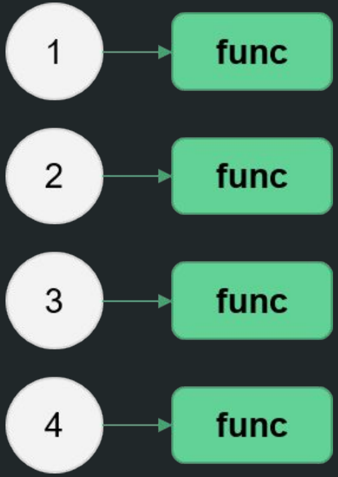


Parallel Python

Jakob Jordan and Jenni Rinker

ASPP 2021, Bordeaux

Clone the repo now!

Outline.

- A bit of theory.
- Embarrasingly parallel and built-in parallelization.
- Case study: Parallelizing a hyperparameter search.
- Case study: Using mpi4py.

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.

Threads, Processes and the GIL.

First step: understand default behavior.

Before we can parallelize, need to know what Python* is doing normally.

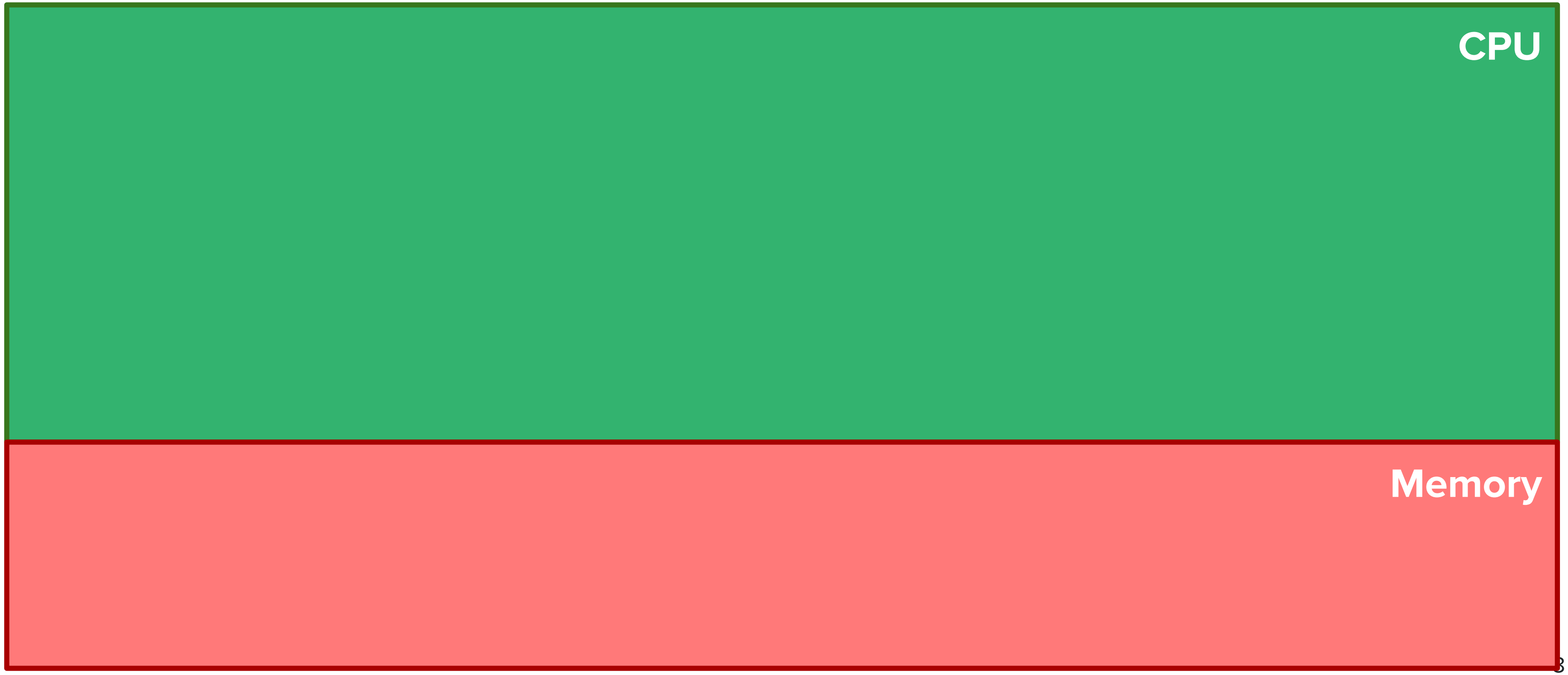
For example, if we enter the following command in a terminal *with no special functions*, what is our computer actually doing?

```
    > python compute_stuff.py
```

