Python and Parallel computing: an overview

Fernando Pérez

Fernando.Perez@berkeley.edu

Helen Wills Neuroscience Institute, UC Berkeley

Python Computing for Science - AY250

Outline

Outline

Data explosion

Our data sets are getting huge!

- Sloan Digital Sky Survey Data Release 7 : ~65 Terabytes
- Neuroimaging: 1 hour $\rightarrow$ a few gigabytes
- etc...

A simple problem: solve $Ax = b$ via Gaussian elimination

- For A an $n \times n$ matrix, cost is $\approx \frac{1}{3}n^3$ floating point operations
- On a computer with ~100MFlops sustained performance

n	time
10	$\mathcal{O}(1)$ μs
100	$\mathcal{O}(1)$ ms
1000	$\mathcal{O}(1)$ s
10000	$\mathcal{O}(1)$ hour
100000	$\mathcal{O}(1)$ month
1000000	$\mathcal{O}(1)$ century

Why do we need fast algorithms?

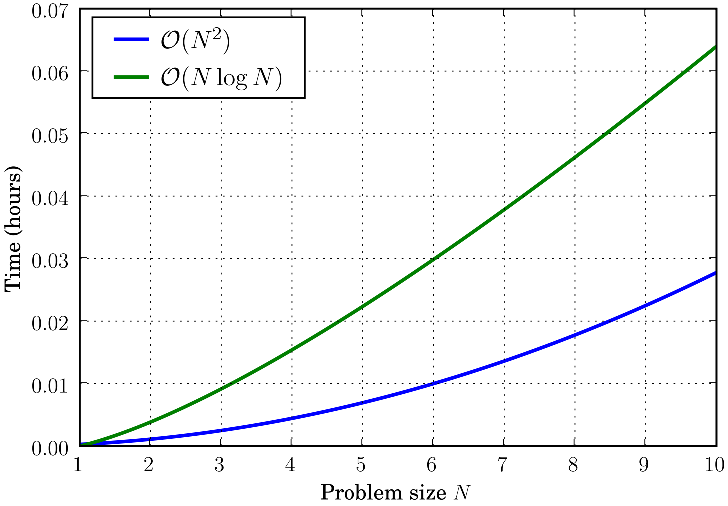
Because computers are getting bigger and faster!

Why do we need fast algorithms?

Because computers are getting bigger and faster!

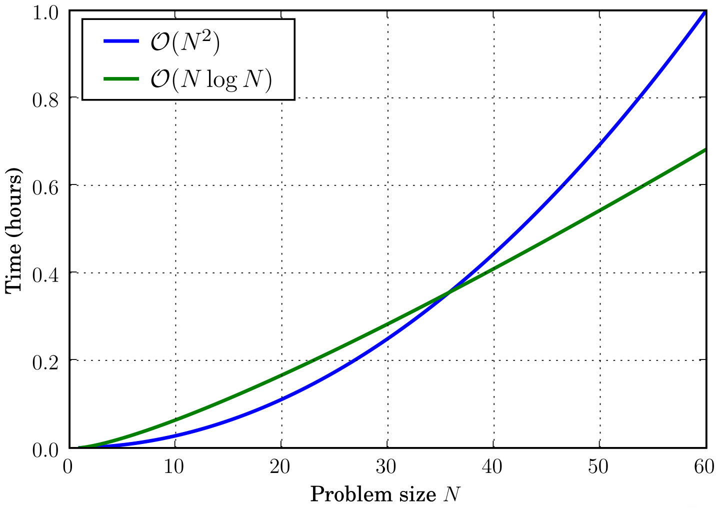
Why do we need fast algorithms?

Because computers are getting bigger and faster!



