Parallel Computing in Python

Current State and Recent Advances

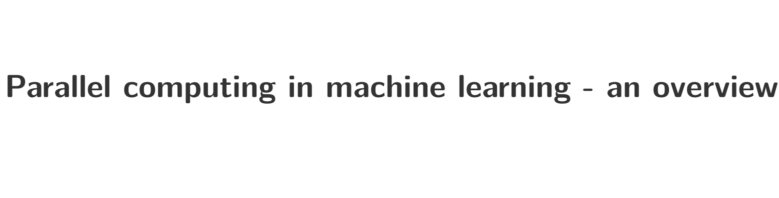
Pierre Glaser, INRIA

July 12, 2019

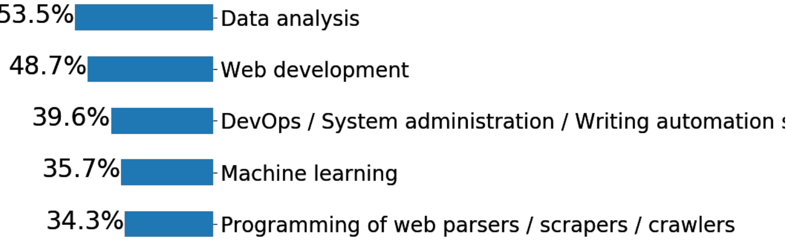
Parallel computing in machine learning - an overview

Built-in and Third Party multiprocessing ressources

Optimizing data communication


Parallel computing in machine learning - an overview

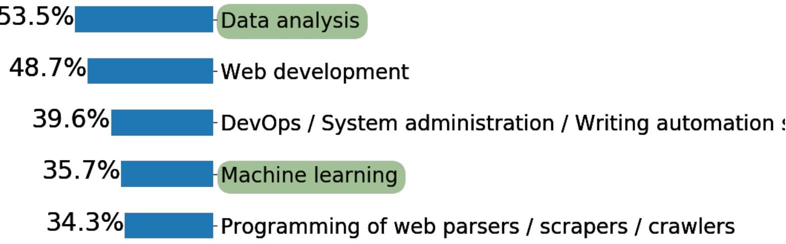
Python? What for?



Python usage among developers [1]

[1] source: https://www.jetbrains.com/research/python-developers-survey-2018/

Python? What for?



Python usage among developers [1]

[1] source: https://www.jetbrains.com/research/python-developers-survey-2018/

A growing data science ecosystem

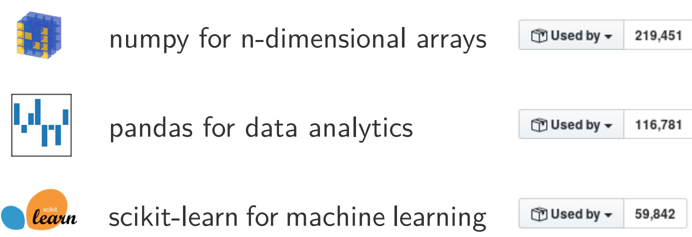
numpy for n-dimensional arrays

A growing data science ecosystem

numpy for n-dimensional arrays

pandas for data analytics

A growing data science ecosystem



Parallel computing? Why?

Independent, similar computation happens a lot in machine learning. We call it embarassingly parallel tasks. Famous examples:

Parallel computing? Why?

Independent, similar computation happens a lot in machine learning. We call it embarassingly parallel tasks. Famous examples:

- cross validation
- random forests
- hyperparameter selection using grid search

Happens for many scikit-learn estimators, but not all.

Parallel computing in scikit-learn made easy

Parallelization in `scikit-learn`:

- ubiquitous
- painlessly toggled using estimators's `n_jobs` option:

Parallel computing in scikit-learn made easy

Parallelization in `scikit-learn`:

- ubiquitous
- painlessly toggled using estimators's `n_jobs` option:

```
clf = RandomForestClassifier(n_estimators=100, n_jobs=4)
X, y = get_data()
clf.fit(X, y)  # runs on 4 cores!
```

Multithreading or multiprocessing?