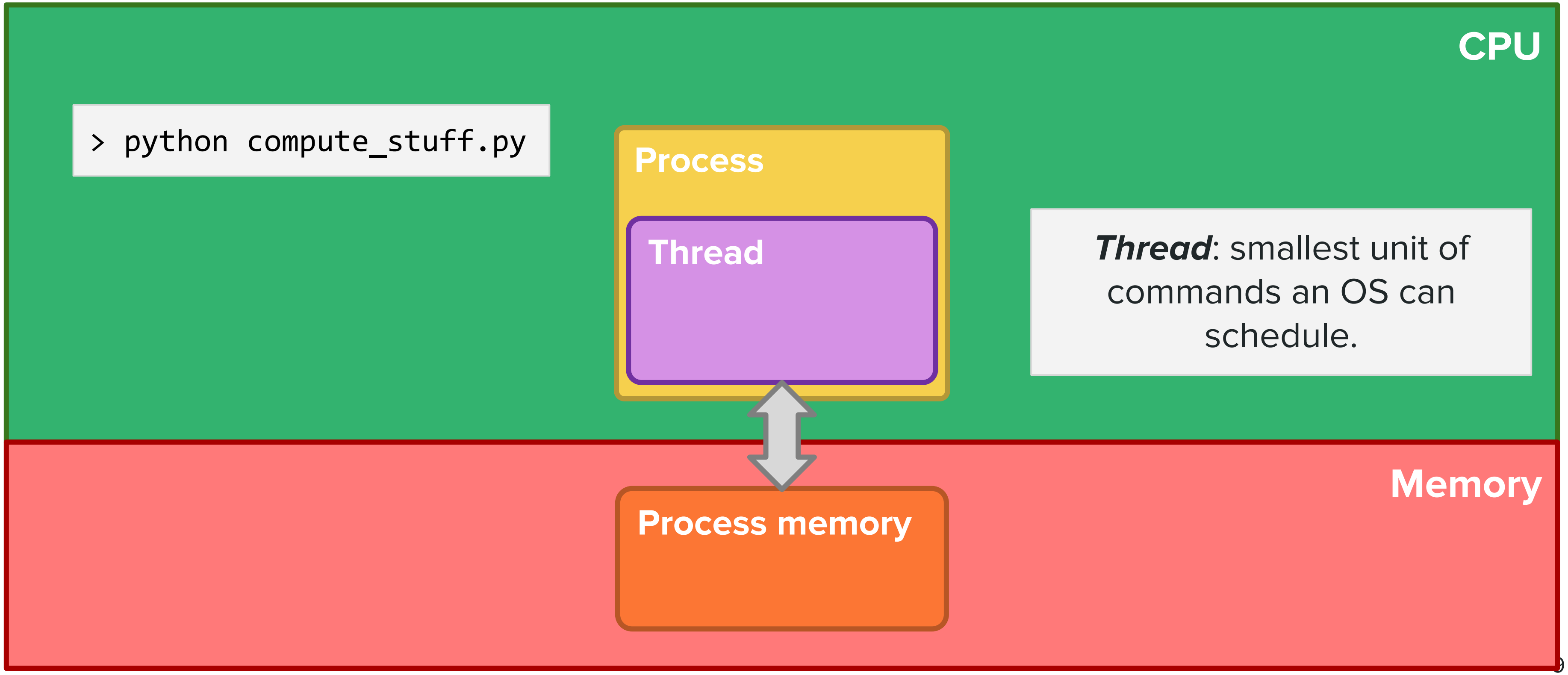
Well, that depends on the architecture.

* “Python” = CPython

Old-school architecture: one CPU.



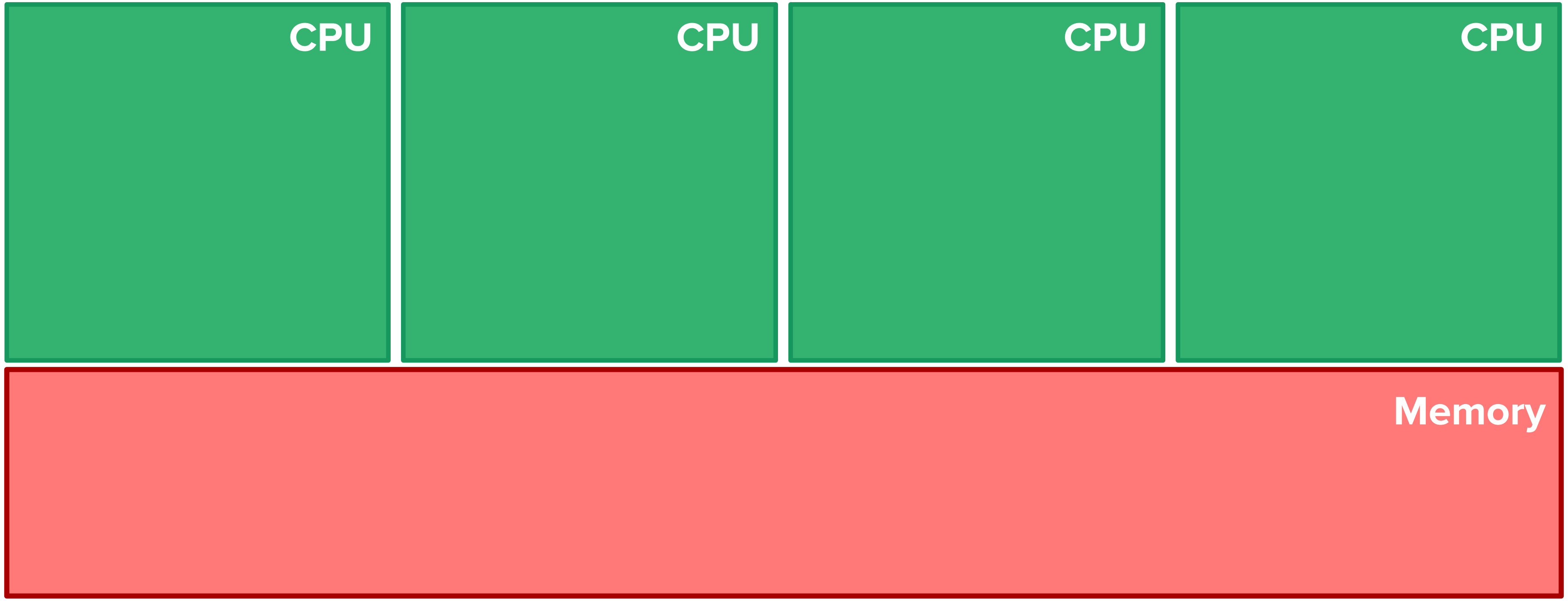
Old-school architecture: one CPU.



