Parallelization of code in Python for beginners

PyData Global 2022

Cheryl Roberts
Allstate Insurance Company

Parallelization helps computation on data run faster

Many Use Cases, Including

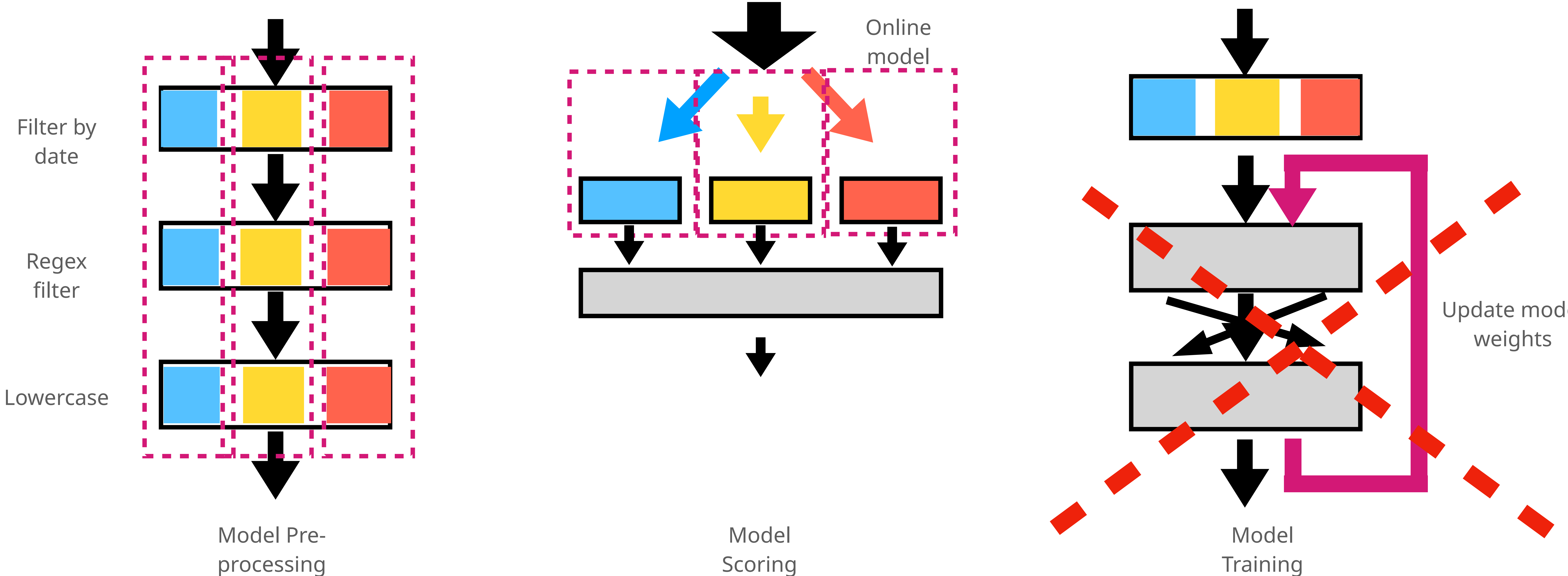
- Large dataset (pre-)processing
- Expensive function calls: Training multiple models at once (ie, hyper-parameter tuning)
- Batch scoring of online machine learning models
- …But a complement, not substitute for ‘vectorizing’ your code! In general try ‘built-in’ functions first if you’re already using a package- these often leverage fast matrix algebra packages and/or built-in parallelization (ie, numpy)

https://upload.wikimedia.org/wikipedia/commons/d/d3/IBM_Blue_Gene_P_supercomputer.jpg

We can use parallelization when we don't have dependencies across data or calculations

If task A and task B don't depend on each other, we can calculate A and B simultaneously

Consider the following graphs of computation:

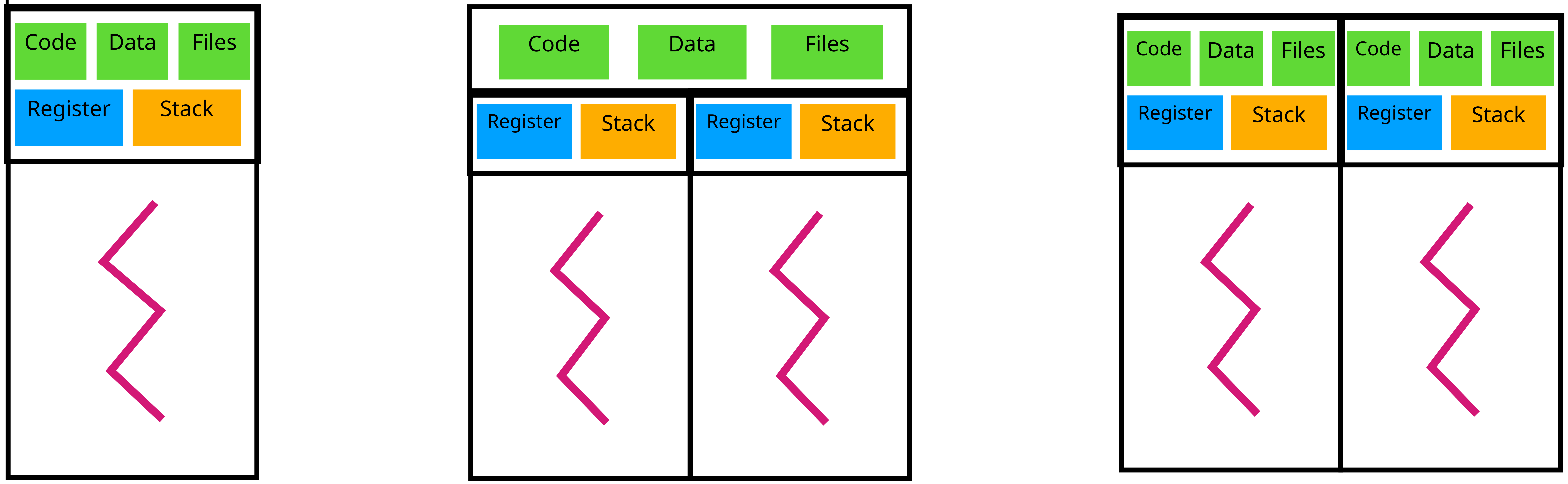


Multithreading versus multiprocessing

Two paradigms that suit different use cases, compute environments

Multiprocessing indicates a paradigm for using multiple processors (ie CPU cores) to execute computation

Multithreading indicates that multiple CPU threads on a single core work together to execute computation



