprocesses
  - both local and remote subprocesses are supported
  - shared memory between subprocesses is risky
    - no coherent types
    - `Array` and `Value` are built in
    - others via multiprocessing.sharedctypes
  - IPC via pipes and queues
    - pipes are not entirely safe
- synchronization via locks
- `Manager` allows for safe distributed sharing, but it's slower than shared memory

Calculating pi with multiprocessing

```
import multiprocessing as mp
import numpy as np
import random

processes = mp.cpu_count()
nsamples = int(12e6/processes)

def calcInside(rank):
    inside = 0
    random.seed(rank)
    for i in range(nsamples):
        x = random.random();
        y = random.random();
        if (x*x)+(y*y)<1:
            inside += 1
    return (4.0*inside)/nsamples

if __name__ == '__main__':
   pool = mp.Pool(processes)
   result = pool.map(calcInside, range(processes))
   print np.mean(result)
```

code: pi-multiproc.py

pi with multiprocessing, optimized

```
import multiprocessing as mp
import numpy as np

processes = mp.cpu_count()
nsamples = int(12e6/processes)

def calcInsideNumPy(rank):
    np.random.seed(rank)

    # "vectorized" sample gen, col 0 = x, col 1 = y
    xy = np.random.random((nsamples,2))
    return 4.0*np.sum(np.sum(xy**2,1)<1)/nsamples

if __name__ == '__main__':
    pool = mp.Pool(processes)
    result = pool.map(calcInsideNumPy, range(processes))
    print np.mean(result)
```

Execution Time

Unoptimized: 4.76±0.23 sec

Vectorized: 1.30±0.14 sec

code: pi-multiprocopt.py

mpi4py

- wraps your native mpi
  - prefers MPI2, but can work with MPI1
- works best with NumPy data types, but can pass around any serializable object
- provides all MPI2 features
- well maintained
- distributed with Enthought Python Distribution (EPD)
- requires NumPy
- portable and scalable
- http://mpi4py.scipy.org/

How mpi4py works...

- mpi4py jobs must be launched with mpirun
- each rank launches its own independent python interpreter
- 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 - management

- the import calls Init()
  - there is generally no reason to manually call mpi_init or mpi_init_thread.
  - calling Init() more than once violates the MPI standard and will cause an exception
  - use Is_initialized to test for initialization if uncertain
- MPI_Finalize() is automatically run when the interpreter exits
  - again there is generally no need to ever call Finalize()
  - use Is_finalized() to test for finalization if uncertain

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!

mpi4py - communicators and operators

Almost all collectives are supported

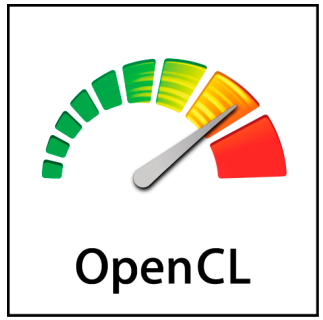
- Collectives operating on Python objects are naive
- For the most part collective reduction operations on Python objects are highly serial
- Notable missing collectives: Reduce_scatter() and Alltoallv()
- Convention applies: lowercased methods will work for general Python objects (albeit slowly) uppercase methods will work for NumPy/MPI data types at near C speed

Calculating pi with mpi4py

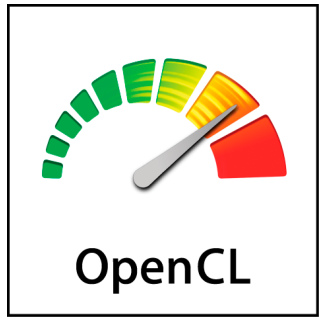
```
from mpi4py import MPI
import numpy as np
import random

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
```

code: *pi-mpi4py.py*

run: *mpirun python pi-mpi4py.py*



- Data & task-parallel computation on homogenous or heterogenous compute resources
- Open specification, maintained by Khronos Group
- C99 subset with added qualifiers, data types and functions
- IEEE-754 compliant rounding (not all hardware provides double-precision floats, though)
- Not device agnostic
  - If your implementation runs well on one platform, it may not on others



Computing Model

- **buffers**
  - used for holding data, shipping data between devices
- **kernels**
  - functions which will operate on buffers in parallel
- **queue**
  - control sequence of operations: transferring buffers, running kernels, waiting on events, etc..

PyOpenCL

- Convenient access to full OpenCL API from Python
- OpenCL errors translated into Python exceptions
- Object cleanup tied to object lifetime (follows Resource Acquisition Is Initialization)
- NumPy ndarrays interact easily with OpenCL buffers
  - as do other objects providing the Python buffer interface
- Include OpenCL kernels in-line in Python code, or use built-in element-wise operators (pyopencl.clmath)

pi with PyOpenCL

```
import pyopencl as cl
import pyopencl.clrandom
import numpy as np

nsamples = int(12e6)

# set up context and queue
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

# create array of random values in OpenCL
xy = pyopencl.clrandom.rand(ctx,queue,(nsamples,2),np.float32)

# square values in OpenCL
xy = xy**2

# 'get' method on xy is used to get array from OpenCL into ndarray
print 4.0*np.sum(np.sum(xy.get(),1)<1)/nsamples
```

