Robustifying `concurrent.futures`

Thomas Moreau - Olivier Grisel

Embarassingly parallel computation in `python` using a pool of workers

Three API available:

- `multiprocessing`: first implementation.
- `concurrent.futures`: reimplementation using `multiprocessing` under the hood.
- `loky`: robustification of `concurrent.futures`.

The `concurrent.futures` API

The `Executor`: a worker pool

```python
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model
```

`fit_model` is the function that we want to run asynchronously.

The `Executor`: a worker pool

```python
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:
```

`fit_model` is the function that we want to run asynchronously.

We instanciate a `ThreadPoolExecutor` with 4 threads.

The `Executor`: a worker pool

```python
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:

    # Submit an asynchronous job and return a Future
    future1 = executor.submit(fit_model, param1)
```

`fit_model` is the function that we want to run asynchronously.

We instanciate a `ThreadPoolExecutor` with 4 threads.

A new job is submitted to the `Executor`.
The `Future` object `future1` holds the state of the computation.

The `Executor`: a worker pool

```python
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:

    # Submit an asynchronous job and return a Future
    future1 = executor.submit(fit_model, param1)

    # Submit other job
    future2 = executor.submit(fit_model, param2)
```

`fit_model` is the function that we want to run asynchronously.

We instanciate a `ThreadPoolExecutor` with 4 threads.

A new job is submitted to the `Executor`.
The `Future` object `future1` holds the state of the computation.

The `Executor`: a worker pool

```
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:

    # Submit an asynchronous job and return a Future
    future1 = executor.submit(fit_model, param1)

    # Submit other job
    future2 = executor.submit(fit_model, param2)

    # Run other computation
    ...
```

`fit_model` is the function that we want to run asynchronously.

We instanciate a `ThreadPoolExecutor` with 4 threads.

A new job is submitted to the `Executor`.
The `Future` object `future1` holds the state of the computation.

The `Executor`: a worker pool

```
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:

    # Submit an asynchronous job and return a Future
    future1 = executor.submit(fit_model, param1)

    # Submit other job
    future2 = executor.submit(fit_model, param2)

    # Run other computation
    ...

    # Blocking call, wait and return the result
    model1 = future1.result(timeout=None)
    model2 = future2.result(timeout=None)
```

`fit_model` is the function that we want to run asynchronously.

We instanciate a `ThreadPoolExecutor` with 4 threads.

A new job is submitted to the `Executor`.
The `Future` object `future1` holds the state of the computation.

Wait for the computation to end and return the result with `f.result`.

The `Executor`: a worker pool

```python
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:

    # Submit an asynchronous job and return a Future
    future1 = executor.submit(fit_model, param1)

    # Submit other job
    future2 = executor.submit(fit_model, param2)

    # Run other computation
    ...

    # Blocking call, wait and return the result
    model1 = future1.result(timeout=None)
    model2 = future2.result(timeout=None)

# The ressources have been cleaned up
print(model1, model2)
```

`fit_model` is the function that we want to run asynchronously.

We instanciate a `ThreadPoolExecutor` with 4 threads.

A new job is submitted to the `Executor`.
The `Future` object `future1` holds the state of the computation.

Wait for the computation to end and return the result with `f.result`.

The ressources are cleaned up.

The `Executor`: a worker pool

```python
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:

    # Submit an asynchronous job and return a Future
    future1 = executor.submit(fit_model, param1)

    # Submit other job
    future2 = executor.submit(fit_model, param2)

    # Run other computation
    ...

    # Blocking call, wait and return the result
    model1 = future1.result(timeout=None)
    model2 = future2.result(timeout=None)

# The ressources have been cleaned up
print(model1, model2)
```

`fit_model` is the function that we want to run asynchronously.

We instanciate a `ThreadPoolExecutor` with 4 threads.

A new job is submitted to the `Executor`.
The `Future` object `future1` holds the state of the computation.

Wait for the computation to end and return the result with `f.result`.

The ressources are cleaned up.

Submitting more than one job returns an iterator: `executor.map`

The `Future` object: an asynchronous result state.

States

`Future` objects hold the state of the asynchronous computations, wich can be in one of 4 states: Not started, Running, Cancelled and Done

The state of a `Future` can be checked using `f.running, f.cancelled, f.done`.

