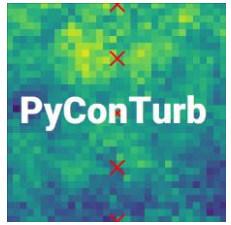
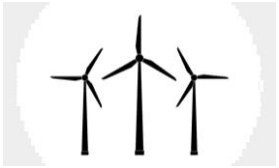
Who I am. Jenni Rinker.

1. Can't overtalk.
2. Please stop me.



- **Student** in 2017. **Lecturer** in 2018: Packaging/Documenting (ASPP-APAC).
- **Researcher**. Loads and Control, Department of Wind Energy.



DTU Technical University of Denmark

- Previously a few other places.

Duke University | Pratt School of Engineering

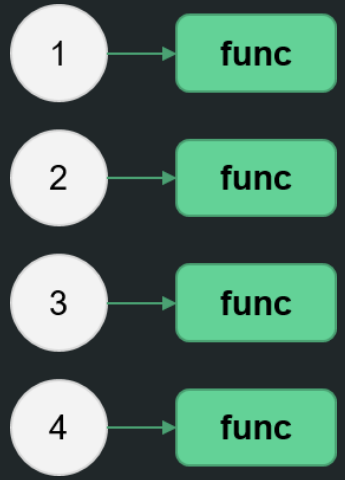
NREL National Renewable Energy Laboratory

Harvey Mudd College

Alaska Grown

- **PyConTurb** (Python Constrained Turbulence) developer.
- **Research interests**. Turbulence-turbine interaction, uncertainty quantification, stochastic systems, structural dynamics, etc.

Parallel Python



Jenni Rinker
Researcher
DTU Wind Energy

Lecture inspired by John Kirkham's 2019 ASPP-APAC lecture. Other content synthesized from the internet.

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.

Part 1: Threads, Processes and the GIL.

First step: understand default behavior.

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

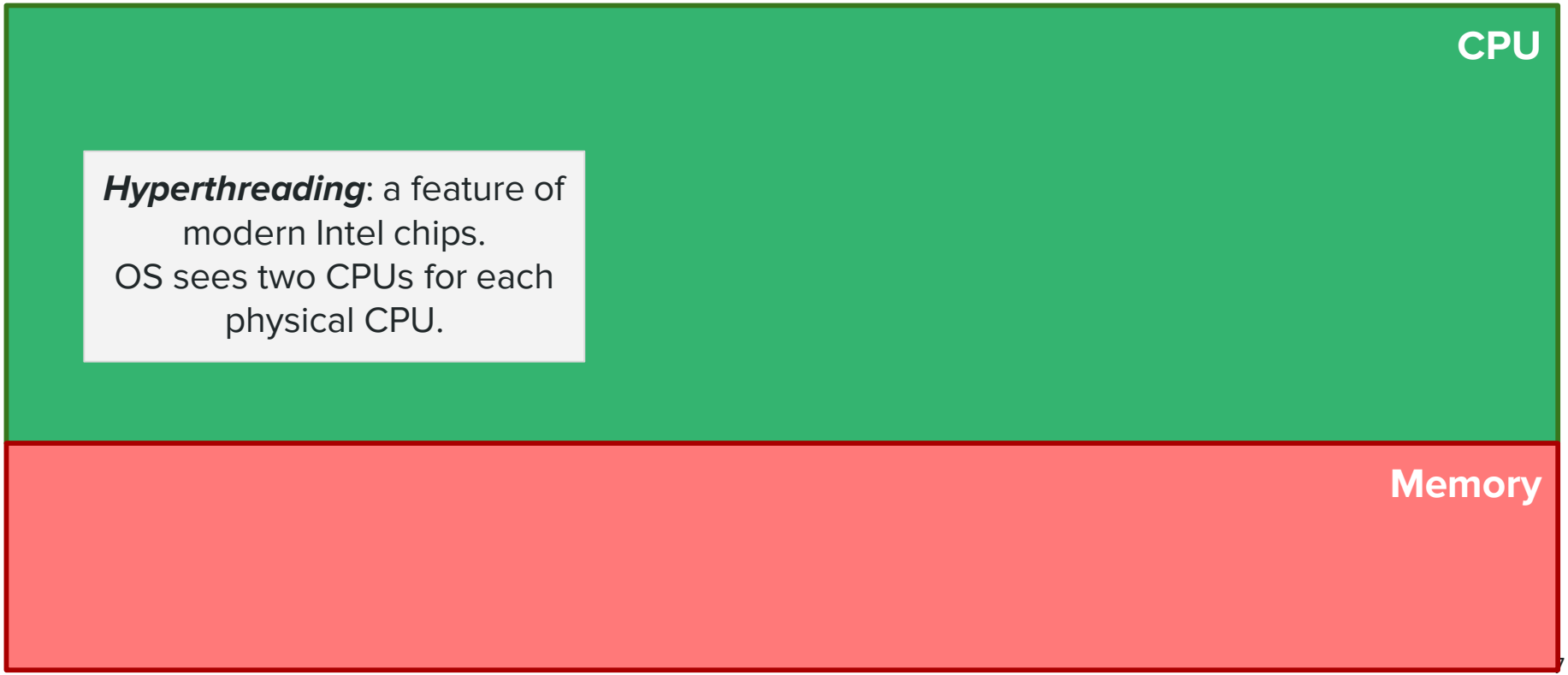
For example, if we enter the following command in a terminal *with no special functions*, what is our computer actually doing?

```
> python compute_stuff.py
```

Well, that depends on the architecture.