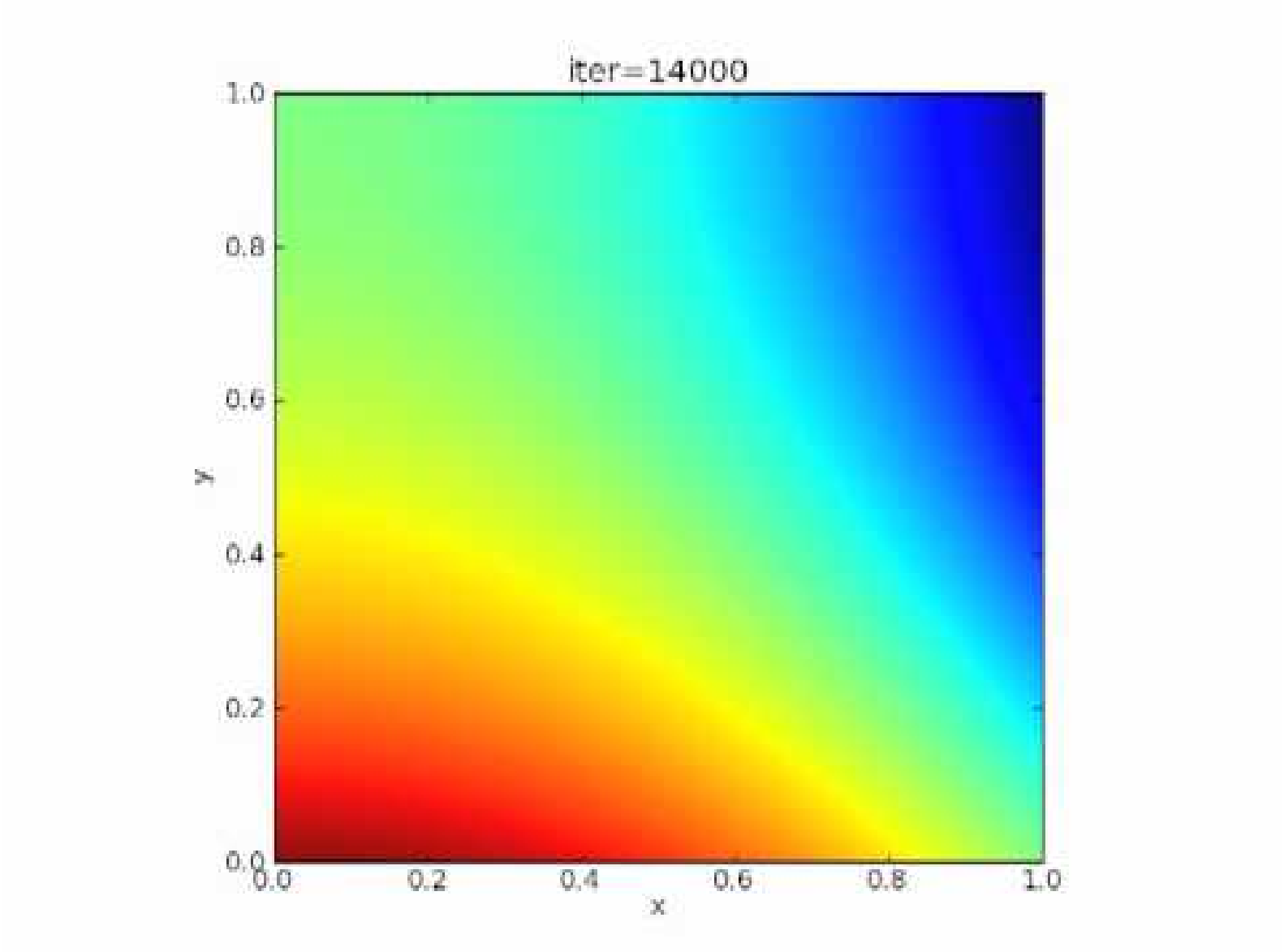
code: pi-opencl.py

2D Laplace Solver

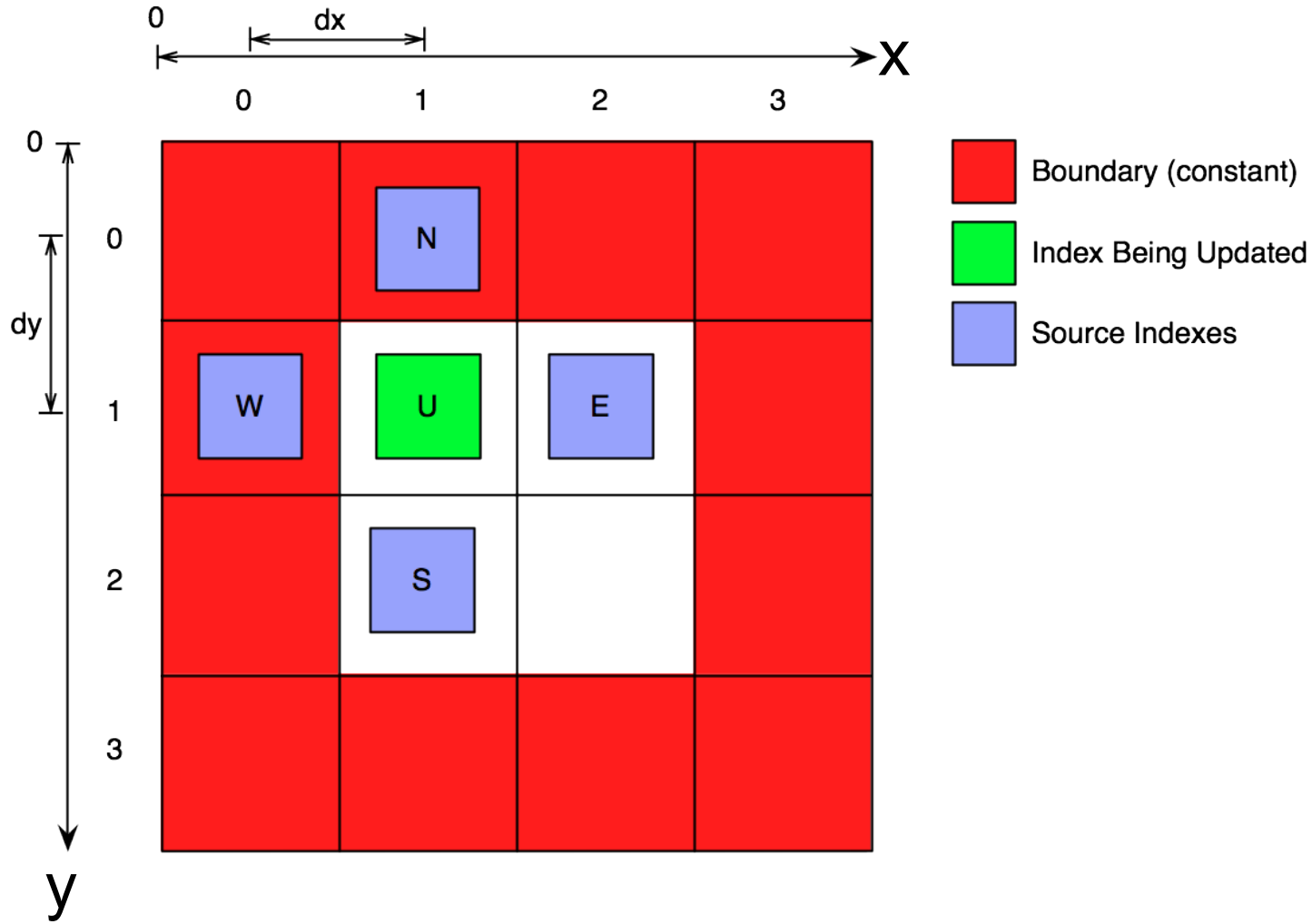
- Solve unknown function u(x,y) such that $\nabla^2 u = 0$ with boundary conditions.
- Map domain onto discretized rectangular grid, boundary conditions are placed at edges
- Use four-point averaging, Gauss-Seidel to solve the equation using an iterative finite-difference approach

Based on: http://www.scipy.org/PerformancePython

2D Laplace Solver



Based on: http://www.scipy.org/PerformancePython



$$U = \frac{(N + S)\,dy + (W + E)\,dx}{2(dy^2 + dx^2)}$$

2D Laplace Solver

- Simple Python solver implementation:

```
for i in range(1, nx-1):
    for j in range(1, ny-1):
        u[i,j] = ((u[i-1, j] + u[i+1, j])*dy**2 +
                  (u[i, j-1] + u[i, j+1])*dx**2)/(2.0*(dx**2 + dy**2))
```

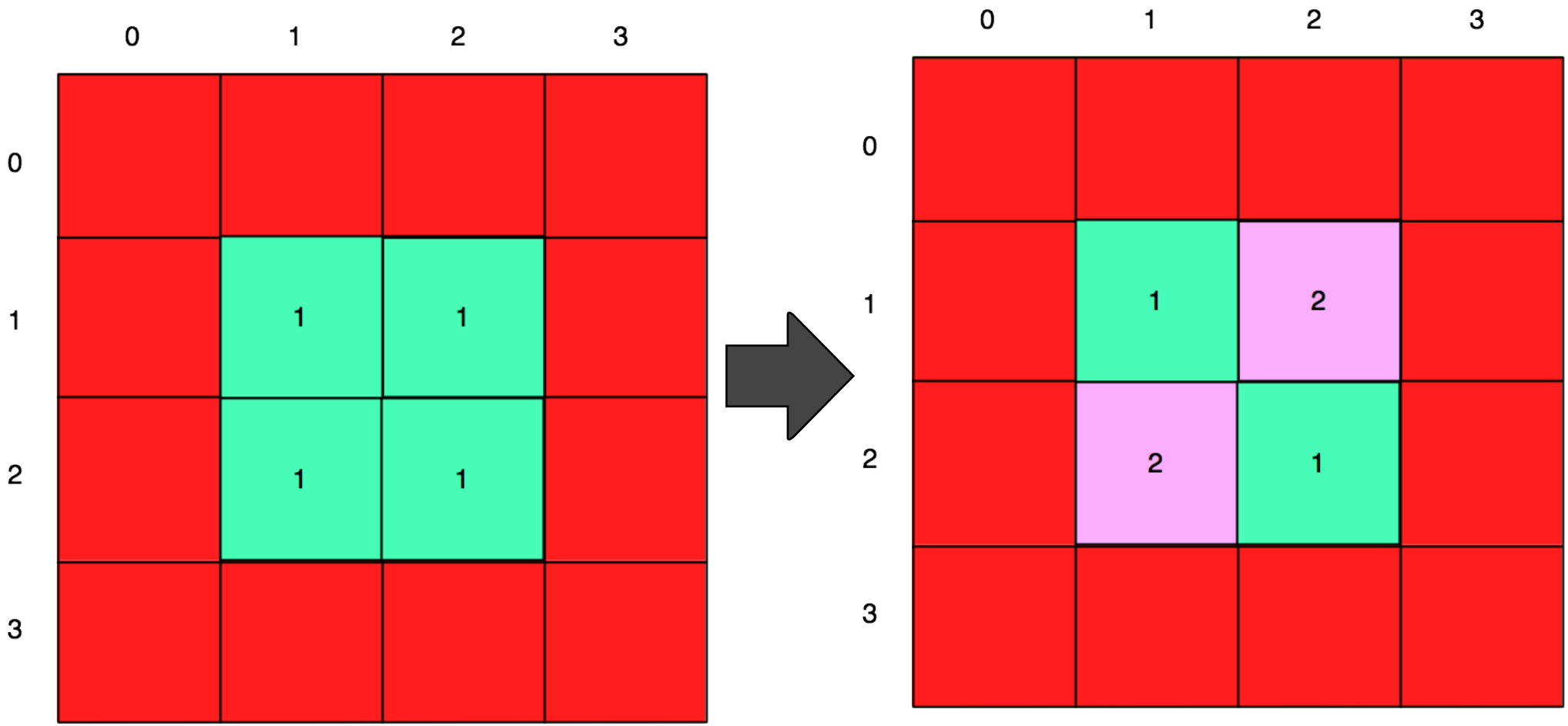
- NumPy implementation:

```
u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy**2 +
                 (u[1:-1,0:-2]  + u[1:-1, 2:])*dx**2)/(2.0*(dx**2 + dy**2))
```

Based on: http://www.scipy.org/PerformancePython

2D Laplace Solver

- Parallelizing by switching to red-black (checkerboard):



2D Laplace Solver

- Parallelizing by switching to red-black (checkerboard) solver, simple, slow Python implementation:

```
  def slowTimeStep(self, dt=0.0):
      g = self.grid
      nx, ny = g.u.shape
      dx2, dy2 = g.dx**2, g.dy**2
      dnr_inv = 0.5/(dx2 + dy2)
      u = g.u

err = 0.0;
      for offset in range(1,3):
        for i in range(1, ny-1):
          for j in range(1 + ( ( i + offset ) % 2), nx-1, 2):
              tmp = u[j,i]
              u[j,i] = ((u[j-1, i  ] + u[j+1, i  ])*dy2 +
                        (u[j  , i-1] + u[j  , i+1])*dx2)*dnr_inv
              diff = u[j, i] - tmp
              err += diff**2
      return np.sqrt(err)
```