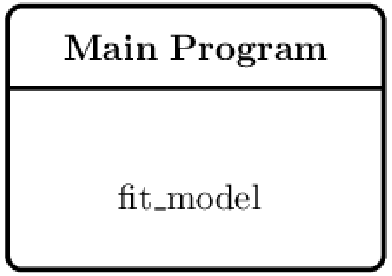
Blocking methods

- `f.result(timeout=None)`
- `f.exception(timeout=None)`

wait for computations to be done.

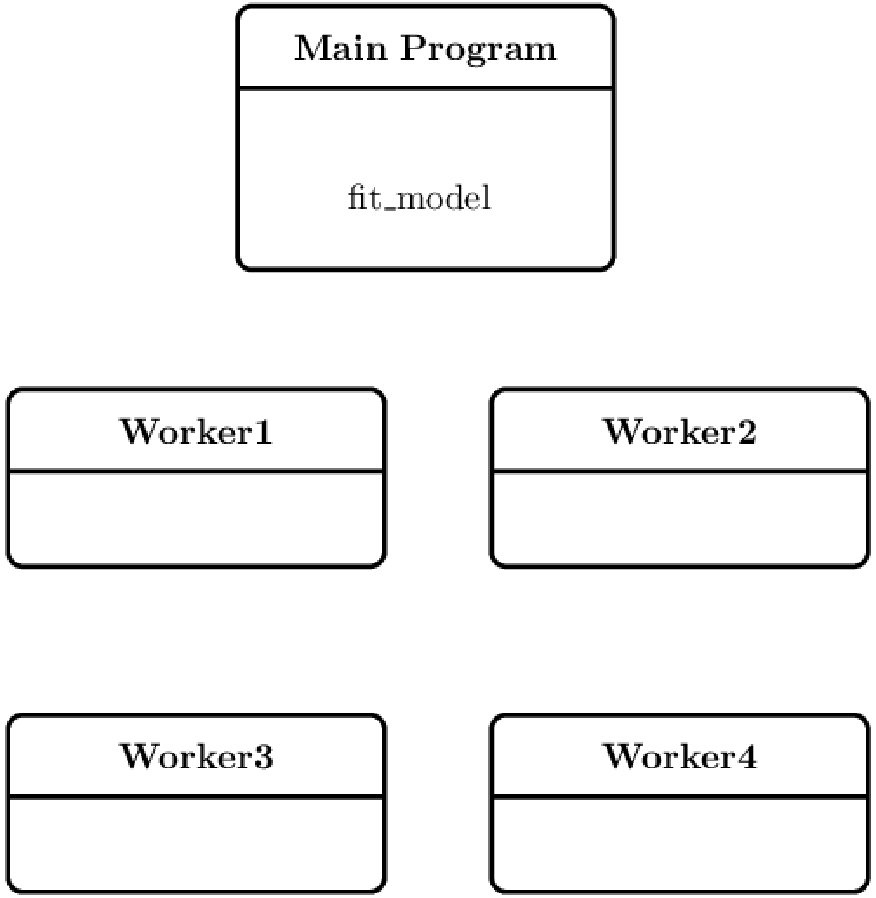
The `Executor`: a worker pool

```python
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:

    # Submit an asynchronous job and return a Future
    future1 = executor.submit(fit_model, param1)

    # Submit other job
    future2 = executor.submit(fit_model, param2)

    # Run other computation
    ...

    # Blocking call, wait and return the result
    model1 = future1.result(timeout=None)
    model2 = future2.result(timeout=None)

# The ressources have been cleaned up
print(model1, model2)
```



