Write faster code

An introduction to profiling, optimisation and parallelisation

Motivation

Running software takes time

- How long should a program take to run?
  - Want an immediate answer? A few seconds
  - Coffee Break? A few minutes
  - Lunch Break? An hour
  - Overnight? Several hours
  - Etc.

Running software takes time

- How often does it need to be run?
    - Want an immediate answer? Many times a day
    - Coffee Break? A few times a day
    - Lunch Break? Once a day
    - Overnight? Once a day
    - Etc.

Not necessarily for quicker results

- Higher throughput
- Larger problem sizes
- More accurate results
- Can experiment with problem inputs

So what can we do?

The plan…

When should we optimise?

- At the design stage
- At the **end** of the development stage
- When you need to!

"Premature optimisation is the root of all evil"

- Donald Knuth

What should we optimise?

- Profiling: Identify performance hotspots
- Look for low-hanging fruit
- Consider a range of relevant problems
    - Different problem sizes
    - Different problem types
    - Running on hardware

How should we optimise?

- Inefficient implementations
  - Algorithm/data structure choice
  - Language choice
  - Redundant computation
  - Can come at the cost of readability

How should we optimise?

- Parallelisation
  - Most computers these days have several computing cores
  - Allows scaling to larger machines/clusters
- Using external libraries
  - May already be optimised
  - Sometimes even parallelised

Profiling

What is profiling?

- Identify regions of the code that are taking significant time
- Usually function-level, sometimes line-level
- Can be graphical or command-line
- Profiling tools available for many languages

What can it tell you?

- Number of calls: How many times each function was called
- Cumulative time: Time spent in the routine, *including* any subsequent function calls
- Exclusive time: Time spent in the routine, *excluding* any subsequent function calls
- Call trees: Which other functions were called by a given function, and how long did they take?

What can it tell you?

- Memory usage: Which routines use the most memory
- Parallel efficiency: How well is your parallel code using the hardware its given
- More specialised information:
  - Operations per clock-cycle
  - Memory copies
  - Etc…

Example: Python's cProfile

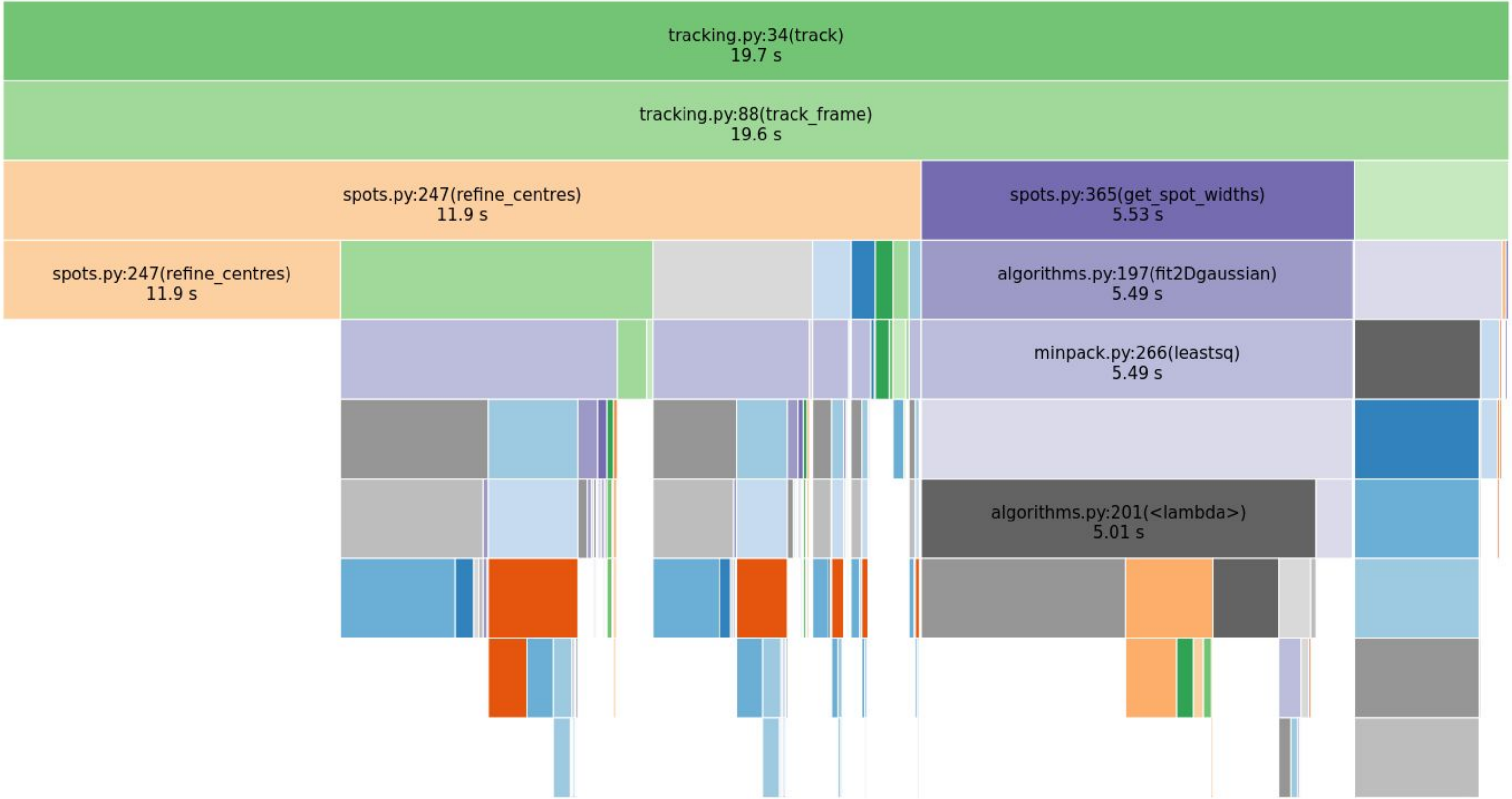
```
Wed Mar  8 11:20:37 2023    pystachio.prof

         16889482 function calls (16717698 primitive calls) in 21.216 seconds

   Ordered by: internal time
   List reduced from 11007 to 5 due to restriction <5>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      100    4.255    0.043   11.856    0.119 spots.py:247(refine_centres)
   159421    2.581    0.000    2.581    0.000 algorithms.py:177(<lambda>)
   883441    1.928    0.000    1.928    0.000 {method 'reduce' of 'numpy.ufunc' objects}
    62939    0.989    0.000    1.636    0.000 _methods.py:196(_var)
   159421    0.859    0.000    5.013    0.000 algorithms.py:201(<lambda>)
```