- Parallelizing by switching to red-black (checkerboard) PyOpenCL implementation:

```
ef openclCheckerTimeStep(self, dt=0.0):
 nx, ny = self.grid.u.shape

 if self.count == 0:
    g = self.grid
    dx2, dy2 = g.dx**2, g.dy**2
    dnr_inv = 0.5/(dx2 + dy2)
    u = g.u

    self.err = np.empty( (ny-2,), dtype=np.float32)
    self.ctx = cl.Context(dev_type=cl.device_type.GPU)

    self.queue = cl.CommandQueue(self.ctx)

    mf = cl.mem_flags
    self.u_buf = cl.Buffer(self.ctx, mf.READ_WRITE, u.nbytes)
    self.err_buf = cl.Buffer(self.ctx, mf.READ_WRITE, self.err.nbytes)

    cl.enqueue_write_buffer(self.queue, self.u_buf, u).wait()
```

- Parallelizing by switching to red-black (checkerboard) PyOpenCL implementation:

```
= cl.Program(self.ctx, """
 void lp2dstep( __global float *u, __global float *err, const uint stidx )

mp, diff;
 get_global_id(0) + 1; # column comes from work item global id

dx == 1 )
1] = 0.0; # only zero out error once per red-black cycle

nt j = 1 + ( ( i + stidx ) %% 2 ); j<( %(nx)d-1 ); j+=2 )

 = u[%(ny)d*j + i];
%(ny)d*j + i] = \
(u[%(ny)d*(j-1) + i] + u[%(ny)d*(j+1) + i])*%(dy2)g +
(u[%(ny)d*j + i-1] + u[%(ny)d*j + i + 1])*%(dx2)g)*%(dnr_inv)g;
 = u[%(ny)d*j + i] - tmp;
i-1] += diff*diff;

 'nx': nx, # use a Python dictionary for formatting string/program
': ny,
2': dx2,
2': dy2,
r_inv': dnr_inv } ).build() # call build on program when we're done
```

- Parallelizing by switching to red-black (checkerboard) PyOpenCL implementation:

```
# Enqueue red step
lp1evt = self.prg.lp2dstep(self.queue, ((ny-2),), None, self.u_buf,
                           self.err_buf, np.uint32(1))
cl.enqueue_wait_for_events(self.queue, [lp1evt])

# Enqueue black step
lp2evt = self.prg.lp2dstep(self.queue, ((ny-2),), None, self.u_buf,
                           self.err_buf, np.uint32(2))
cl.enqueue_wait_for_events(self.queue, [lp2evt])

# Get updated error and return
cl.enqueue_read_buffer(self.queue, self.err_buf, self.err).wait()
return np.sqrt(np.sum(self.err))
```

Performance

- Best practices with pure Python & NumPy
- Optimization where needed (we'll talk about this in GPAW)
  - profiling
  - inlining

Python Best Practices for Performance

If at all possible...

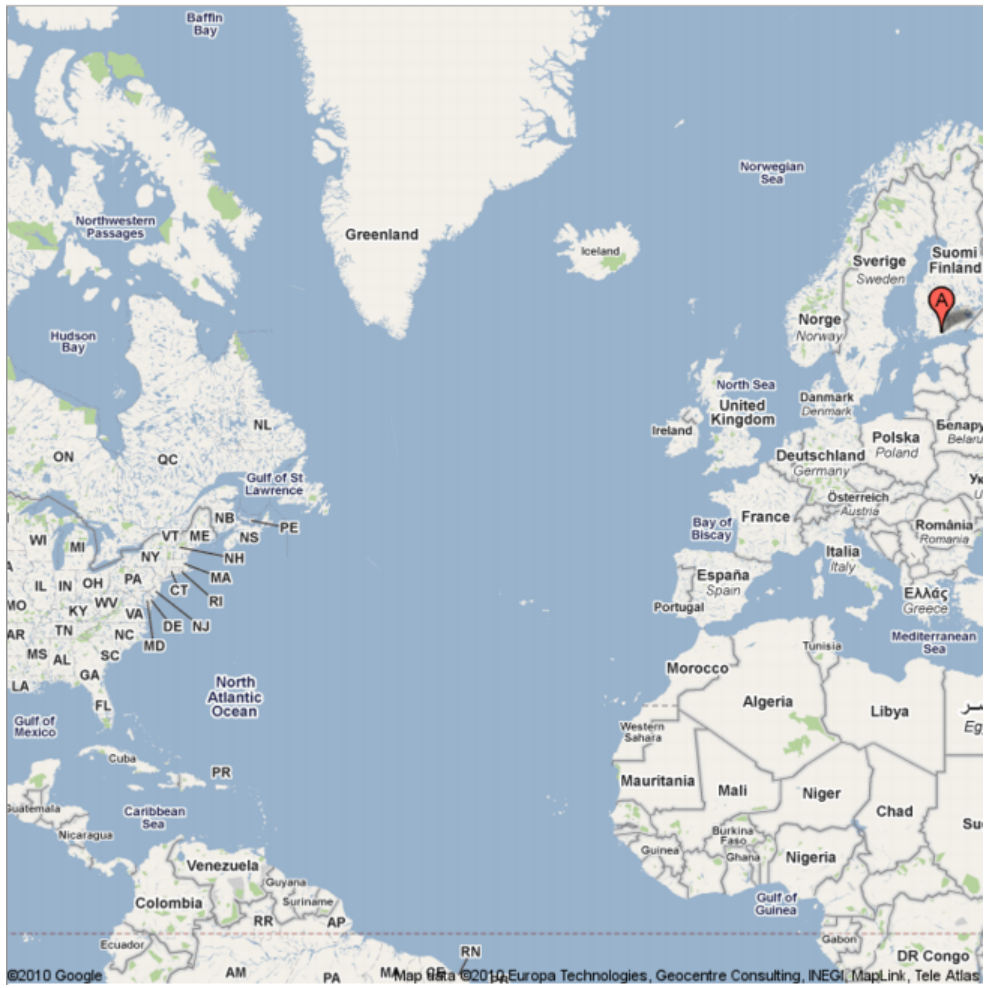
- Don't reinvent the wheel.
  - someone has probably already done a better job than your first (and probably third) attempt
- Build your own modules against optimized libraries
  - ESSL, ATLAS, FFTW, MKL
- Use NumPy data types & functions instead of built-in Python ones for homogeneous vectors/arrays
- "vectorize" operations on ≥1D data types.
  - avoid for loops, use single-shot operations
- Pre-allocate arrays instead of repeated concatenation
  - use numpy.zeros, numpy.empty, etc..

a massively parallel Python-C code
for electronic structure calculations

Jussi Enkovaara
CSC - IT Center for Science
Espoo, Finland

CSC - IT Center for Science

- Finnish national supercomputing center



Outline

- Overview of GPAW
  - performance and parallel scalability
- Programming model
  - Python, Numpy, C, libraries
- Challenges
  - "non-standard" operating systems (BG, Cray XT)
  - profiling and debugging
  - Python initialization
- Summary and outlook

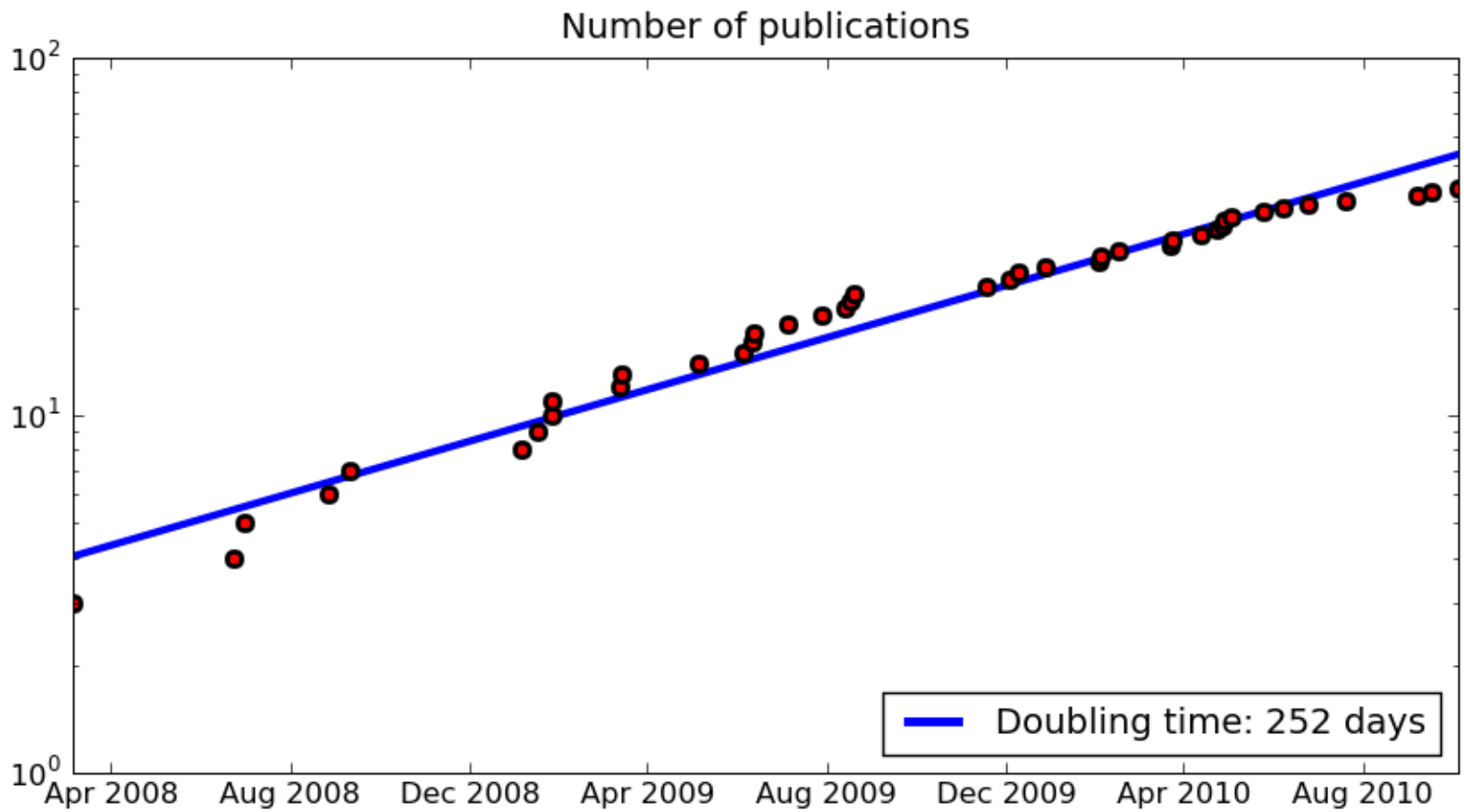
GPAW Overview

- *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
  - ~10^6 grid points, ~10^3 eigenvalues
- Written in Python and C using the NumPy library
- Massively parallel using MPI
- Open source (GPL)

