

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_tutorial
-)
- 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

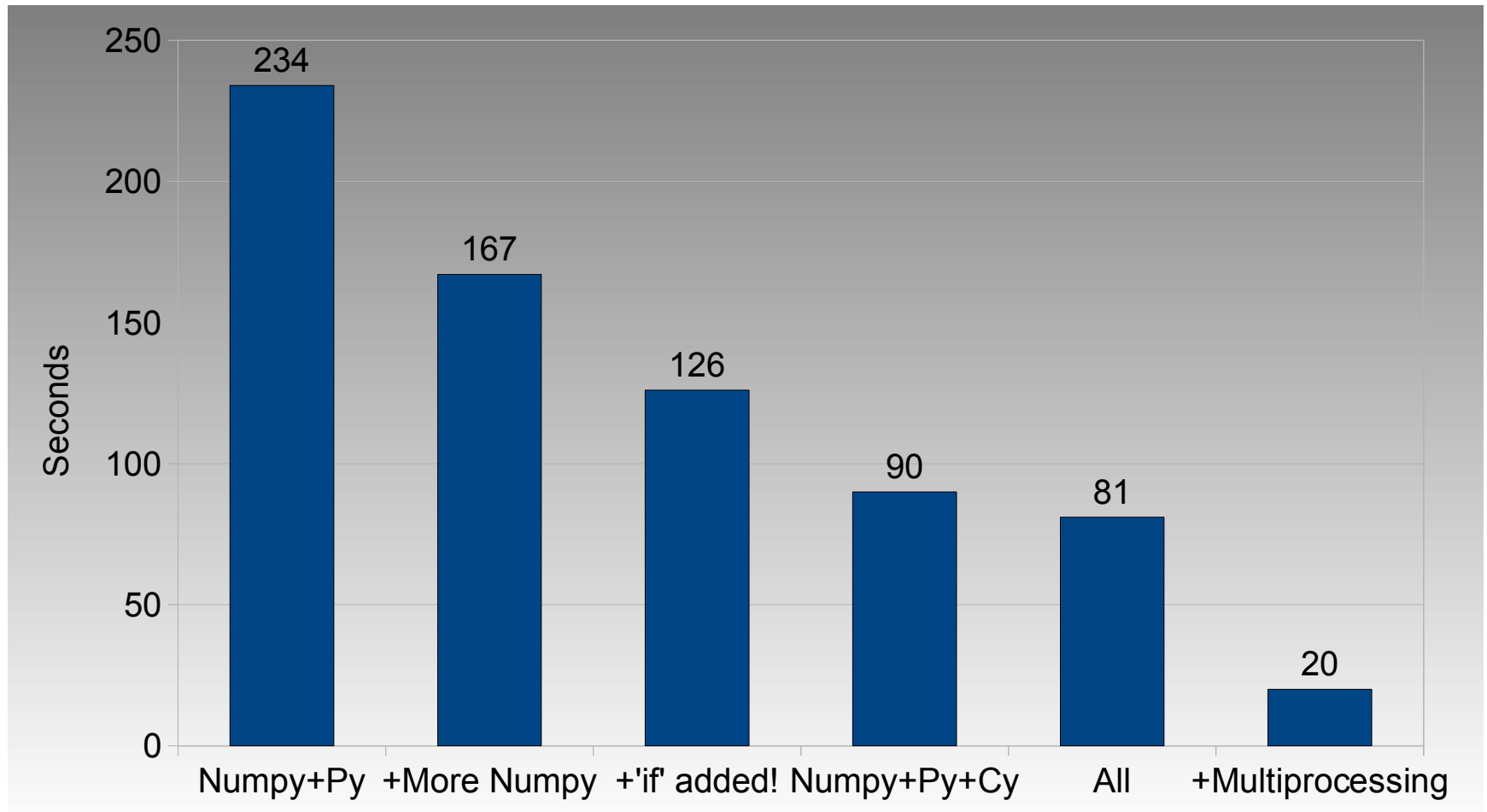
Something to consider

- “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

