Argonne NATIONAL LABORATORY

Illustrated Tactics for Productivity, Performance, and Parallelism at Scale with Python

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

U.S. DEPARTMENT OF ENERGY

Why this talk?

Conventions used in these slides!

- Anything in `monospace font indicates code()`
- Most bold text is a **`language keyword()`**
  - May also be used for **`emphasis`**
- Ellipsis … is used for omitted code
- White space and formatting is meaningful in Python code
- Most code samples are available at https://github.com/wscullin/ACCA-CS
- URLs are plain text
- Book and article titles are *italics*
- Licensing:
  - All code samples are MIT and CRAPL licensed
  - Applications mentioned have varying licenses
  - I accept patches and feedback enthusiastically

What do we mean by performance?

The New Oxford American dictionary defines performance as:

2 the action or process of carrying out or accomplishing an action, task, or function: *the continual performance of a single task reduces a man to the level of a machine*.
• an action, task, or operation, seen in terms of how successfully it was performed: *pay increases are now being linked more closely to performance | a dynamic performance by Davis*.
• the capabilities of a machine, vehicle, or product, esp. when observed under particular conditions: *the hardware is put through tests that assess the performance of the processor*.
• the extent to which an investment is profitable, esp. in relation to other investments.

We're going to run with these.

Performance as FLOPs

The Oxford English Dictionary notes that the term FLOPS first arose in a 1976 paper titled “Algorithmic and Architectural Issues Related to Vector Processors” by Donald A. Calahan:

“The most common [vector] performance measure is the number of floating point operations per second (FLOPS), obtained by dividing the number of floating point operations - known from the algorithmic complexity - by the computation time.”

Dr. Calahan wasn’t the first to measure performance in operations per second, but he was the first to reduce systems performance to FLOPS - and to do so in the context of multiple cpus and vectorization.

Performance as FLOPs

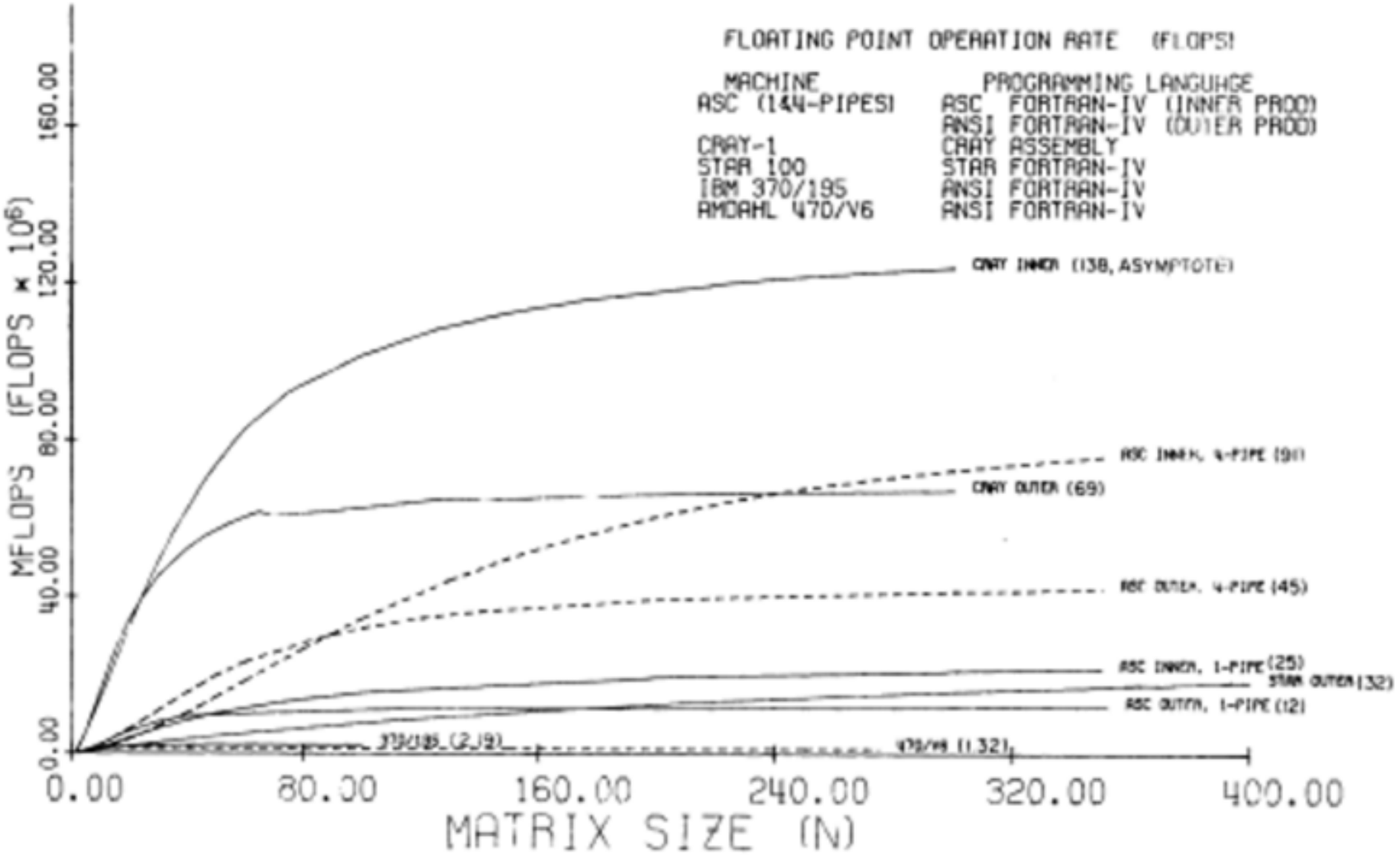
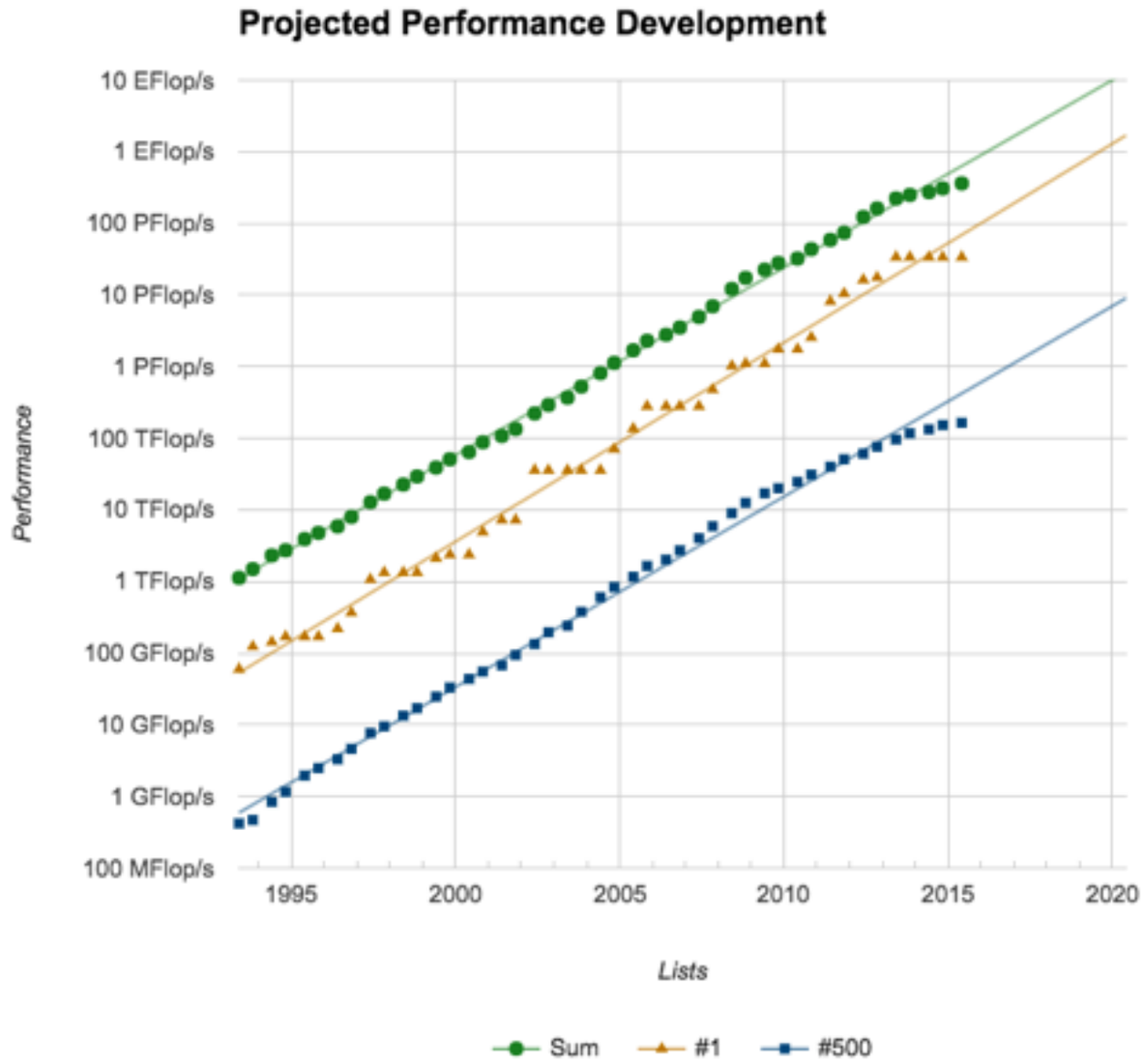


