High Performance Computing with Python (4 hour tutorial)

EuroPython 2011

Goal

- Get you writing faster code for CPU-bound problems using Python
- Your task is probably in pure Python, is CPU bound and can be parallelised (right?)
- We're not looking at network-bound problems
- Profiling + Tools == Speed

Get the source please!

- http://tinyurl.com/europyhpc
- (original: http://ianozsvald.com/wp-content/hpc_tutoria
-)
- google: “github ianozsvald”, get HPC full source (but you can do this after!)

About me (Ian Ozsvald)

- A.I. researcher in industry for 12 years
- C, C++, (some) Java, Python for 8 years
- Demo'd pyCUDA and Headroid last year
- Lecturer on A.I. at Sussex Uni (a bit)
- ShowMeDo.com co-founder
- Python teacher, BrightonPy co-founder
- IanOzsvald.com - MorConsulting.com

Overview (pre-requisites)

- cProfile, line_profiler, runsnake
- numpy
- Cython and ShedSkin
- multiprocessing
- ParallelPython
- PyPy
- pyCUDA

We won't be looking at...

- Algorithmic choices, clusters or cloud
- Gnumpy (numpy->GPU)
- Theano (numpy(ish)->CPU/GPU)
- CopperHead (numpy(ish)->GPU)
- BottleNeck (Cython'd numpy)
- Map/Reduce
- pyOpenCL