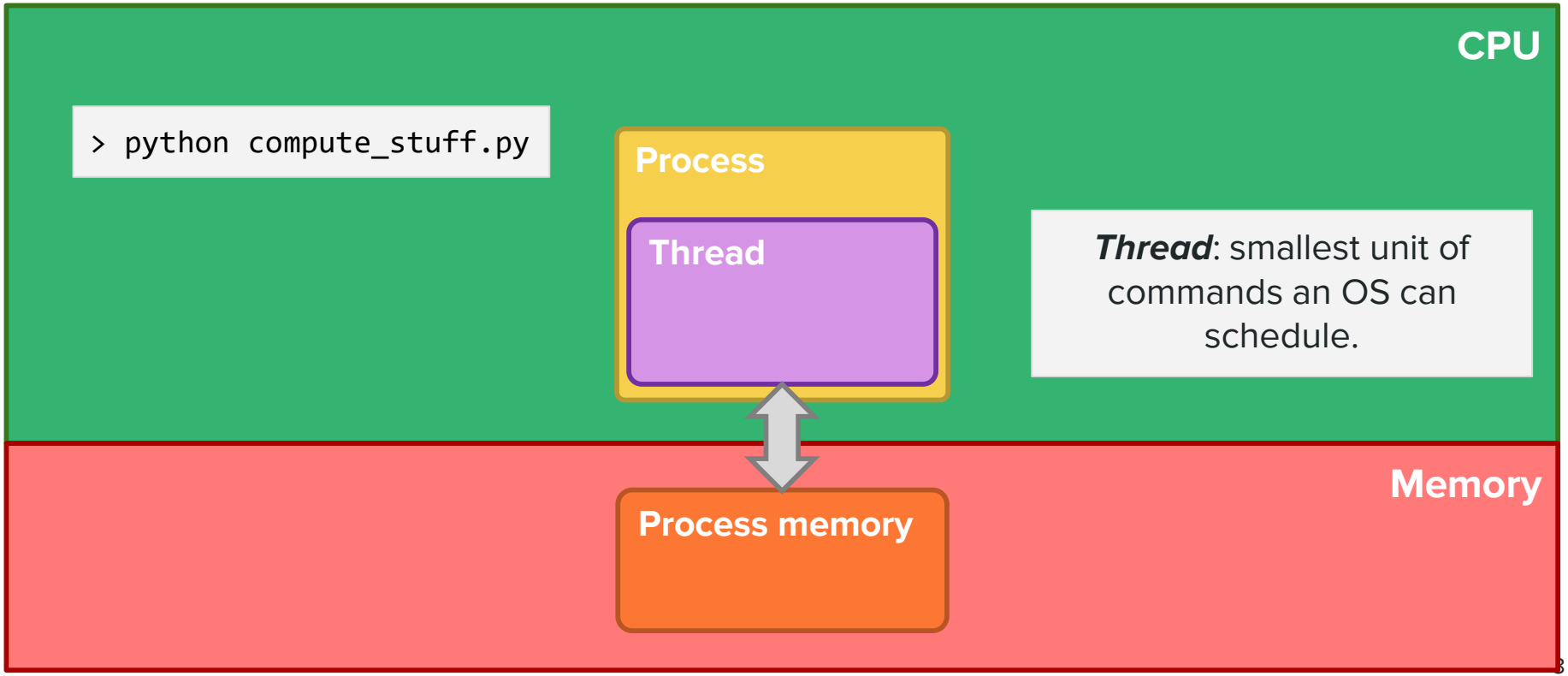
* “Python” = CPython

Old-school architecture: one CPU, no hyperthreading.

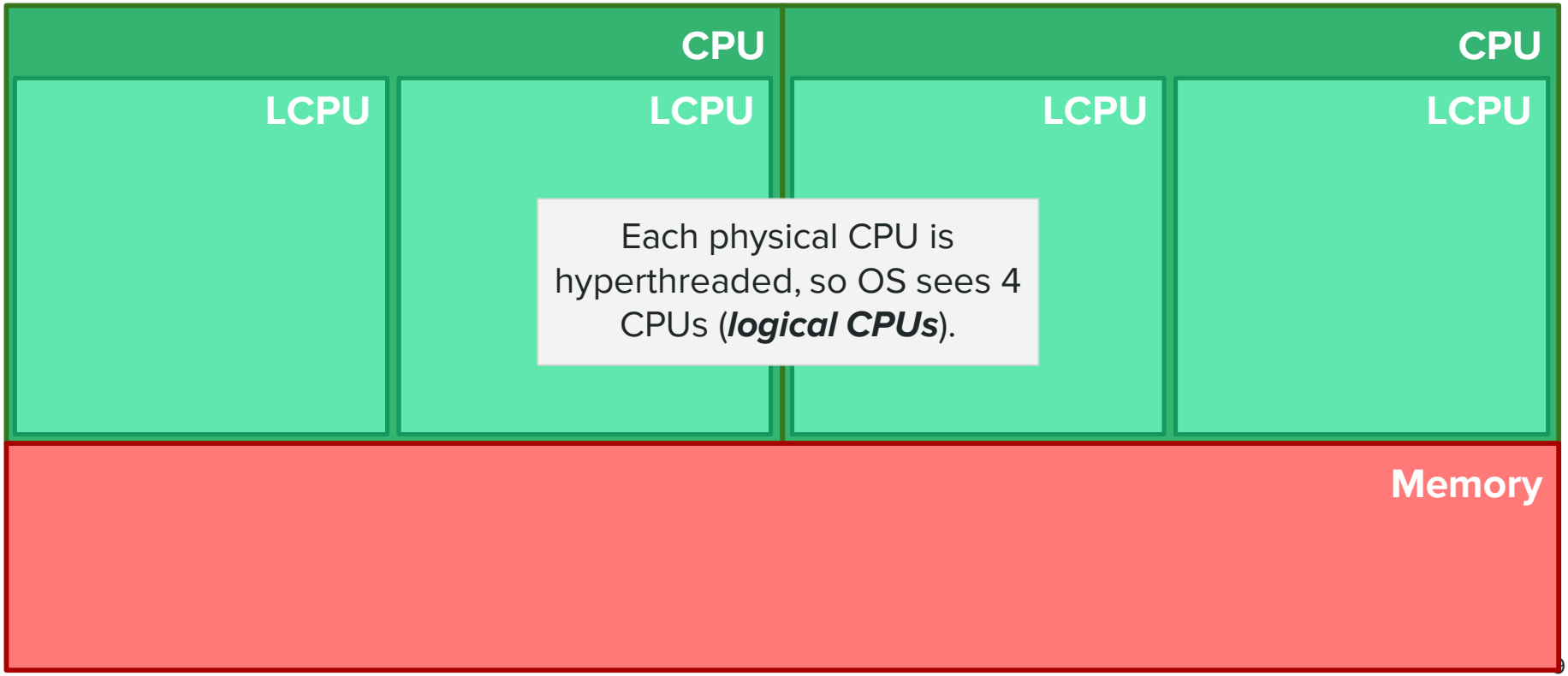


Hyperthreading: a feature of modern Intel chips.
OS sees two CPUs for each physical CPU.