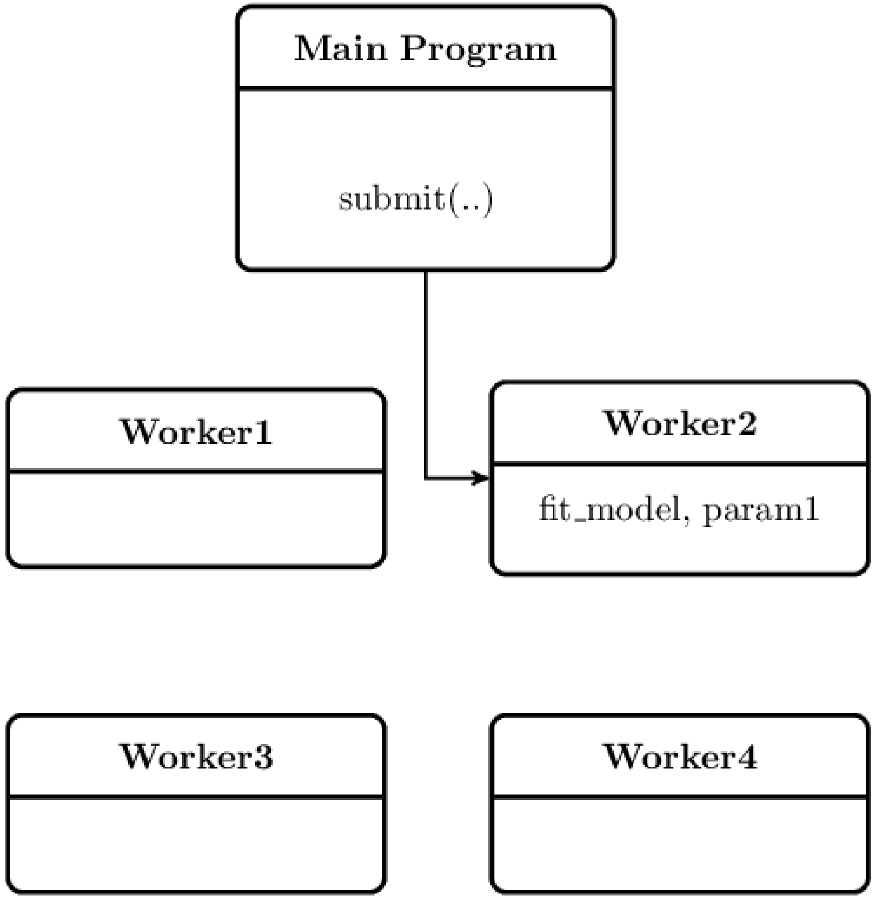
The `Executor`: a worker pool

```python
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:

    # Submit an asynchronous job and return a Future
    future1 = executor.submit(fit_model, param1)

    # Submit other job
    future2 = executor.submit(fit_model, param2)

    # Run other computation
    ...

    # Blocking call, wait and return the result
    model1 = future1.result(timeout=None)
    model2 = future2.result(timeout=None)

# The ressources have been cleaned up
print(model1, model2)
```



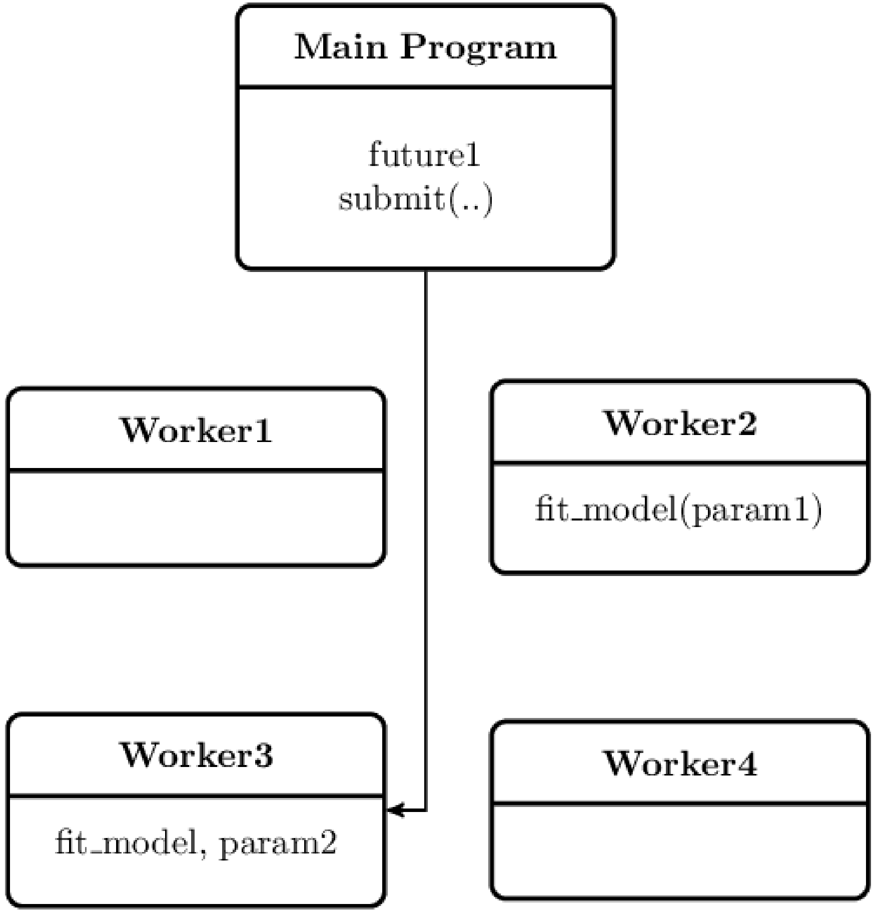
The `Executor`: a worker pool

```python
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:

    # Submit an asynchronous job and return a Future
    future1 = executor.submit(fit_model, param1)

    # Submit other job
    future2 = executor.submit(fit_model, param2)

    # Run other computation
    ...

    # Blocking call, wait and return the result
    model1 = future1.result(timeout=None)
    model2 = future2.result(timeout=None)

# The ressources have been cleaned up
print(model1, model2)
```