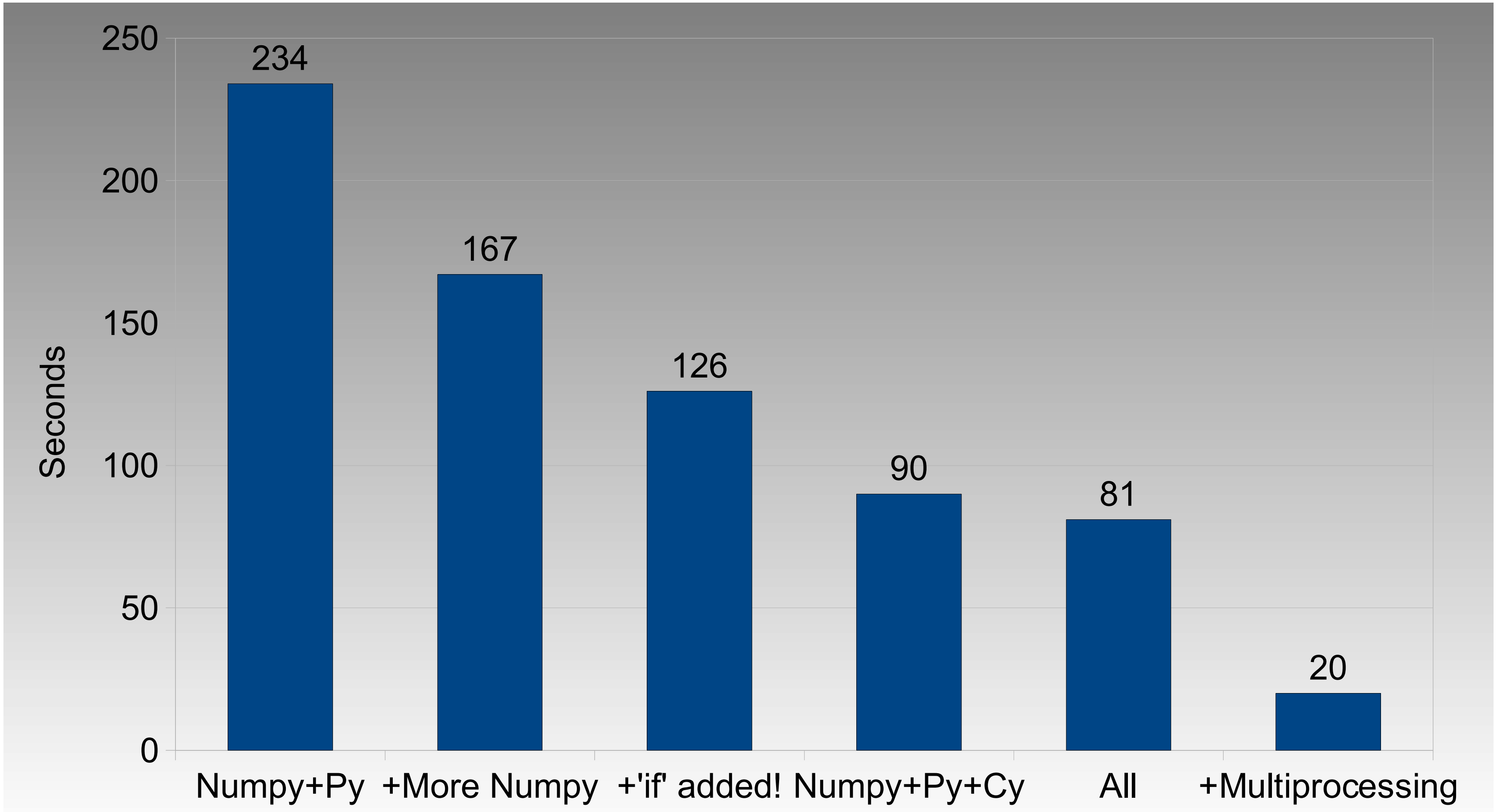
Something to consider

- “Proebsting's Law”
- http://research.microsoft.com/en-us/um/people
- Compiler advances (generally) unhelpful (sort-of – consider auto vectorisation!)
- Multi-core common
- Very-parallel (CUDA, OpenCL, MS AMP, APUs) should be considered

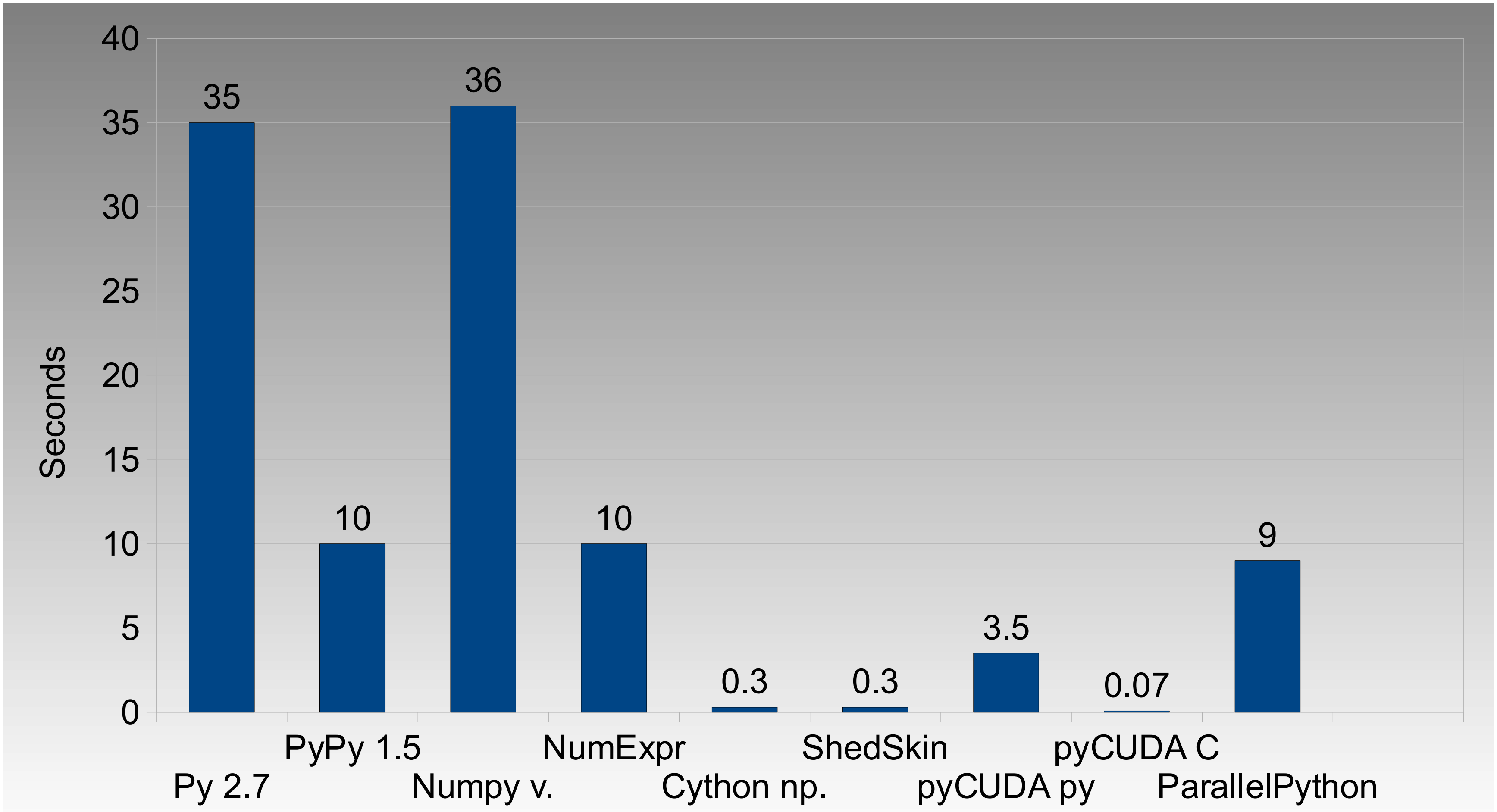
What can we expect?