Parallelism usually exist under two different forms:

Multithreading or multiprocessing?

Parallelism usually exist under two different forms:

- executing multiple processes (python programs) in parallel
- executing multiple threads of a same process in parallel

Fitting a random forest

I want to fit many small decision trees independently, on the same data.

Fitting a random forest

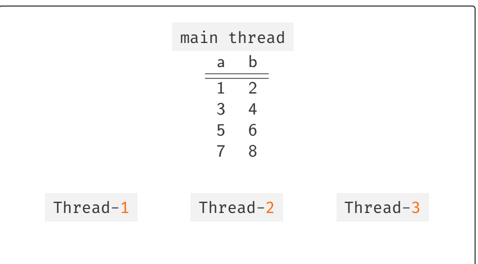
I want to fit many small decision trees independently, on the same data.

thread-based parallelism:

+: threads share memory

- no data copies
- no data transfer

-: By default, Python forces threads to run sequentially
The GIL can be released when calling native code (`numpy`, `scipy`...)



Fitting a random forest

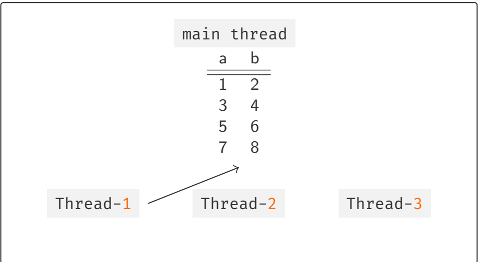
I want to fit many small decision trees independently, on the same data.

thread-based parallelism:

+: threads share memory

- no data copies
- no data transfer

-: By default, Python forces threads to run sequentially
The GIL can be released when calling native code (`numpy`, `scipy` ...)



Fitting a random forest

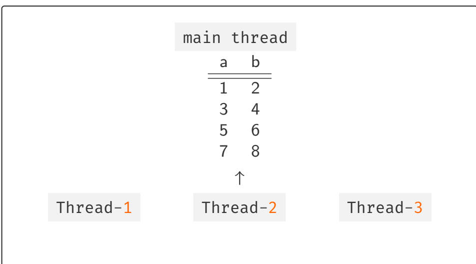
I want to fit many small decision trees independently, on the same data.

thread-based parallelism:

+: threads share memory

- no data copies
- no data transfer

-: By default, Python forces threads to run sequentially
The GIL can be released when calling native code (`numpy`, `scipy` ...)



Fitting a random forest

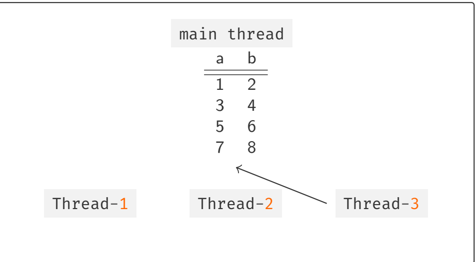
I want to fit many small decision trees independently, on the same data.

thread-based parallelism:

+: threads share memory

- no data copies
- no data transfer

-: By default, Python forces threads to run sequentially
The GIL can be released when calling native code (`numpy`, `scipy` ...)



Fitting a random forest

I want to fit many small decision trees independently, on the same data.

process-based parallelism:

+: processes are assured to run in parallel

-: need to pass and copy data around

- larger memory footprint
- data transfer overhead

main process

a	b
1	2
3	4
5	6
7	8

Process-1 Process-2 Process-3

: single python interpreter

Fitting a random forest

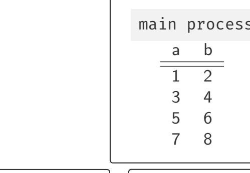
I want to fit many small decision trees independently, on the same data.

process-based parallelism:

+: processes are assured to run in parallel

-: need to pass and copy data around

- larger memory footprint
- data transfer overhead



Process-1

a	b
1	2
3	4
5	6
7	8

Process-2

a	b
1	2
3	4
5	6
7	8

Process-3

a	b
1	2
3	4
5	6
7	8

: single python interpreter

Fitting a random forest

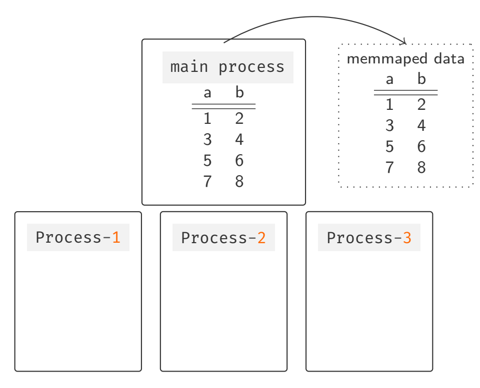
I want to fit many small decision trees independently, on the same data.

process-based parallelism:

+: processes are assured to run in parallel

-: need to pass and copy data around

- larger memory footprint
- data transfer overhead



In practice

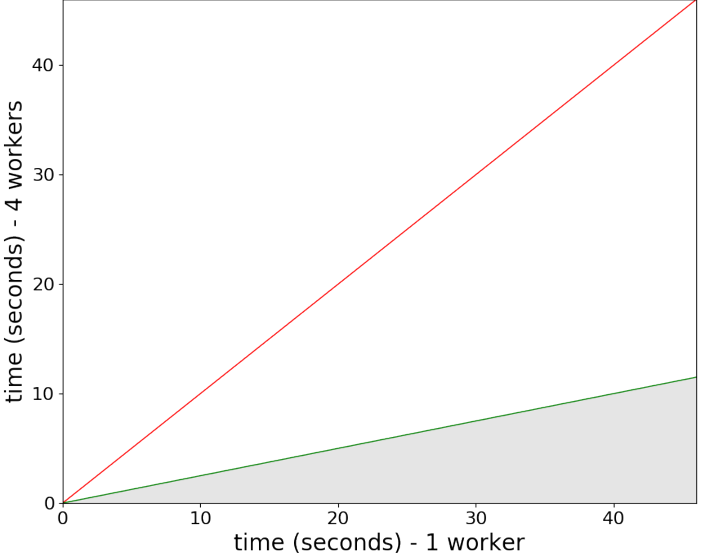


Figure: Model fitting time - parallel vs. sequential

In practice

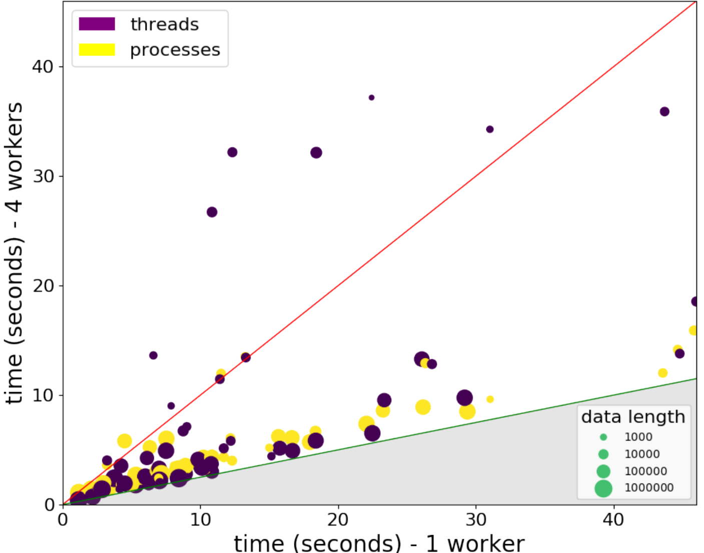


Figure: Model fitting time - parallel vs. sequential

In practice

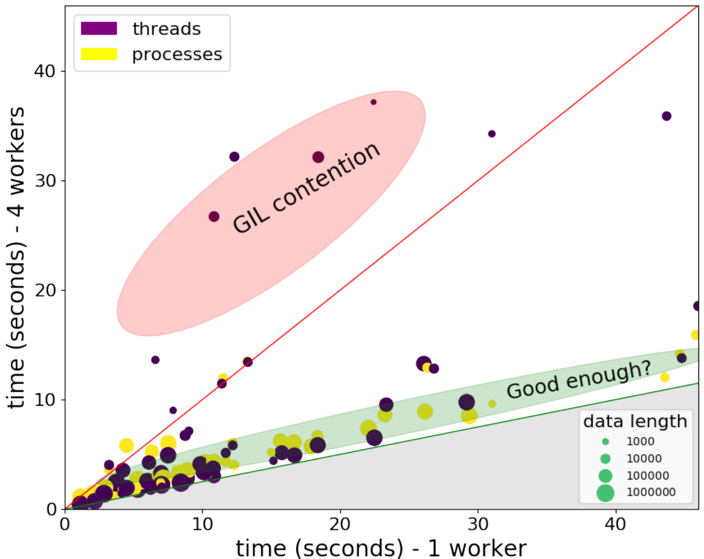


Figure: Model fitting time - parallel vs. sequential

Built-in and Third Party multiprocessing ressources

`multiprocessing`

provides all necessary constructs for creating and handling proccesses in `Python` programs

`multiprocessing`

provides all necessary constructs for creating and handling proccesses in `Python` programs

- creation

main process

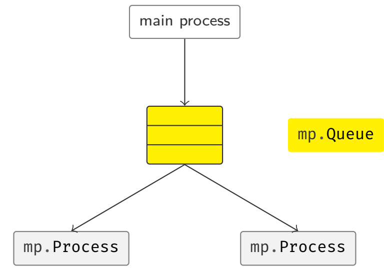
`mp.Process`

`mp.Process`

`multiprocessing`