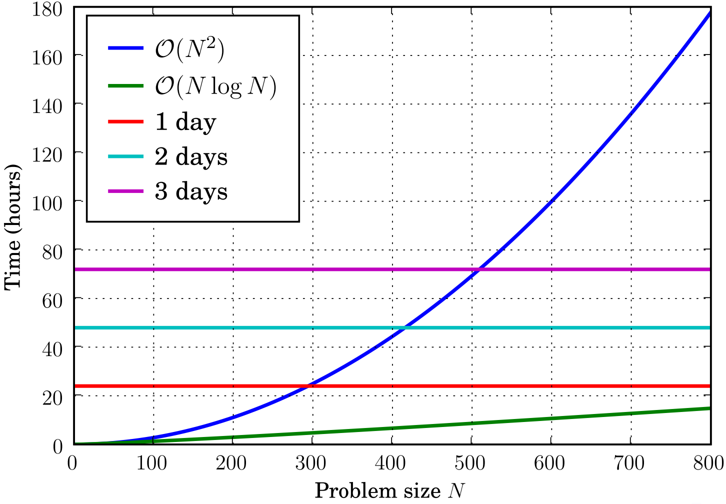
Why do we need fast algorithms?

Because computers are getting bigger and faster!



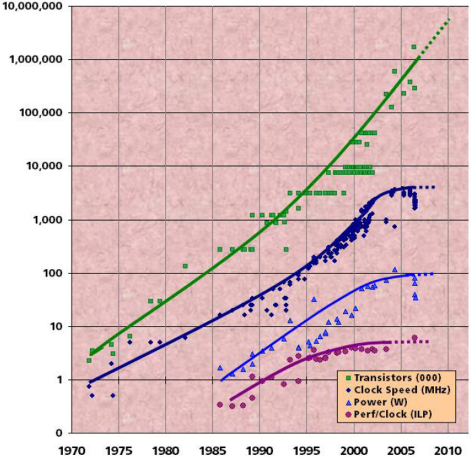
Why do we need fast algorithms?

Because computers are getting bigger and faster!



Parallel computing: why should we care?

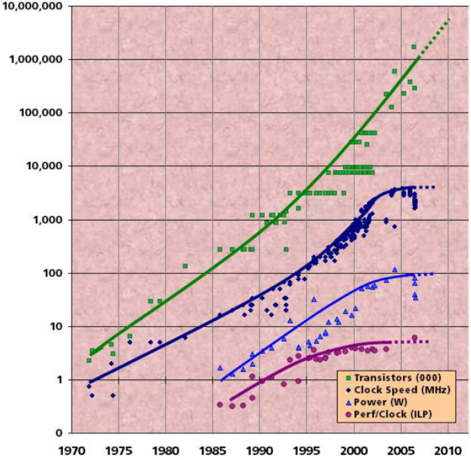
Because reality looks like this:



Sources: Intel, Microsoft (Sutter), Stanford (Olukotun, Hammond) & Berkeley (Yelick)

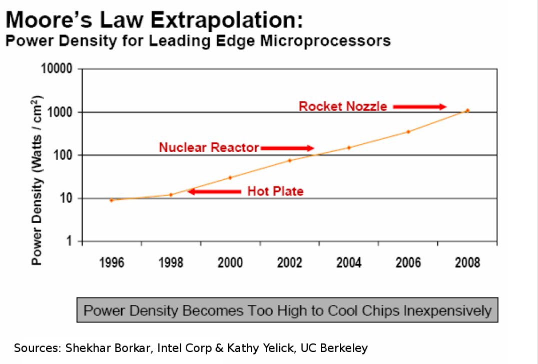
Parallel computing: why should we care?

Because reality looks like this:



Sources: Intel, Microsoft (Sutter), Stanford (Olukotun, Hammond) & Berkeley (Yelick)

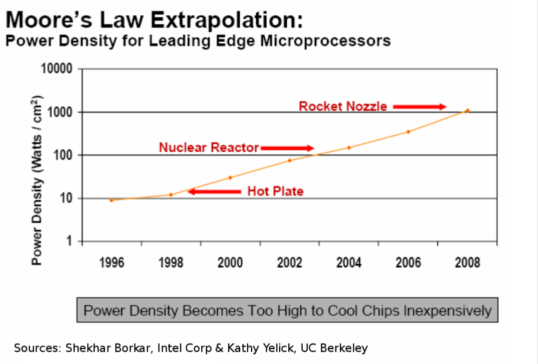
We can't escape thermodynamics



The vendor's solutions:

- Multicore chips: even in your laptop.
- Graphics cards for general computing: $>$ 128 'processors' per card.
- Clusters, cloud, ...