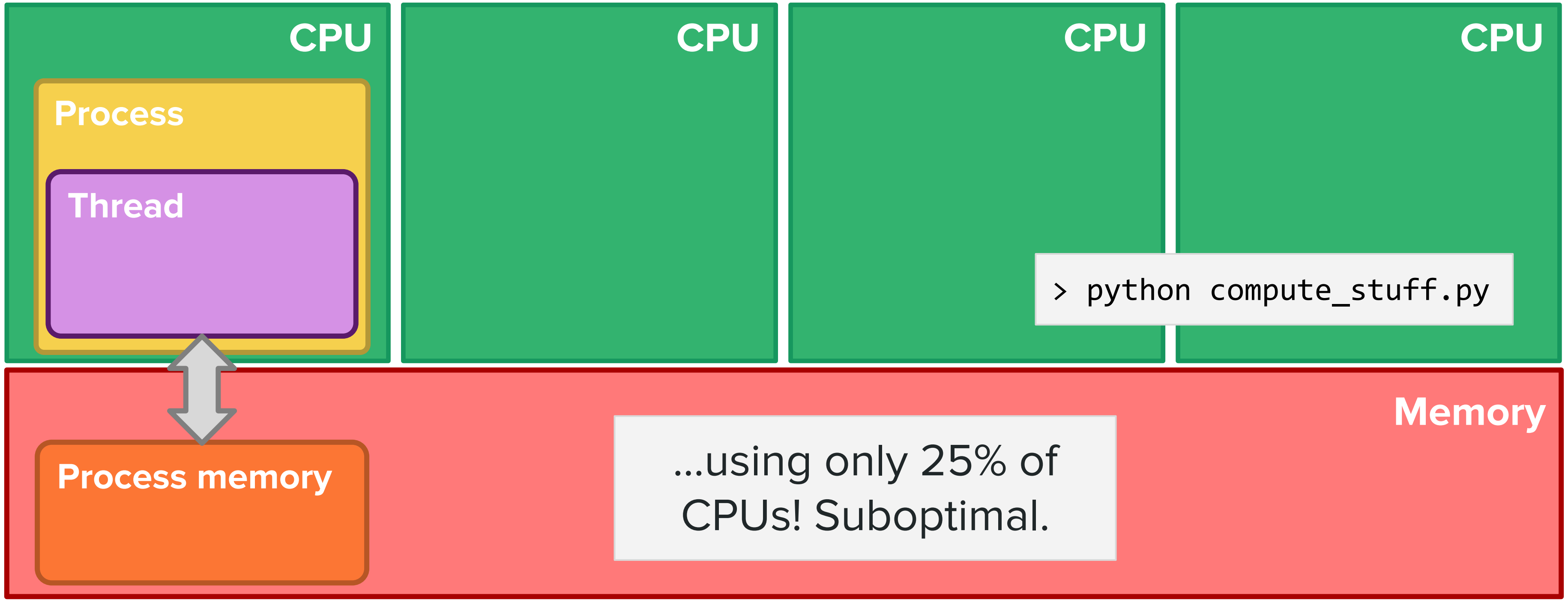
```
> python compute_stuff.py
```

Thread: smallest unit of commands an OS can schedule.

Modern-day architecture: multiple CPUs.