- Close to C speeds (shootout):
  - http://attractivechaos.github.com/plb/
  - http://shootout.alioth.debian.org/u32/which-p
- Depends on how much work you put in
- nbody JavaScript much faster than Python but we can catch it/beat it (and get close to C speed)

Practical result - PANalytical



Mandelbrot results (Desktop i3)



Our code

- `pure_python.py`
- `numpy_vector.py`
- `pure_python.py 1000 1000 # RUN`
- Our two building blocks
- Google “github ianozsvald” -> EuroPython2011_HighPerformanceComputing
- https://github.com/ianozsvald/EuroPython2011

Profiling bottlenecks

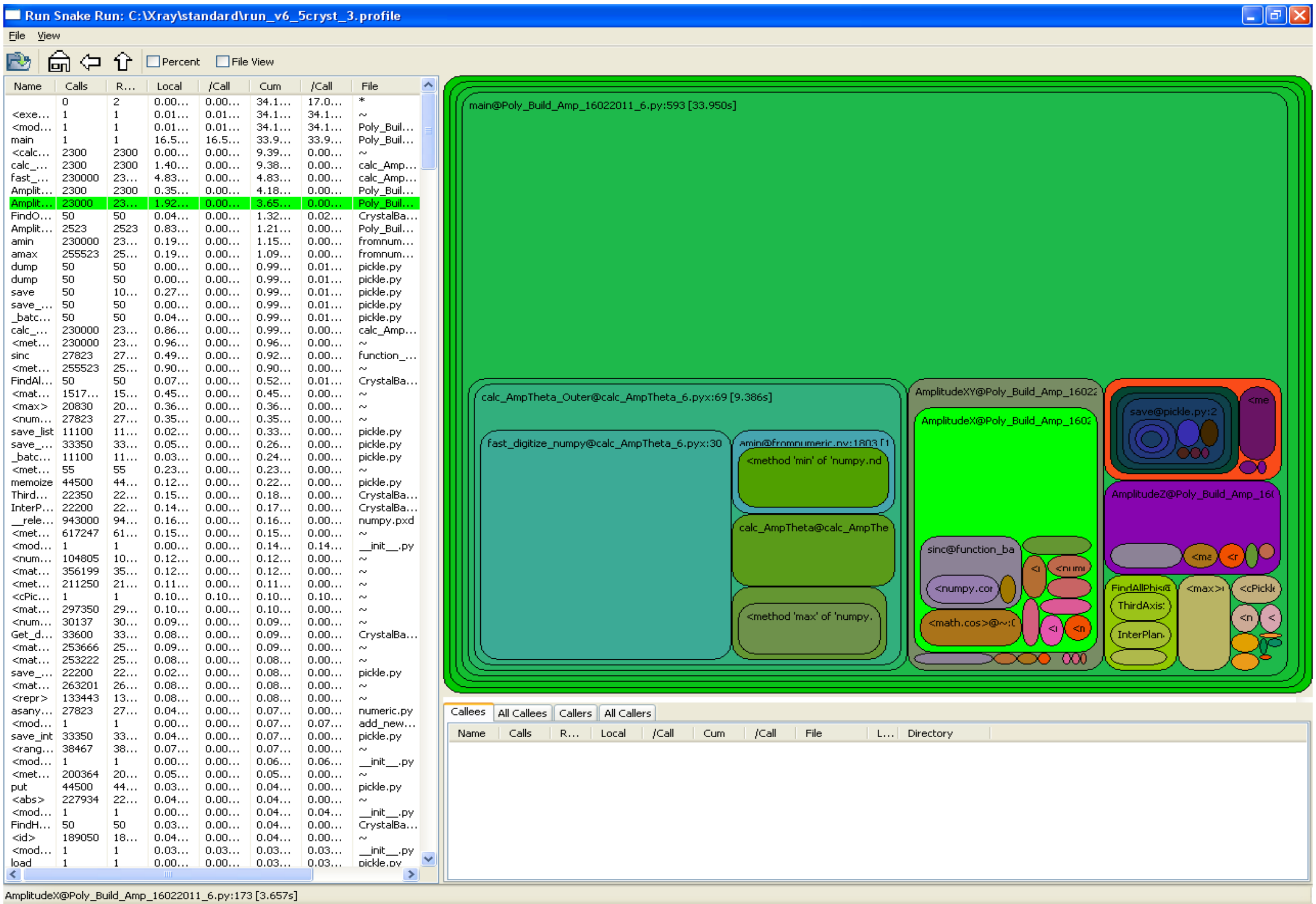
- `python -m cProfile -o rep.prof pure_python.py 1000 1000`
- `import pstats`
- `p = pstats.Stats('rep.prof')`
- `p.sort_stats('cumulative').print_stats(10)`

