Parallel Part III

Outline

- ~~Processes, threads and THE GIL~~
- ~~Hands-on investigations of embarrassingly parallel problems~~
    A. ~~Multithreading with NumPy~~
    B. ~~The `multiprocessing` package~~
    C. ~~Blending processes and threads~~
- Going further
- Wrap-up

Going further

Topics

- `Asyncio` and coroutines
- The near-future: sub-interpreters
- Larger-than-memory problems
- Not-embarrassingly-parallel problems
- Other tools
- Parallelism on clusters

Concurrency in Python

Concurrency: multiple tasks making progress at the same time.

- Parallelism is a type of concurrency.

1. Multithreading (<=2.3)
2. Multiprocessing (<=2.6)
3. Coroutines and `asyncio` (3.4)
4. Sub-interpreters (3.12)

See [1] for more discussion.

The `asyncio` package

As of Python 3.4.

Concurrent but not *parallel*.

Single thread that can switch between tasks at points you specify.

Use case: slow I/O with many connections [2].

- E.g., web scraping.

A bit of a learning curve.

Pseudocode

Inspired by [3]

```
async def main():

  urls = ['www.yes.com',
          'www.no.com']

  # asynchronously collect
  # url content with 1 thread
  content = await scrape(urls)

  # process content here...
```

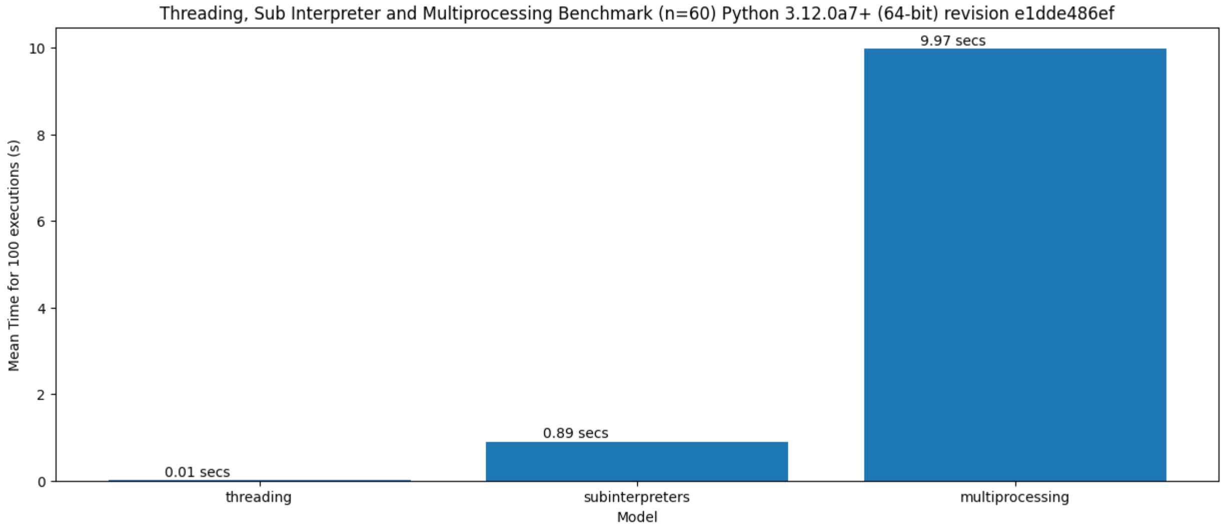
Dakos example in extras/! Serial, naïve async, async.

Sub-interpreters.

Different Python interpreters within the same process.

- Own GIL, shared memory.

Less time to spin up compared to multiprocessing [4]:



Released with 3.12. Still undergoing bug fixes. [1]

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.

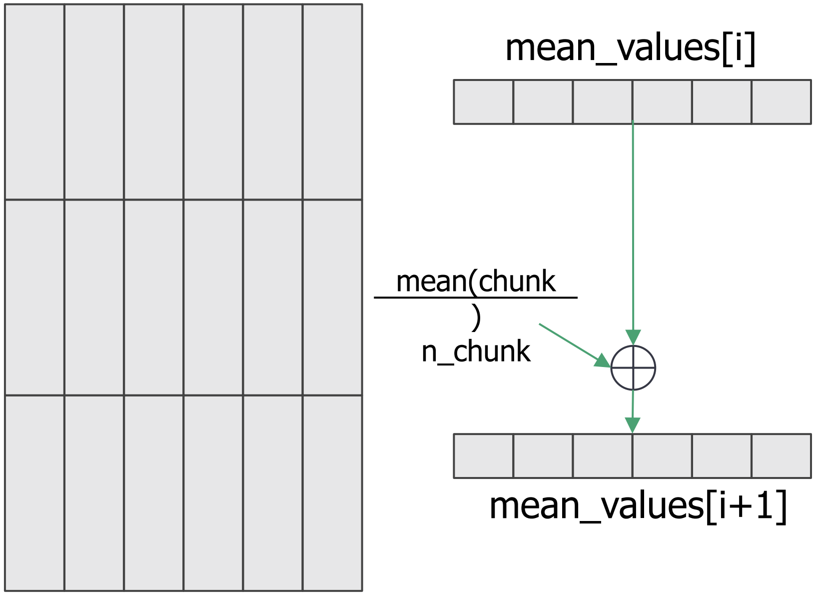
Discuss for a minute with your partner: what code could you write to get around this problem?