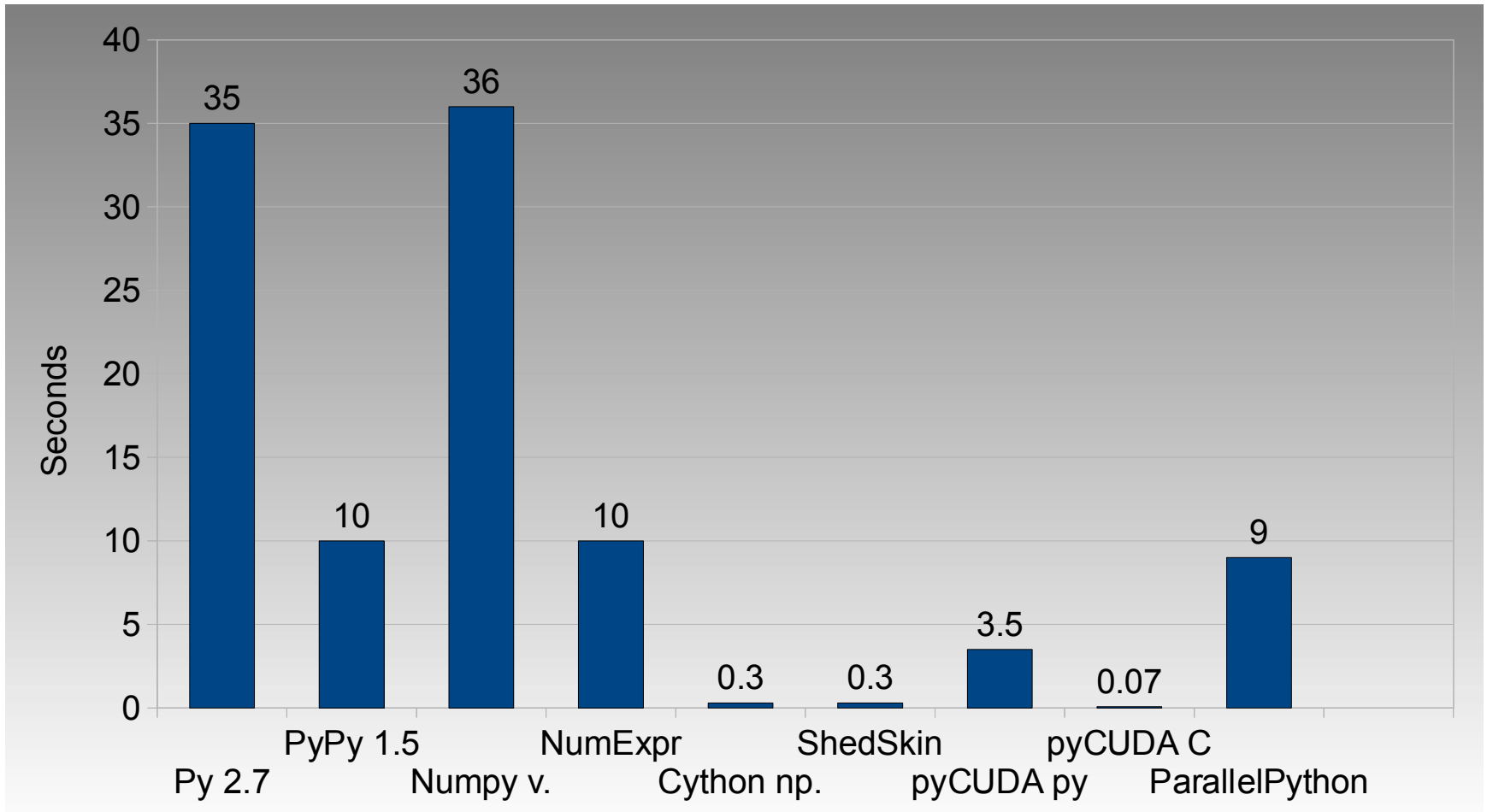
What can we expect?

- 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_HighPerformanceCom
puting
- <https://github.com/ianozsvald/EuroPython2011>