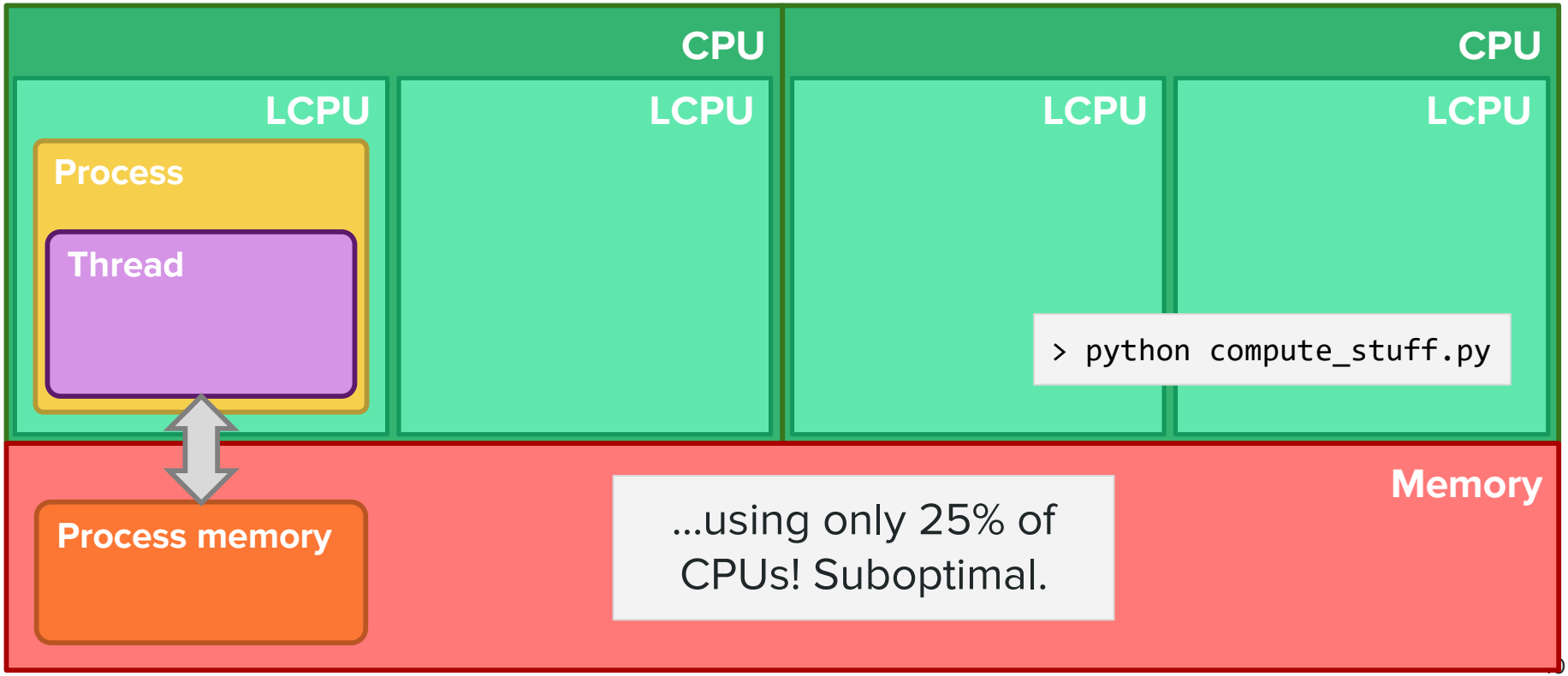
Old-school architecture: one CPU, no hyperthreading.



Modern-day architecture: multiple CPUs, hyperthreading.



Modern-day architecture: multiple CPUs, hyperthreading.

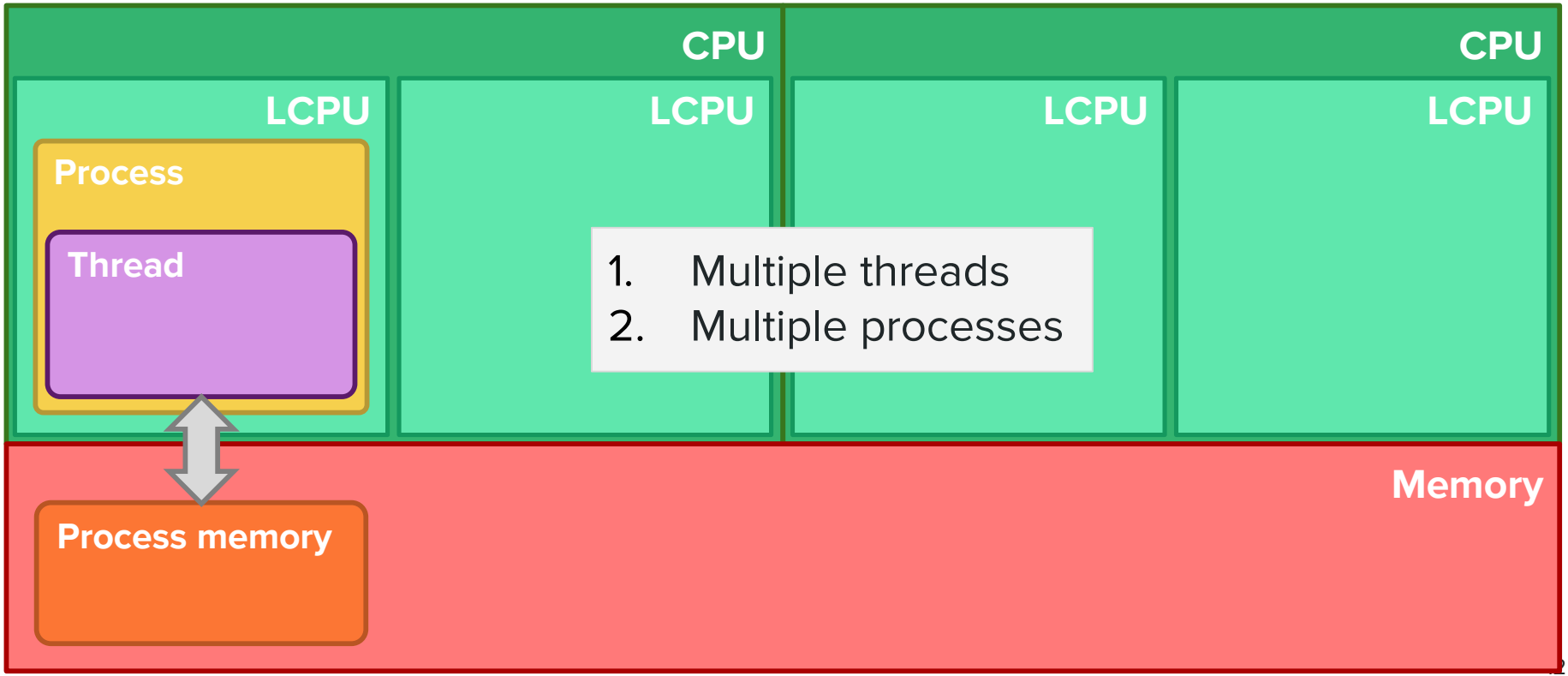


Quick review of terminology before we move on.

