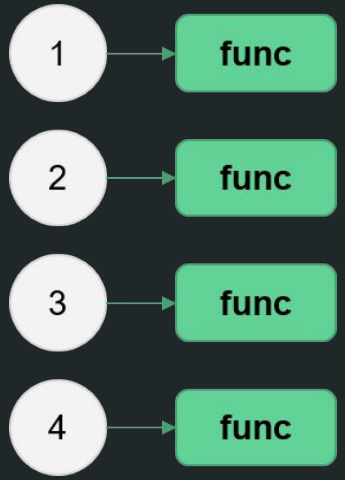
Parallel Python



Jakob Jordan / Jenni Rinker / Zbigniew Jędrzejewski-Szmek

ASPP LatAm 2023, CDMX

Fork/clone the repo now!

Outline

- Multithreading, the GIL, and multiprocessing
- Embarrassingly parallel problems with the built-in `multiprocessing` module
- Investigation of NumPy threading on execution time
- Going further
- Wrapping up

Exercise: brainstorm

Why do we parallelize?

Talk to your partner and come up with three practical examples of where parallelization could be beneficial (in your work or another application).

Exercise: brainstorm

Why do we parallelize?

Talk to your partner and come up with three practical examples of where parallelization could be beneficial (in your work or another application).

In short, two reasons why:

- Speed up computations.
- Process “big” things.

As for the “how”...we’ll come back to that later.

Multithreading, the GIL, and multiprocessing

Parallelization as...

Parallelization as a kitchen

Chef Elena Reygadas
Taken from here

Parallelization as a kitchen

We need to make a single dish: **TACOS**. This requires*:

- Cooking the tortilla
- Warming the filling
- Making fresh salsa

The three things are returned to Chef Elena for the final plating.

I know this is a terrible oversimplification of making tacos...please forgive me.

Chef Elena Reygadas
Taken from here

The kitchen consists of:

- Head chef Elena
- Sous chefs
- Fixed number of workstations
- Dish logbook

Tortilla	Done
Filling	*In progress*
Salsa	*In progress*

But how is a kitchen related to a computer???

Key concepts:

- **Process**
  - An instance of a program being executed, for us the Python interpreter interpreting your script. Can spin up threads.
- **Thread**
  - A unit of computation sent to CPU for execution.
- **Number of cores**
  - The number of independent units in your CPU that execute threads.
- **Memory**
  - Where information/variables are kept/updated during execution.

Elements of our kitchen metaphor:

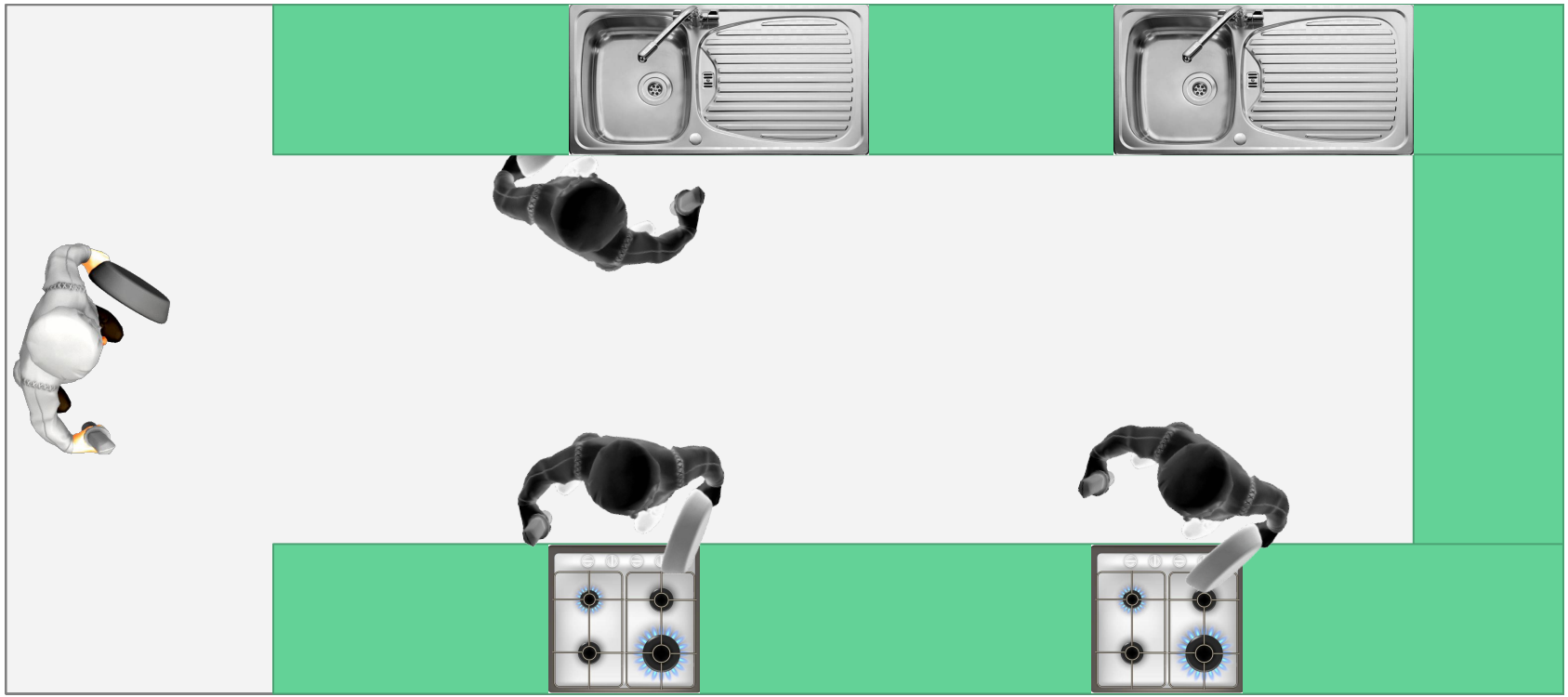
- Sous chefs, each with their own taco task
- Fixed number of workstations
- Dish logbook
- Head chef Elena

So which element is which?

- Process:
- Thread:
- Number of cores:
- Memory:

Most efficient way for a single taco

Ideal situation: All 3 sous chefs are working on their tasks simultaneously



But there is a problem...

...and that problem is tomatoes.

Consider a scenario:

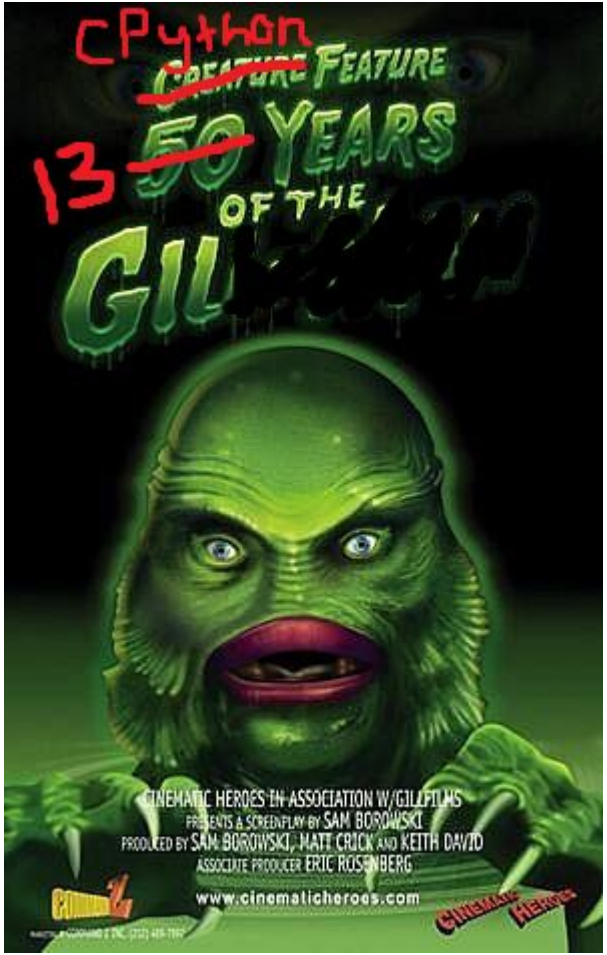
- We're trying to make 2 tacos at once.
- We have plenty of workstations (cores) and sous chefs (threads).
- The "make salsa" task includes a sub-task: count all the tomatoes in the pantry, take X tomatoes for the salsa, and tell Chef Elena how many tomatoes are left in the pantry. (Past trauma when she ran out of vegetables on opening night.)
- What happens if two sous chefs making salsa execute their tasks at the same time?
    - The number of tomatoes reported to Chef Elena will be off by 1!