provides all necessary constructs for creating and handling proccesses in `Python` programs

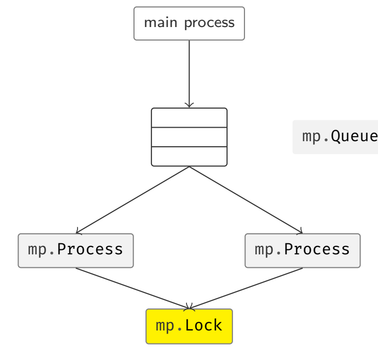
- creation
- communication



multiprocessing

provides all necessary constructs for creating and handling proccesses in `Python` programs

- creation
- communication
- synchronization



It's a very rich library!

multiprocessing

Programs executing embarassingly parallel tasks share a common multiprocessing strucure:

```python
import multiprocessing as mp

# Q: how can i parallelize this effortlessly?
results = map(greet, ["alice", "bob"])
```

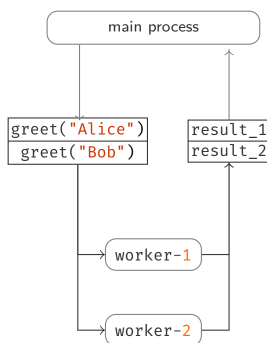
multiprocessing

Programs executing embarassingly parallel tasks share a common multiprocessing strucure:

```python
import multiprocessing as mp

# Q: how can i parallelize this effortlessly?
results = map(greet, ["alice", "bob"])

# A: worker pools
pool = mp.Pool(2)
results = pool.map(greet, ["Alice", "Bob"])
```



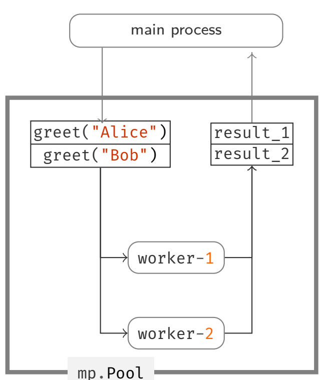
multiprocessing

Programs executing embarassingly parallel tasks share a common multiprocessing strucure:

```python
import multiprocessing as mp

# Q: how can i parallelize this effortlessly?
results = map(greet, ["alice", "bob"])

# A: worker pools
pool = mp.Pool(2)
results = pool.map(greet, ["Alice", "Bob"])
```