Profiling bottlenecks

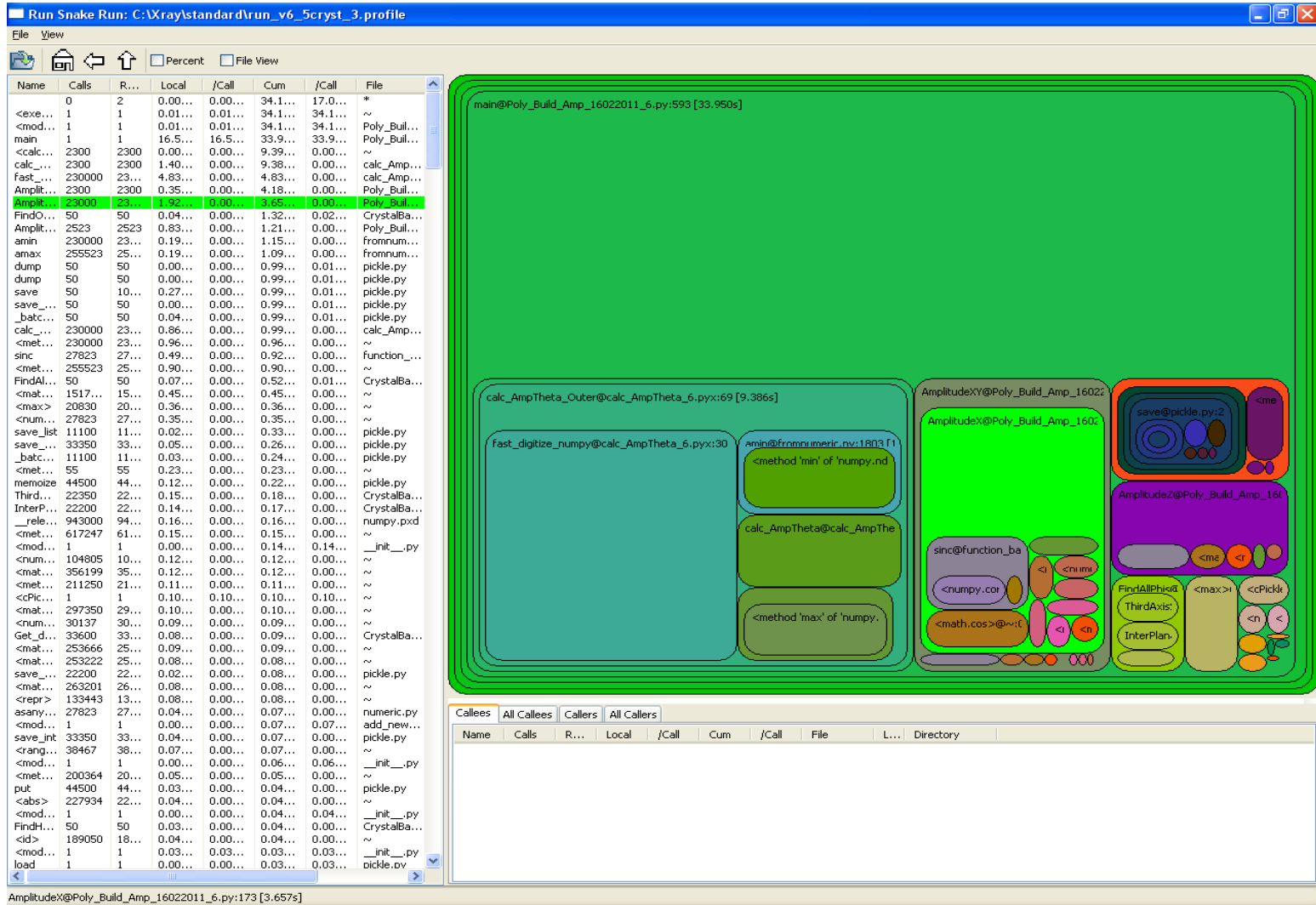
- `python -m cProfile -o rep.prof
pure_python.py 1000 1000`
- `import pstats`
- `p = pstats.Stats('rep.prof')`
- `p.sort_stats('cumulative').pri
nt_stats(10)`

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



The screenshot shows the RunSnakeRun application interface. On the left is a call stack table with columns: Name, Calls, R..., Local, /Call, Cum, /Call, File. The table lists various function calls, with the top entry being 'main@Poly_Build_Amp_16022011_6.py:593 [33.950s]'. Below the table is a call graph visualization showing a hierarchy of function calls. The root node is 'main@Poly_Build_Amp_16022011_6.py:593 [33.950s]'. It branches into several nodes, including 'calc_AmpTheta_Outer@calc_AmpTheta_6.py:69 [9.386s]', 'AmplitudeXY@Poly_Build_Amp_1602...', and 'save@picde.py:2'. The graph uses different colors to represent different parts of the call stack. At the bottom, there is a 'Callees' section with a table showing the relationship between callees and callers.

