Threading and Multiprocessing

Tom Dalton

GitHub: http://bit.ly/1DqIw5s

Parallelising Problems

Modern commodity computer hardware good at parallel computation

- PC Processors
- Graphic Cards

Why do stuff in Parallel?

- Some problems naturally structured this way
  - E.g. Calculate first 100 square numbers

- Can reduce the *real* time taken to get a result

Who not do stuff in Parallel?

- Some problems do not naturally parallelise
  - Calculating fibonacci sequence

- It’s Complicated!
  - Synchronisation

Threading

Lightweight parallelism supported by the OS

- share process memory
- separate processor

Threading

```
File  Edit  View  Search  Terminal  Help
top - 13:11:20 up  3:50,  4 users,  load average: 1.80, 0.59, 0.28
Tasks: 223 total,   1 running, 221 sleeping,   0 stopped,   1 zombie
%Cpu(s): 78.8 us,  2.0 sy,  0.0 ni, 19.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   8082836 total,  4126952 used,  3955884 free,    90824 buffers
KiB Swap:  8294396 total,        0 used,  8294396 free.  2024336 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
20185 tom       20   0 1081188  28004  11740 S 314.4  0.3   0:06.93 python
 1355 root      20   0  646592 229236 207764 S   4.0  2.8   5:09.39 Xorg
 2470 tom       20   0  491580  32476  13624 S   4.0  0.4   1:45.64 compiz
 2997 tom       20   0  889616  32616  14900 S   2.0  0.4   0:19.85 gnome-terminal
```

Multiprocessing

Separate processes can run together

- Often supported by OS
  - Shared memory, pipes, semaphores etc

In the context of this talk

- Working together for a common goal

Multiprocessing

```
File  Edit  View  Search  Terminal  Help
top - 13:14:58 up  3:53,  4 users,  load average: 0.69, 0.61, 0.36
Tasks: 230 total,   6 running, 223 sleeping,   0 stopped,   1 zombie
%Cpu(s): 99.0 us,  0.7 sy,  0.0 ni,  0.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:   8082836 total,  4351416 used,  3731420 free,    92744 buffers
KiB Swap:  8294396 total,        0 used,  8294396 free.  2107944 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
21373 tom       20   0  437876  17060   1104 R  94.4  0.2   0:04.82 python
21374 tom       20   0  437876  17064   1108 R  94.4  0.2   0:04.46 python
21372 tom       20   0  437876  17056   1100 R  93.4  0.2   0:04.45 python
21375 tom       20   0  437876  17052   1096 R  93.4  0.2   0:04.48 python
21368 tom       20   0  660252  31032  12088 S  14.9  0.4   0:04.25 python
 2470 tom       20   0  491784  32620  13624 S   3.0  0.4   1:50.37 compiz
 1355 root      20   0  660900 244432 222620 S   2.0  3.0   5:20.23 Xorg
```

Example Problems

1. File Search
2. Fractal Plotting

Problem 1: File Search

Find regexp matches in a file

Problem 1: File Search

Quantitative comparisons

- Test file 25M lines (~1.1GB)
- Lines of 10-80 chars
- Each char in [a-z] or space

Problem 1: File Search

Find all regexp matches in a file

```
~/python_edinburgh/parallelism_talk master › grep -n "abcde[a-m]" big_sample_file
211365:kxalvvufy ymqh aaa rplyjqdc bragfirzhwsrvibbxzabcdeirgsjxvn mksecvaswjt xeim
1112241:dlympvzxkcleabcdefobjglpcg igxg qvsx kuwxrvqnpmzjmhmekoifhbzwingd hkklk
2640629:bqsvvid zxv rx  fsowabcdehy  ag vydpbzfb
3523191:lho  rusezzx rlhbg qxwijlxnbxm az ezmmqoa edto abcdeetqf
4060083: sitdaqixucjz sptnztpyaozmbtu kazdeg  gaabcdehmr araanosh e  cfbnxqxmzcqz
4299996:mmkmhyzfecuyboicaphv eogljualf pdlhvabcdedawmszfzbfexkiankz  chmrri kwcsrtchd
4350217:glrxjwufmvcdxeaanzegvknddlcfmfihlqnsdmphczsababcdekcwihtudv opdwsqcf
6461019: x bkkicd pxm fwegjqylpsgabcdeibg ybmkf
7039561:jtgshvtvznm mngxueimhyiwtntdsgwxpabcdehwnk
9105581:nh abcdem gkjmigoj   z
10711291:ny wy fqpobgszflbdjhukmabcdeiiqlundayx elkc ii wq
10879350:dxeapeaqvlftr exhbh  d omufi nfoilctvpu mhzidljf nihmtf  wmem fo abcdemf
12363608:oa xprgf fjkabcdemqvlmid
13986724: ovfgn hricppdqnabcdedaq rfr vwzbfaorvt yoqtily
15527376: pyddudntqdrjeqlfccabcdee fllwdfntkrmmakn bztlwe  qthlio
18075538:nfgcjabcdeaeecdykmgk
18802776:bx tm  iwozcstfahgskbymmjfeut  ws sklpiouq qmo nabcdebljd pyhoayjyrh
20627195:d kygjmsrq xecr ibnsabcdefwbehbbsoewglrtvc nfcxrma stah ib evmvt
22332650:zamwf abcdehuxgjtgrrmesblhnunrrobad anxvjasddmpnqt kxxdl
~/python_edinburgh/parallelism_talk master ›
```