We can't escape thermodynamics



The vendor's solutions:

- Multicore chips: even in your laptop.
- Graphics cards for general computing: >128 'processors' per card.
- Clusters, cloud, ...

How fast can we go?

Exercise: Derive and plot Amdahl's law

Upper bound on speedup achievable via parallelization

- s : serial fraction of total work to be done
- $1 - s$: parallelizable fraction
- p : number of processors used
- Let T_n be the time for a job with n processors in use.
- Derive the possible speedup T_1 / T_p.
- Compare to a simple function of s.

Amdahl's law: a logical limit

Consider solving a problem where communication has zero cost

- Using 1 processor: $T_1 = s + (1 - s) = 1$
- Using p processors: $T_p = s + \frac{1-s}{p}$

The total speedup possible:

$$\frac{T_1}{T_p} = \frac{1}{s + \frac{1-s}{p}} < \frac{1}{s}$$

Lesson: The serial fraction is a hard limit!

Amdahl's law: a logical limit

Consider solving a problem where communication has zero cost

- Using 1 processor: $T_1 = s + (1-s) = 1$
- Using p processors: $T_p = s + \frac{1-s}{p}$

The total speedup possible:

$$\frac{T_1}{T_p} = \frac{1}{s + \frac{1-s}{p}} < \frac{1}{s}$$

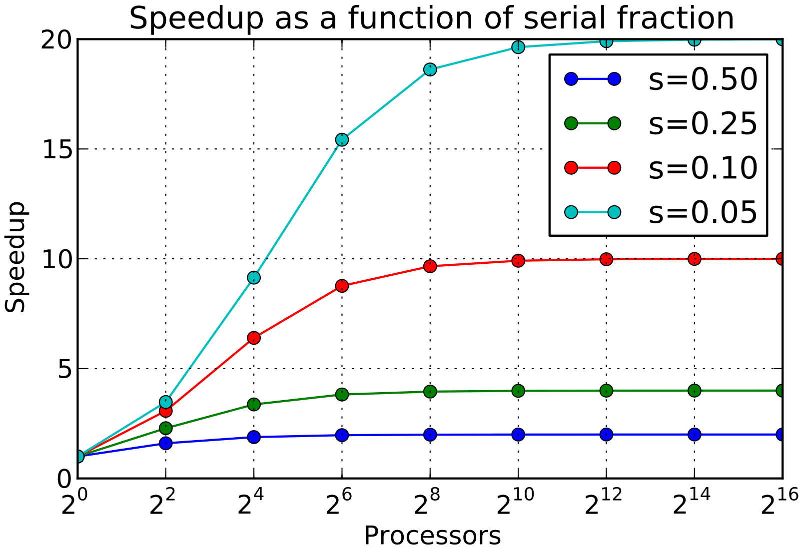
Lesson: The serial fraction is a hard limit!

Code

Plot Amdahl's law

```
def amdahl(s, p):
    return 1.0/(s+(1.0-s)/p)

p = np.logspace(0, 16, 9, base=2)
ax = plt.subplot(111)

for s in [0.5, 0.25, 0.1, 0.05]:
    sp = amdahl(s, p)
    ax.semilogx(p, sp,'-o', label='s=%.2f' % s, basex=2.0)

ax.set_xlabel('Processors')
ax.set_ylabel('Speedup')
ax.set_title('Speedup as a function of serial fraction')
ax.legend()
ax.grid()

plt.show()
```

Amdahl's law



Speedup as a function of serial fraction
Speedup
Processors
20
15
10
5
0
2^0
2^2
2^4
2^6
2^8
2^{10}
2^{12}
2^{14}
2^{16}
s=0.50
s=0.25
s=0.10
s=0.05

General Principles of Parallel Programmig

Credit: Kathy Yelick, UCB.

- Finding enough parallelism (Amdahl's law)
- Granularity: bite-sized chunks for each unit...
  - But need large enough amount of work to hide the overhead
- Locality
  - large memories are slow, fast memories are small.
- Load balance
- Coordination and synchronization
  - Communication is expensive...
  - But gettting the wrong answer fast doesn't cut it.
- Model performance, profile, profile, profile...
  - If you didn't measure it, you don't actually know.
  - Your intuition is wrong.