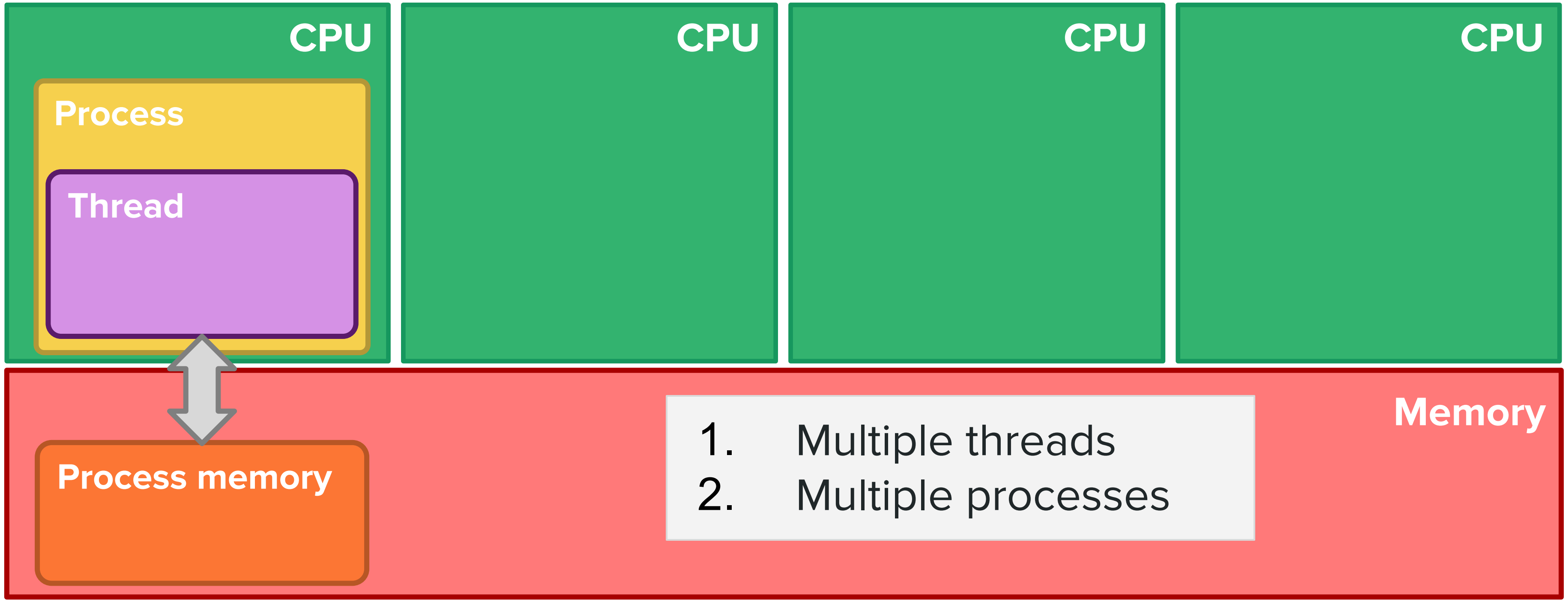
Modern-day architecture: multiple CPUs.



```
> python compute_stuff.py
```

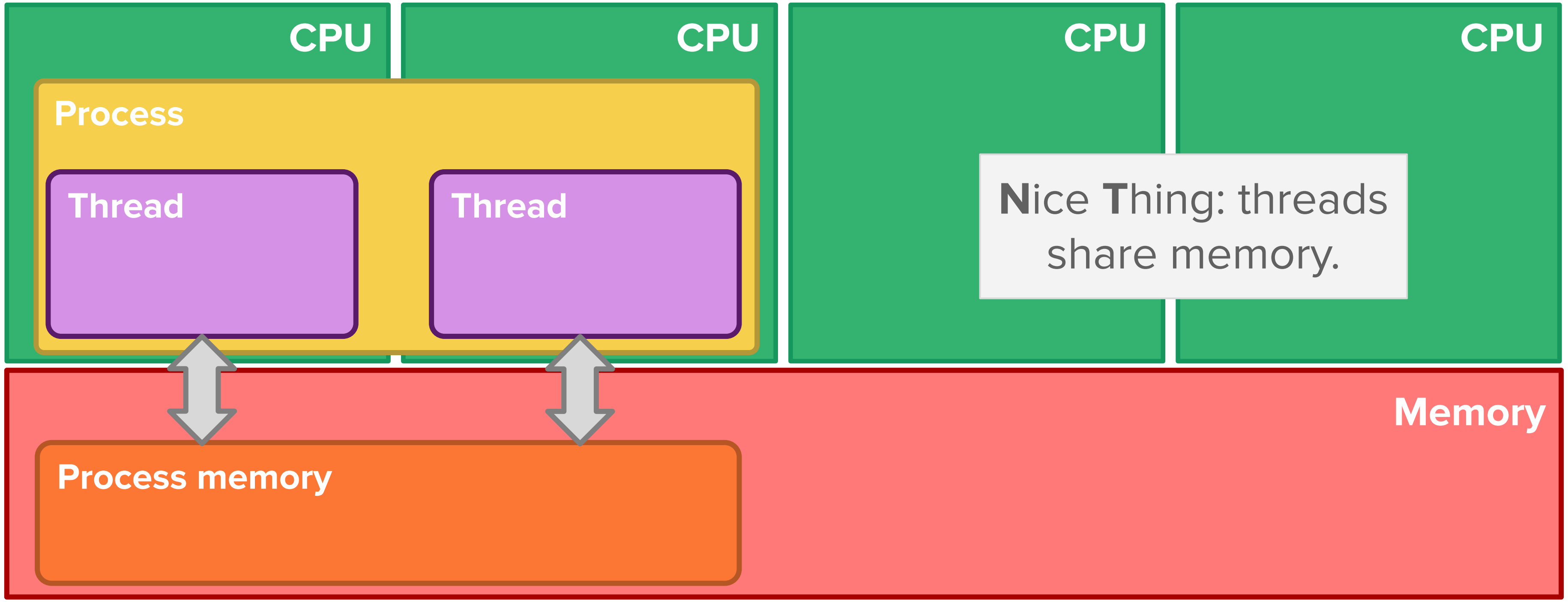
...using only 25% of CPUs! Suboptimal.

How can we use it more efficiently?