Problem 1: File Search

Trivial solution

```
for each line in the file:
    if line matches regexp:
        output line number and line
```

Problem 1: File Search

```
import re

class FileSearcher(object):

    def __init__(self, filename, regexp_str):
        self.filename = filename
        self.regexp = re.compile(regexp_str)

    def find_matches(self):
        results = {}
        with open(self.filename, "rb") as f:
            for i, line in enumerate(f):
                matchobj = self.regexp.match(line)
                if matchobj:
                    results[i] = line
        return results
```

Problem 1: File Search

Results (Elapsed real time):

Algorithm	Regexp matches no lines	Regexp matches all lines
Trivial	~10s	~14s

Problem 1: File Search

Parallelised solutions (1)

Read file into chunks

- regexp-match chunks in parallel

Problem 1: File Search

Results (Elapsed real time):

Algorithm	Regexp matches no lines	Regexp matches all lines
Trivial	~10s	~14s
Threaded Regexp	~21s	~27s
Multiprocess Regexp	~30s	~60s

Problem 1: File Search

Why are these results so bad?

Threading:

- GIL

Multiprocessing:

- multiprocessing.Queue -> Pickle

Problem 1: File Search

What can we do about these problems?

GIL

- Use multiprocessing (!)
- C extensions (more later)

Problem 1: File Search

What can we do about these problems?

Pickle

- Avoid pickling large data sets
- Use shared memory or other IPC

Problem 1: File Search

Parallelised solutions (2)

Every thread/process reads the file

- Calculates its own chunk and processes it

Problem 1: File Search

Results (Elapsed real time):

Algorithm	Regexp matches no lines	Regexp matches all lines
Trivial	~10s	~14s
Threaded Regexp	~21s	~27s
Multiprocess Regexp	~30s	~60s
Threaded Chunking	~26s	~37s

Problem 1: File Search

Results (Elapsed real time):

Algorithm	Regexp matches no lines	Regexp matches all lines
Trivial	~10s	~14s
Threaded Regexp	~21s	~27s
Multiprocess Regexp	~30s	~60s
Threaded Chunking	~26s	~37s
Multiprocess Chunking	~6s	~37s

Problem 1: File Search

Results (Elapsed real time):

Algorithm	Regexp matches no lines	Regexp matches all lines
Trivial	~10s	~14s
Threaded Regexp	~21s	~27s
Multiprocess Regexp	~30s	~60s
Threaded Chunking	~26s	~37s
Multiprocess Chunking	~6s	~37s
Grep	~2s	~3s