This structure is abstracted away in the `mp.Pool` class

`multiprocessing` portability

`multiprocessing` portability

`Windows` (start method; spawn) causes issues with interactive sessions (`IPython, Jupyter`)

```
>>> import multiprocessing as mp
>>> mp.set_start_method("spawn")
>>> p = mp.Pool(2)
>>> def greet_friend(name):
...     print("hello {}!".format(name))
...
>>> p.map(greet_friend, ("Alice", "Bob"))
```

```
AttributeError: Cant get attribute 'greet_friend' on <module '__main__'>
```

`multiprocessing` portability

- ❌ `Windows` (start method; spawn) causes issues with interactive sessions (`IPython, Jupyter`)

```
>>> import multiprocessing as mp
>>> mp.set_start_method("spawn")
>>> p = mp.Pool(2)
>>> def greet_friend(name):
...     print("hello {}!".format(name))
...
>>> p.map(greet_friend, ("Alice", "Bob"))
```

```
AttributeError: Cant get attribute 'greet_friend' on <module '__main__'>
```

- ❌ posix (start method; fork) causes crash with external libraries (GNU `openmp`)

`multiprocessing` portability

- ✗ `Windows` (start method; spawn) causes issues with interactive sessions (`IPython, Jupyter`)

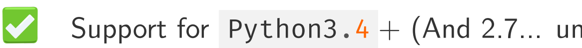
```
>>> import multiprocessing as mp
>>> mp.set_start_method("spawn")
>>> p = mp.Pool(2)
>>> def greet_friend(name):
...     print("hello {}!".format(name))
...
>>> p.map(greet_friend, ("Alice", "Bob"))
```

```
AttributeError: Cant get attribute 'greet_friend' on <module '__main__'>
```

- ✗ posix (start method; fork) causes crash with external libraries (GNU `openmp`)
- ✗ recovering from child processes crashes

loky

`loky` is a third party package, that provides a more robust process pool implementation.



- ✅ Support for `Python3.4`+ (And 2.7... until next year)
- ✅ Consistent behavior on all 🐧 , and
- 📈 Works in interactive shells

It is also the default backend of `scikit-learn`

`loky` - API

`concurrent futures` and `loky` **only expose (the same) worker pool** objects

`loky` - API

`concurrent futures` and `loky` **only expose (the same) worker pool** objects

using `concurent.futures`

```
>>> from concurrent.futures import ProcessPoolExecutor
>>> executor = ProcessPoolExecutor(max_workers=2)
>>> def greet_friend(name):
...     return "hello {}!".format(name)
...
>>> results = executor.map(greet_friend, ("Alice", "Bob"))  # non-blocking
>>> for r in results:  # blocking until the next task completes.
...     print(r)
```

`loky` - API

`concurrent futures` and `loky` **only expose (the same) worker pool** objects

using `loky`

```
>>> from loky import ProcessPoolExecutor
>>> executor = ProcessPoolExecutor(max_workers=2)
>>> def greet_friend(name):
...     return "hello {}!".format(name)
...
>>> results = executor.map(greet_friend, ("Alice", "Bob"))  # non-blocking
>>> for r in results:  # blocking until the next task completes.
...     print(r)
```

`joblib`

`joblib` is a parallel computing library built on top of of `loky`. It provides many useful features and optimizations for data scientists, including:

`joblib`

`joblib` is a parallel computing library built on top of of `loky`. It provides many useful features and optimizations for data scientists, including:

disk-based memoization of expensive computations

```
>>> @memory.cache
... def f(x):
...     print('Running f(%s)' % x)
...     return x
>>> print(f(1))  # computes f(1), dumps the result to disk
Running f(1)
1
>>> print(f(1))  # does not re-run f, simply grabs the result from the disk
1
```