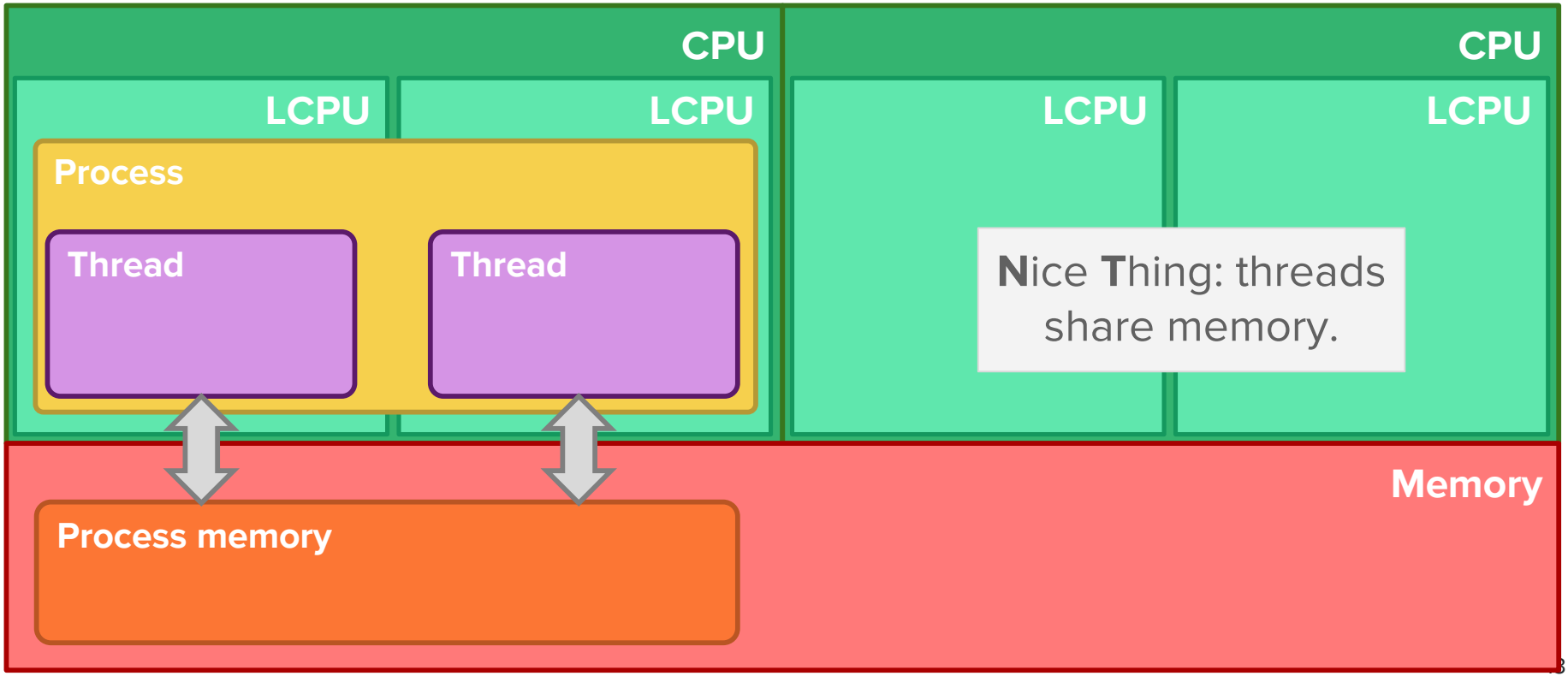
- ***CPU (processor)***: a unit on your computer that performs computations. (Tautological, I know.)
- ***Hyperthreading***: a feature of Intel chips that makes one physical CPU into two logical CPUs.
- ***Process***: an instance of a program (e.g., Python interpreter). Empress of the calculations.
- ***Thread***: a chain of executable commands, smallest that can be scheduled by the OS. Spawned by a process.

Any questions so far?

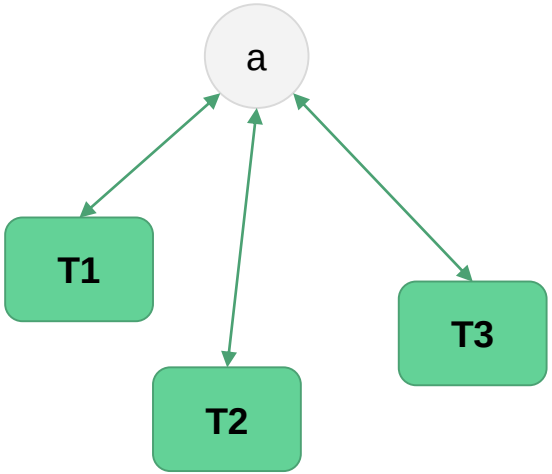
Back to our computer. How can we use it more efficiently?



Option 1: Multithreading.

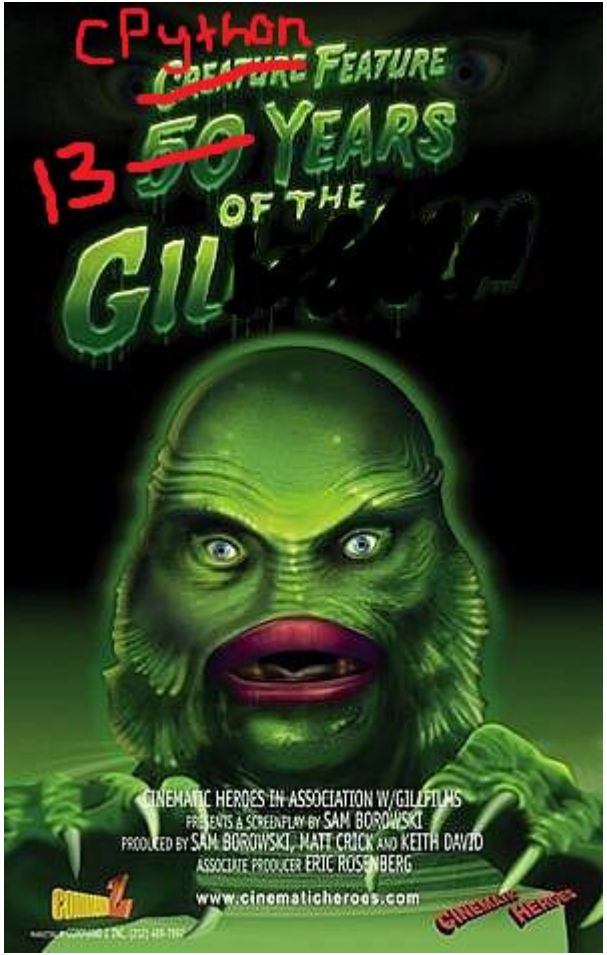


...but blindly sharing memory can be dangerous.



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

CPython’s solution...