This sort of data corruption is called a **race condition**

Race conditions are problematic in any multithreaded code. In Python, race conditions can lead to memory leaks (reference counting [3]) and data corruption.

To avoid race conditions, *Python implemented something special called...

**To be specific, only CPython – Python built on the C language. Other types of Python may not have this, but you are almost certainly using CPython.*



The “Global Interpreter Lock” (GIL)

- A “mutex” (mutual exclusion) lock. Forces each thread in **pure-Python code** to acquire the lock before execution.
- In other words, only one thread is allowed to run at a time!*

**For 5 ms, then Python forces the thread to release. [1]*

Hypothesize with your partner:

NumPy can (and by default does) run code with multiple threads in parallel. How is this possible?

NumPy's trick

NumPy interfaces with non-Python libraries that, by default, use as many threads as you have cores.

In other words, it is *many* sous chefs disguised as one!

What does this all mean?

Computationally heavy, pure-Python code will generally have **0 speed-up** with multiple threads.

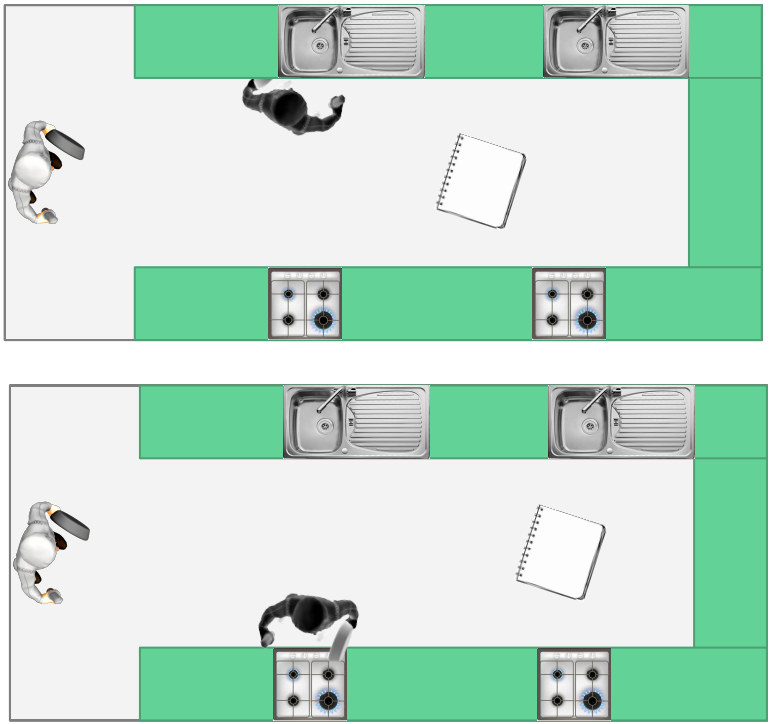
Some specific packages (NumPy) get around this by spinning up multiple threads without the Python interpreter knowing.

Note that network- and IO-bound problems can be handled with multithreading.

So how can we get around this?

- Instead of multiple threads, use multiple *processes*.