`joblib`

`joblib` is a parallel computing library built on top of of `loky`. It provides many useful features and optimizations for data scientists, including:

disk-based memoization of expensive computations

optimized transfer of `numpy` arrays

`joblib`

`joblib` is a parallel computing library built on top of of `loky`. It provides many useful features and optimizations for data scientists, including:

disk-based memoization of expensive computations

optimized transfer of `numpy` arrays

a backend-agnostic user API

```python
with parallel_backend("loky", n_jobs=2):
    do_stuff_in_parallel()
```

`joblib`

`joblib` is a parallel computing library built on top of of `loky`. It provides many useful features and optimizations for data scientists, including:

disk-based memoization of expensive computations

optimized transfer of `numpy` arrays

a backend-agnostic user API

```
with parallel_backend("threading", n_jobs=2):
    do_stuff_in_parallel()
```

The challenges of multiprocessing (and beyond)

Improvements in python multiprocessing mostly concern:

speed (of data communication)

memory footprint (of duplicated data)

ease of use, robustnews(deadlocks)

Optimizing data transfer

disclaimer

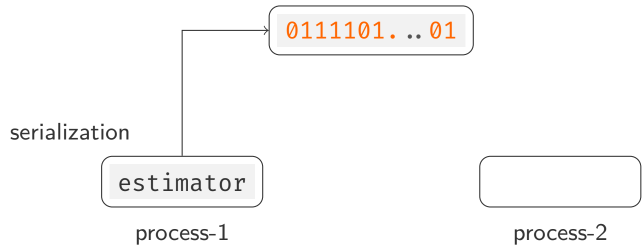
The optimizations mentionned now are `CPython` specific.

Serialization

Serialization defines the process of transforming an in-memory object into a sequence of bytes.

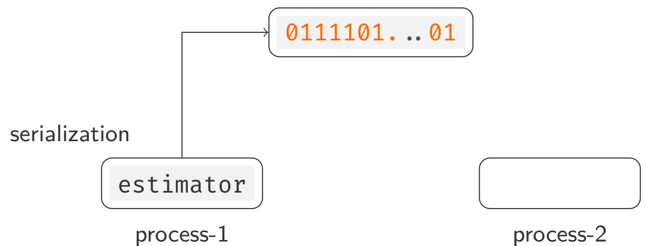
Serialization

Serialization defines the process of transforming an in-memory object into a sequence of bytes.



Serialization

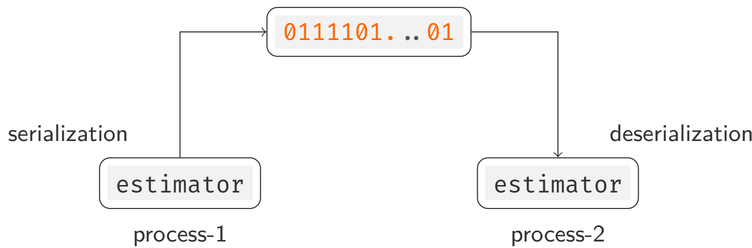
Serialization defines the process of transforming an in-memory object into a sequence of bytes.



The bytes string contains the instructions sequence that has to be executed to reconstruct the graph of objects in a fresh python environment

Serialization

Serialization defines the process of transforming an in-memory object into a sequence of bytes.



The bytes string contains the instructions sequence that has to be executed to reconstruct the graph of objects in a fresh python environment

the `pickle` protocol

`Python` defines a serialization protocol called `pickle`, and provides an implementation of it in the standard library.

```
>>> import pickle
>>> s = pickle.dumps([1, 2, 3])  # serialization (pickling) step
>>> s
b'\x80\x03]q\x00(K\x01K\x02K\x03e.'
```

```
>>> depickled_list = pickle.loads(b'\x80\x03]q\x00(K\x01K\x02K\x03e.')
>>> depickled_list
[1, 2, 3]
```

`pickle` extensions

- by design, the pickle implementation blocks the serialization of some `Python` constructs