Single Processor/Thread

Multithreading

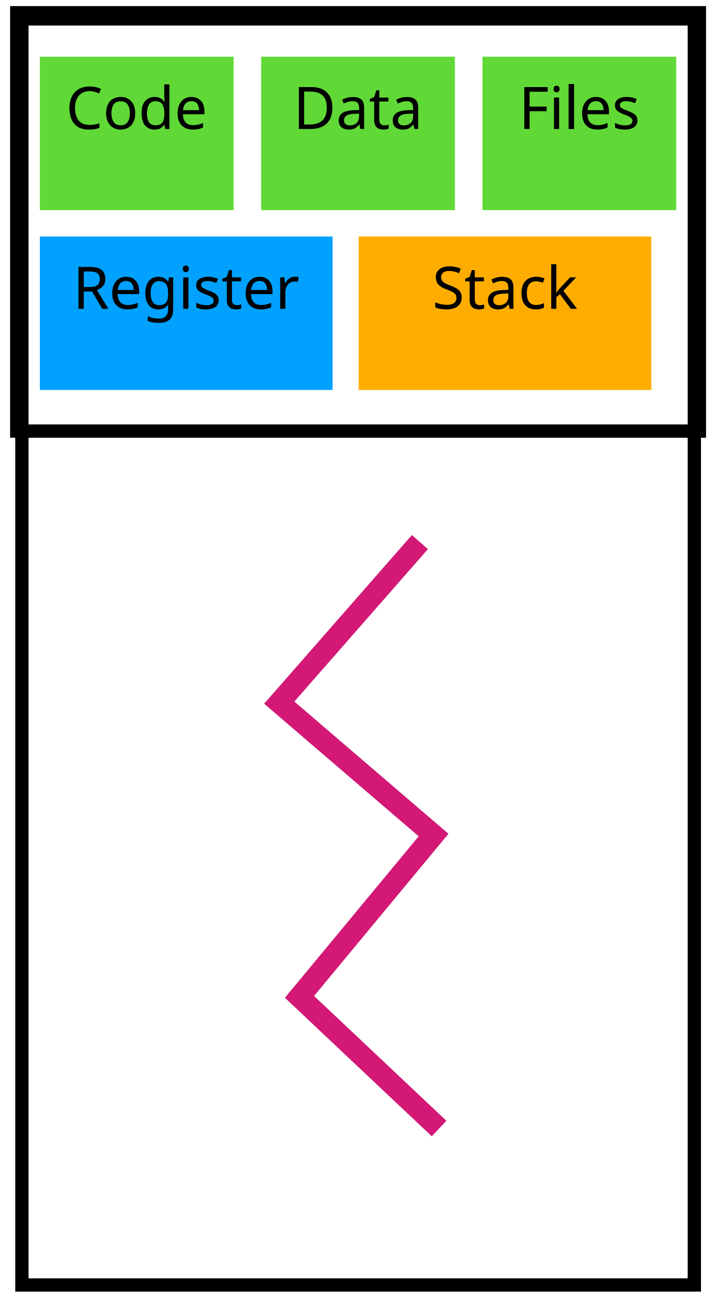
Multiprocessing

Multithreading versus multiprocessing

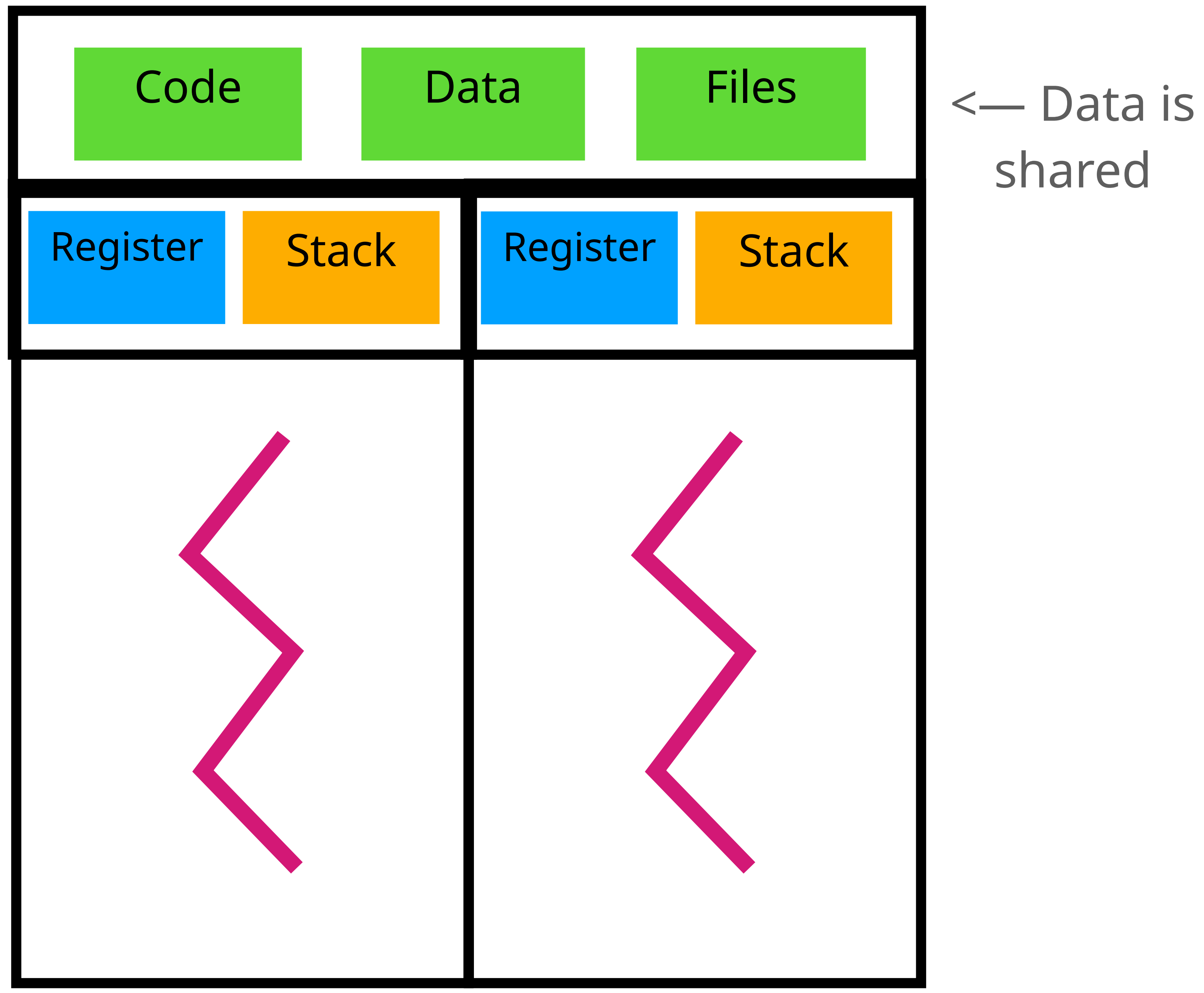
Two paradigms that suit different use cases, compute environments