Instead of multiple sous chefs, multiple kitchens



Each process is an instance of the Python interpreter and therefore has its own GIL!

BUT processes have separate memory, so data must be duplicated in each process.

Multiple processes therefore have additional computational overhead and memory usage.

To wrap things up...a pop quiz!

On your pair computer, please navigate to **kahoot.it** and enter game pin

XXX XXXX

Next up: let’s learn how to apply it

Embarrassingly parallel problems with the built-in `multiprocessing` module

Python's `multiprocessing` module

This builtin module allows you to easily parallelize so-called "embarrassingly-parallel" problems.

Jakob will explain some basics before the first exercise. Live coding! 😬

For now, sit back and watch. And please don't hesitate to ask questions!

Exercise 0 (5 min)

Open* `multiprocessing_exercise_0.ipynb` and follow the instructions.

*`jupyter notebook multiprocessing_exercise_0.ipynb`

Exercise 1 (15 min)

Open* `multiprocessing_exercise_1.ipynb` and follow the instructions.

*`jupyter notebook multiprocessing_exercise_1.ipynb`

You can always do this lecture again!

We'll upload Jakob's notebook, but there is already another filled one on the repo with some additional comments.

```
jupyter notebook multiprocessing_filled.ipynb
```

Investigation of NumPy threading on execution time

Python threads vs. threads in general

In Python the GIL prevents threads from providing a benefit. In general, however, this is not the case.

Compiled code (remember yesterday?) can avoid the GIL and can use threads to speed up computations*.

NumPy, by default, uses as many threads as your computer has CPUs.

* (that don’t interact with the Python interpreter state, e.g. the non-yellow stuff yesterday)

Objectives of this section

Learn how you can control the number of threads that NumPy uses.

Measure the effect of the number of threads on execution time.
Guess the architecture of the laptop processor based on the results.

Start a multiprocessing job, where each of the processes uses multiple NumPy threads.
Explore the effect of processes and threads on execution time.

How to control the number of threads used by NumPy

Interactive demo by Zbyszek.

How to control the number of threads used by NumPy

On the command line on Linux:

```
OMP_NUM_THREADS=4 python script.py
```

On the command line on Windows:

```
set OMP_NUM_THREADS=4
python script.py
```

Note that you can also set this variable within Python. We'll show this later.

Time for an exercise (10–15 min)

We'll repeat same calculation, varying the number of threads. Based on the results, we can hypothesize about the architecture (i.e., the number of cores) of the processor.

Python script `just_threads/timing.py` does a calculation using NumPy.
Run it with different numbers of threads, using the `OMP_NUM_THREADS` variable, and write down the reported execution time.
Use the file `just_threads/timing_plot.py` to plot a graph of the speeds.
Take a screenshot of your plot and send it in the Telegram group.

Detailed instructions in `processes_and_threads/README.md`.

What did you find?

(show Telegram)

Based on this, how many cores do you think your laptop has?

What can we conclude from the plots

The speed-up is less than we might think.

The maximum speed-up occurs for number of threads == number of cores.

So it doesn't make sense to use more threads than the number of cores.

Note that the number of CPUs shown by `lscpu` is twice as the number of physical cores – this is due to something called hyperthreading (not discussed in detail). So if you have 4 cores, NumPy spins up 8 threads.

Setting the number of NumPy threads within Python

We can set `OMP_NUM_THREADS` from Python, but this must be done **before NumPy is imported for the first time**.

For example:

```
import os
os.environ['OMP_NUM_THREADS'] = '1'
import numpy as np
```

Now what happens with multiple processes

We have 12 input images.

Script `processes_and_threads/process_image.py` analyzes a single image.
Script `processes_many_images.py` calls `process_image.py` over all images, with a configurable number of processes and threads.

Figure out the fastest combination of processes and threads.

Detailed instructions in `processes_and_threads/README.md`.

What did you find?

(insert findings here)

Going further

Larger-than-memory problems

Consider a problem: calculating the mean value of each column in a numpy array, where the array is so big that you cannot hold it in memory.

