Applied Parallel Computing with Python – List of Tasks

PyCon 2013

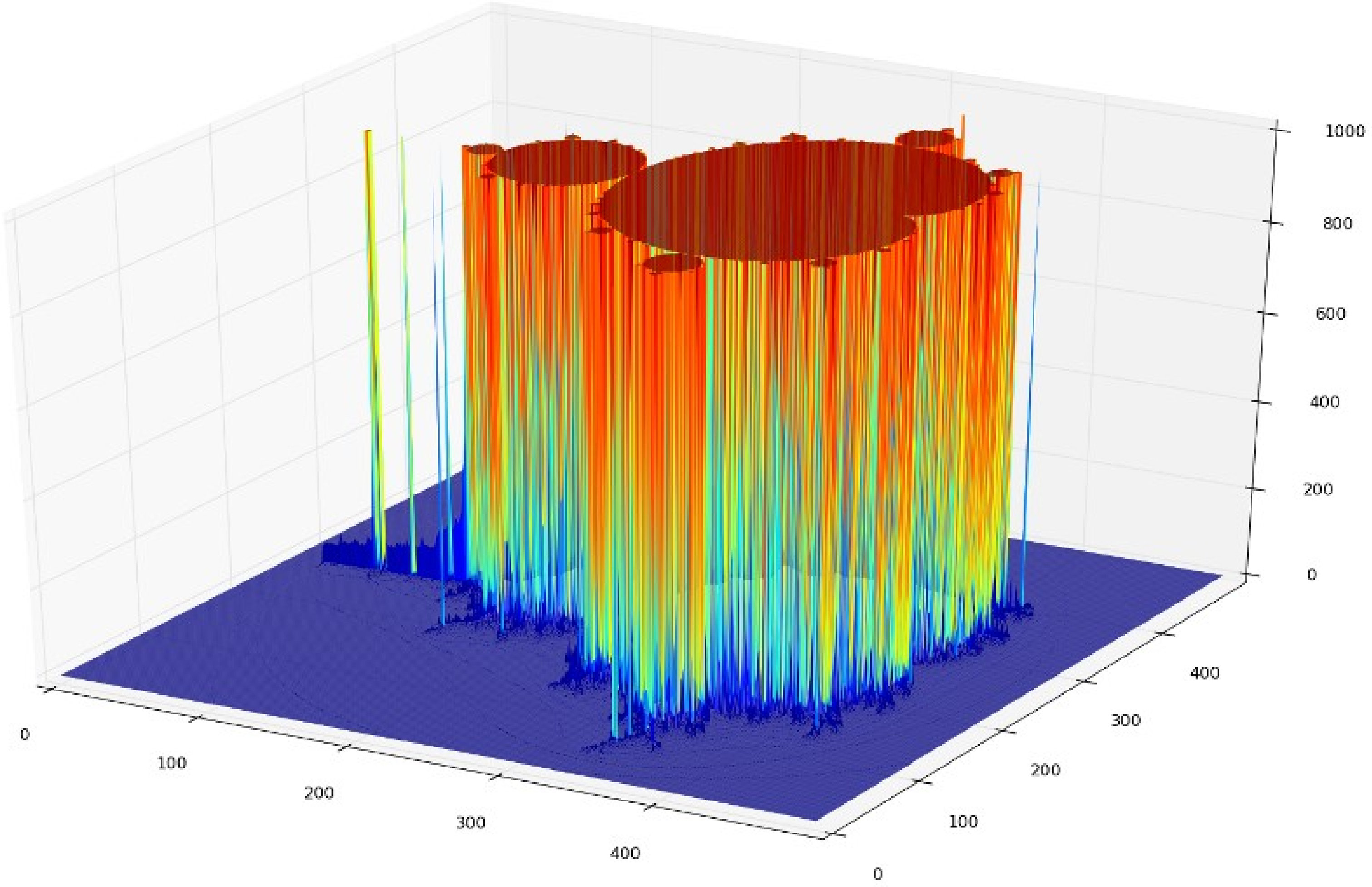
Goal

- Tackle CPU-bound tasks
- Accept the GIL
- Utilise many cores on many machines
- Maybe utilise many languages too

Overview (pre-requisites)

- multiprocessing
- ParallelPython
- hotqueue, redis (and Redis system)
- Matplotlib (for visualisations)

Mandelbrot as surface plot



Serial single thread

-
  ```
  $ python serial_python.py --plot3D --size 100
  ```
- 2500 elements

-
  ```
  $ python serial_python.py
  ```
- 250,000 elements
- 11 seconds on 1 core

Amdahl's law

- Max speed-up is limited to the parallelisable portions and resources
- What serial constraints do we have?
- How many data elements?
- How much memory?
- What affects transmission speed? Gigabit? Switches? Traffic?

Memory usage?

- `import sys`
- `sys.getsizeof(0+0j)   # 32 bytes`
- `250,000 * 32 == ? # lower-bound`
- Pickling and sending will take time
- Assembling the result will take time

Profile memory usage

- `Github fabianp memory_profiler`
- `$ python -m memory_profiler serial_python_temp.py #argparse`
- Output (takes a while):
- `61:q.append(complex...) # +25MB`
- `65:...=calculate_z(...) # +7MB`

multiprocessing

- Using all our CPUs is cool, 4 are common, 32 will be common
- Global Interpreter Lock (isn't our enemy)
- Silo'd processes are easiest to parallelise
- Forks on local machine (1 machine only)
- http://docs.python.org/library/multiprocessing

Making chunks of work

- Split the work into chunks
- Start splitting by number of CPUs
- Submit the jobs with map_async
- Get the results back, join the lists
- Profile and consider the results...

multiprocessing Pool

- `2_mandelbrot_multiprocessing/`
- `multiproc.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

multiprocessing

- 1 process takes 12 secs
- 2 takes 6 secs (watch System Monitor)
- 4 takes about 5 – what's happening?
- What about 32?

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?

Running ParallelPython

- Run

-
  ```
  $ python parallelpy.py #chunks
  ```
- Now to run server separately:

-
  ```
  $ ppserver.py -d -a # uses all CPUs
  ```

-
  ```
  $ python parallelpy_manymachines.py
  ```

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)

“timeout: timed out”

- Beware the timeout problem, the default timeout isn't helpful:
  - `pptransport.py`
  - `TRANSPORT_SOCKET_TIMEOUT = 60*60*24 # from 30s`
- Remember to edit this on *all* copies of `pptransport.py`

Redis queue

- Queue is persistent, architect. agnostic
- Server/client model, time shift ok
- `1$ python hotq.py # worker(s)`
- `2$ python hotq.py --server`
- What if many jobs get posted and you're consumers aren't running?
- Also->Amazon Simple Queue Service