Name	Calls	R...	Local	/Call	Cum	/Call	File
0	2	0.00...	0.00...	34.1...	17.0...	*	~
<exe...	1	1	0.01...	0.01...	34.1...	34.1...	~
<mod...	1	1	0.01...	0.01...	34.1...	34.1...	Poly_Bul...
main	1	1	16.5...	16.5...	33.9...	33.9...	Poly_Bul...
<calc...	2300	2300	0.00...	0.00...	9.39...	0.00...	~
calc_...	2300	2300	1.40...	0.00...	9.38...	0.00...	calc_Amp...
fast_...	230000	23...	4.83...	0.00...	4.83...	0.00...	calc_Amp...
Amplit...	23000	2300	0.35...	0.00...	4.18...	0.00...	Poly_Bul...
Amplit...	23000	23...	1.92...	0.00...	3.65...	0.00...	Poly_Bul...
FindO...	50	50	0.04...	0.00...	1.32...	0.02...	CrystalBa...
Amplit...	2523	2523	0.83...	0.00...	1.21...	0.00...	Poly_Bul...
amin	230000	23...	0.19...	0.00...	1.15...	0.00...	fromnum...
amax	255523	25...	0.19...	0.00...	1.09...	0.00...	fromnum...
dump	50	50	0.00...	0.00...	0.99...	0.01...	pickle.py
dump	50	50	0.00...	0.00...	0.99...	0.01...	pickle.py
save	50	10...	0.27...	0.00...	0.99...	0.01...	pickle.py
save_...	50	50	0.00...	0.00...	0.99...	0.01...	pickle.py
_bata...	50	50	0.04...	0.00...	0.99...	0.01...	pickle.py
calc_...	230000	23...	0.86...	0.00...	0.99...	0.00...	calc_Amp...
<met...	230000	23...	0.96...	0.00...	0.96...	0.00...	~
sinc	27823	27...	0.49...	0.00...	0.92...	0.00...	function_...
<met...	255523	25...	0.90...	0.00...	0.90...	0.00...	~
FindAl...	50	50	0.07...	0.00...	0.52...	0.01...	CrystalBa...
<mat...	1517...	15...	0.45...	0.00...	0.45...	0.00...	~
<max>	20830	20...	0.36...	0.00...	0.36...	0.00...	~
<num>	27823	27...	0.35...	0.00...	0.35...	0.00...	~
save_list	11100	11...	0.02...	0.00...	0.33...	0.00...	pickle.py
save_...	33350	33...	0.05...	0.00...	0.28...	0.00...	pickle.py
_bata...	11100	11...	0.03...	0.00...	0.24...	0.00...	pickle.py
<met...	55	55	0.23...	0.00...	0.23...	0.00...	~
memoize	44500	44...	0.12...	0.00...	0.22...	0.00...	pickle.py
Third...	22350	22...	0.15...	0.00...	0.18...	0.00...	CrystalBa...
InterP...	22200	22...	0.14...	0.00...	0.17...	0.00...	CrystalBa...
_rele...	943000	94...	0.16...	0.00...	0.16...	0.00...	numpy.pxd
<met...	617247	61...	0.15...	0.00...	0.15...	0.00...	~
<mod...	1	1	0.00...	0.00...	0.14...	0.14...	__init__.py
<num...	104805	10...	0.12...	0.00...	0.12...	0.00...	~
<mat...	356199	35...	0.12...	0.00...	0.12...	0.00...	~
<met...	211250	21...	0.11...	0.00...	0.11...	0.00...	~
<cPic...	1	1	0.10...	0.10...	0.10...	0.10...	~
<mat...	297350	29...	0.10...	0.00...	0.10...	0.00...	~
<num...	30137	30...	0.09...	0.00...	0.09...	0.00...	~
Get_d...	33600	33...	0.08...	0.00...	0.09...	0.00...	CrystalBa...
<mat...	253666	25...	0.09...	0.00...	0.09...	0.00...	~
<mat...	253222	25...	0.08...	0.00...	0.08...	0.00...	~
save_...	22200	22...	0.02...	0.00...	0.08...	0.00...	pickle.py
<mat...	263201	26...	0.08...	0.00...	0.08...	0.00...	~
<repr>	133443	13...	0.08...	0.00...	0.08...	0.00...	~
asany...	27823	27...	0.04...	0.00...	0.07...	0.00...	numeric.py
<mod...	1	1	0.00...	0.00...	0.07...	0.07...	~
save_int	33350	33...	0.04...	0.00...	0.07...	0.00...	pickle.py
<rang...	38467	38...	0.07...	0.00...	0.07...	0.00...	~
<mod...	1	1	0.00...	0.00...	0.06...	0.06...	__init__.py
<met...	200364	20...	0.05...	0.00...	0.05...	0.00...	~
put	44500	44...	0.03...	0.00...	0.04...	0.00...	pickle.py
<abs>	227934	22...	0.04...	0.00...	0.04...	0.00...	~
<mod...	1	1	0.00...	0.00...	0.04...	0.04...	__init__.py
FindI...	50	50	0.03...	0.00...	0.04...	0.00...	CrystalBa...
<id>	189050	18...	0.04...	0.00...	0.04...	0.00...	~
<mod...	1	1	0.03...	0.03...	0.03...	0.03...	__init__.py
load	1	1	0.00...	0.00...	0.03...	0.03...	pickle.py