Example: Python's SnakeViz



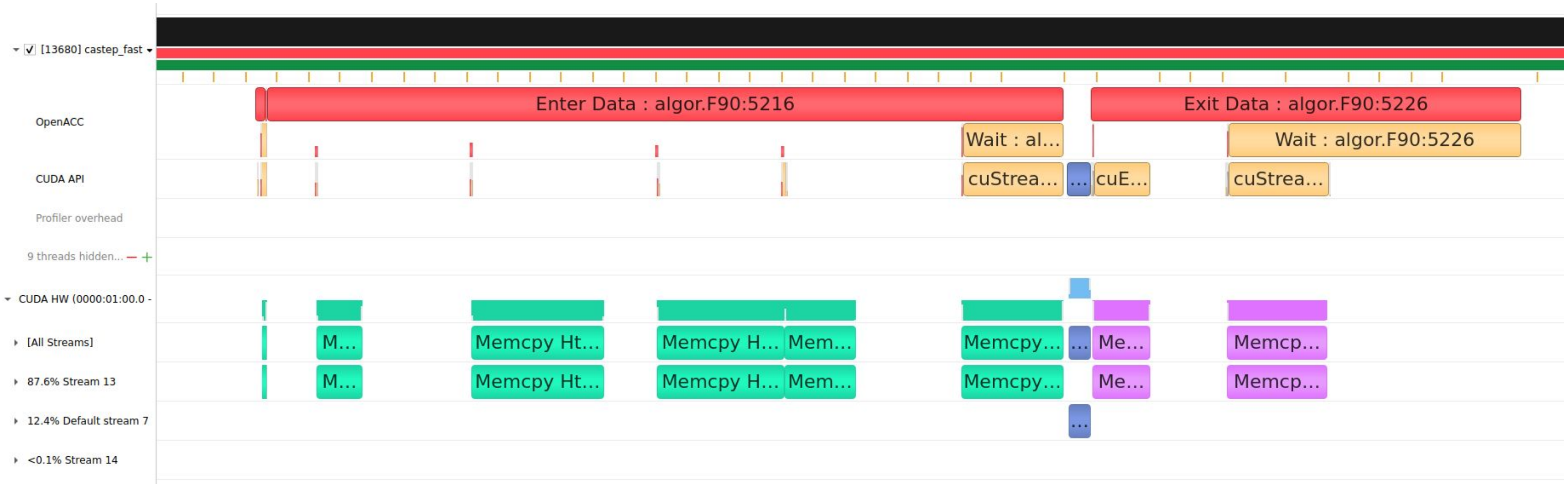
Example: R Studio Profvis

Flame Graph | Data | Options

Code	File	Memory (MB)		Time (ms)
▼ print	<expr>	-32.7	38.3	1930
▼ print.ggplot		-32.7	38.3	1930
▼ grid.draw		-10.1	4.4	1460
▶ grid.draw.gTree		-10.1	4.4	1460
▼ ggplot_gtable		-3.8	9.3	140
▶ element_render		0	0.2	10
▶ facet_render		-0.1	0.6	20
▶ Map		-3.7	8.4	100
▼ ggplot_build		-18.8	22.1	310
▶ by_layer		0	2.6	10
▶ train_ranges		-3.5	0	10
▶ map_position		-7.1	15.1	230
▶ train_position		-8.3	4.4	60
▶ grid.newpage		0	2.4	20
▶ ggplot	<expr>	0.0	4.7	60