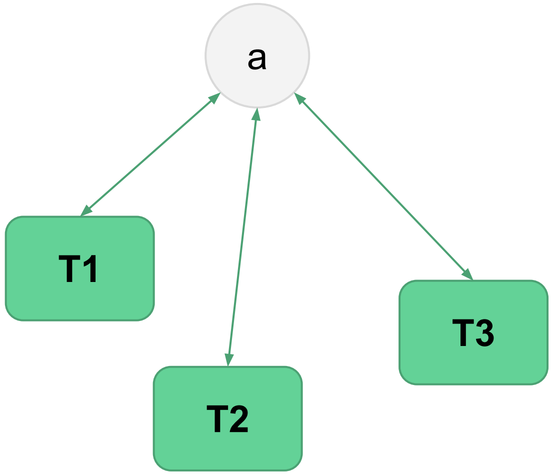
1. Multiple threads
2. Multiple processes

Option 1: Multithreading.



Nice **T**hing: threads share memory.

…but blindly sharing memory can be dangerous.

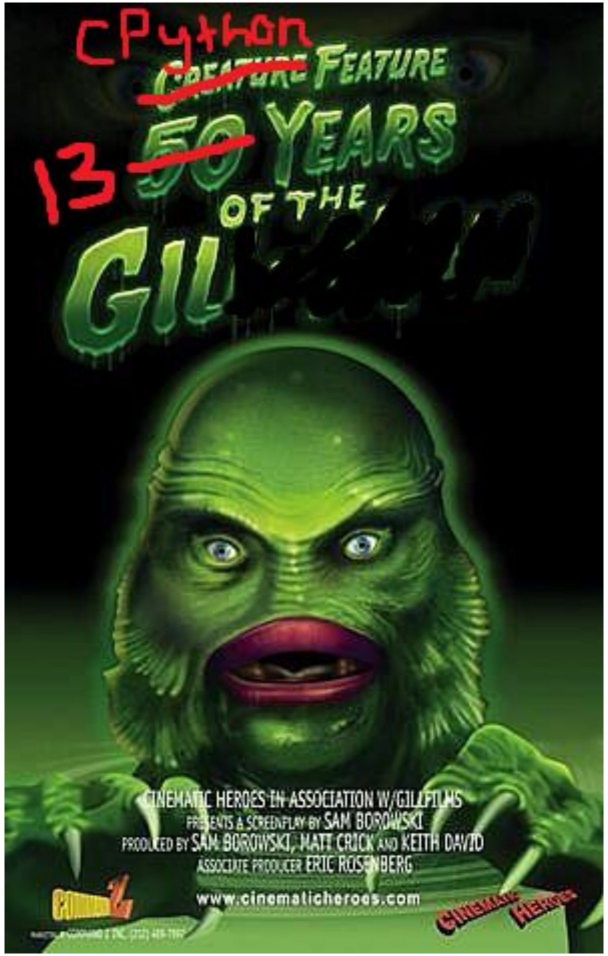


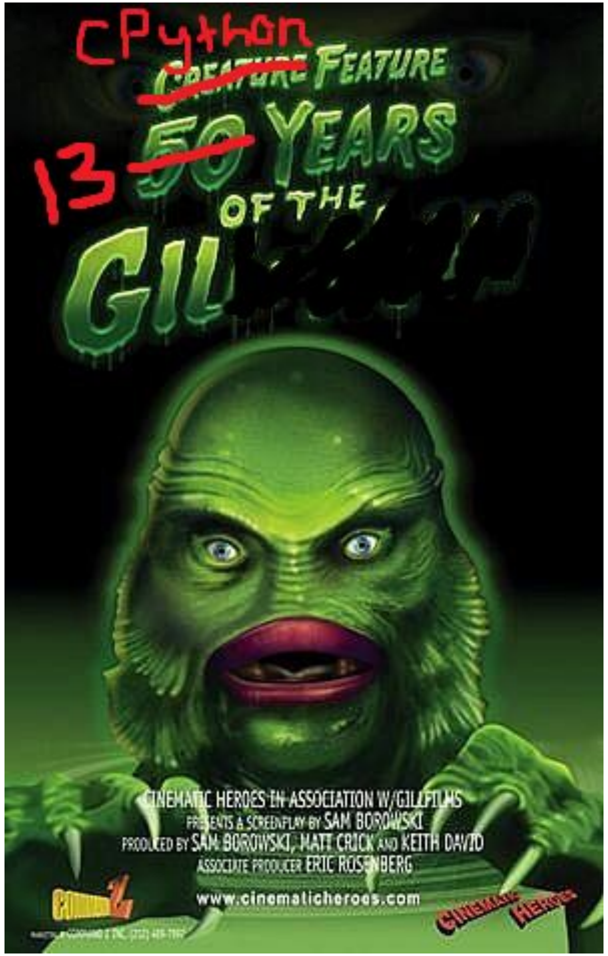
Data corruption, “race conditions” with reference counts resulting in leaked memory or incorrect memory release.

CPython’s solution…

The “Global Interpreter Lock” (GIL).

- Lock is like a speaking stone in a noisy group.
- Requires threads in a Python process to acquire the lock before execution.
- Implemented in CPython and PyPy, not other types of Python.