Figure 3

Performance as FLOPs

FLOPS now refers almost exclusively to double precision floating point operations. Since the creation of the Top500 list in 1993, it's become **the way** HPC centers talk about system performance.



Theoretical Peak FLOPS = cores*clock*FLOPS/clock

Measured FLOPS = HPL's Rmax

Efficiency = Measured / Theoretical

What do flops tell us?

Almost nothing, really.*

So why do we care anyway?

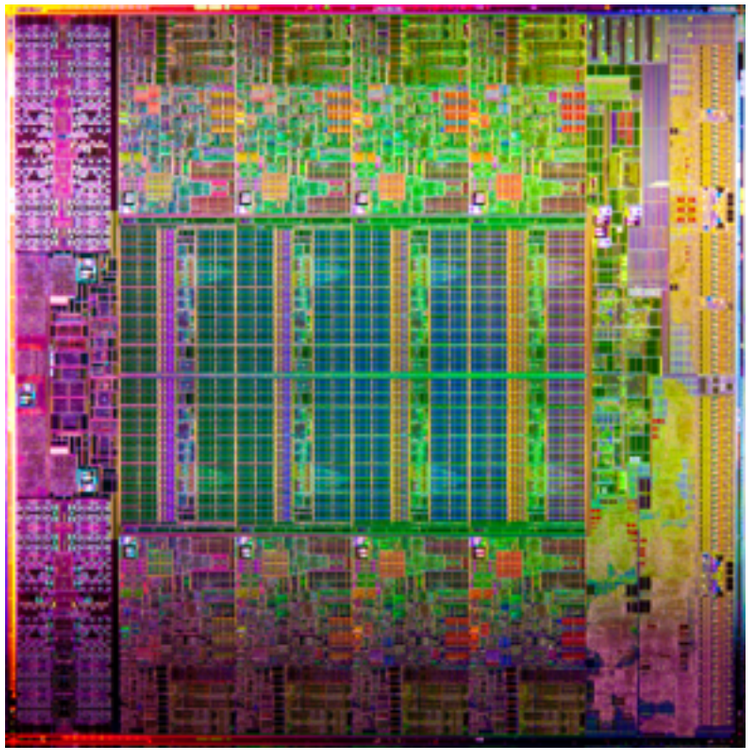
- Meaningful benchmarks let us know what's possible
- FLOP rates aren't useless, just the HPL (Linpack) benchmark
  - HPL's main flaw is that it's not representative of real codes
  - HPCC and other suites contain multiple applications representative of many different flavors of code
  - Knowing the relative efficiency of a piece of code lets us characterize and fully explain a benchmark result
- Synthetic and simple workloads make the debugging of complex systems much easier