wiki.fysik.dtu.dk/gpaw

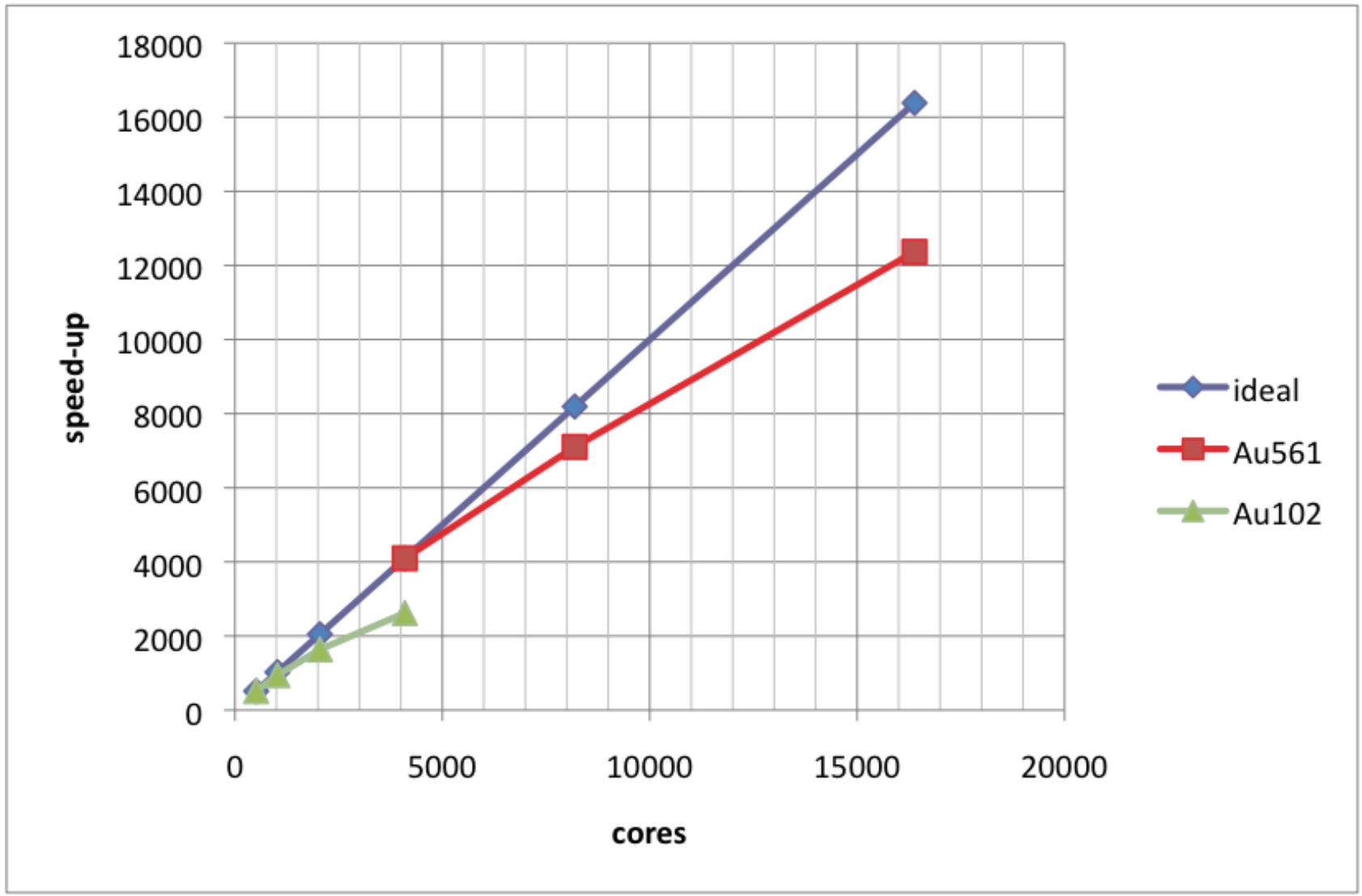
J. Enkovaara *et al.* J. Phys.: Condens. Matter **22**, 253202 (2010)

Science done with GPAW



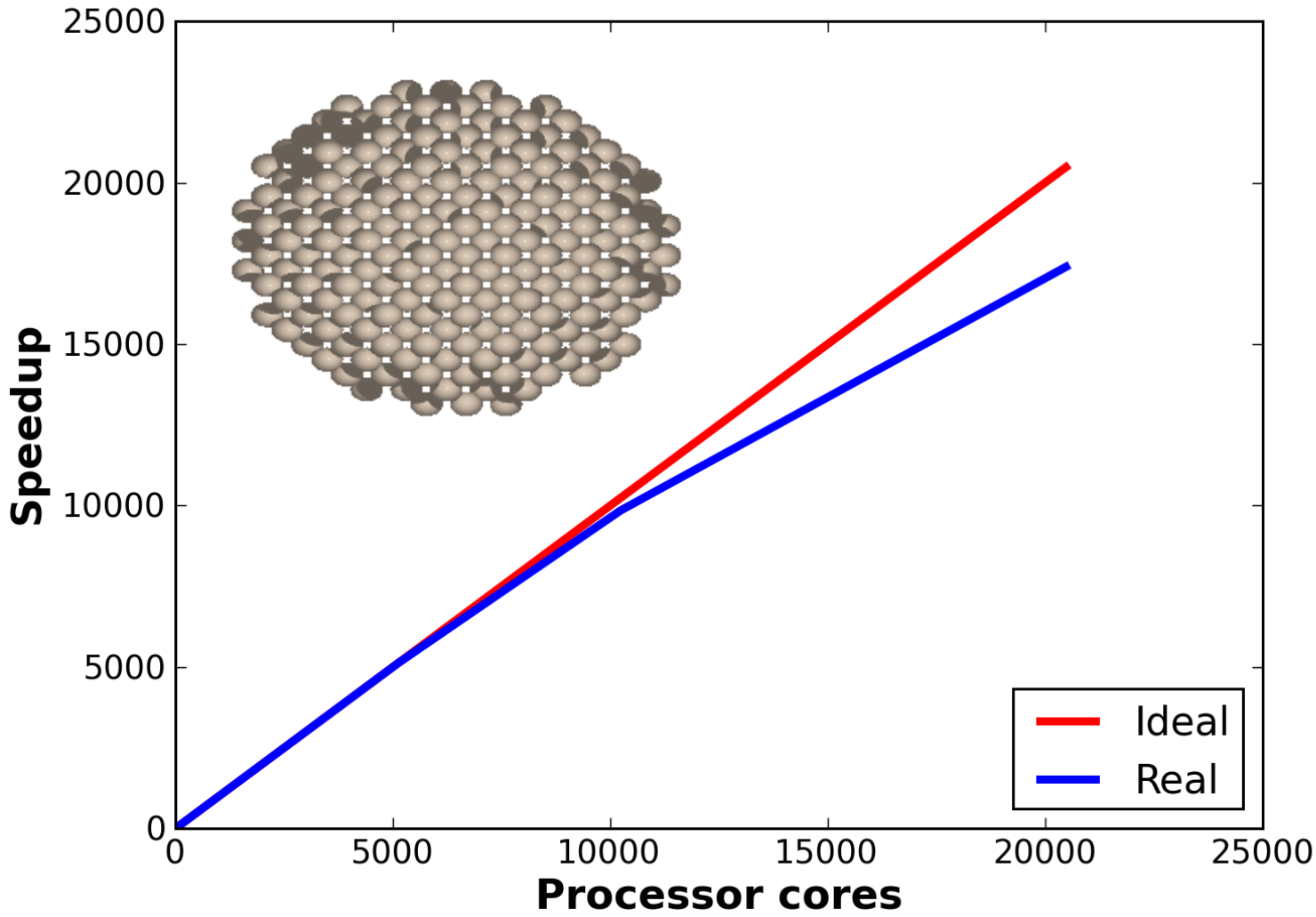
Nature Chemistry, PRL, JACS, PNAS, PRB, ...