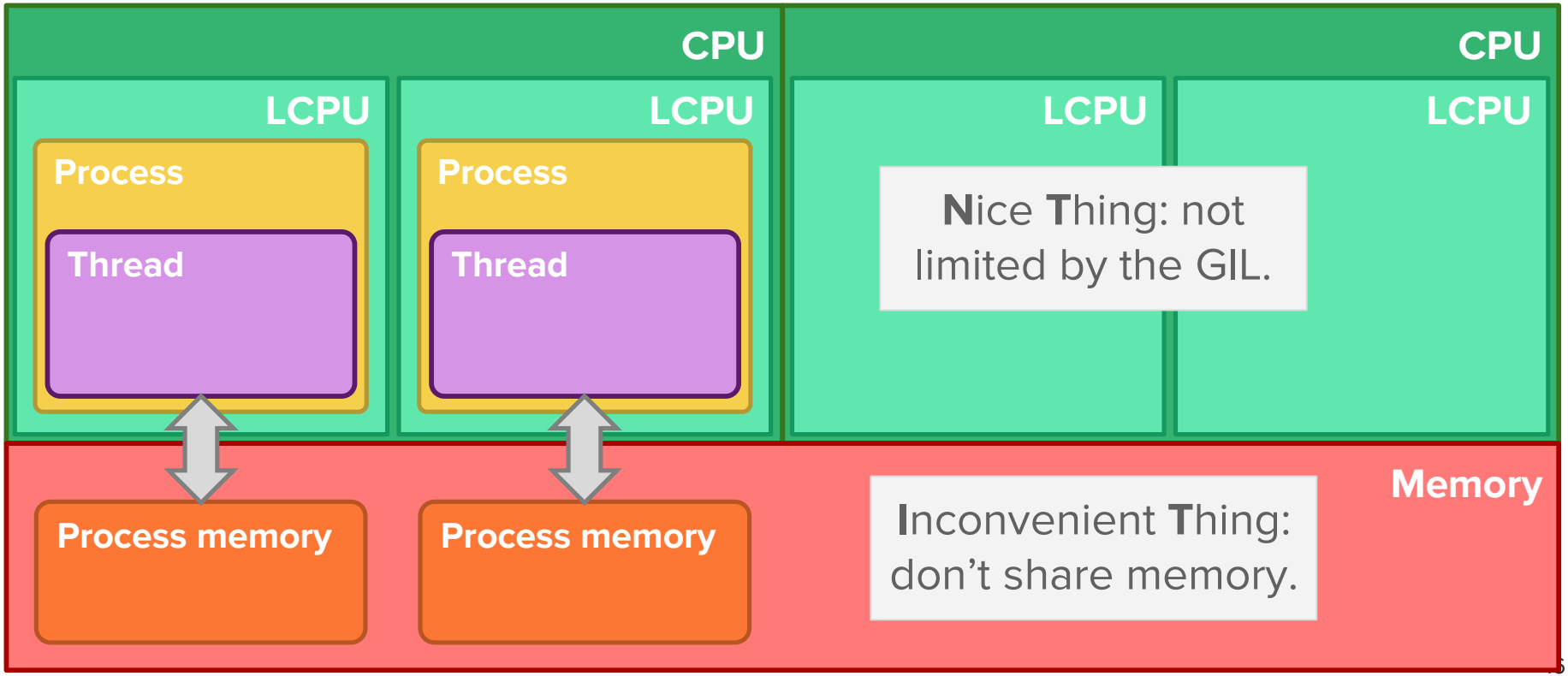
The “Global Interpreter Lock” (GIL).

- 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.

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.



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.

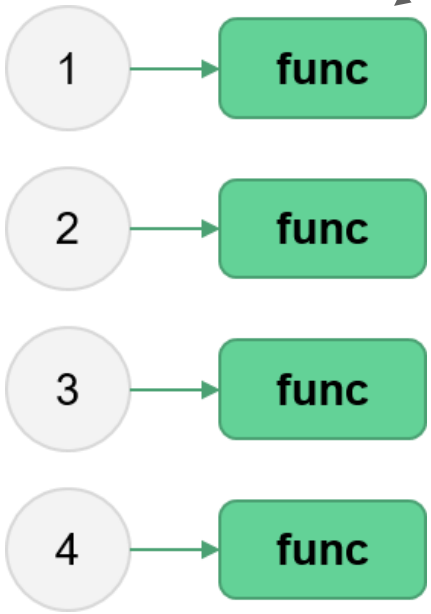
Use with:

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

Exercise. Embarrassingly parallel: multiple threads vs. multiple processes.

1. Write this function

2. Evaluate this workflow using
a) single thread
b) multiple threads
c) multiple processes



3. What do you observe? Document your results. Commit **ONLY** your solution file. Push your branch.

Note! This demo uses dask functions we'll explain later.

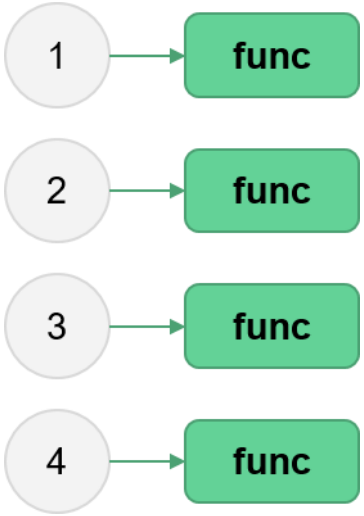
Exercise. Embarrassingly parallel: multiple threads vs. multiple processes.

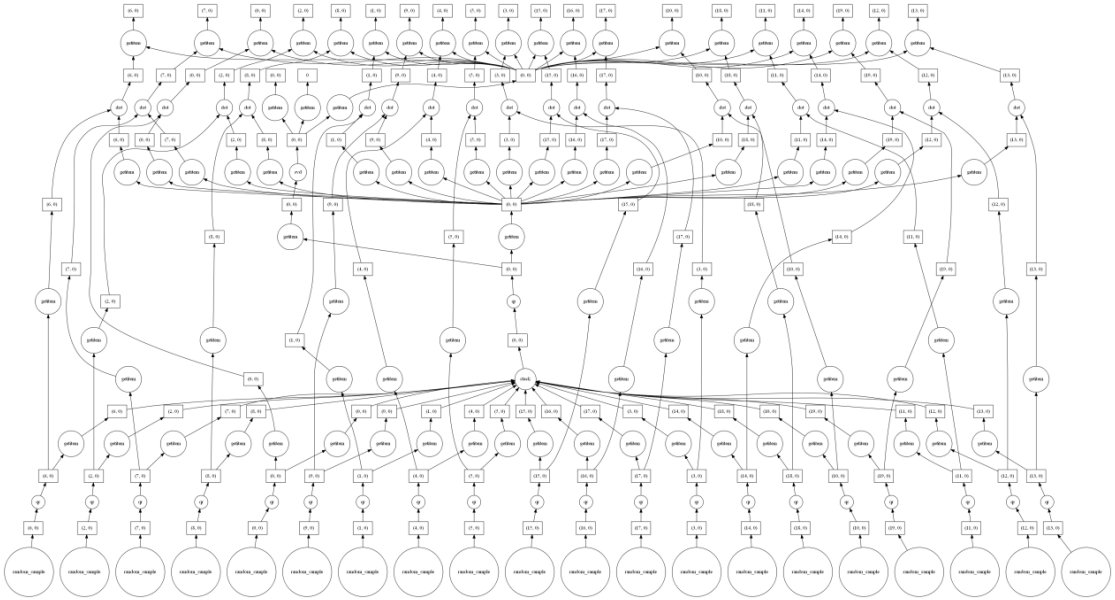
- Let’s go over it together. Who’s got a branch name for me?
- Difference when running with processes on Windows.

