Benefits of efficient programming

Lincoln Colling

lincoln@colling.net.nz

research.colling.net.nz

Software Sustainability Institute

The problem

- Powering compute infrastructure is burning through our carbon budget
- Mining for resources to build compute infrastructure is destroying ecosystems

A solution?

The simple solution is to **use less**

- Use less carbon intensive energy for running our code
- Use fewer computing resources for running our code
- Run less code

The solution is simple, but putting it into action isn't

We could send people to the moon using the computing power of a pocket calculator and 4kb of ram

Now we need millions of transistors and gigabtyes of ram just to run a web browser

Back then, **compute was expensive** and **engineers were cheap**, so you could get a load of smart people to squeeze the most out of a little

Now, **compute is "cheap"** and **engineers are expensive**, so it's cheaper so throw (*sometimes bad*) code at a load of compute

The idea that **compute is cheap** is illusory:

- The monetary cost might be low, but the environmental cost is high

Maybe, we can go back to the way things were...

- Make **compute expensive** by recognizing the true cost, so we can try to minimize it
- Make **engineers "cheaper"**, not by paying them less, but by getting them to produce **higher quality** code

Recognizing the true cost of compute

CATS: The Climate-Aware Task Scheduler

CATS is a Python package for scheduling compute jobs so that you can take advantage of low carbon electricity

- The UK National Grid provides real-time estimates (and predictions) of the Carbon Intensity of electricity
- Using this information, it should be possible to estimate the relative carbon cost of **a compute jobs** that have scheduled to start at different times

For example, it might be the case that we could schedule a 4-hour compute job **now** when carbon intensity in high, or we could wait **6 hours** when carbon intensity is lower

*Solving this **start time optimization problem** is the job of CATS...*

How does CATS work?

- Given a compute job duration (in minutes), CATS finds a block of time (of that length) where the average Carbon Intensity is the lowest
- It uses the predicted Carbon Intensity of the next 48 hrs (as provided by the National Grid API)
- It outputs the time-stamp for the optimal start time
- The output can then be piped to a standard Linux scheduler like `at`

How does CATS work?

```
> python -m cats -d 5 --loc BN11

Using carbonintensity.org.uk for carbon intensity forecasts
Using location provided: BN11
Best job start time: 2023-11-05 23:19:31.653139+00:00
202311052319
```

- You can also give CATS information about your CPU and GPU
- And job-specific information about CPU, GPU, and memory usage and cats will give you an estimate of your carbon savings...

How does CATS work?

- By using CATS, you're able to *see* the carbon cost of your compute jobs, so you're not fooled into thinking that *compute is cheap*
- And you're able to **reduce** these costs by picking a better start time

To find out more about CATS, check out the Github page: github.com/GreenScheduler/cats

Or come talk to me or Loïc during the break

Getting more out of code

Code and energy efficiency

Python, the most popular language for scientific computing, does not do well when it comes to energy efficiency

Table 1: Pereira et al (2021) Ranking programming languages by energy efficiency, *Science of Computer Programming*

Language	Energy	Time	Memory
C	1.00	1.00	1.17
Rust	1.03	1.04	1.54
C++	1.34	1.56	1.34
Julia[1]	1.80	3.39	3.71
Fortran	2.52	4.20	1.24
Python	79.58	71.90	2.80

[1] Note that Julia was added after publication

Does this mean we should abandon Python?

Not necessarily...

1. We can pay closer attention to run times and memory usage (time and space complexity) and get more out of Python[2]

But we shouldn't be afraid to pick up other languages either

2. We can partially re-write compute intensive parts of our Python code in more energy-efficient languages like Rust

[2]This is something we should care about anyway, but because "compute" is cheap, and engineering time is expensive, we sometimes ignore it.

Some bad code is wild

"Science is amateur software engineering"

Most scientific code is written by amateurs

- In many ways this is **great!**
- But it also has it downsides

The downside is that you often see lots of bad code in software packages written by scientists for scientists

- And a lot of this bad code could be spotted by trained software engineers

```
data = [2.2, 2.23, 4.0, 5.0]
means = []
sds = []
for i in range(len(data)):
    d = data[0:i+1]
    m = mean(d)
    s = stdev(d) if len(d) > 1 else None
    sds.append(s)
    means.append(m)

# mean: [2.2, 2.215, 2.81, 3.3575]
# sd: [None, 0.021, 1.030, 1.381]
```

This code might look reasonable, but if we dive deeper we see something terrible...

Hidden loops...

- To work out a *mean*, we need a *sum*. And to work out a *sum*, we need a loop
- To work out an *std dev*, we need a mean
  - To get that mean, we need a sum, so we need another loop
  - And we then need to loop over the numbers again, to get the square differences
  - And then we need to sum those, so we need another loop

```
def sum(d):
    s = 0
    for n in d:
      s += n
    return s
```

```
mean = sum(d) / length(d)
sum_sq = sum([(x - m)**2 for x in d])
```

```
from itertools import accumulate
def welfords(existing_aggregate, new_value):
    (count, mean, M2, sd) = existing_aggregate
    count += 1
    delta = new_value - mean
    mean += delta / count
    delta2 = new_value - mean
    M2 += delta * delta2
    sd = (M2 / (count - 1))**0.5 if count > 1 else None
    return (count, mean, M2, sd)

output = list(accumulate(data, welfords, initial=(0,0,0,0)))
```

Calculating the running mean and std dev is a common problem, and a solution can be found in many classic algorithms textbooks[3]

[3] For example, Knuth's *The Art of Computer Programming*, Vol 2

What changed?