Sample Interval: 10ms 1990ms

Example: NVIDIA NSight



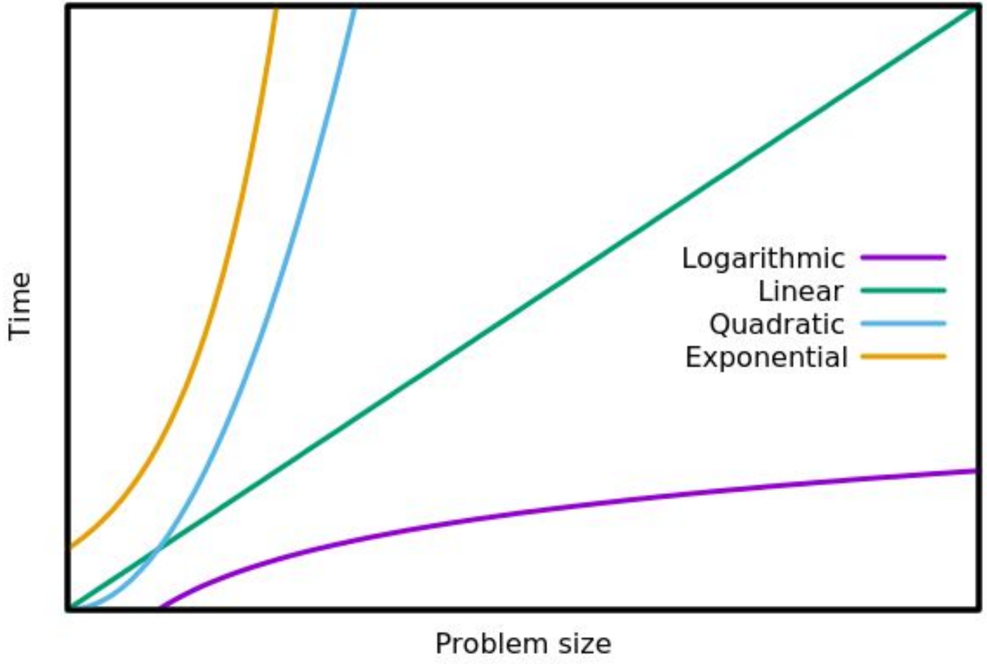
Further reference

- Python:
    - cProfile: https://docs.python.org/3/library/profile.html
    - SnakeViz: https://jiffyclub.github.io/snakeviz/
- R:
    - ProfVis: http://rstudio.github.io/profvis
- C/C++/Fortran:
    - CPU performance: Intel VTune
    - GPU performance: NVIDIA NSights
- Other languages:
    - Have a look around, google is often the best place to start

Optimisation

Choosing the right algorithm

- Algorithm time:
    - O(f(N)) => prefactor * f(N)
- Logarithmic
  < Polynomial
  < Exponential
- Need to consider problem size



Choosing the right algorithm

- Classic example: Sorting N elements
    - Bubble sort is rarely the right choice. $O(N^2)$
    - Quick sort is good for large lists. $O(N \log(N))$
    - Insertion sort better for small lists. $O(N^2)$

Choosing the right data structures

- Does the data have some underlying structure?
- What kinds of operations are being performed?

Data structure	Insertion	Access	Searching
Array	O(N)	O(1)	O(N)
Linked List	O(1)	O(N)	O(N)
Dictionary	O(1)	n/a	O(1)
Binary tree	O(log (N))	n/a	O(log (N))

Making code more efficient

- Use optimised libraries
  - Someone else has already put the time in
  - Often written in low-level languages
  - E.g.
    - Tensorflow/PyTorch for AI
    - NumPy/Pandas for Python
    - Lapack for C/Fortran
    - Using your language’s standard library

Making code more efficient

- Compile your code
  - Try Just-In-Time (JIT) compilation tools such as Numba (Python)
  - Rewrite computational kernels in a compiled language
    - Most languages have a C interface, although this can be difficult to use
    - Some languages provide tools for writing packages in Fortran, such as Python’s F2Py
    - Use sparingly, it can make for very complicated codebases