Part 2: Task Graphs and Not-So-Embarrassing Problems.

But what about more complicated computations?

We can use multithreading or multiprocessing for this...





But how do we apply that to something like a SVD of a massive matrix?

Different parallelizing methodologies.

Note: this is shamelessly copied from Matthew Rocklin's 2017 PyCon.DE lecture. Check it out.

1. **Embarrassingly parallel**: multiprocessing, joblib
   - Map a function onto a list of data, 90% of problems
   - **Pros**: easy to install, lightweight, familiar
   - **Cons**: doesn't scale, can't handle complex algorithms
2. **BigData collections**: MapReduce, Flint, Spark, SQL
   - Collection of scalable functions useful for big data
   - **Pros**: lots of new operations, scales nicely, mature and trusted
   - **Cons**: heavyweight, can't handle complex computations
3. **Task schedulers**: Airflow, Luigi, Celery, Make
   - Define a graph of functions with data dependencies, scheduler handles parallelization
   - **Pros**: arbitrarily complex problems, Python native
   - **Cons**: no inter-worker storage, possible long latencies, not designed for user interaction

In this tutorial, we use dask.

A Python package that “provides advanced parallelism for analytics, enabling performance at scale for the tools you love.”

More practically, a package developed by developers at Continuum Analytics for easy parallelization of stuff.

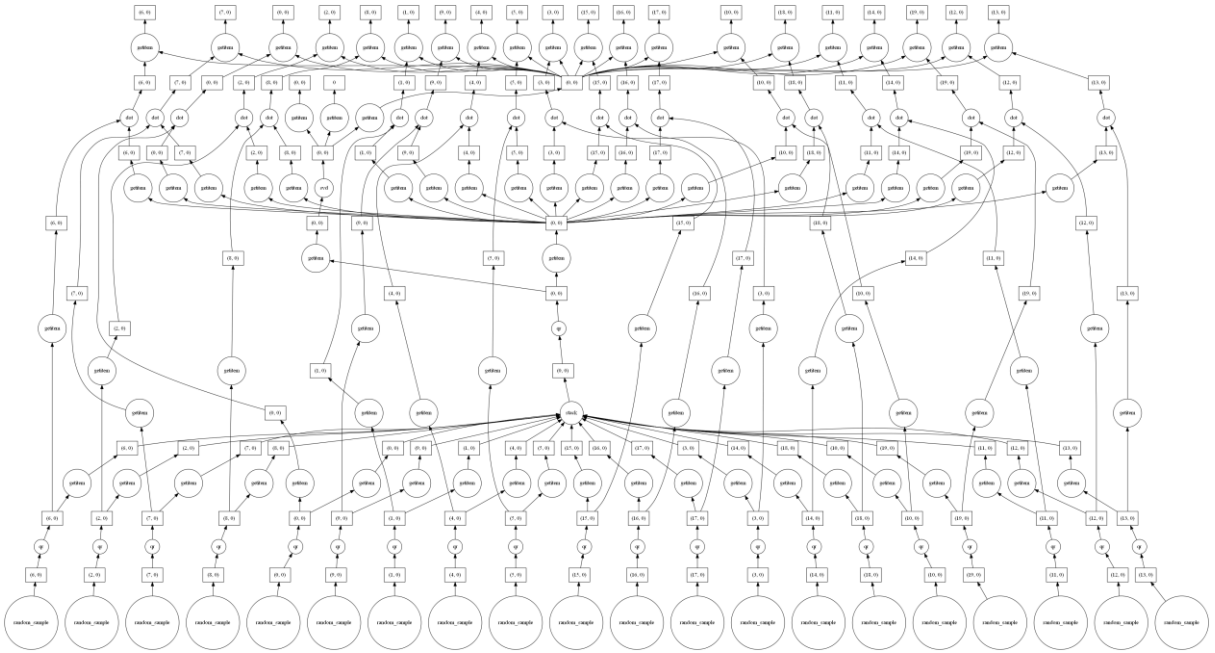
In the words of Rocklin (PyCon.DE 2017),

> *“We want:*
> - *Scalability of Spark/Flink/Databases*
> - *Flexibility of Airflow/Celery for complex dependencies*
> - *Familiarity and lightweight nature of multiprocessing”*

Two main steps.

1. Creation of a task graph.

2. Parallel computation of the task graph (scheduler).



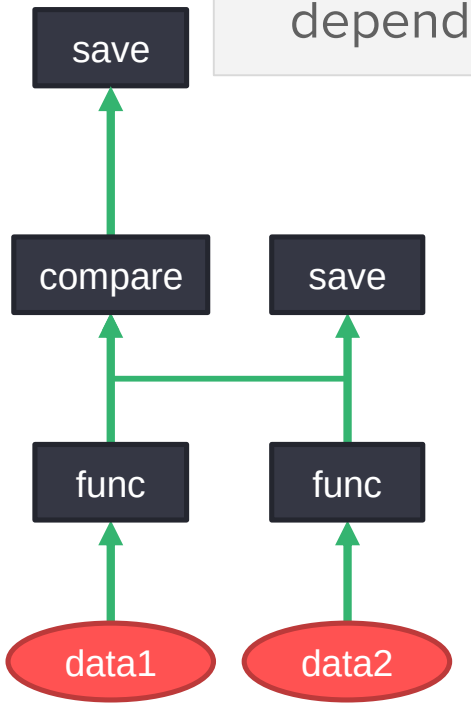
Task graphs.

```
% run a function on datasets
out1 = func(data1)
out2 = func(data2)

% compare outputs
diff = compare(out1, out2)

% save useful results
save(out2)
save(diff)
```

In short, a visual representation of functions and data dependencies.



Demo: dask delayed.

Live coding demo.