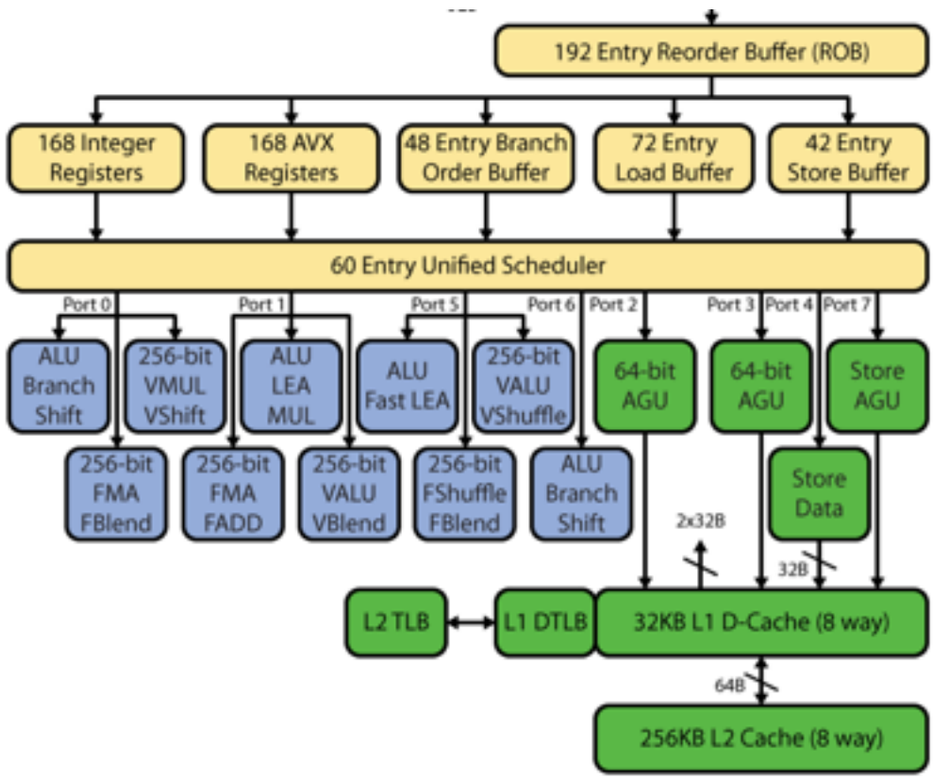
Why is our system slow?

- Approach everything with a methodology in mind
- Define questions to be answered: i.e.: Why am I getting 2% of theoretical peak performance? Why do I get only 30% of the performance of a very similar application?
- Document your approaches and operating conditions
  - Compilers, libraries, OS versions
  - Hardware
  - Concurrently running processes
- Brendan Gregg (http://www.brendangregg.com/) advocates the **USE** methodology for all resources:
  - Utilization - time a resource is busy
  - Saturation - how deep the queue length or wait time is
  - Errors - is something throwing errors

Modern CPUs look a lot like early big iron

- Modern systems have multiple CPUs
- Modern CPUs have multiple cores
- Modern cores have vector units that deliver >80% of FLOPS
- There's contention for resources and overheads that prevent ideal execution!