cProfile output

```
51923594 function calls (51923523 primitive calls)
in 74.301 seconds

ncalls  tottime  percall  cumtime  percall

pure_python.py:1(<module>)
     1    0.034    0.034   74.303   74.303

pure_python.py:23(calc_pure_python)
     1    0.273    0.273   74.268   74.268

pure_python.py:9(calculate_z_serial_purepython)
     1   57.168   57.168   73.580   73.580

{abs}
 51,414,419 12.465    0.000   12.465    0.000

...
```

RunSnakeRun

Mor Consulting



Let's profile python.py

- `python -m cProfile -o res.prof pure_python.py 1000 1000`
- `runsnake res.prof`
- Let's look at the result

What's the problem?

- What's really slow?
- Useful from a high level...
- We want a line profiler!

line_profiler.py

- `kernprof.py -l -v pure_python_lineprofiler.py 1000 1000`
- Warning...slow! We might want to use `300 100`

kernprof.py output

```
...% Time   Line Contents
=====================
             @profile
             def calculate_z_serial_purepython(q,
  maxiter, z):
     0.0         output = [0] * len(q)
     1.1         for i in range(len(q)):
    27.8             for iteration in range(maxiter):
    35.8                 z[i] = z[i]*z[i] + q[i]
    31.9                 if abs(z[i]) > 2.0:
```

Dereferencing is slow

- Dereferencing involves lookups – slow
- Our '`i`' changes slowly
- `zi = z[i]; qi = q[i] # DO IT`
- Change all `z[i]` and `q[i]` references
- Run `kernprof` again
- Is it cheaper?

We have faster code

- pure_python_2.py is faster, we'll use this as the basis for the next steps
- There are tricks:
  – sets over lists if possible
  – use dict[] rather than dict.get()
  – build-in sort is fast
  – list comprehensions
  – map rather than loops

PyPy 1.5

- Confession – I'm a newbie
- Probably cool tricks to learn
- `pypy pure_python_2.py 1000 1000`
- PIL support, numpy isn't
- My (bad) code needs numpy for display (maybe you can fix that?)
- `pypy -m cProfile -o runpypy.prof pure_python_2.py 1000 1000 # abs but no range`

Cython

- Manually add types, converts to C
- .pyx files (built on Pyrex)
- Win/Mac/Lin with gcc, msvc etc
- 10-100* speed-up
- numpy integration
- http://cython.org/

Cython on pure_python_2.py

- `# ./cython_pure_python`
- Make `calculate_z.py`, test it works
- Turn `calculate_z.py` to `.pyx`
- Add `setup.py` (see Getting Started doc)
- `python setup.py build_ext --inplace`
- `cython -a calculate_z.pyx` to get profiling feedback (.html)

Cython types

- Help Cython by adding annotations:
  - `list q z`
  - `int`
  - `unsigned int # hint no negative indices with for loop`
  - `complex and complex double`
- How much faster?