Multiprocessing walls off computation (and often data) on separate CPU cores. This is safe but potentially slower/more memory intensive

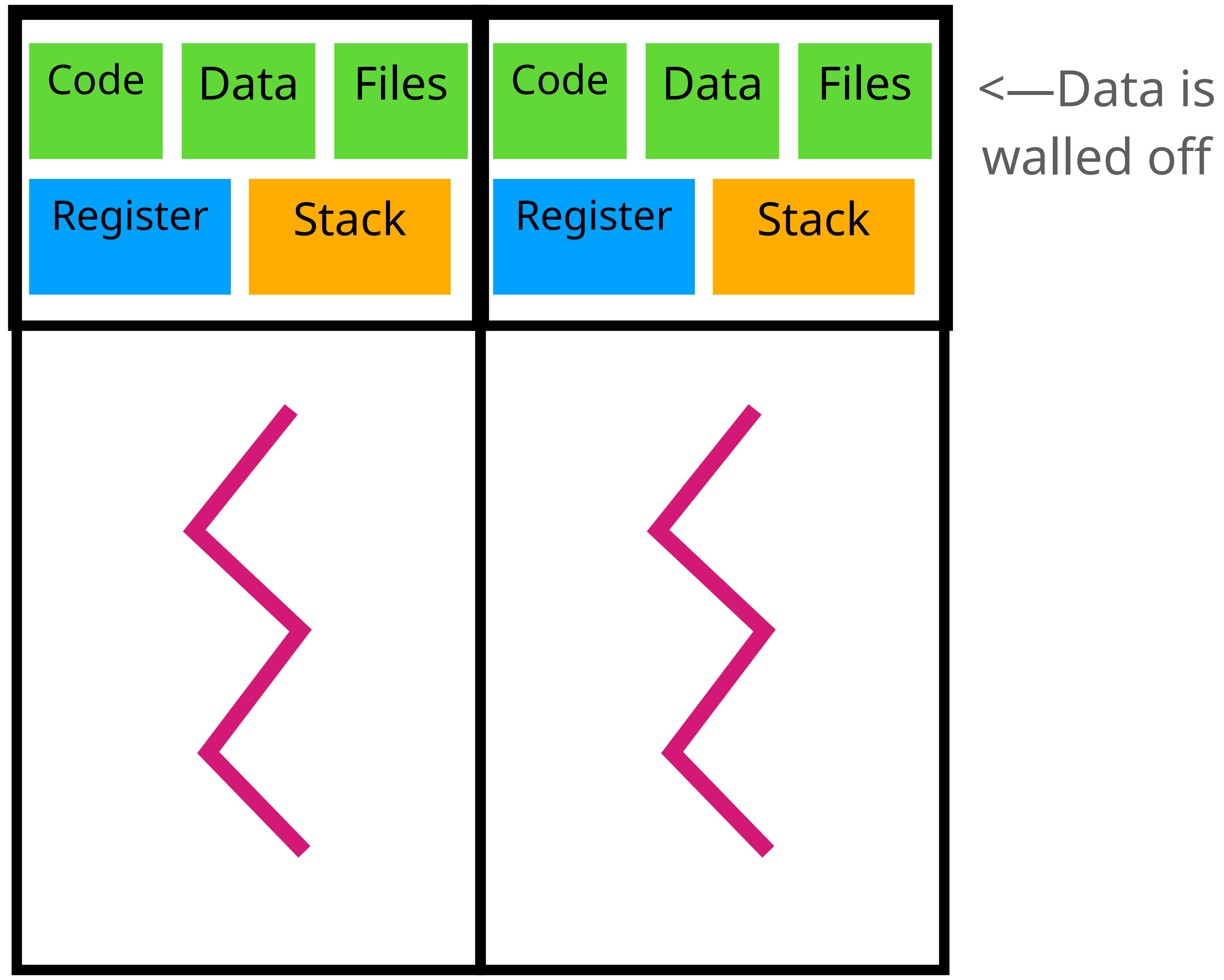
Multithreading reduces overhead by sharing some of the data needed for computation across threads. However, **multithreading** may not be appropriate for every job as naive application can lead to resource contention or even crashes