The key to success: Multilevel Parallelism in a Compiled Language!

```
/*3456789012345678901234567890123456789012345678901234567890123456789012345678*/

#define GAE 114474615732576576.000000


/*************************************************************
 2015-02-10 intcurve

 Calculate an approximation of the
 integral of x^8-x^6+x^4-x^2+1 from min to max

 This is a single file program without a makefile. It is built:

 mpicc -lm -g intcurve.c -o intcurve

*************************************************************/


/*************************************************************

 included headers

*************************************************************/
#include <mpi.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <limits.h>
#include <omp.h>

typedef struct result_record result_record;

struct result_record
{
char * solver;
double area;
int size;
};


/*************************************************************
pdiv:
    In: x1 - a double for the first point
        x2 - a double for the second point
        div - a divisor
    Out: return - a double for x1+x2/divisor

*************************************************************/
double pdiv(double x1, double x2, double div)
{

    return ((x1+x2)/div);
}


/*************************************************************
line:

   Print a line of 78 = signs to set off program output

   In: none

   Out: return - none
        stdout - to screen

*************************************************************/
void line()
{
    int m;

    for (m=0; m<=78; m++) printf("=");
    printf("\n");
}


/*************************************************************
startup_message:

   Print a message at program startup

   In: none

   Out: return - none
        stdout - to screen

*************************************************************/
void startup_message()
{
       line();
       printf("\n\tStarting calculation\n\n");
       line();
}


/*************************************************************
final_output:

   Print the final output

   In: the calulated area

   Out: return - none
        stdout - to screen

*************************************************************/
void final_output(result_record * results)
{

    int i;
    double error;
    double offset;

    line();
    for (i=0;i<results[0].size;i++)
    {

        error=results[i].area-GAE;
        offset=((GAE-results[i].area)/GAE)*100.0;

        printf("\n");
        printf("\tFinal area (%8s):\t%30.6lf\n",results[i].solver,results[i].area);
        printf("\tError (s-A):\t\t%30.6f\n",error);
        printf("\tError (a/A)%%:\t\t%30.10lG\n",offset);

    }
    printf("\n");
    line();
}


/*************************************************************
f:

   Return the value of the polynomial equation:
   f(x)=x^8-x^6+x^4-x^2+1

   In: x - a double precision value to evaluate the equation for

   Out: return - a double for the value of the equation at x

*************************************************************/
double f( double x )
{
    return (pow(x,8)-pow(x,6)+pow(x,4)-pow(x,2)+1);
}


/*************************************************************
pavg:

   Return the average value of an equation at a point

   In: f - a double precision function
       x1 - a double for the first point
       x2 - a double for the second point

   Out: return - a double for average of the points

*************************************************************/
double pavg( double(*f)(const double), double x1, double x2 )
{
    return pdiv(f(x1),f(x2),2);
}


/*************************************************************
trule:

   Return the average value of an equation at a point

   In: f - a double precision function
       x1 - a double for the first point
       x2 - a double for the second point

   Out: return - a double for average of the points

*************************************************************/
double trule(double x1, double x2 )
{
    return pavg(&f,x1,x2)*(x2-x1);
}


/*************************************************************
default_solve:

   Approximate solver

   In: xmin - a double for lowest point in range
       xmax - a double for highest point in range
       rank - integer for the rank or thread
       nrank - total number of ranks or threads
       samples - total number of samples

   Out: return - a double for average of the points

*************************************************************/
double default_solve(double xmin, double xmax, int rank, int nranks, long int samples)
{
    long int localsamples=0;
    double ss=0.0;
    double lxmin=0.0;
    double lfxs=0.0;
    double range=0.0;


    /* Setup for local evaluation */
    localsamples=samples/nranks;
    range=pdiv(xmax,-xmin,nranks);
    ss=range/localsamples;
    lxmin=(range*rank)+xmin;

    /* Caculate rank-local values */
    #pragma omp parallel
    /* loop counter*/
    long int j;
    for (j=0; j<localsamples; j++  )
    {
        lfxs=pavg(&f,lxmin,lxmin+ss)+lfxs;
        lxmin=lxmin+ss;
    }

    return (lfxs*((xmax-xmin)/samples));
}


/*************************************************************
simpson:

   Solve via simpson's method... donuts!

   In: xmin - a double for lowest point in range
       xmax - a double for highest point in range
       rank - integer for the rank or thread
       nrank - total number of ranks or threads
       samples - total number of samples

   Out: return - a double for average of the points

*************************************************************/
double simpson( double xmin, double xmax, int rank, int nranks, long int samples)
{
    long int localsamples=0;
    double lxmin=0.00;
    double lxmax=0.00;
    double xdiff=(xmax-xmin);
    double xoffset=0.0;
    double lsect=0.0;



    /* make values evenly divisible */
    do
    {
        localsamples=samples/nranks;
        samples++;
    }
    while((localsamples%2)!=0);

    xoffset=xdiff/samples;
    lxmax=(xoffset*localsamples)*(rank+1)+xmin;
    lxmin=(xoffset*localsamples)*(rank)+xmin;
    xoffset=(lxmax-lxmin)/localsamples;

    lsect=f(lxmin)+f(lxmax);

    #pragma omp parallel
    {
    long int i;
    for(i=1;i<localsamples;i=i+2)
        lsect=( 4 * f(lxmin+(i*xoffset)))+lsect;


    for(i=2;i<localsamples-1;i=i+2)
        lsect=( 2 * f(lxmin+(i*xoffset)))+lsect;
    }

    lsect=(xoffset/3)*lsect;

    return lsect;
}


/*************************************************************
main:

   Main routine

   In: argv and argc - both unused

   Out: return - an integer value for the return code

*************************************************************/
int main( int argc, char *argv[] )
{

    /* range across which the integral will be evaluated */
    double xmax=100.3333333;
    double xmin=-10.2666667;

    /* set the number of samples used in the approximation */
    long int samples=pow(10,9);

    /* values used in the solve */
    double fdfs=0;
    double ldfs=0;
    double fsimpson=0;
    double lsimpson=0;

    /* loop counters */
    int i,j,k;

    /* things done to wreck havoc */
    result_record results[2];

    /* variables for process MPI information */
    int nranks=1;
    int rank=0;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&nranks);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

MPI_Status status;
MPI_Request request[nranks*2];

/* Status to let us know things have started */
if (rank == 0)
{
    startup_message();
}

MPI_Barrier(MPI_COMM_WORLD);

/* Setup for local evaluation */
for (i=0; i<nranks; i++)
{
    if ( rank == i)
    {
        ldfs=default_solve(xmin,xmax,rank,nranks,samples);
        lsimpson=simpson(xmin,xmax,rank,nranks,samples);
    }
}

/* Output local values for each rank */
for (k=0; k<nranks; k++)
{
    if ( rank == k)
    {
        printf("rank %3d deflt: % 26.6f simpson: % 26.6f\n",rank,ldfs,lsimpson);
    }
}

MPI_Reduce(&ldfs,&fdfs,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

/* Sum for default solver */
if (rank == 0)
{

        results[0].solver="default";
        results[0].area=fdfs;
        results[0].size=2;
}


MPI_Reduce(&lsimpson,&fsimpson,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

/* Sum for simpson */
if (rank == 0)
{
    results[1].solver="simpson";
    results[1].area=fsimpson;
    results[1].size=2;
}


/* Output final results */
if (rank == 0)
{
    final_output( &results);
}


MPI_Finalize();
```

Translating the screen of spew

- From the 1960s into the 1990s, Fortran was the dominant language in HPC and scientific computing. We went with C circa 1999.
  - Other languages appeared later with C, Algol, Lisp, Ada, and vendor specific dialects of the aforementioned being about it.
  - This example was originally in Fortran 77 in a single block.
  - The MPI (Message Passing Interface) 1.0 standard only appeared in June 1994 and defined a portable interface for parallel programming through message passing
  - OpenMP 1.0 appeared in 1997 giving an easy to use multiprocessing API for shared-memory systems that generally implements threads
- The code calculates an approximation of the integral of: x^8-x^6+x^4-x^2+1 by two methods and compares results
  - Mixes MPI and OpenMP to use both message passing and threading
  - Counts on compiler optimization to generate vector instructions
  - Contains an intentional error in the use of OpenMP ;-)
  - Could use a rewrite to make the for loops easier to vectorize
  - Debugging is a pain
  - It meets the first two definitions of performance from slide 4, but not the third…
    "the extent to which an investment is profitable, esp. in relation to other investments"

Can a language improve performance?