GPAW Strong-scaling Results



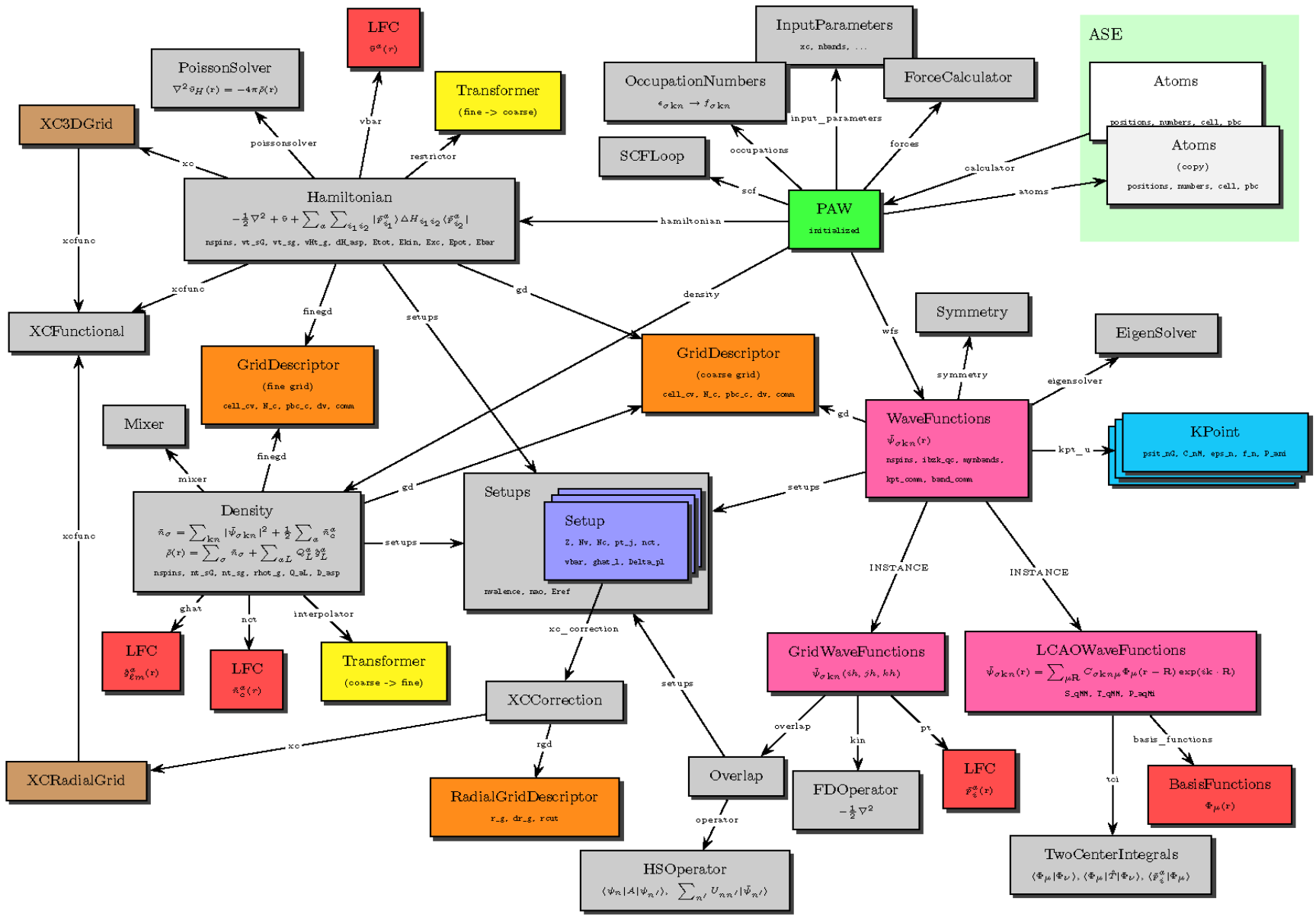
Ground state DFT on Blue Gene P

GPAW Strong-scaling Results



TD-DFT on Cray XT5

DFT is a complex algorithm!



Performance Mantra

People are able to code complex algorithms in much less time by using a high-level language like Python. *There can be a performance penalty in the most pure sense of the term.*

"The best performance improvement is the transition from the nonworking to the working state."
--John Ousterhout

"Premature optimization is the root of all evil."
--Donald Knuth

"You can always optimize it later."
-- Unknown

GPAW code structure

- Built on top of **NumPy** library.
- Not simply a Python wrapper on legacy Fortran/C code
- Python for coding the high-level algorithm
- C for coding numerical intense operations
- Use BLAS and LAPACK whenever possible

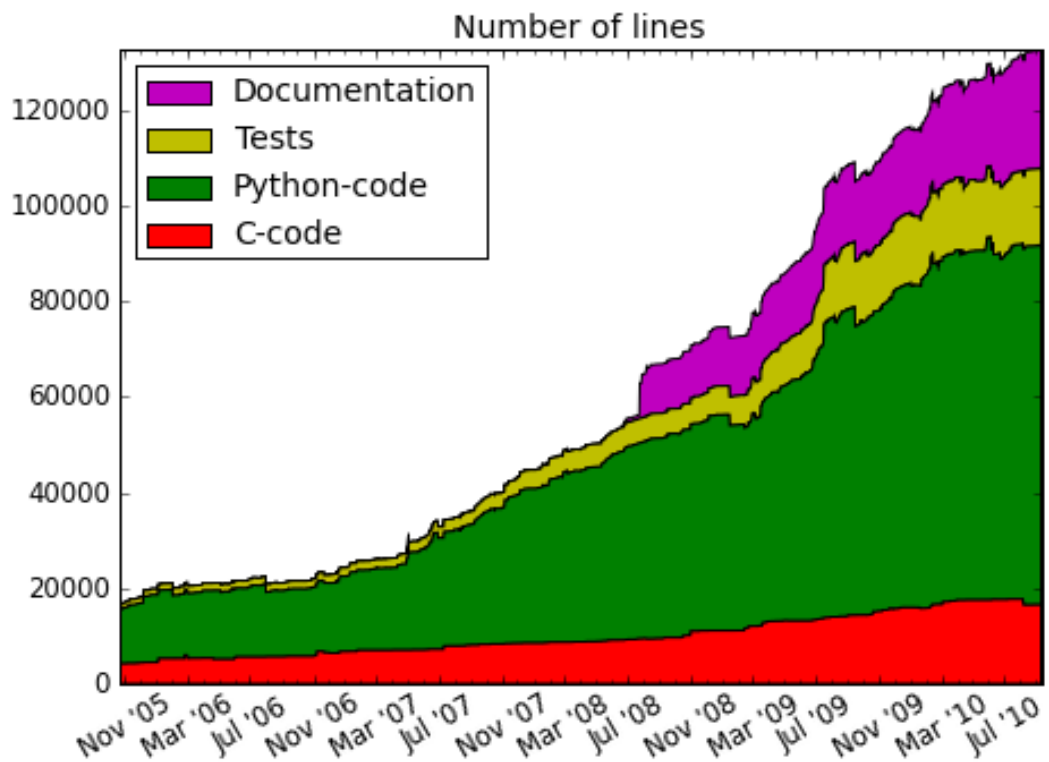
Sniplet from eigensolver:

```
 hamiltonian.kin.apply(psit_nG, Htpsit_nG)                    #
Call to C-function
 Htpsit_nG += psit_nG * hamiltonian.vt_G
 r2k(0.5 * self.gd.dv, psit_nG, Htpsit_nG, 1.0, H_nn)    #
Interface to BLAS
```

DFT algorithms are well-known and computationally intensive parts are known *a priori.*

Source Code Timeline

- **Mostly Python-code**, ~10% C-code.
- over 90% of wall-clock time spend in C, BLAS, LAPACK, MPI,...



NumPy - Weakly-typed data structures

Weakly-typed data structures are handy

- In DFT, both real and complex double precision are needed
  - Fortran77/Fortran90 - lots of if-statements and modules
  - C++ - handles this with templating and operator overloading
  - Python - doesn't care, but your C extensions will, however, that is only 10% of your code.

```
if kd.gamma:
    dtype=float
else:
    dtype=complex
H_nn = np.zeros((n, n),
dtype)
```

Memory overheads

- Python introduces some memory overheads
  - Python Interpreter ~ 12 MB
  - NumPy library ~ 8 MB
  - GPAW modules ~ 50 MB
  - some cases unexpted memory usage, Python's object allocation mechanism to blaim?
  - Together with OS and MPI overheads can add up to 100-150MB
- Blue Gene P has 512 MB per core
  - at worst case only 350 MB left for the actual application data!
- Less severe on architectures with more memory per core

NumPy - temporary arrays

Temporary arrays are created in expressions like:

```
for i in xrange(N):
    Y[i] += alpha*X[i]# temp. array for alpha * X[i]
    C[i] += A[i]*B[i] # temp. array for A[i] * B[i]
```

- Unexpexted memory usage
- Cache usage may worsen
- Fused multiply-add instructions provided by some architectures (e.g. PPC) cannot be used
- Vectorization depends on the NumPy build

NumPy - FLOPS

- Optimized BLAS available via NumPy **np.dot**
  - general inner product of multi-dimensional arrays.
- Very difficult to cross-compile. Blame distutils!
  - core/_dotblas.so is a sign of optimized np.dot
  - Python wrapper overhead is negligible
- For very large matrices (~50 MB), there is a big performance difference
  - unoptimized - 1% single core peak performance
  - optimized - 80% single core peak performance
- For matrix * vector products, np.dot can yield better performance than direct call to GEMV!

NumPy - FLOPS

WARNING: If you make heavy use of BLAS & LAPACK type operations...