Single Processor/Thread



Multithreading



Multiprocessing

To use multiprocessing/multithreading, we need to know a bit about our computer
The computer

Cores/CPUs: for each CPU core, we can allocate a process in multiprocessing. Desktop computers today typically have 4+ cores (**4x speedup?**).

- Processes are computing tasks, like running a python function or a Jupyter notebook cell
- Threads in contrast are fundamental computing units of a process; multithreading is typically used to make a single process/task run faster or more efficiently
- We can have parallel computing by running processes in parallel on separate CPU cores

Memory: Consider that if doing many tasks in parallel, need memory for **"scratch space"** for intermediate data structures produced by the computation as well as to hold **"parallelized" input data**

On Linux, running **top** command yields information on computer specs and running threads/processes

- You may want to check this after launching a joblib job, to make sure things are running as expected

In general, choose ‘multiprocessing’ and not ‘multithreading’ when using joblib or other multiprocessing/threading packages that wrap around black-box code

Multiprocessing suits more use cases, given today’s computers

- Computers are likely to have multiple cores that can be easily multi-processed; shared data for a typical job is usually small
  - Caveat: multiprocessing may duplicate data/objects shared across parallelized processes. This uses up **more memory** than a single process (in some cases, multithreading can help with this)
- Multithreading carries risks if not built into the specific package code
  - During computation, will multiple tasks **read or write from a shared dataset/object**?
    - Multithreading may carry a **higher risk of contention** here; each “worker” may have to queue to access the shared object. Object could be locked for the entire duration of the task, negating the benefit of multithreading!
    - It may be difficult to determine if your code does this, as workings of external packages may be opaque