"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 distributed 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 i;
 int n = atoi(argv[1]);
 for(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, and others
SWIG
Cython
ctypes
f2py
RPy

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
Coopr
OpenOpt

Symbolic Math
SymPy

Electromagnetics
PyFemax

Astronomy
AstroLib
PySolar

Dynamic Systems
Simpy
PyDSTool

Finite Elements
SfePy

Big Data
Pandas
PySpark

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

Why Shouldn't You Use Python: The Language

- Low learning curve makes it easy to write un-Pythonic code
  - A lot of new users write Fortran / C / C++ / Java in Python
  - There are a lot of reimplementations of good ideas because it's easy to do
  - Because bindings to C/C++/Fortran are easy to write, there are a lot of un-Pythonic bindings

- White space rules are strict and there's peer pressure to follow coding standards
  - PEP 8 isn't the law, just a really good idea
  - http://www.python.org/dev/peps/pep-0008/

Why Shouldn't You Use Python: The Language

- Language maintainers strive for philosophical consistency
  - Backwards compatibility is seldom guaranteed
  - 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, but aren't guaranteed to make it in
- Language maintainers strive for "principle of least surprise"
  - Web folks are fighting for decimal numerics by default which might actually a surprise to others

Why Shouldn't You Use Python: The Language

- The language is constantly evolving through the PEP process

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

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
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.

Why Not Use Python: CPython

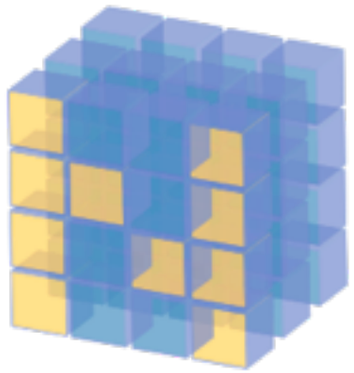
- It's horribly inefficient
  - Python 2.x is a true interpreter
  - Pure Python is interpreted line-by-line
  - "If you want your code to run faster, you should probably just use PyPy."
    — Guido van Rossum

- The Global Interpreter Lock (GIL) kills threading performance
  - David Beazley covers it better than anyone:
    http://www.dabeaz.com/python/GIL.pdf
    http://www.dabeaz.com/python/NewGIL.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

How about a tour of the important bits?



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…
- This gets you vectorization for math functions in CPython!

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.])
```


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
```


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?

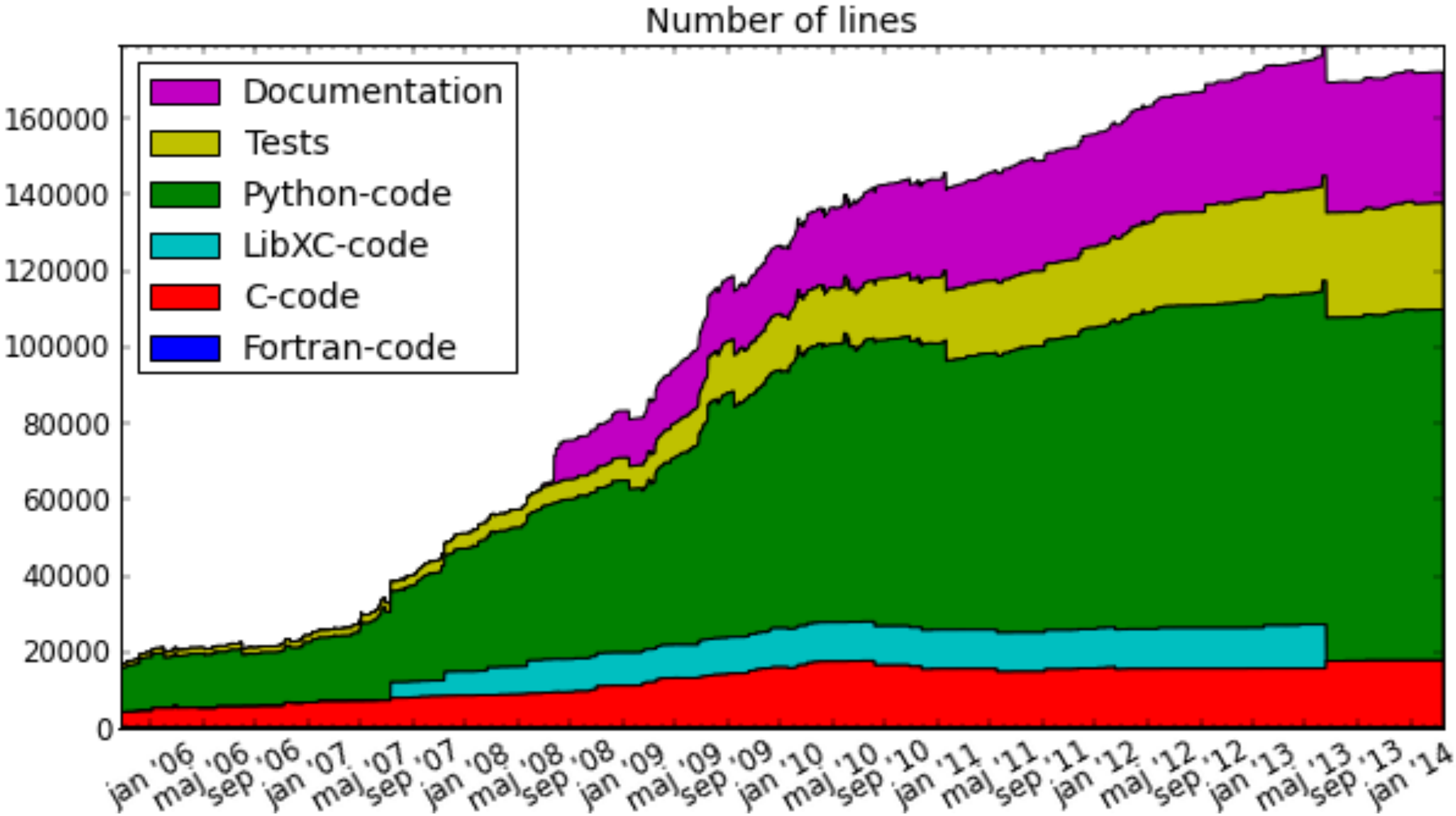
a massively parallel Python-C code
for electronic structure calculations