Let's profile python.py

- `python -m cProfile -o res.prof
pure_python.py 1000 1000`
- `runsake 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/compilerc>
- 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)
```

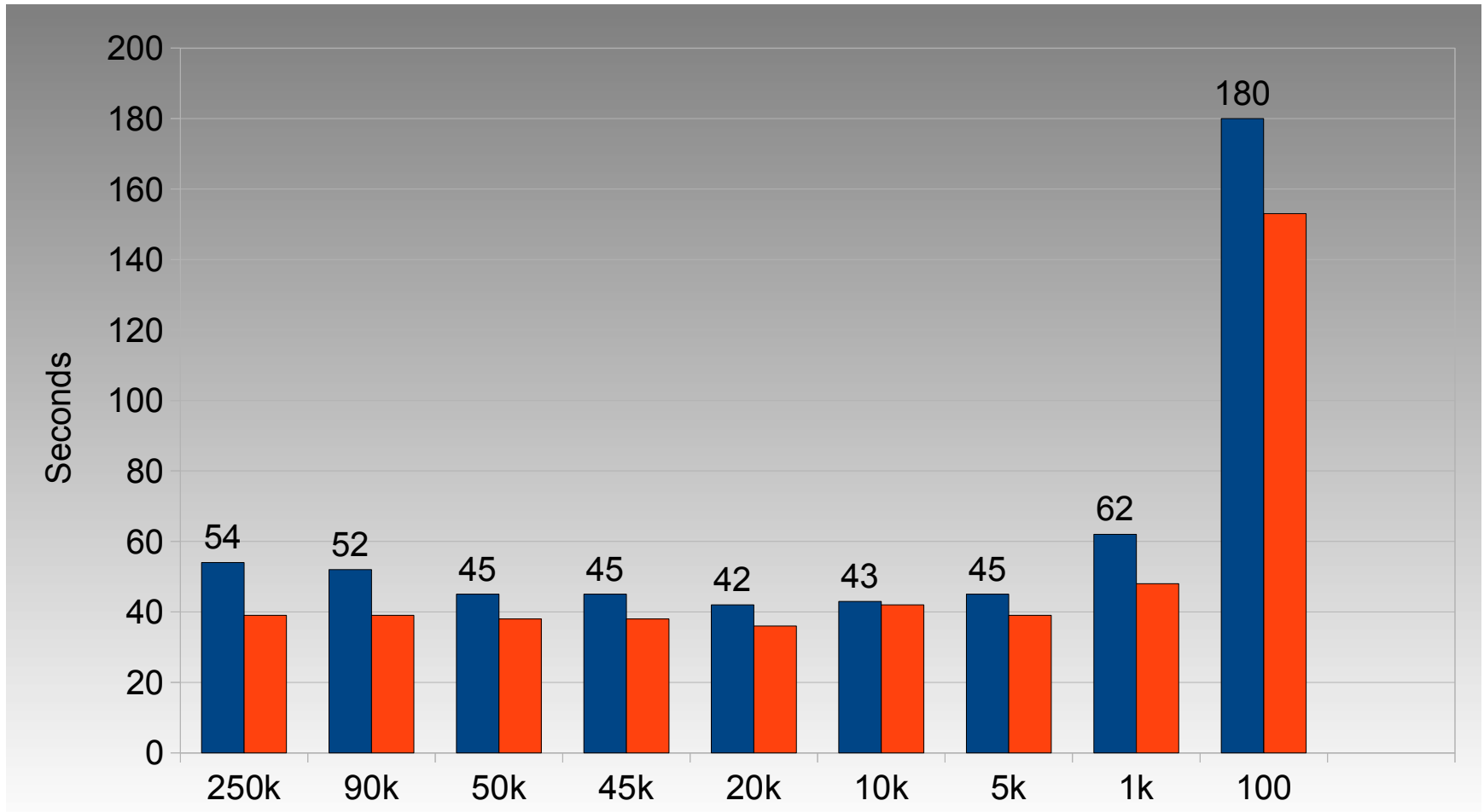
Profiling some more

- `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.html>

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://mathematician.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 :-)