- Plan on investing a significant amount of time cross-compiling optimized NumPy.
- Safest thing is to write your own C-wrappers.
- If all your NumPy arrays are < 2-dimensional, Python wrappers will be simple.
- Wrappers for multi-dimensional arrays can be challenging:
  - SCAL, AXPY is simple
  - GEMV more difficult
  - GEMM non-trivial
- Remember C & NumPy arrays are row-ordered by default, Fortran arrays are column-ordered!

Python BLAS Interface

```
void dscal_(int*n, double* alpha, double* x, int* incx); // C prototype for Fortran
void zscal_(int*n, void* alpha, void* x, int* incx);  // C prototype for Fortran
#define DOUBLEP(a) ((double*)((a)->data))  // Casting for NumPy data struc.
#define COMPLEXP(a) ((double_complex*)((a)->data))  //  Casting for NumPy data struc.

PyObject* scal(PyObject *self, PyObject *args)
{
  Py_complex alpha;
  PyArrayObject* x;
  if (!PyArg_ParseTuple(args, "DO", &alpha, &x))
    return NULL;
  int n = x->dimensions[0];
  for (int d = 1; d < x->nd; d++)  // NumPy arrays can be multi-dimensional!
    n *= x->dimensions[d];
  int incx = 1;

  if (x->descr->type_num == PyArray_DOUBLE)
    dscal_(&n, &(alpha.real), DOUBLEP(x), &incx);
  else
    zscal_(&n, &alpha, (void*)COMPLEXP(x), &incx);
  Py_RETURN_NONE;
}
```

Parallelization

- Message passing with MPI
- Custom Python interface to needed MPI routines
  - MPI calls both from C and from Python

```
hamiltonian.apply(psi, hpsi)  # MPI calls within the apply C-function
norm = gd.comm.sum(np.vdot(psi,psi)) # Python interface to MPI Reduce
```

- Lots of parallelization concerns can be hidden from the high level algorithms
- Python syntax and more higher level of programming may speed up development
- All the issues of normal MPI programming remain
  - dead-locks
  - communication/computation ratio
  - load balancing

Parallel dense linear algebra

Custom Python interface to ScaLAPACK and BLACS

Mostly non-Python related challenges:

- Best way to understand ScaLAPACK is to read the source code.
- DFT leads to complicated scenarios for ScaLAPACK. Matrices exist on a small subset of MPI_COMM_WORLD.
- Arrays must be distributed by application developer
- Distributed arrays must be compatible with their native 2D-block cyclic layout
- Distributed arrays assumed to be Fortran-ordered.
- Python interface makes playing with ScaLAPACK easier

Parallel IO

- DFT is not very IO intensive
- In massively parallel scale IO may become bottleneck
- GPAW can use parallel HDF5 utilizing h5py
- h5py does not support parallel features of HDF5
- Only one (two for collective IO) custom C-functions are needed for enabling parallel IO with h5py

```
plist = h5py.h5p.create(h5py.h5p.FILE_ACCESS)
_gpaw.h5_set_fapl_mpio(plist.id, comm) # Custom interface for enabling parallel IO
self.fid = h5py.h5f.open(name, h5py.h5f.ACC_RDONLY, fapl=plist)
```

- Slicing access to datasets allows different processes to write easily to different parts of dataset

  ```
  dset[rank:rank+size] = mydata[:]
  ```

- Low level API of h5py is needed for collective IO

Challenges for HPC with Python

- Special operating systems
- Debugging and profiling of mixed Python/C applications
- Python's import mechanism

Special operating systems

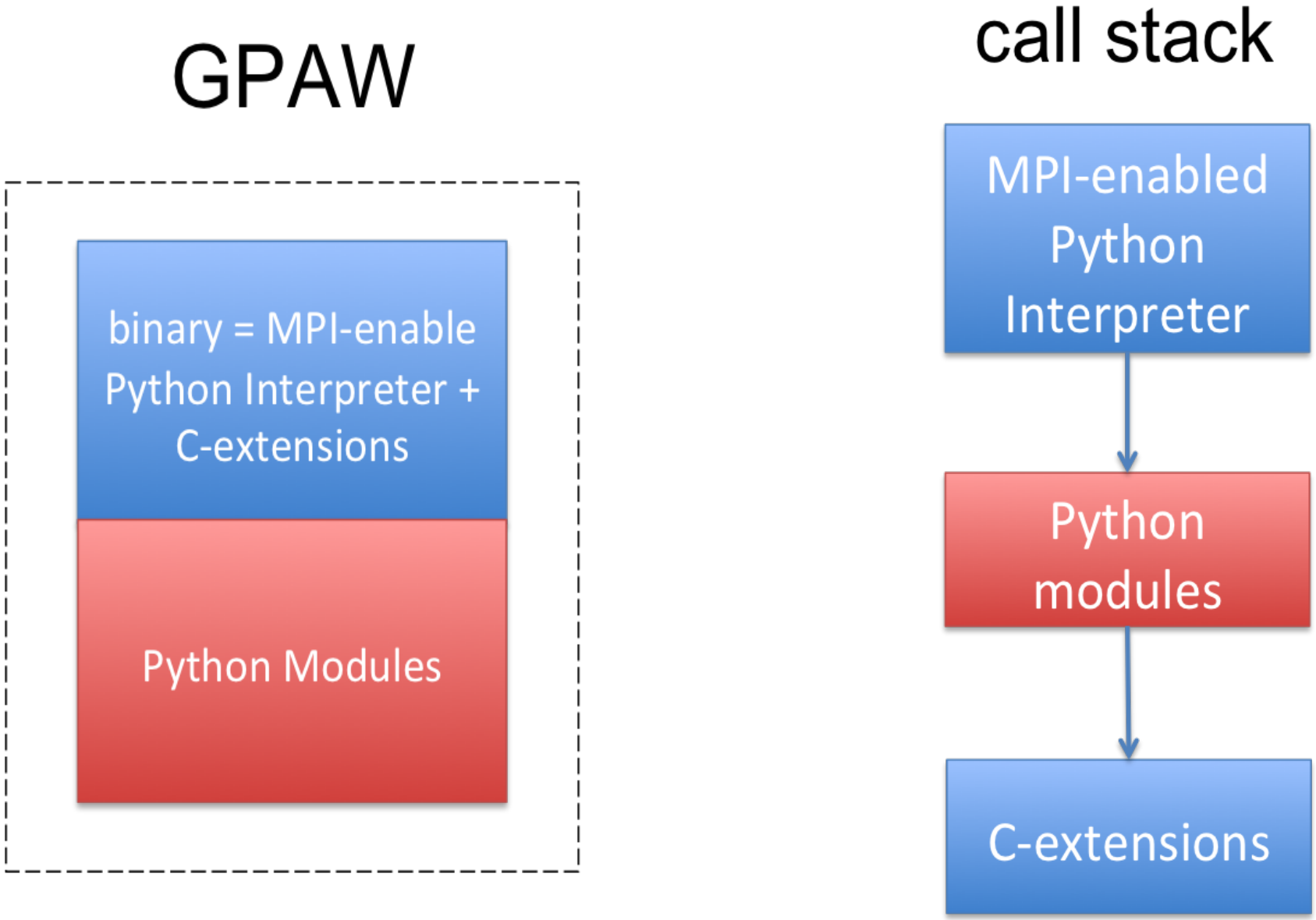
- Some supercomputing systems (BG, Cray XT) have special light-weight kernels on compute nodes
- Lack of "standard" features
  - dynamic libraries
- Python relies heavily on dynamic loading
  - static build of Python (including all needed C-extensions) is possible
  - modification of CPython is needed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

Custom interpreter

- GPAW uses a custom MPI-enabled "embedded" Python Interpreter:

```
int main(int argc, char **argv)
{
   int status;
   MPI_Init(&argc, &argv); // backwards compatible with MPI-
1
   Py_Initialize(); // needed because of call in next line
   PyObect* m = Py_InitModule3("_gpaw", functions,

extension for GPAW\n\n...\n");
   import_array1(-1); // needed for NumPy C-API
   MPI_Barrier(MPI_COMM_WORLD); // sync up
   status = Py_Main(argc, argv); // call to Python Interpreter
   MPI_Finalize();
   return status;
}
```