- *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)

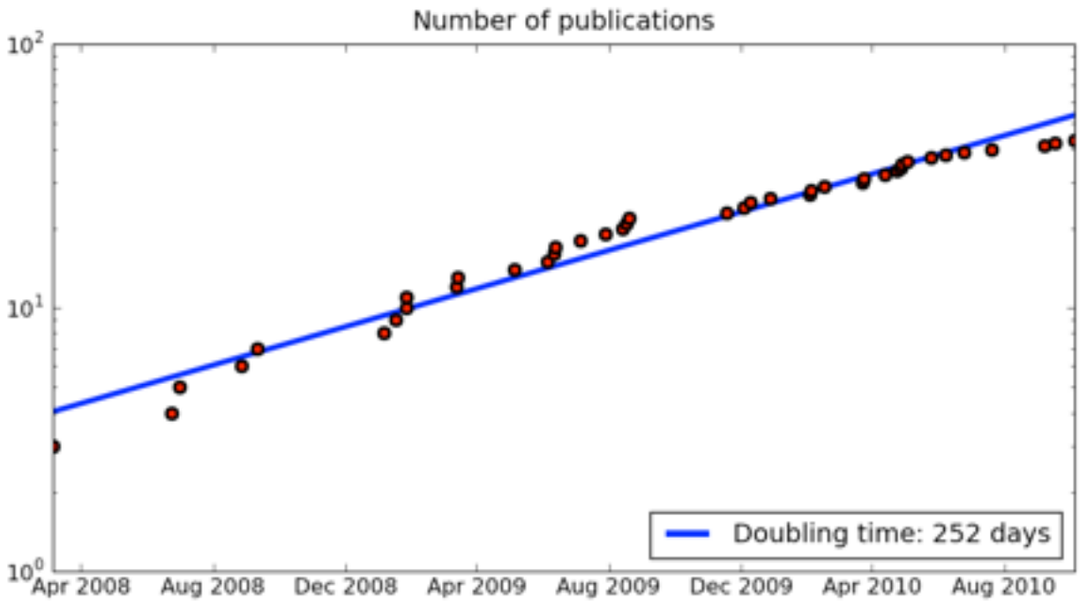
http://wiki.fysik.dtu.dk/gpaw

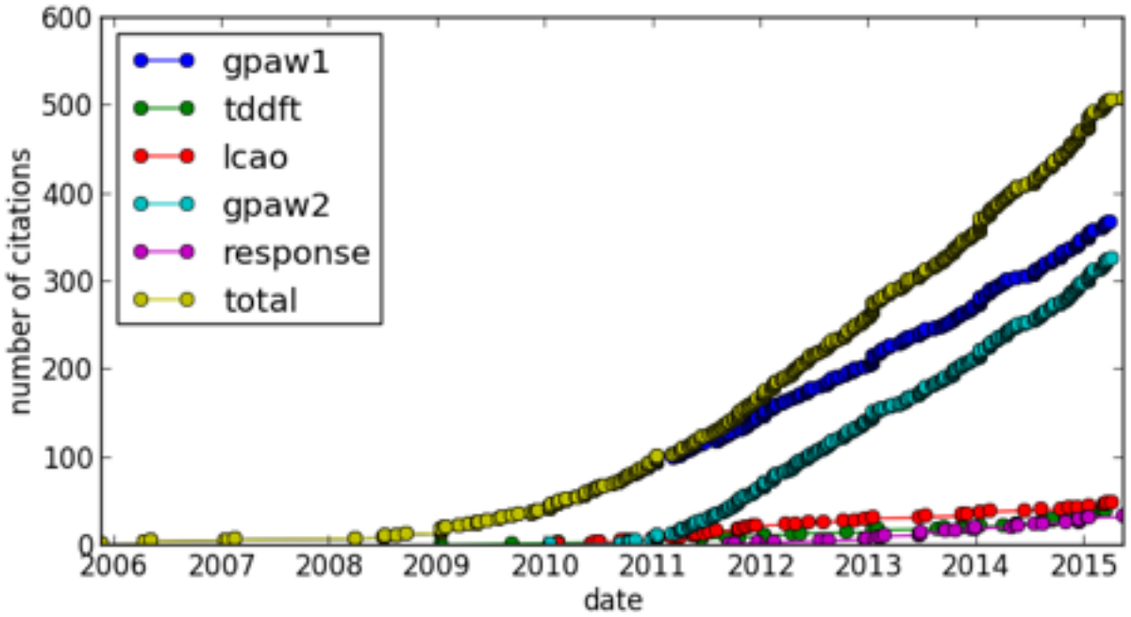
J. Enkovaara *et al.* J. Phys.: Condens. Matter **22**, 253202 (2010)