CPython
~~CREATURE~~ FEATURE
13 ~~50~~ YEARS
OF THE
GIL
CINEMATIC HEROES IN ASSOCIATION W/GILLFILMS
PRESENTS A SCREENPLAY BY SAM BOROWSKI
PRODUCED BY SAM BOROWSKI, MATT CRICK AND KEITH DAVID
ASSOCIATE PRODUCER ERIC ROSENBERG
www.cinematicheroes.com
CINEMATIC HEROES


CPython ~~CREATURE~~ FEATURE
13 ~~50~~ YEARS
OF THE
GIL
CINEMATIC HEROES IN ASSOCIATION W/GILLFILMS
PRESENTS A SCREENPLAY BY SAM BOROWSKI
PRODUCED BY SAM BOROWSKI, MATT CRICK AND KEITH DAVID
ASSOCIATE PRODUCER ERIC ROSENBERG
www.cinematicheroes.com

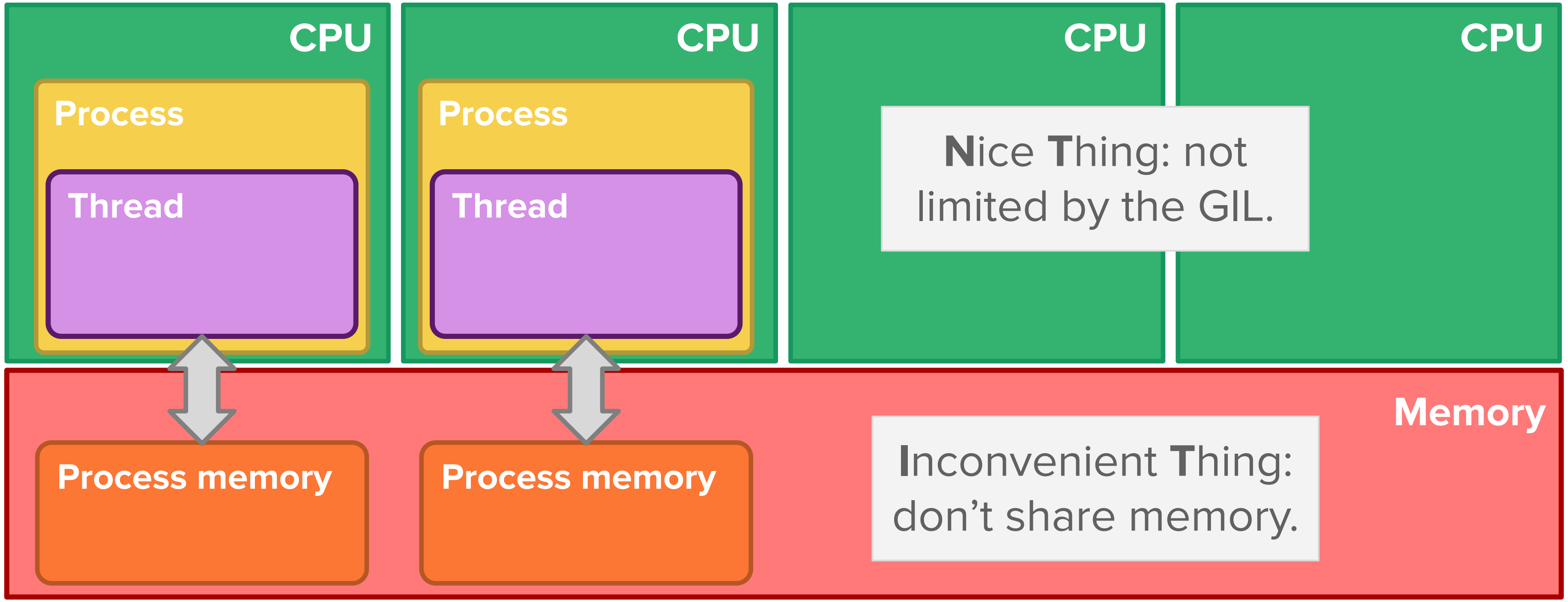
The “Global Interpreter Lock” (GIL).

- Lock is like a speaking stone in a noisy group.
- Requires threads in a Python process to acquire the lock before execution.
- Implemented in CPython and PyPy, not other types of Python.

Bottom line for multithreading:

The GIL may still cause multithreaded code to run sequentially. Non-Python code can get around the GIL (e.g., NumPy). Network-bound problems are also good.

Option 2: Multiprocessing.



Nice **T**hing: not limited by the GIL.

Inconvenient **T**hing: don’t share memory.

So, when to use multithreading or multiprocessing?

Multiple threads.

- Shares memory.
- Limited by the Python GIL.

Use with:

- Not-pure-Python code that releases the GIL.
- Network-bound problems.

Multiple processes.

- Doesn’t share memory (overhead).
- Gets around the GIL.

Use with:

- Pure Python code hindered by GIL.
- CPU-bound problems.

So, when to use multithreading or multiprocessing?