`pickle` extensions

by design, the pickle implementation blocks the serialization of some `Python` constructs

```
>>> import pickle
>>> import cloudpickle
>>> pickle.dumps(lambda x: x + 1)  # would cause arbitrary code execution
```

`pickle` extensions

by design, the pickle implementation blocks the serialization of some `Python` constructs

```
>>> import pickle
>>> import cloudpickle
>>> pickle.dumps(lambda x: x + 1)  # would cause arbitrary code execution
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
_pickle.PicklingError: Can't pickle <function <lambda> at 0x7fd0b36631e0>:
attribute lookup <lambda> on __main__ failed
```

pickle extensions

by design, the pickle implementation blocks the serialization of some Python constructs

```
>>> import pickle
>>> import cloudpickle
>>> pickle.dumps(lambda x: x + 1)  # would cause arbitrary code execution
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
_pickle.PicklingError: Can't pickle <function <lambda> at 0x7fd0b36631e0>:
attribute lookup <lambda> on __main__ failed
```

In practice, data scientists need remote code execution of interactively defined functions (jupyter + dask , Zeppelin + (py)spark ...)
Such frameworks require pickle extensions such as cloudpickle

```
>>> cloudpickle.dumps(lambda x: x + 1)
b'\x80\x04\x958\x01\x00\x00\x00\x00\x00\x00\x8c\x17cloudpickle.cloudpickle ... '
```

`pickle` extensions (2)

The `pickle` module is implemented both as a pure `Python` module, and as a `C`-optimized module.

`pickle` extensions (2)

The `pickle` module is implemented both as a pure `Python` module, and as a `C`-optimized module.

`pickle` extensions however could only extend the slow pythonic `pickle`

`pickle` extensions (2)

The `pickle` module is implemented both as a pure `Python` module, and as a `C`-optimized module.

`pickle` extensions however could only extend the slow pythonic `pickle`

```
$python3.7 -m timeit  'import pickle; pickle.dumps(list(range(100000)))'
50 loops, best of 5: 4.39 msec per loop
$python3.7 -m timeit  'import cloudpickle; cloudpickle.dumps(list(range(100000)))'
2 loops, best of 5: 119 msec per loop
```

extending the C-optimized `pickle`

in `Python 3.8`, `pickle` extensions can now extend the C-optimized `pickle` module [2]

[2] joint work with ogrisel and Antoine Pitrou

extending the C-optimized `pickle`

in `Python 3.8`, `pickle` extensions can now extend the C-optimized `pickle` module [2]

```
$python3.8 -m timeit  'import pickle;pickle.dumps(list(range(100000)))'
50 loops, best of 5: 4.69 msec per loop
$python3.8 -m timeit  'import cloudpickle;cloudpickle.dumps(list(range(100000)))'
100 loops, best of 5: 3.73 msec per loop
```

[2] joint work with ogrisel and Antoine Pitrou

extending the C-optimized `pickle`

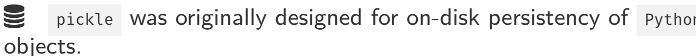
in `Python 3.8`, `pickle` extensions can now extend the C-optimized `pickle` module [2]

```
$python3.8 -m timeit  'import pickle;pickle.dumps(list(range(100000)))'
50 loops, best of 5: 4.69 msec per loop
$python3.8 -m timeit  'import cloudpickle;cloudpickle.dumps(list(range(100000)))'
100 loops, best of 5: 3.73 msec per loop
```

30x faster than on python3.7!

[2] joint work with ogrisel and Antoine Pitrou

`pickle` protocol 5



`pickle` was originally designed for on-disk persistency of `Python` objects.

`pickle` protocol 5

`pickle` was originally designed for on-disk persistency of `Python` objects.

Now, it is used wildely to communicate objects between workers, which is done in-memory. **RAM usage becomes a critical concern.**

`pickle` protocol 5

`pickle` was originally designed for on-disk persistency of `Python` objects.