GPAW Source Code Timeline



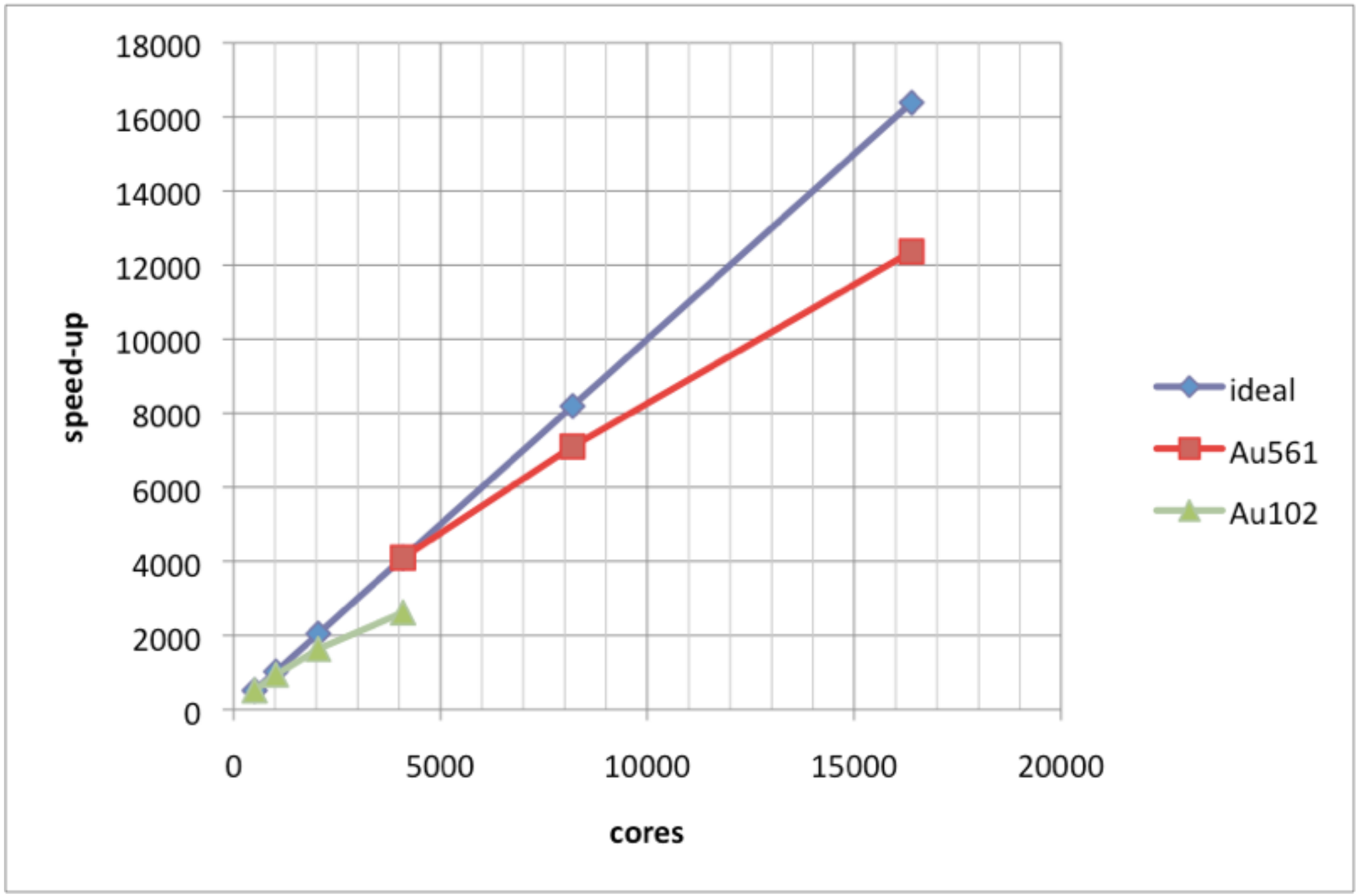
Science done with GPAW





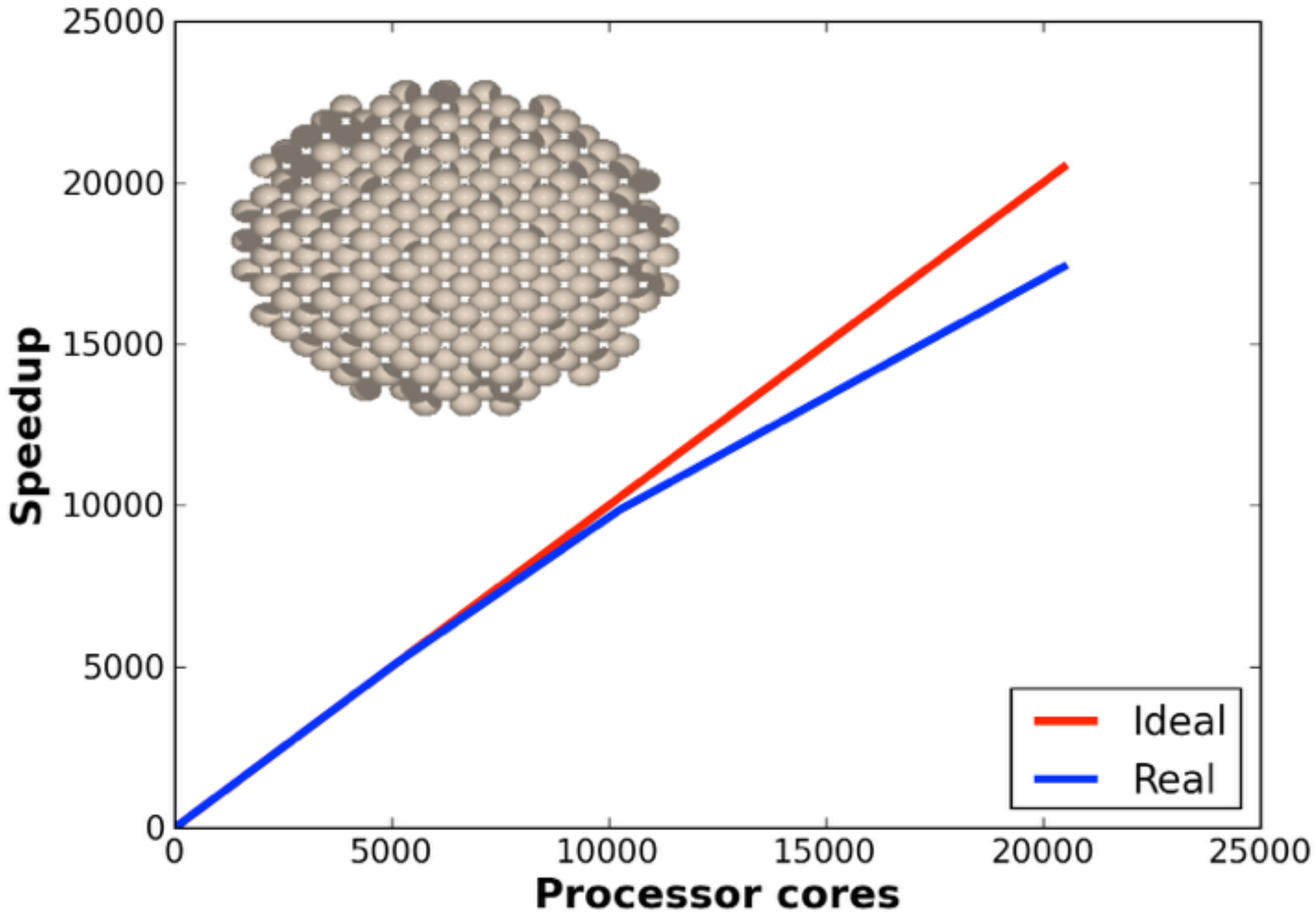
Nature Chemistry, PRL, JACS, PNAS, PRB, ...