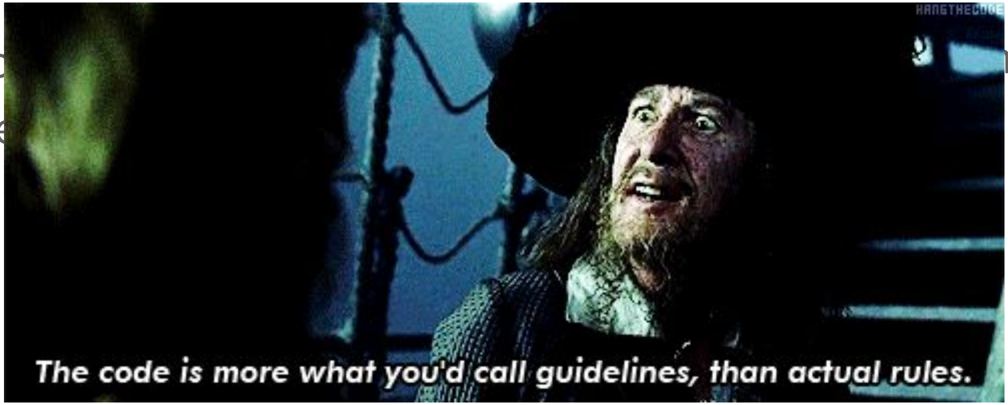
Multiple threads.

- Shares memo
- Limited by the

Multiple processes.

nory (overhead).

IL.


HANGTHECODE
The code is more what you'd call guidelines, than actual rules.

Use with:

- Not-pure-Python code that releases the GIL.
- Network-bound problems.

- Pure Python code hindered by GIL.
- CPU-bound problems.

Next up: let’s learn how to apply it.

Embarrasingly parallel and built-in parallelization.

Some background first.

Jakob will explain a few fundamental concepts before the first exercise.

For now, sit back and watch.

Now time to get your hands dirty.

Open `01-multiprocessing_exercises.ipynb`.

Exercises in parallelization. Outline.

- Exercise 1. Serial Python.
    - Summary here.
- Exercise 2. Thread-parallel Python.
    - Summary here.
- Exercise 3. Process-parallel Python.
    - Summary here.

You can always do this lecture again!

We’ll upload Jakob’s notebook, but there is already another filled one on the repo.

Check it out:

`01-multiprocessing_lecture_filled.ipynb`

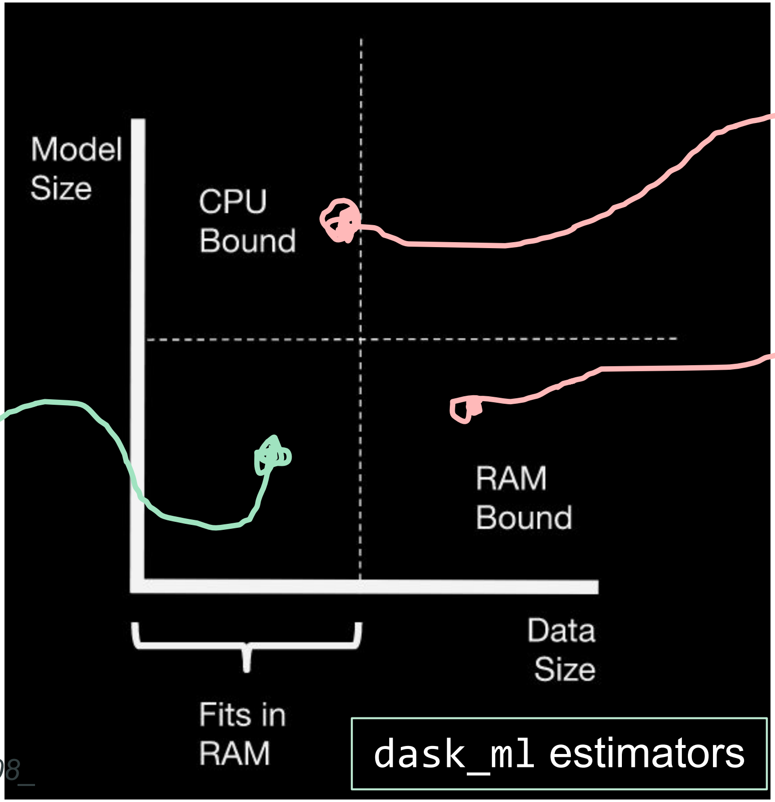
Case study: Machine learning.

Machine learning can complicate things.

multiprocessing

Use scikit-learn

???



Taken from https://github.com/dask/dask-tutorial/blob/main/08_machine_learning.ipynb

Common task: hyperparameter grid search.

- Hyperparameters are “meta” parameters passed into neural nets.
    - Specified *a priori*, affect the quality of the resulting fit.
    - Example: number of layers in a deep net.
- We often want to tune these hyperparameters.
    - Produce the model with the lowest error.
- One method: grid search.

Hyperparameter grid search.

Want to (1) fit model and (2) find model error for each value.

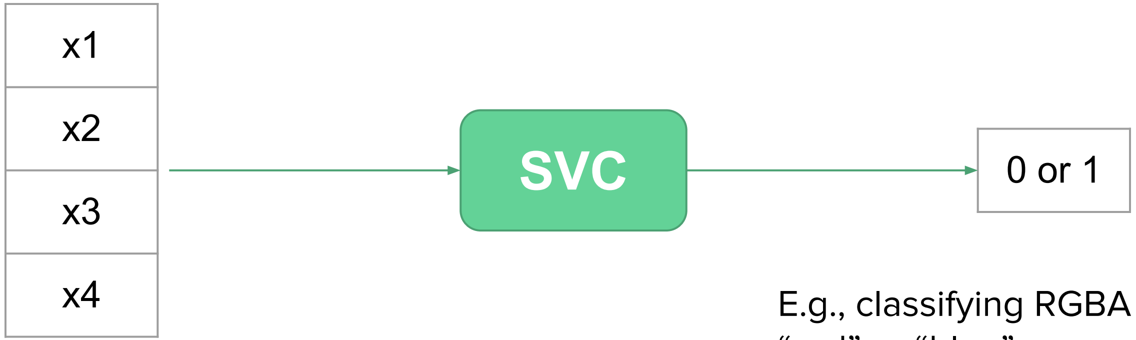
Return the parameters that result in the best score.