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
```



Not-embarrassingly parallel problems

It gets complicated. ☹

If I/O bound, perhaps `asyncio`.

CPU-bound...

- MPI: Message Passing Interface. De facto standard for low-level massive parallelization. https://github.com/mpi4py/mpi4py/
- Slides at end of deck.

Other Python packages for parallel computing

- https://www.dask.org/ (data objects [arrays, dataframes] for scaled computation, from laptop to cluster)
- https://snakemake.readthedocs.io/en/stable/index.html (tool to create reproducible and scalable analyses)
- 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)

On the cluster at your institute

If a parallel Python workflow is available, use it.

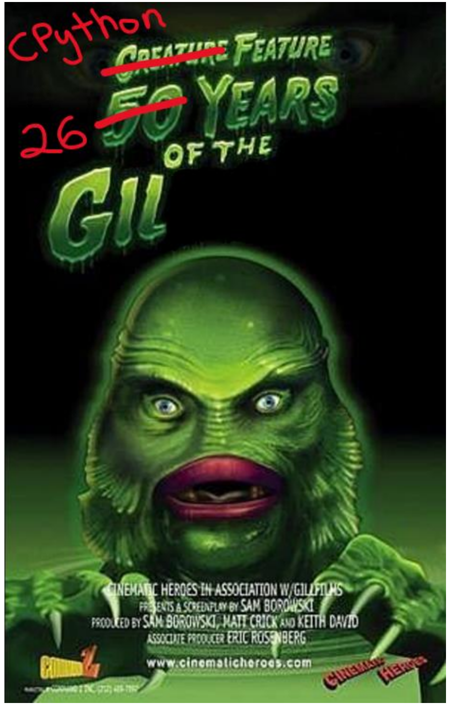
If no workflow is available, be ready for challenges.

- Efficacy of certain packages is cluster-dependent.

Wrapping up

Outline

- Processes, threads and THE GIL
- Hands-on investigations of embarrassingly parallel problems
    - A. Multithreading with NumPy
    - B. The `multiprocessing` package
    - C. Blending processes and threads
- Going further
- Wrap-up



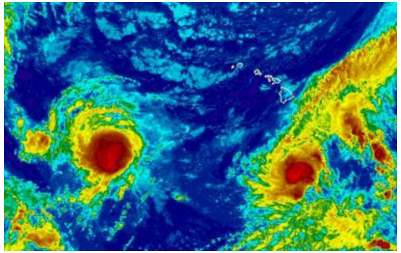
Thanks!

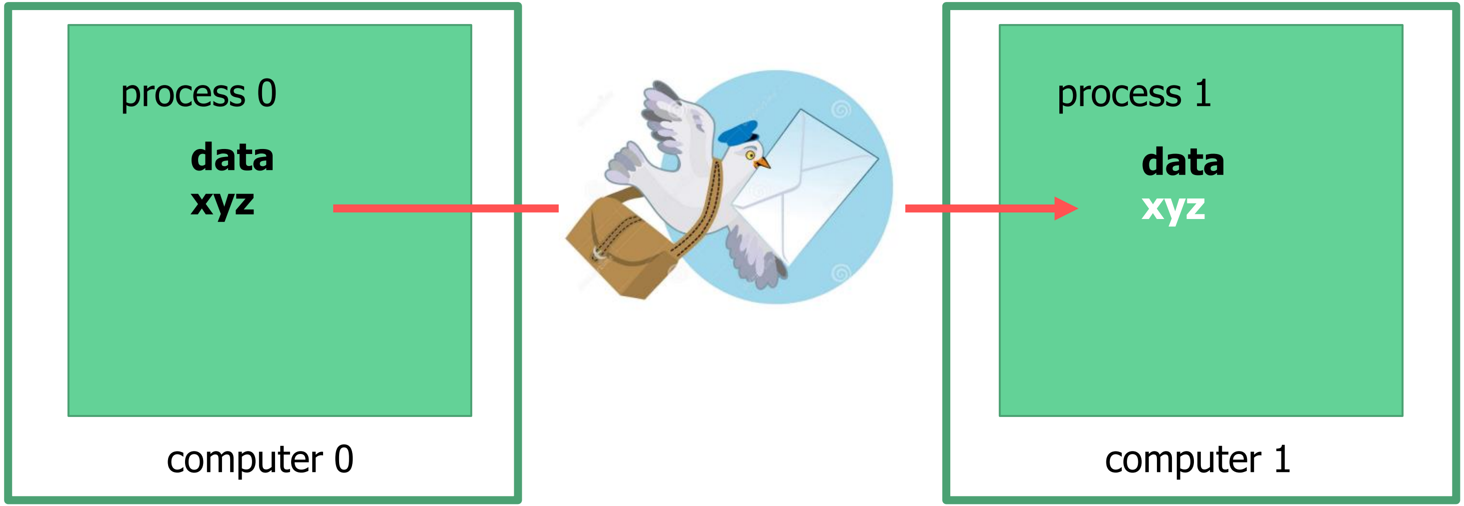
Supplementary material

Further reading

1. 4 concurrency in Python and subinterpreters Python subinterpreters and free-threading [LWN.net]
2. Concurrency in Python. Note that code snippets are a little outdated. Async Python: The Different Forms of Concurrency · Abu Ashraf Masnun
3. Example of webscraping with asyncio Asynchronous Web Scraping in Python [2024] - ZenRows
4. Sub-interpreters in 3.12 https://tonybaloney.github.io/posts/sub-interpreter-web-workers.html

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.

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)