Problem 1: File Search

Conclusion

For simple problems, it can be hard to beat the overheads

Problem 1: File Search

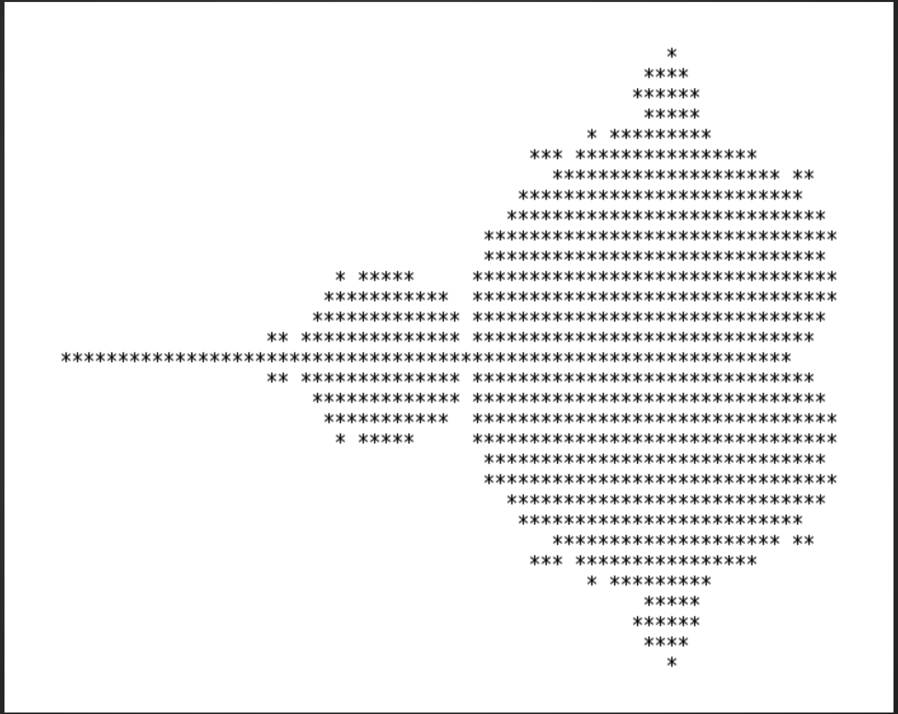
Problem 2: Plotting Fractals

Mandelbrot fractal 101

- Each point represents a complex number
- Points may diverge or not
- Because maths!
- Pixel colour relates to divergence speed

TLDR Each pixel requires doing a hard sum

Problem 2: Plotting Fractals



http://commons.wikimedia.org/wiki/File:Mandel.png

Problem 2: Plotting Fractals

The baseline/control

- Single process/thread
- Calculates each pixel value
- Pixel calc implemented in pure python

Problem 2: Plotting Fractals

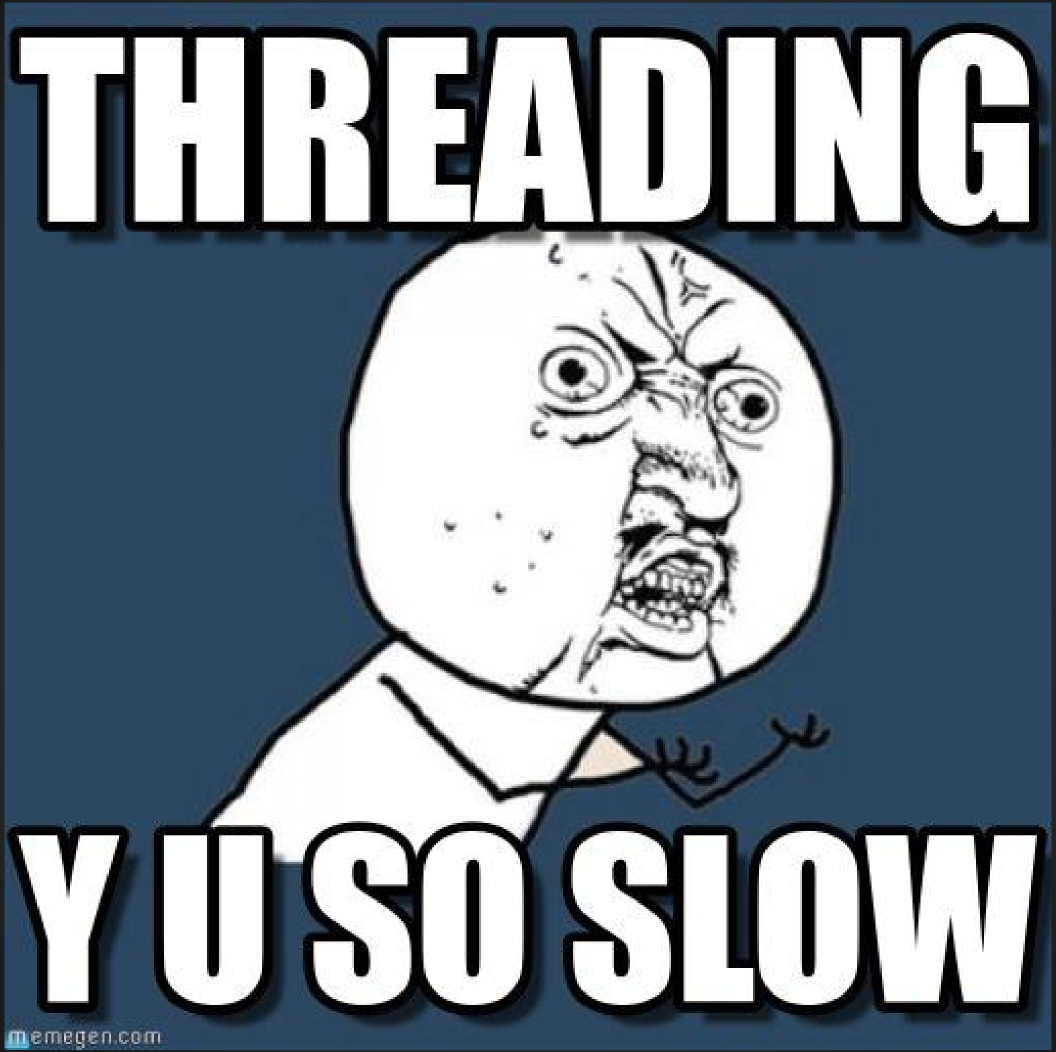
Results (Elapsed real time):