Parallel Python Debugging



Debugging Python and C

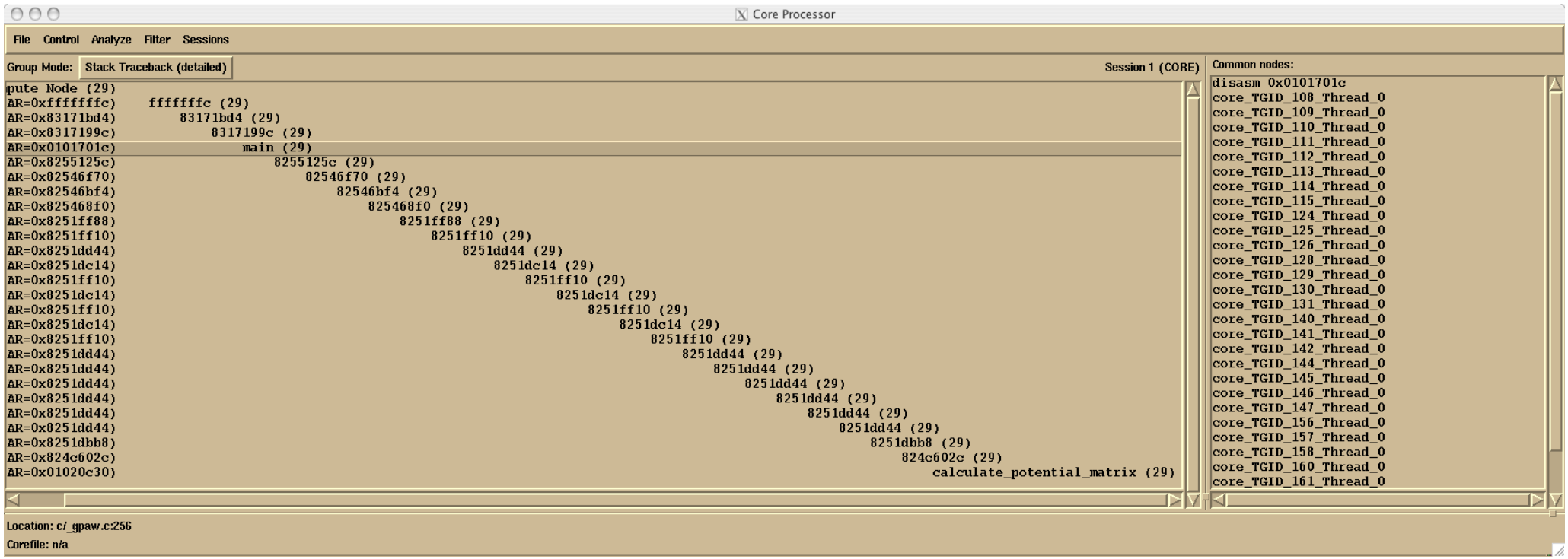
- Python comes with gdb-like command line debugger **pdb**
- Helps in debugging serial Python applications
- C-extensions can be debugged with standard debuggers
- In serial case it is even possible to debug Python and C parts in the same debugger session:

```
murska ~/gpaw/trunk/test> gdb python
GNU gdb Red Hat Linux (6.1post-1.20040607.52rh)
(gdb) break Operator_apply
Function "Operator_apply" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y

Breakpoint 1 (Operator_apply) pending.
(gdb) run -m pdb H.py
Starting program: /usr/bin/python -m pdb H.py
[Thread debugging using libthread_db enabled]
[New Thread -1208371520 (LWP 1575)]
> /home/jenkovaa/test/H.py(1)?()
-> from gpaw import GPAW
(Pdb)
```

Runtime errors and core dumps

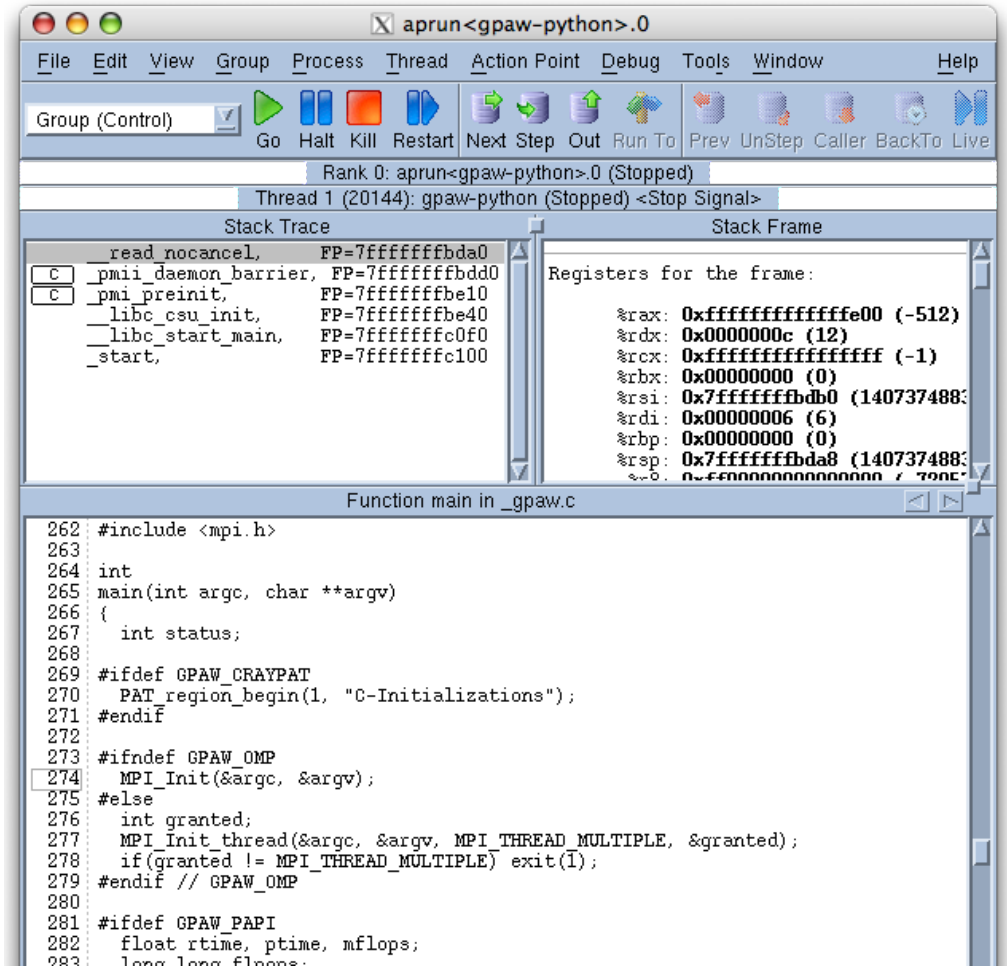
Errors in Python modules are OK, core dumps in C extensions are problematic:

- Python call stack is hidden; this is due to Python's interpreted nature.
- Totalview won't help, sorry.



Parallel debugging

- Totalview can be used for debugging C-functions
- Custom interpreter provides easy initial breakpoint



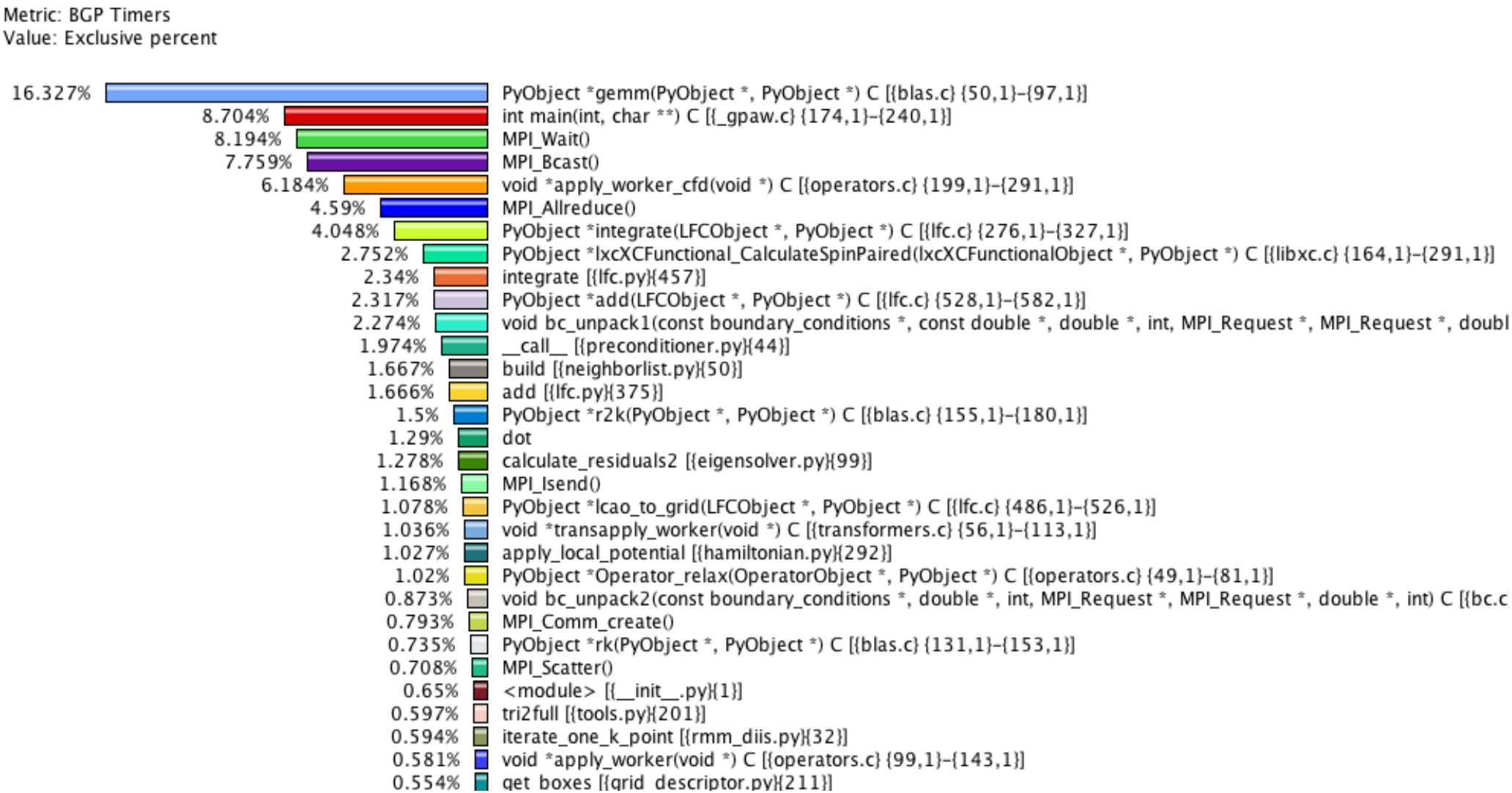
- For Python parts one is left mostly with print statements