Further reference

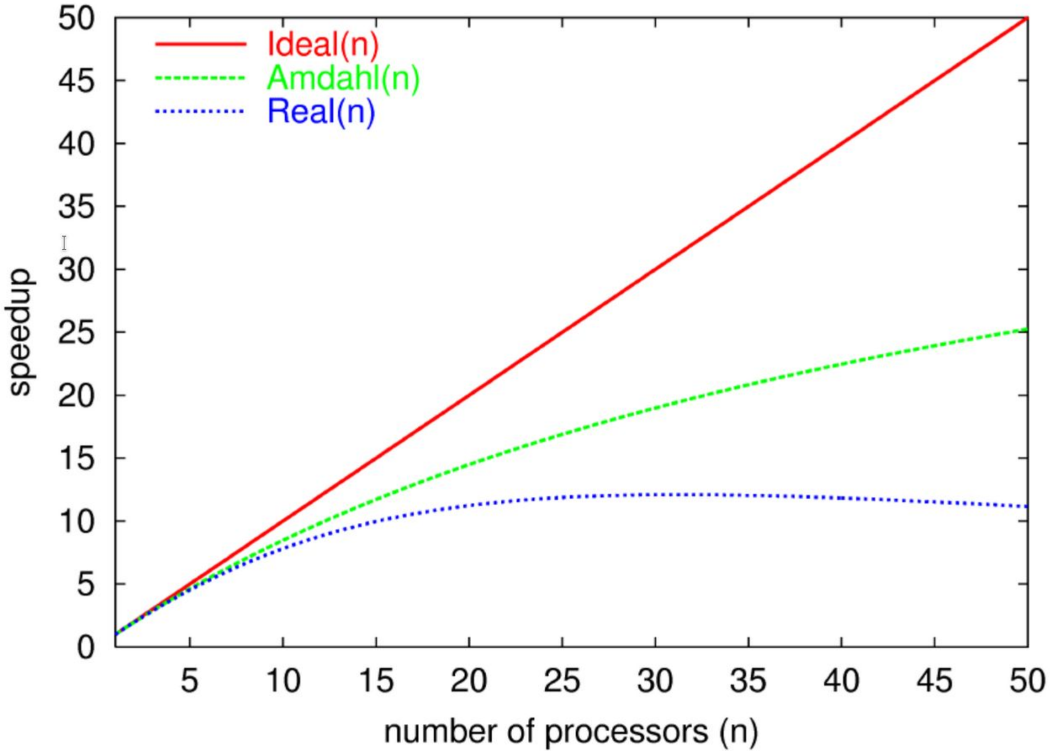
- Python:
  - Numba: https://numba.pydata.org/
  - F2Py: https://numpy.org/doc/stable/f2py/
- R:
  - https://masuday.github.io/fortran_tutorial/r.html
- C/Fortran:
  - Prof. Matt Probert's Intro. To HPC lecture course:https://www-users.york.ac.uk/~mijp1/teaching/4th_year_HPC
- Other languages:
  - Have a look around, google is often the best place to start

Parallelisation

Considerations for parallel programs

Amdahl's law:

- All code has a serial portion (S) and a parallelisable portion (P)
- Time = S + P / #cores
- Ignores network latency/bandwidth



High-level parallelism

- Run your program multiple times with different inputs
- Easiest form of parallelism!
- Considerations
  - Make sure your program runs are independent
  - Make sure your entire problem can be specified at run-time
  - Will all problems take the same/a similar amount of time?

Using parallel libraries

- Some libraries will already run in parallel
- E.g.
  - Tensorflow/PyTorch for AI/Machine Learning
  - Some LAPACK implementations provide threading/distributed parallelism
  - R's "parallel" package provides parallel versions of standard functions

Parallelising loops

- Research software often involves looping over large amounts of data
- Considerations:
  - Easiest when loop iterations are independent
  - Does each loop iteration take the same amount of time?
  - Fewer iterations → less scope for parallelism
  - Quicker iterations → parallel overheads may dominate the run time
- Examples:
  - C/C++/Fortran: OpenMP
  - Python: Multiprocessing module

Parallelising loops

```
def refine_centres(self, frame, params):
    image = frame.as_image()
    # Refine the centre of each spot independently
    for i_spot in range(self.num_spots):
        r = params.subarray_halfwidth
        N = 2 * r + 1

        ...
```

Other types of parallelism

- Task-level parallelism
  - Can many tasks be done simultaneously?
- Domain parallelism:
  - Splitting up your problem across multiple processes
- Hardware acceleration:
  - GPUs
  - FPGAs
  - Etc.

Summary

Summary

- Optimising & Parallelising your programs can allow you to get work done faster and tackle larger problem sizes
- Parallelising your program can allow you to use larger computers/clusters
- Profiling your program will tell you where to focus your time when optimising
- Choosing the right algorithms can make huge differences to runtime
- Using existing libraries/compiled languages can often provide better performance than interpreted ones

“