Algorithm	Fractal Calc	Iterations	Render Time
Simple	Python	256	~26s

Problem 2: Plotting Fractals

Results (Elapsed real time):

Algorithm	Fractal Calc	Iterations	Render Time
Simple	Python	256	~26s
Threaded	Python	256	~41s
Multiprocess	Python	256	~15s


THREADING
Y U SO SLOW
memegen.com

Problem 2: Plotting Fractals

Threaded slowest!

- GIL

Problem 2: Plotting Fractals

What can we do about these problems?

GIL

- Use multiprocessing
- C extensions

Problem 2: Plotting Fractals

```
int _calc_point(double x0, double y0, int max_iterations) {
      int i;
      double x_temp;
      double x = 0.0;
      double y = 0.0;
      for (i=0; i < max_iterations; i++) {
            if (x * x + y * y >= 4.0) {
                  return i;
            }
            x_temp = x * x - y * y + x0;
            y = 2 * x * y + y0;
            x = x_temp;
      }
      return -1;
}
```

Problem 2: Plotting Fractals

```
static PyObject * calc_point(PyObject *self, PyObject *args) {
	double x0, y0;
	int i, max_iterations;
	if (!PyArg_ParseTuple(args, "ddi", &x0, &y0, &max_iterations)) {
		return NULL;
	}
	Py_BEGIN_ALLOW_THREADS;
	i = _calc_point(x0, y0, max_iterations);
	Py_END_ALLOW_THREADS;
	if (i == -1) {
		Py_RETURN_NONE;
	} else {
		return Py_BuildValue("i", i);
	}
};
```

Problem 2: Plotting Fractals

```
static PyMethodDef c_mandelbrot_methods[] = {
      {"calc_point",  calc_point, METH_VARARGS, "Calculate a point on a mandelbrot fractal."},
      {NULL, NULL, 0, NULL}
};

PyMODINIT_FUNC initcmandelbrot(void) {
      (void) Py_InitModule("cmandelbrot", c_mandelbrot_methods);
};
```