GPAW Strong-scaling Results



Ground state DFT on Blue Gene P

GPAW Strong-scaling Results



TD-DFT on Cray XT5

Special operating systems

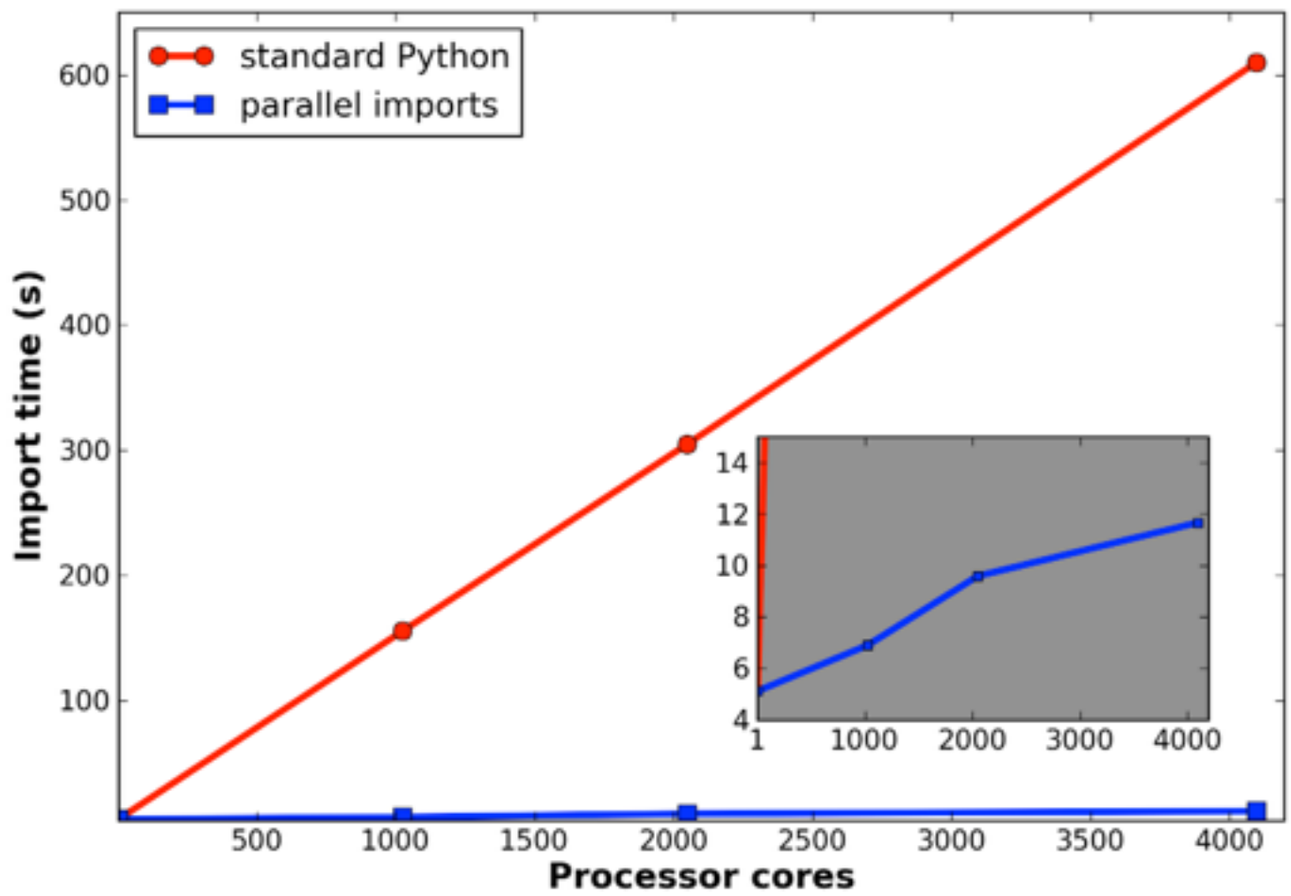
- Some supercomputing systems (BG, Cray XT) have special light-weight kernels on compute nodes
- Lack of "standard" features
  - dynamic libraries
  - lots of missing system calls
  - did we mention all I/O is forwarded?
- Python relies heavily on dynamic loading
  - static build of Python (including 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