Summary:

- Delayed is a lazy version of a function.
- Can visualize using `dask.visualize(z)` or `z.visualize()`.
- Compute using `dask.compute(z)` or `z.compute()`.

Exercise. Build a better reduce.

Exercise located in `2_better_reduce.ipynb`.

1. Create a task graph for reduce using a delayed function.

2. Make a new reduce function that is more parallelizable.

3. Look at the task graph with your new function, then test it. Does it work?

4. Add/commit **ONLY** your solution in a new branch. Push it. Push it real good.

Inspiration in `demo_dask_delayed.ipynb`.

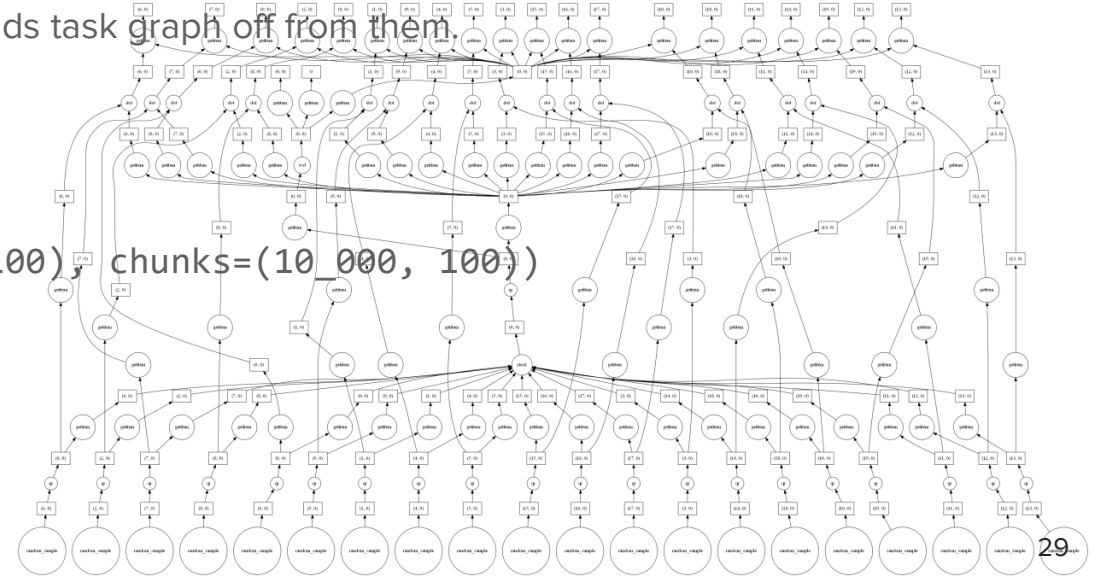
Exercise. Build a better reduce.

Who has a branch with a solution?

Still a pretty simple workflow. Can dask do anything cooler?

- Dask array.
  - Essentially a NumPy array, but ready to deployable/parallelizable in a distributed manner.
  - Separates array into chunks, builds task graph off from them.

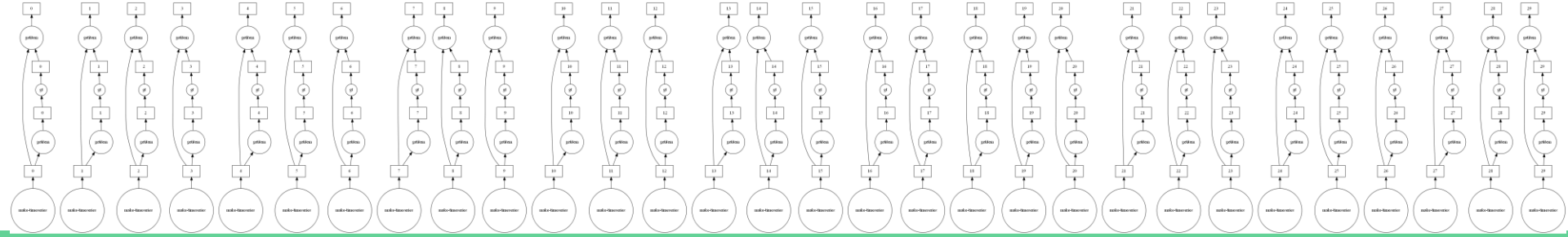
```
> from dask import array as da
> X = da.random.random((200_000, 100), chunks=(10_000, 100))
> u, s, v = da.linalg.svd(X)
> dask.visualize(v)
```



Hm...yeah...still not convinced.

- Dask dataframe.
  - Essentially a Pandas dataframe, but ready to deployable/parallelizable in a distributed manner.
  - A collection of dataframes.

```
> df = dask.datasets.timeseries()
> df2 = df[df.y > 0]
> df2.visualize()
```



What this means for you.

- Potentially can parallelize your code by replacing NumPy arrays with dask arrays, Pandas dataframes with dask dataframes.

- Setting up task graphs is very easy with dask. Either low-level (delayed) or high-level (arrays/dataframes).

- The tricky part: launching the computations efficiently. (On your computer, on a cluster, etc.). So now we need to talk about **schedulers**.

Part 3: Schedulers and Running Computations.

Single-machine scheduler.

Default scheduler. Optimized for larger-than-memory problems.

Scheduler options:

- `single-threaded`: for-loop in the current thread.
  - Useful for debugging.
- `threads`: uses `multiprocessing.pool.ThreadPool` in the local process.
  - Default for `dask.array`, `dask.dataframe` and `dask.delayed`.
- `processes`: uses `multiprocessing.Pool` to spawn multiple processes.
  - Default for `dask.bag`, sometimes useful for `dask.dataframe`.
  - For GIL-bound code, the `dask.distributed` scheduler is better. (Coming next...)

Single-machine scheduler.

Multiple ways to change the options.

- When calling compute.

```
z.compute(scheduler='threads')
```

- With a context manager.

```
with dask.config.set(scheduler='threads'):
    z.compute()
```

- Globally.

```
dask.config.set(scheduler='threads')
```

A different scheduler: distributed scheduler.