Compiler directives

- http://wiki.cython.org/enhancements/compilerd
- We can go faster (maybe):
  - `#cython: boundscheck=False`
  - `#cython: wraparound=False`
- Profiling:
  - `#cython: profile=True`
- Check profiling works
- Show `_2_bettermath # FAST!`

ShedSkin

- http://code.google.com/p/shedskin/
- Auto-converts Python to C++ (auto type inference)
- Can only import modules that have been implemented
- No numpy, PIL etc but great for writing new fast modules
- 3000 SLOC 'limit', always improving

Easy to use

- `# ./shedskin/`
- `shedskin shedskin1.py`
- `make`
- `./shedskin1 1000 1000`
- `shedskin shedskin2.py; make`
- `./shedskin2 1000 1000 # FAST!`
- No easy profiling, complex is slow (for now)

numpy vectors

- http://numpy.scipy.org/
- Vectors not brilliantly suited to Mandelbrot (but we'll ignore that...)
- numpy is very-parallel for CPUs
- `a = numpy.array([1,2,3,4])`
- `a *= 3 ->`

```
numpy.array([3,6,9,12])
```

Vector outline...

```
# ./numpy_vector/numpy_vector.py
for iteration...
  z = z*z + q
  done = np.greater(abs(z), 2.0)
  q = np.where(done,0+0j, q)
  z = np.where(done,0+0j, z)
  output = np.where(done,
    iteration, output)
```

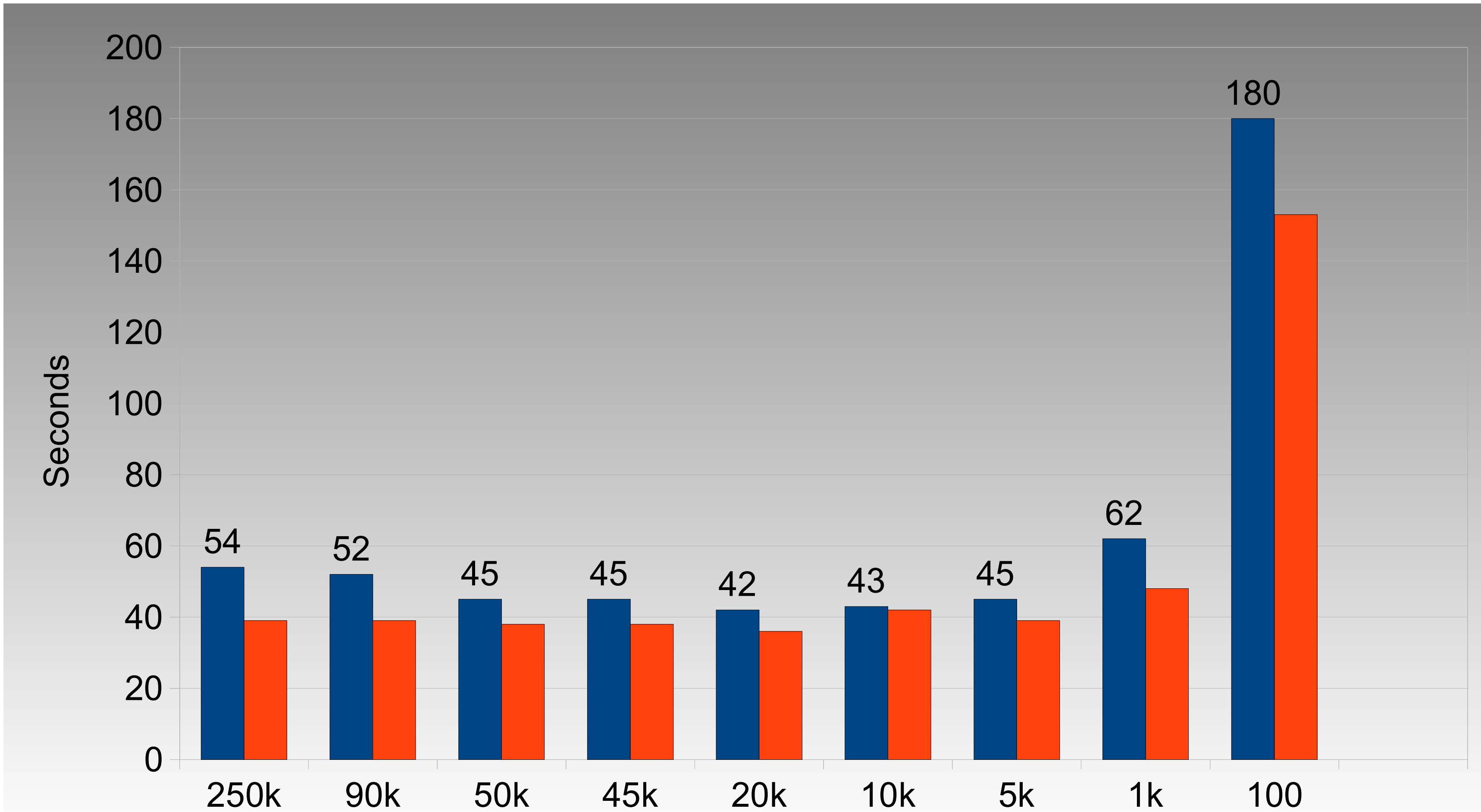
Profiling some more

- `python numpy_vector.py 1000 1000`
- `kernprof.py -l -v numpy_vector.py 300 100`
- How could we break out early?
- How big is 250,000 complex numbers?
- `# .nbytes, .size`