Now, it is used wildely to communicate objects between workers, which is done in-memory. **RAM usage becomes a critical concern.**

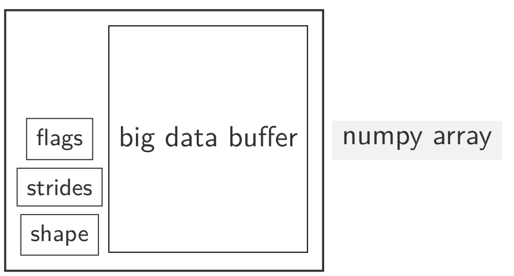
`pickle` protocol 5 [a] addition: `PickleBuffer`

ensures no copy operations when dumping or loading objects with large `numpy` arrays and `Arrow` tables. (pandas `DataFrame`, `scikit-learn` estimators...)

[a] work by Antoine Pitrou

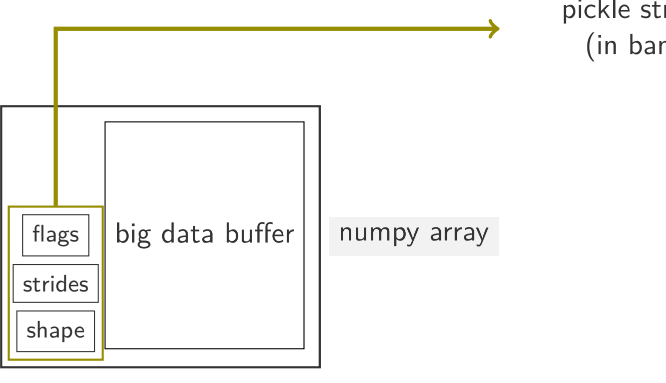
out-of-band serialization

`pickle` protocol 5 goes even one step further: It allows delegation of `PEP 3118`-compatible objects serialization to third-party code.



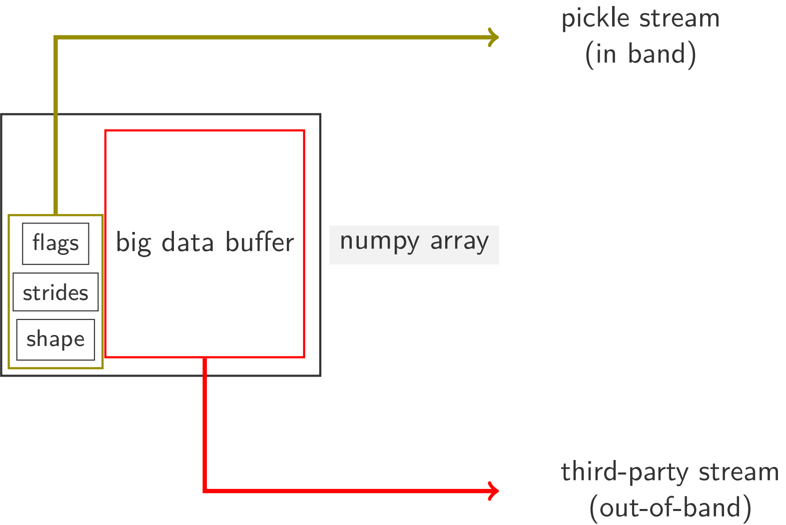
out-of-band serialization

`pickle` protocol 5 goes even one step further: It allows delegation of `PEP 3118`-compatible objects serialization to third-party code.



out-of-band serialization

`pickle` protocol 5 goes even one step further: It allows delegation of `PEP 3118`-compatible objects serialization to third-party code.



Conclusion

parallel computing often generates significant speedups when executing machine learning code

Conclusion

parallel computing often generates significant speedups when executing machine learning code

which backend (processes vs. threads) to use can be a problem-specific question: what's the size of your data, does your compute-hungry code releases the GIL...

Conclusion

parallel computing often generates significant speedups when executing machine learning code

which backend (processes vs. threads) to use can be a problem-specific question: what's the size of your data, does your compute-hungry code releases the GIL...

⬆ **working with upstream is worth the hassle**

Questions?