Kernel/C	0.001	0.025	2.5	5	10
‘rbf’					
‘poly’					

Question 1: Is this an embarrassingly parallel problem?

Question 2: Is this better suited to multithreading or multiprocessing?

Our neural net: Support Vector Classifier.



E.g., classifying RGBA values into "red" or "blue".

- SVC model has many hyperparameters. We'll consider two.
  - C: Regularization parameter.
  - kernel: Kernel type to be used in the classification.

You don't need to know what these mean. Just that we want to find good values.

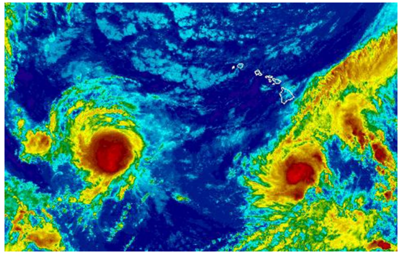
Let’s try parallelizing our grid search.

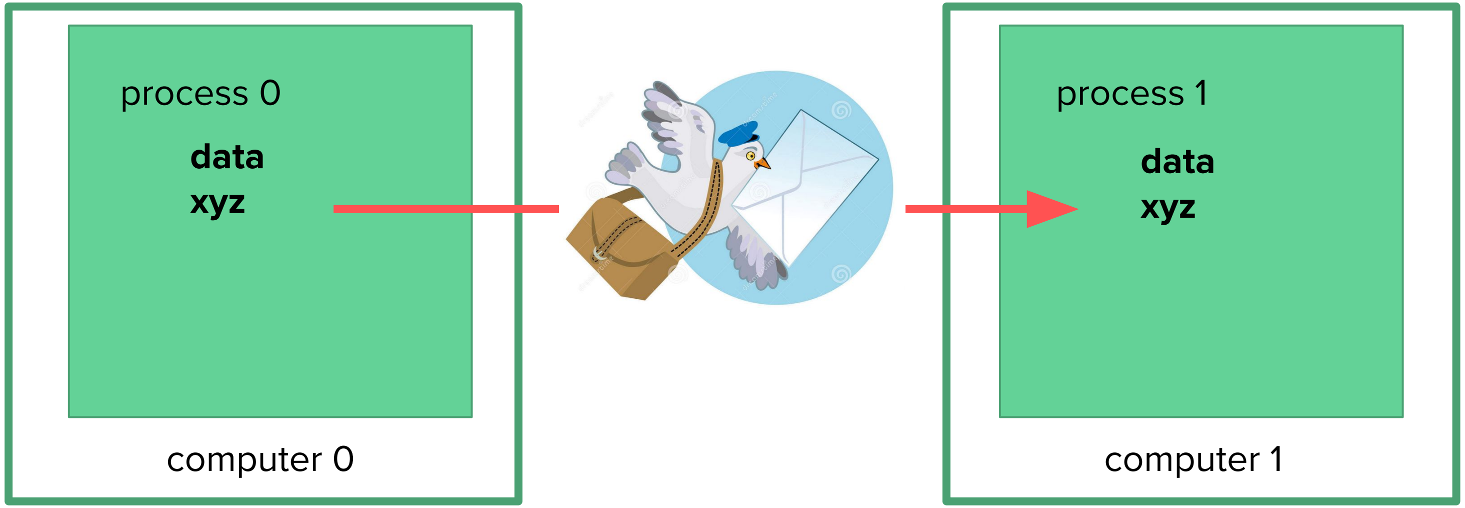
- Open the notebook `02-parallel_machine_learning.ipynb`.
- The notebook will walk you through:
  - Fitting an SVC model for a single set of hyperparameters.
  - Using the parallelization options in `scikit-learn`.
  - Using `dask` to parallelize.
- Keep track of the runtimes!

Let’s go through the notebook together.

Case study: Using mpi4py.

MPI (Message Passing Interface)





- MPI is a standard for passing data ("messages") between processes.
- OpenMPI/MPICH/... are C-libraries following the MPI standard.
- mpi4py allows you to use these libraries from Python to communicate between processes.

A **brief** introduction to mpi4py.

Jakob will explain a few fundamental concepts.

Sit back and watch.

Why (not) MPI?

+ high-level of flexibility
+ high performance (when used correctly)
+ leverage the combined power of thousands of computers

- cognitive overhead
- difficult to use effectively
- debugging ~~can~~ is a nightmare

(pictured: typical MPI user)

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.

Other resources

- Multithreading vs. multiprocessing: what you need to know. (Article)
- Advanced Python topics: Threads in Python. (Article)
- Matthew Rocklin at PyCon.DE. Dask: Next Steps in Parallel Python. (Video)