Cache sizes

- Modern CPUs have 2-6MB caches
- Tuning is hard (and may not be worthwhile)
- Heuristic: Either keep it tiny (<64KB) or worry about really big data sets (>20MB)
- `# numpy_vector_2.py`

Speed vs cache size (Core2/i3)



NumExpr

- http://code.google.com/p/numexpr/
- This is magic
- With Intel MKL it goes even faster
- `# ./numpy_vector_numexpr/`
- `python numpy_vector_numexpr.py 1000 1000`
- Now convert your `numpy_vector.py`

numpy and iteration

- Normally there's no point using numpy if we aren't using vector operations
- `python numpy_loop.py 1000 1000`
- Is it any faster?
- Let's run `kernprof.py` on this and the earlier `pure_python_2.py`
- Any significant differences?

Cython on numpy_loop.py

- Can low-level C give us a speed-up over vectorised C?
- `# ./cython_numpy_loop/`
- http://docs.cython.org/src/tutorial/numpy.html
- Your task – make .pyx, start without types, make it work from `numpy_loop.py`
- Add basic types, use `cython -a`

multiprocessing

- Using all our CPUs is cool, 4 are common, 8 will be common
- Global Interpreter Lock (isn't our enemy)
- Silo'd processes are easiest to parallelise
- http://docs.python.org/library/multiprocessing.h

multiprocessing Pool

- `# ./multiprocessing/multi.py`
- `p = multiprocessing.Pool()`
- `po = p.map_async(fn, args)`
- `result = po.get() # for all po objects`
- join the result items to make full result

Making chunks of work

- Split the work into chunks (follow my code)
- Splitting by number of CPUs is good
- Submit the jobs with map_async
- Get the results back, join the lists

Code outline

- Copy my chunk code

```
output = []
for chunk in chunks:
  out = calc...(chunk)
  output += out
```

ParallelPython

- Same principle as multiprocessing but allows >1 machine with >1 CPU
- http://www.parallelpython.com/
- Seems to work poorly with lots of data (e.g. 8MB split into 4 lists...!)
- We can run it locally, run it locally via ppserver.py and run it remotely too
- Can we demo it to another machine?

ParallelPython + binaries

- We can ask it to use modules, other functions and our own compiled modules
- Works for Cython and ShedSkin
- Modules have to be in PYTHONPATH (or current directory for ppserver.py)
- `parallelpython_cython_pure_python`

Challenge...

- Can we send binaries (.so/.pyd) automatically?
- It looks like we could
- We'd then avoid having to deploy to remote machines ahead of time...
- Anybody want to help me?

pyCUDA

- NVIDIA's CUDA -> Python wrapper
- http://mathema.tician.de/software/pycuda
- Can be a pain to install...
- Has numpy-like interface and two lower level C interfaces

pyCUDA demos

- `# ./pyCUDA/`
- I'm using float32/complex64 as my CUDA card is too old :-((Compute 1.3)
- numpy-like interface is easy but slow
- elementwise requires C thinking
- sourcemodule gives you complete control
- Great for prototyping and moving to C

Birds of Feather?

- numpy is cool but CPU bound
- pyCUDA is cool and is numpy-like
- Could we monkey patch numpy to auto-run CUDA(/openCL) if a card is present?
- Anyone want to chat about this?

Future trends

- multi-core is obvious
- CUDA-like systems are inevitable
- write-once, deploy to many targets – that would be lovely
- Cython+ShedSkin could be cool
- Parallel Cython could be cool
- Refactoring with rope is definitely cool

Bits to consider

- Cython being wired into Python (GSoC)
- CorePy assembly -> numpy
  http://numcorepy.blogspot.com/
- PyPy advancing nicely
- GPUs being interwoven with CPUs (APU)
- numpy+NumExpr->GPU/CPU mix?
- Learning how to massively parallelise is the key

Feedback

- I plan to write this up
- I want feedback (and maybe a testimonial if you found this helpful?)
- ian@ianozsvald.com
- Thank you :-)