Problem 2: Plotting Fractals

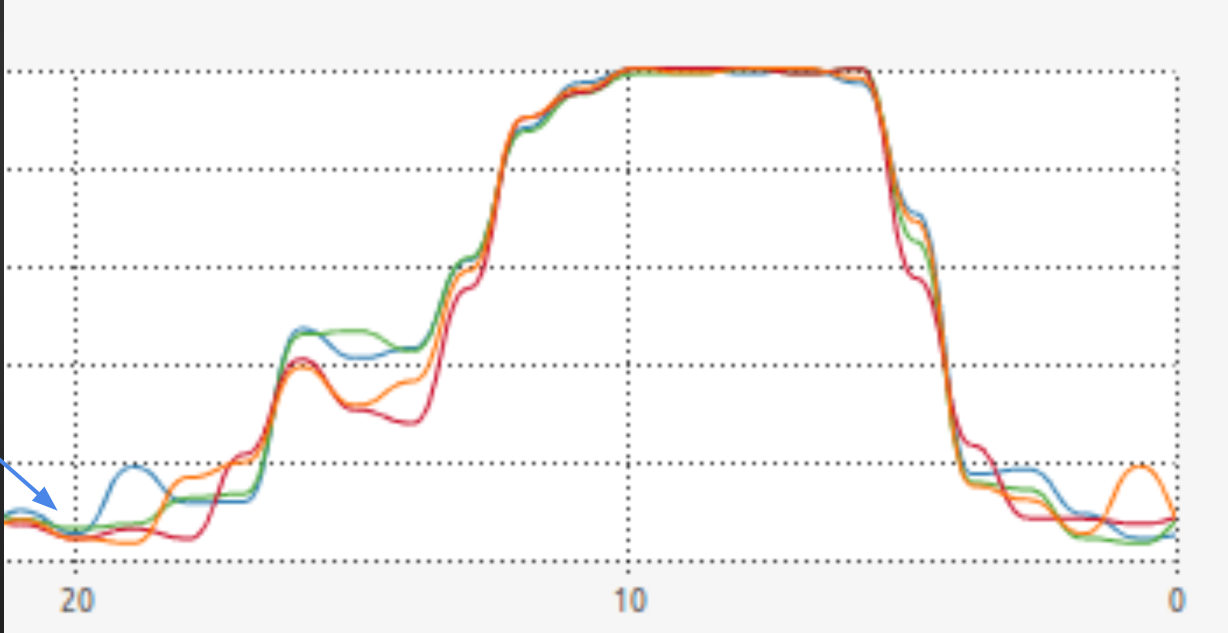
Results (Elapsed real time):

Algorithm	Fractal Calc	Iterations	Render Time
Simple	C (optimised)	20000	~26s
Threaded	C (optimised)	20000	~17s
Multiprocess	C (optimised)	20000	~12s

Problem 2: Plotting Fractals

Multiprocess CPU Graph

Render Start



Problem 2: Plotting Fractals

Final Optimisations

- Line-centric
- Random render order

Problem 2: Plotting Fractals

Results (Elapsed real time):

Algorithm	Fractal Calc	Iterations	Render Time
Multiprocess	C (optimised)	20000	~12s
Threaded (line)	C (optimised)	20000	~10s
Multiprocess (line)	C (optimised)	20000	~10s
Threaded (line, random)	C (optimised)	20000	~9s
Multiprocess (line, random)	C (optimised)	20000	~8s

Problem 2: Plotting Fractals

Results (Elapsed real time):

Algorithm	Fractal Calc	Iterations	Render Time
Multiprocess	C (optimised)	20000	~12s
Multiprocess (line)	C (optimised)	20000	~10s
Multiprocess (line, random)	C (optimised)	20000	~8s
Multiprocess (line, random)	Python	20000	~819s

Conclusions

Threading

- Simple
- Lots of Caveats

Multiprocessing

- More complex/less performant IPC
- Overall faster for both examples

WAT

https://www.destroyallsoftware.com/talks/wat

- Gary Bernhardt
- CodeMash 2012

WAT

```
#!/usr/bin/env python
from multiprocessing import Process, Queue

def thing(queue, x):
    result = x * 2
    queue.put(result)

q = Queue()
p = Process(target=thing, args=(q, 10, ))

p.start()

p.join()
print q.get()
```

WAT

```
› ./wat.py
20
```

WAT

```
#!/usr/bin/env python
from multiprocessing import Pool, Queue

def thing(queue, x):
    result = x * 2
    queue.put(result)

q = Queue()
pool = Pool(1)

print pool.apply(thing, (q, 10, ))
```

WAT

```
› ./wat2.py
Traceback (most recent call last):
  File "./wat2.py", line 11, in <module>
    print pool.apply(thing, (q, 10, ))
  File "/usr/lib/python2.7/multiprocessing/pool.py", line 244, in apply
    return self.apply_async(func, args, kwds).get()
  File "/usr/lib/python2.7/multiprocessing/pool.py", line 558, in get
    raise self._value
RuntimeError: Queue objects should only be shared between processes through
inheritance
```


WAT.

Questions?

Thanks!

Slides and Code:

http://bit.ly/1DqIw5s

https://github.com/tom-dalton-fanduel/python-parallelism-talk

Parallelism vs Concurrency:

http://yosefk.com/blog/parallelism-and-concurrency-need-different-tools.html