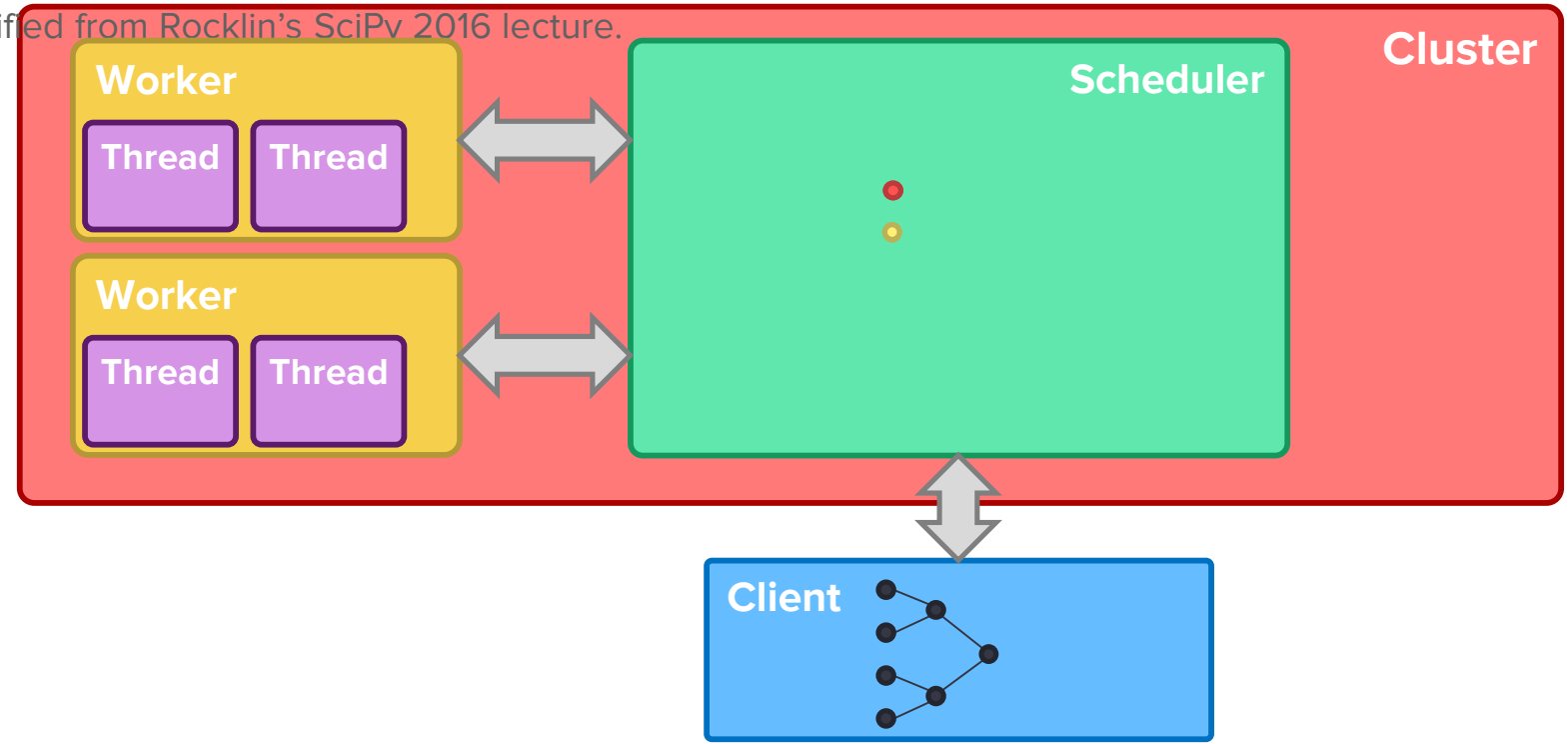
- Also works on a single machine, but other nice things:
  - Works on a cluster or a personal computer equally well.
  - Access to asynchronous API. (Don’t worry about this.)
  - Handles data locality better, so better than multiple processes option in SMS.
  - Very cool thing: dashboard. (We’ll see this soon.)

- For more info on this scheduler, highly recommend Rocklin’s SciPy 2016 lecture.

- Slightly more complicated than the single-machine scheduler. Let’s look.

Distributed scheduler diagram. (Default)

Modified from Rocklin's SciPy 2016 lecture.



Choosing number of workers and threads.

- In general, `n_cores = n_workers * threads_per_worker`.

- Similar rules as before:
  - Code that releases the GIL: fewer processes, more threads.
  - Pure-Python, CPU-bound code: more processes, fewer threads.

- "*In short, if you're using mostly numpy/pandas-style data, try to get at least eight threads or so in a process. Otherwise, maybe go for only two threads in a process.*" – Rocklin in Stackoverflow comment.

Demo/Exercise. Using a distributed scheduler.

- `demo_distributed_scheduler.ipynb`. Based on our `myreduce` example.

- **Exercise**. Try the following variations. Document the values for each attempt, time to simulate, observations and any other interesting behavior.
  - Change the number of workers (`n_workers`).
  - Change the number of threads per worker (`threads_per_worker`).
  - Try using threads instead of processes (`processes=False`, `n_workers`=?).

- Note: dask doesn't always shut a cluster's dashboard down cleanly. If your Graph tab stops working, restart your kernel.

Demo/Exercise. Using a distributed scheduler.

n_workers	threads_per_worker	processes	Time	Observations
1	1	True	Slow	Progresses serially through graph, but shows multiple things being “processed” at the same time.
1	4	True	Slower	
4	1	True	Faster	
1		False	Slower	Possibly irrelevant number of workers?
8				Slower than with 4

Exercise. If time.

- Using your distributed cluster, mess around with the following examples:
  - Dask array
  - SVD of a large matrix
  - Dask dataframe
  - Machine learning with dask (requires scikit-learn)

Final notes and best practices.

- Try to avoid dask. Better algorithms, compiled code, profiling, etc.
- Use the dashboard. Parallelism with dask can be counterintuitive.
- In a script, be sure to place any Client() calls in an `if __name__ == ‘__main__’` block.
- The syntax is redundant; there are probably 3 other ways to do everything we did.
- Set global variable `OMP_NUM_THREADS` to 1 when using NumPy operations. (Or use a single thread per worker...) BLAS is also multithreaded.

Things we didn’t discuss.

- Dask also has a `bag` object, which is like a dictionary or list.
- Parallelize machine learning using `dask-ml` (`scikit-learn`).
- Launch dask computations on an HPC cluster with a job scheduler (e.g., PBS/SLURM/etc.) using dask-jobqueue.
- Active work is underway for getting dask on GPUs. See the blog.
- Honestly? A lot.

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

- Simple computer architecture. CPUs, hyperthreading, processes, threads.
- The GIL, multithreading and packages that get around the GIL.
- Task graphs and designing functions that are more parallelizable.
- How to use the single-machine and distributed schedulers in dask.
- Best practices and unmentioned things.

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 – I’d like to know how to improve.

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)
- Dask and Distributed Computing, SciPy2016. (Video)
- ”Distributed: basic setup and use” screencast. (Video)
- Dask tutorial. (Jupyter notebooks)
- Dask dashboard walkthrough. (Video)
- Nice stackoverflow comment on number of workers. (Text)
- Dask best practices. (Article; links to best practices for arrays, dataframes and bags)
- State of the art on the dask blog. (Articles)
- Dask and machine learning. (Example code)