Profiling Mixed Python-C code

- Finding performance bottlenecks is critical to scalability on HPC platforms
- Number of profiling tools available:
  - gprof, CrayPAT, scalasca - C, Fortran
    - can be used for C-extensions
  - import profile - Python
- TAU Performance System, http://www.cs.uoregon.edu/research/tau/home.php
  - Exclusive time for C, Python, MPI are reported simultaneously.
  - Heap memory profiling.
  - Interfaces with PAPI for performance counters.
  - Manual and automatic instrumentation available.
  - Cross platform tool

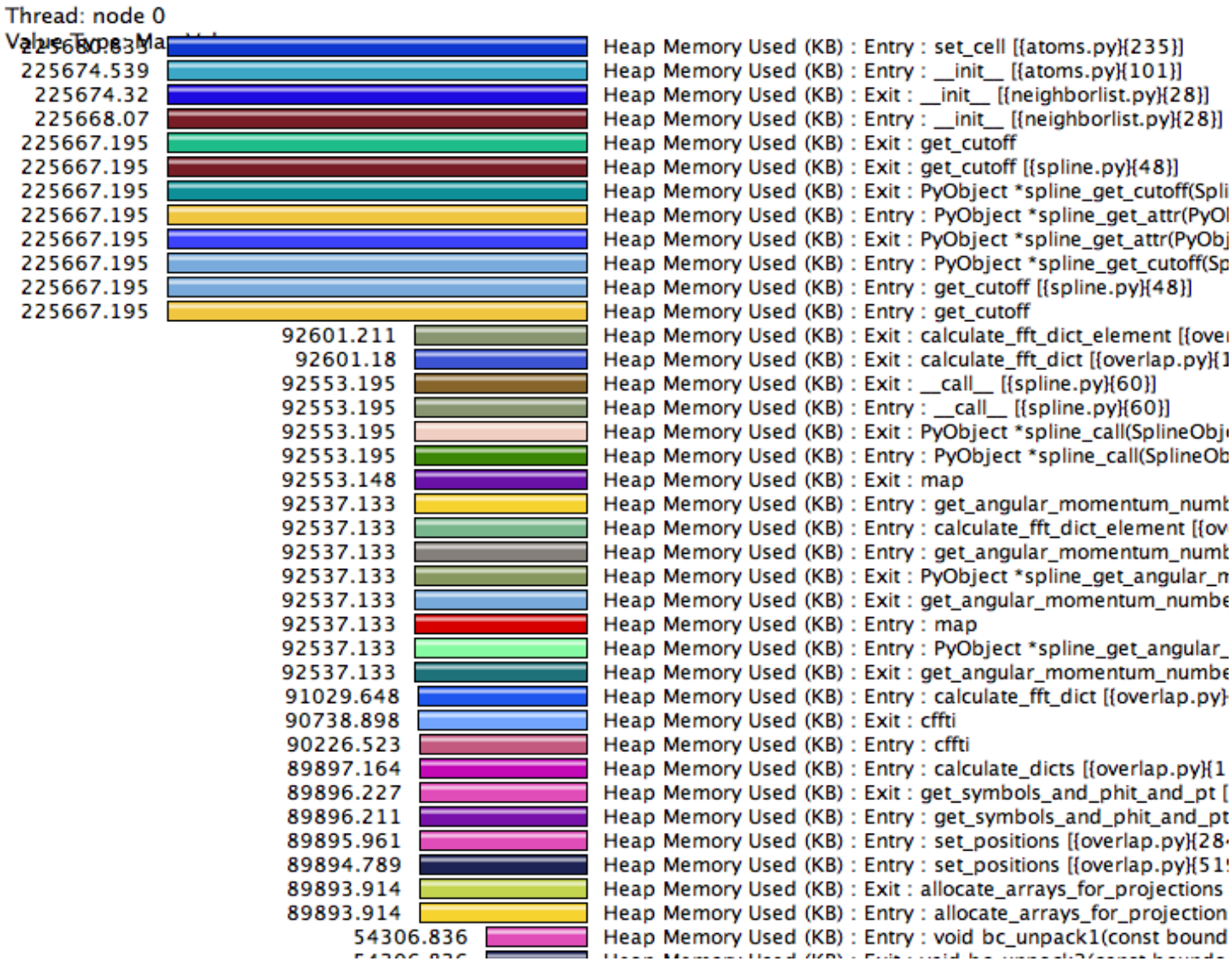
Profiling Mixed Python-C code

Flat profile shows time spent in Python, C, and MPI simultaneously:



Profiling Mixed Python-C code

Measure heap memory on subroutine entry/exit:

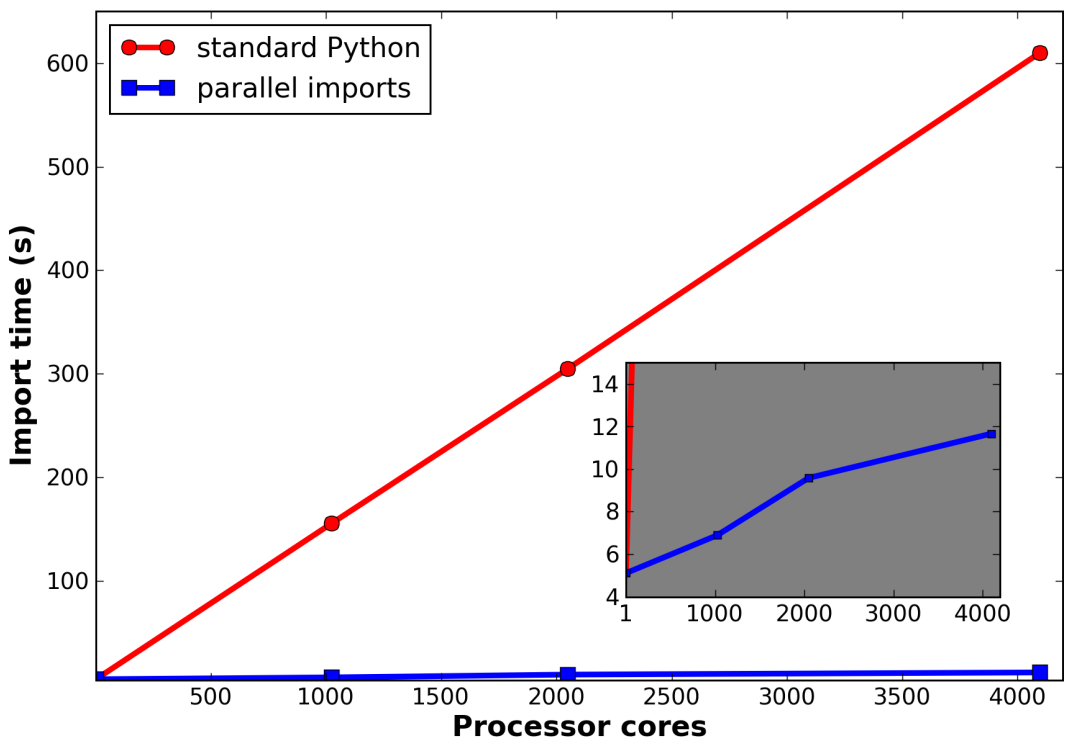


Python's import mechanism and parallel scalability

- Import statement trickers lot 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
  - 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



Summary

The Bad & Ugly:

- Python's import mechanism is not scalable
- optimized NumPy cross-compile
- C extensions require learning NumPy & C API
- Parallel debugging can be difficult
- OpenMP-like threading challenging due to GIL
- Support for GPU acceleration? Python level, C level or both?

The Good:

- GPAW has an extraordinary amount of functionality and scalability.
- Python makes coding complex algorithms easy:
  - object-oriented programming
  - weakly-typed data structures
  - ...
- Interfaces to efficient C-extensions and libraries provide good performance

Massively parallel HPC programming
with Python is feasible!

More info: wiki.fysik.dtu.dk/gpaw

Acknowledgements

- GPAW team: J. J. Mortensen, N. A. Romero, C. Rostgaard, M. Dulak, C. Glinsvad, A. H. Larsen, V. Petzold, M. Vanin, M. Strange, L. Lehtovaara, M. Kuisma, J. Ojanen, O. Lopez-Acevedo, M. Walter, L. Ferrighi, H. H. Kristoffersen, J. Stausholm-Möller, M. Ljungberg, G. K. H. Madsen, H. Häkkinen, T. Rantala, M. Puska, R. M. Nieminen
- ANL staff: V. A. Morozov, J. P. Greeley
- Cray Center of Excellence: J. Levesque

Acknowledgements

Funding and Computational resources:
This research used resources at: Argonne Leadership Computing Facility and the Center for Nanoscale Materials at Argonne National Laboratory, which is supported by the office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357; High Performance Computing Center North (HPC2N).
The Center for Atomic-scale Materials
Design is sponsored by the Lundbeck Foundation. The authors acknowledge support from the Danish Center for Scientific Computing and Finnish Technology Agency (TEKES)

Let's review!

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
- the families of the presenters
- Sameer Shende, ParaTools, Inc.
- Enthought, Inc. for their continued support and sponsorship of SciPy and NumPy
- Lisandro Dalcin for his work on mpi4py and tolerating a lot of questions
- the members of the Chicago Python User's Group (ChiPy) for allowing us to ramble on about science and HPC
- the Python community for their feedback and support
- CCT at LSU
- numerous others at HPC centers nationwide