The thirteen dwarves: patterns in parallel programming

The Landscape of Parallel Computing Research: A View from Berkeley

1. Dense Linear Algebra
2. Sparse Linear Algebra
3. Spectral Methods
4. N-Body Methods
5. Structured Grids
6. Unstructured Grids
7. MapReduce
8. Combinational Logic
9. Graph Traversal
10. Dynamic Programming
11. Backtrack and Branch-and-Bound
12. Graphical Models
13. Finite State Machines

http://view.eecs.berkeley.edu/wiki/Dwarf_Mine

Outline

Threading and parallelism in Python: overview

- Multiple implementations of the Virtual Machine:
    - CPython: pure C, 'reference'
    - IronPython: .NET
    - Jython: Java
- Their threading behaviors differ, I'll focus on CPython
- Native threads supported, but of limited use.
- Global interpreter lock (GIL): only **one** thread can modify any python data structure
- No language-specific primitives for parallelism.

Parallelism in Python

- **In-process (mind the GIL)**
  - Data parallellism with threaded libraries
  - Numpy/scipy can use a threaded ATLAS
  - Numexpr: a 'numpy VM'
  - Theano: a library that thinks it's a compiler
  - GPU-based solutions: PyCuda/PyOpenCL, scikits.cuda.
  - Hand-written threaded code...
- **Out-of-process**
  - The `multiprocessing` module
  - Python `futures` (only in Python 3.2 or newer)
  - Communicating Sequential Processes, ParallelPython, ... many more
  - IPython (I'm obviously biased). For more on IPython, see: `https://github.com/ipython/ipython-in-depth`.

Multiprocessing

Module: multiprocessing

- Built-in since version 2.6 (available for earlier versions)
- An API that closely follows the threading API, but using processes
- Useful high-level objects
  - Process, Process pool, Namespaces, Listeners, ...
- Uses fork() on posix (hence there are some limitations)

A simple example

```
from multiprocessing import Process

def f(name):
    print 'hello', name

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

Outline

Numpy and Scipy: 'Out of the box' parallelism?

- **Not great...**
- Can be built against a threaded ATLAS or the Intel Math Kernel Library (MKL)
  - This can give multithreaded support to many linear algebra operations.
- Manual effort with C/Fortran + OpenMP can give you some gains...
  - but with a fair amount of pain

Numexpr

An expression compiler for numpy

Approach

- Compile Numpy expressions to equivalent Python code...
- Block operations carefully
- execute on a special-purpose mini-VM (written in C)

Benefits

- Reduce the use of temporaries.
- Be cache-friendly.
- Support threads natively for all operations.
- Support Intel Vector Math Library and MKL.

Numexpr usage

Evaluating simple expressions

```
>>> import numpy as np
>>> import numexpr as ne

>>> a = np.arange(1e6)   # Choose large arrays for high
    performance
>>> b = np.arange(1e6)

>>> ne.evaluate("a + 1")   # a simple expression
array([  1.00000000e+00,   2.00000000e+00,   3.00000000e+00, ...,
         9.99998000e+05,   9.99999000e+05,   1.00000000e+06])

>>> ne.evaluate('a*b-4.1*a > 2.5*b')   # a more complex one
array([False, False, False, ..., True, True, True], dtype=bool)
```

Numexpr timings

Comparisons to Numpy and thread usage

```
>>> timeit a**2 + b**2 + 2*a*b
10 loops, best of 3: 35.9 ms per loop

>>> ne.set_num_threads(1)  # use 1 thread (on a 6-core machine)

>>> timeit ne.evaluate("a**2 + b**2 + 2*a*b")
100 loops, best of 3: 9.28 ms per loop   # 3.9x faster than NumPy

>>> ne.set_num_threads(4)  # use 4 threads (on a 6-core machine)

>>> timeit ne.evaluate("a**2 + b**2 + 2*a*b")
100 loops, best of 3: 4.17 ms per loop   # 8.6x faster than NumPy
```