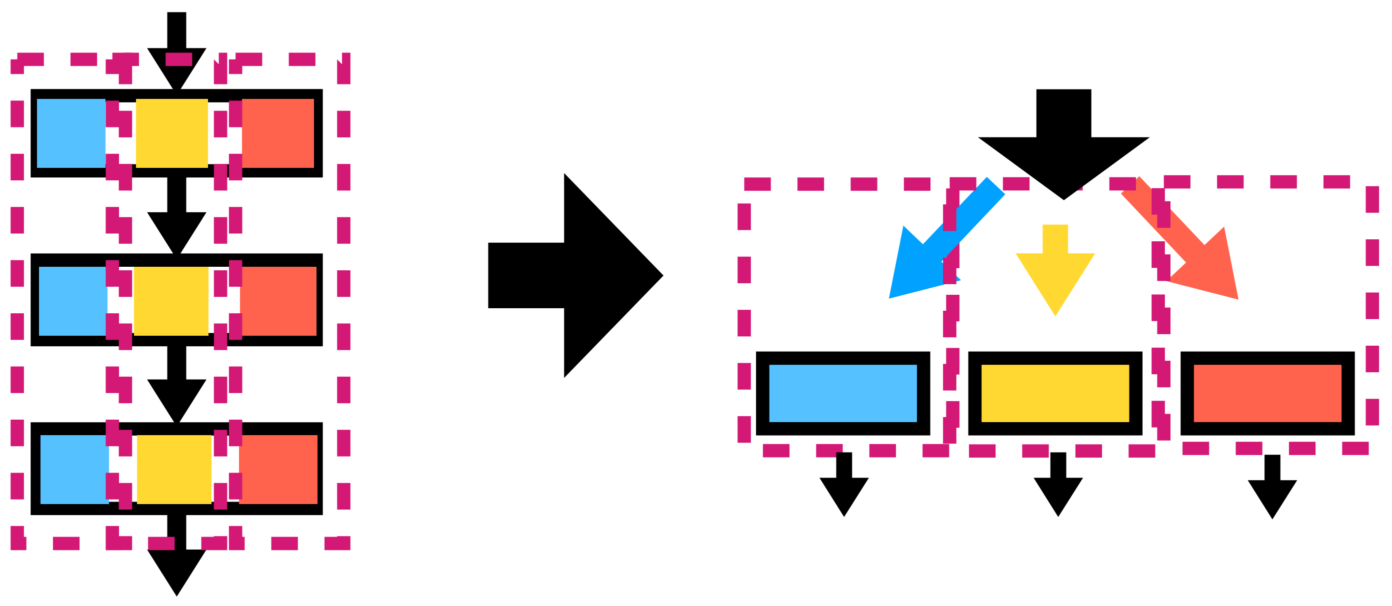
Image by Drazen Zigic on Freepik

“Embarrassingly simple” for loop

Joblib example

How one breaks up a single big computational task into many smaller tasks determines the benefits of parallelization

How do we design a Joblib job?



Example on Amazon Fine Foods dataset: Data science workflow

Git repo at https://github.com/kopant/pydata22-joblib

Gotchas: the need to “lock down” data in memory (GIL/mutex)

Writing overlapping segments of data in memory is to be avoided

Risk: your results may be corrupted

Ideally (and also to avoid contention issues), the code you parallelize should have **walled off write dependencies:** writes are to a segment in memory owned solely by that worker

Generally* taken care of by **joblib**, as long as you **don’t use “in-place” operations** in parallelized code

Exception*: shared large Numpy data caveat

Potential solution: write to separate Numpy arrays per process, and aggregate results as a separate step

Gotchas: multiprocessing of multithreading

May not work

Risk: your job may crash

Some packages/libraries themselves already integrate multiprocessing/multithreading. Can we add another layer of parallelization on top of this? **The answer is maybe…**

- Numpy BLAS, Tensorflow…
- Joblib recommends using the **Loky backend** to handle this (see “Avoiding over-subscription of CPU resources” section of documentation, as well as the section on the older multiprocessing backend of Joblib)

Also to treated with caution: multiprocessing calls to servers/external services, which themselves have separately owned mechanisms for handling multiple requests (ie, all your simultaneous calls may go to a queue)

Tips

Practical hints

Start out small and see how runtime scales (also check that code works as expected in the first place on a small sample, before launching a long job!)

If your job crashes, it is likely that you have run out of memory in some form; try increasing the number of tasks within reason (and consequently feeding smaller chunks of data to a single worker at a time)

If this fails, perhaps also try processing data in batch (multiple sequential calls to Joblib)

Be aware/wary of data skew: a single more complex record can cause resource (and memory) spikes when encountered by the CPU; when running in a parallelized situation the effects can be compounded. This can act as a bottleneck and slow down performance, or even cause a crash. Perhaps filter out such records and handle them separately

Thank you!

PyData Global 2022

Cheryl Roberts, LinkedIn

Git repo contains the joblib demo