1. In the first version, we had a loop with in a loop
   - The outer loop selected 1, then 2, then 3, then 4 bits of data
   - The inner loop (the list comprehension) looped over these values multiple times to calculate sum, means, std dev etc (hidden loops)
2. In the second version, we replaced the loops with an `accumulate`
   - An `accumulate` takes a list of values (our data), a binary function (a function that takes two arguments), and an initial value
   - The function is applied to each element of the list only once

How much of a difference does it make?

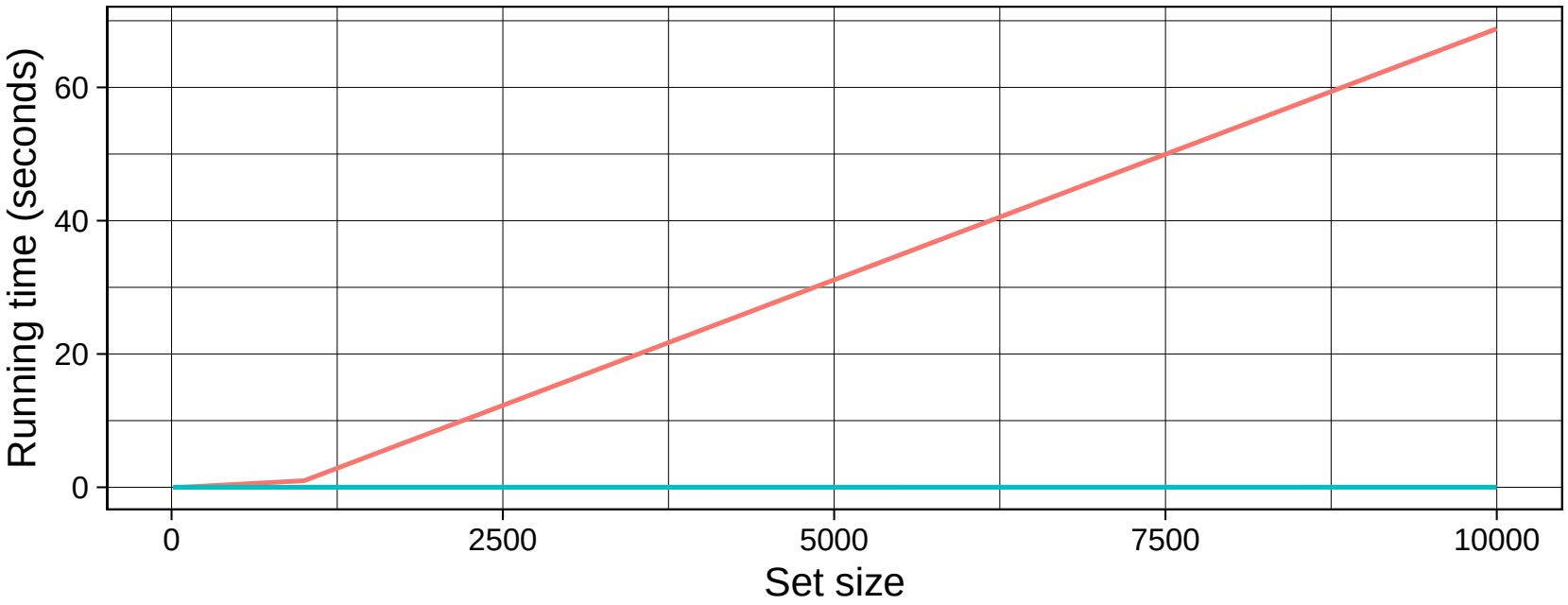

Figure 1: Comparison of the two algorithms

How much of a difference does it make?

- This might not seem like it matters that much because 60 seconds is still pretty quick
- But this code is meant to run on datasets of over 1,000,000 data points
- In a typical use-case, it's a difference between 6 minutes and $<$ 1.5 minutes
- And this is code in a library, so it could be run thousands and thousands of times, which leads to a huge difference!

But can we do even better?

Yes!

- When we re-write the core in a more energy efficient language Rust, we can get the typical use-case compute time down from ~1.5 minutes to ~1.5 seconds!

1. By writing better code with fewer loops, we could go from
   - 6 minutes to ~1.5 minutes
2. By switching out of python for the bits that really mattered, we could go from
   - ~1.5 minutes to ~1.5 seconds

Starting at ~6 minutes and ending at ~1.5 seconds, multiplied by thousands of runs works out to a HUGE saving in energy, and carbon

My recommendations

- We need to "*make compute expensive*" by recognising the true cost
  - We can mitigate some of this cost, by using tools like CATS
- And we need to "*make engineers cheaper*" by having them produce a higher quality product
  - We can do this by making sure that we have better training for those writing code in our labs (through organisations like the Software Sustainability Institute)
  - We can have departments fund research software engineers (possibly shared between labs) to ensure that high quality efficient code is produced

And we can use venues like this to alert researchers, funders, and institutions, to the scale of the problems