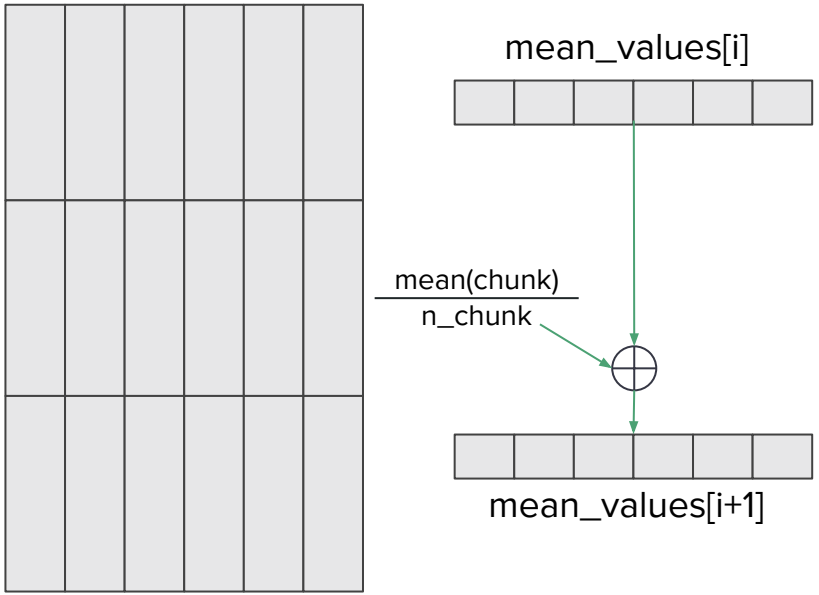
Discuss for a minute with your partner: what code could you write to get around this problem?

A pseudocode solution

Break the array into n_chunks vertical chunks.

```
initialize mean_values to array of zeros

for each chunk in the array:
    load the chunk into memory
    calculate the mean values of all columns
    divide the chunk mean values by n_chunks
    add the result to mean_values
```



That’s nice, but what about more complicated problems?

For example, an eigenvalue decomposition or Singular Value Decomposition (SVD)?

Any ideas?

One library that is potentially useful is dask. I will present a demo of dask as a very brief introduction. You are not expected to understand everything, it is just to make you aware of the package. Please also note that the example probably doesn’t work on the student laptops.

Final notes on dask

In theory, the same dask code can be executed on your laptop or on a cluster by just instantiating a different Client.

Dask has a bit of a learning curve. If you need to use it, be prepared to invest time to learn its quirks.

It may not be the best tool for your problem. Or it may be the only tool for your problem.

Other Python packages for parallel computing

- https://github.com/mpi4py/mpi4py/ (de facto standard for low-level massive parallelization)
- https://ipyparallel.readthedocs.io/en/latest/ (parallel IPython)
- https://github.com/ray-project/ray (parallelization for ML-style workflows)
- https://github.com/modin-project/modin (parallel pandas)
- https://www.bodo.ai/ (SQL & data processing)
- https://spark.apache.org/docs/latest/api/python/index.html (big data analytics)

Wrapping up

Let’s wrap it up. What have we learned?

- <insert cool things here>

Thanks for your patience and attention.

Please take time today and record your notes/impressions of the lecture.

When the evaluation is sent out, use the notes to jog your memory.

And please be honest with your feedback – we'd like to know how to improve.

Further reading

1. When Python releases the GIL When Python can't thread: a deep-dive into the GIL's impact (pythonspeed.com)
2. NumPy and the GIL NumPy vs the Global Interpreter Lock (GIL) (superfastpython.com)
3. How the GIL prevents memory leaks in Python programs What Is the Python Global Interpreter Lock (GIL)? – Real Python
4. Advanced Python topics: Threads in Python
5. Sharing data in threads and processes Threads are 4x Faster at Sharing Data Than Processes in Python (superfastpython.com)
6.