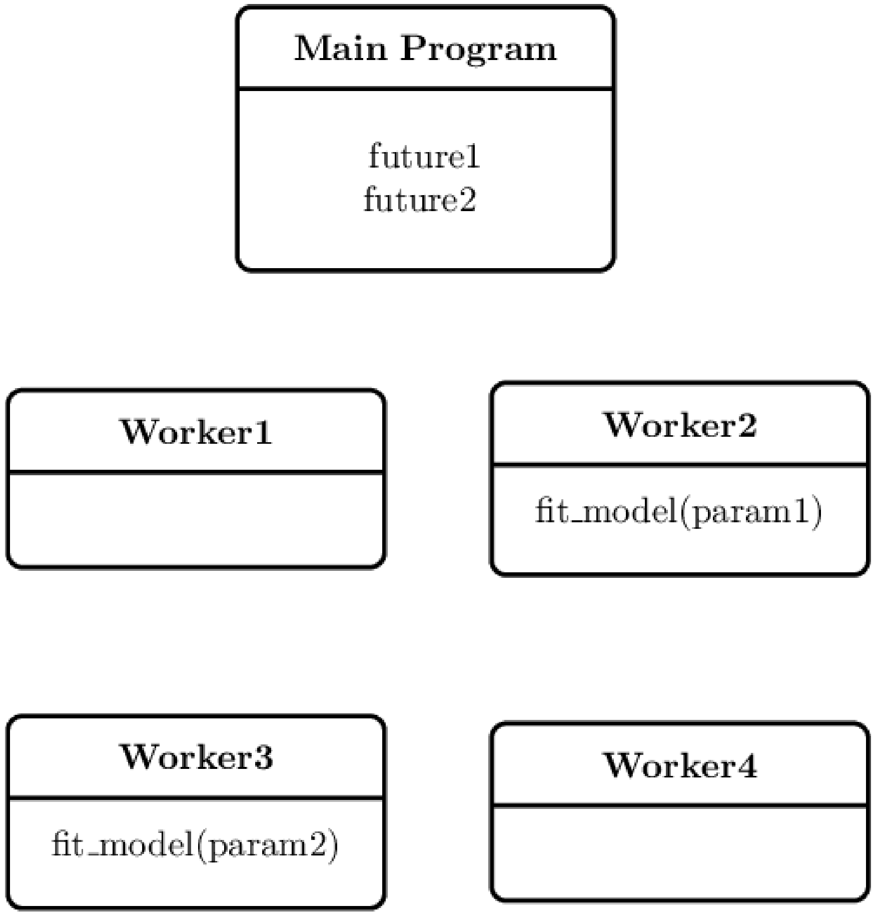
The `Executor`: a worker pool

```python
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:

    # Submit an asynchronous job and return a Future
    future1 = executor.submit(fit_model, param1)

    # Submit other job
    future2 = executor.submit(fit_model, param2)

    # Run other computation
    ...

    # Blocking call, wait and return the result
    model1 = future1.result(timeout=None)
    model2 = future2.result(timeout=None)

# The ressources have been cleaned up
print(model1, model2)
```



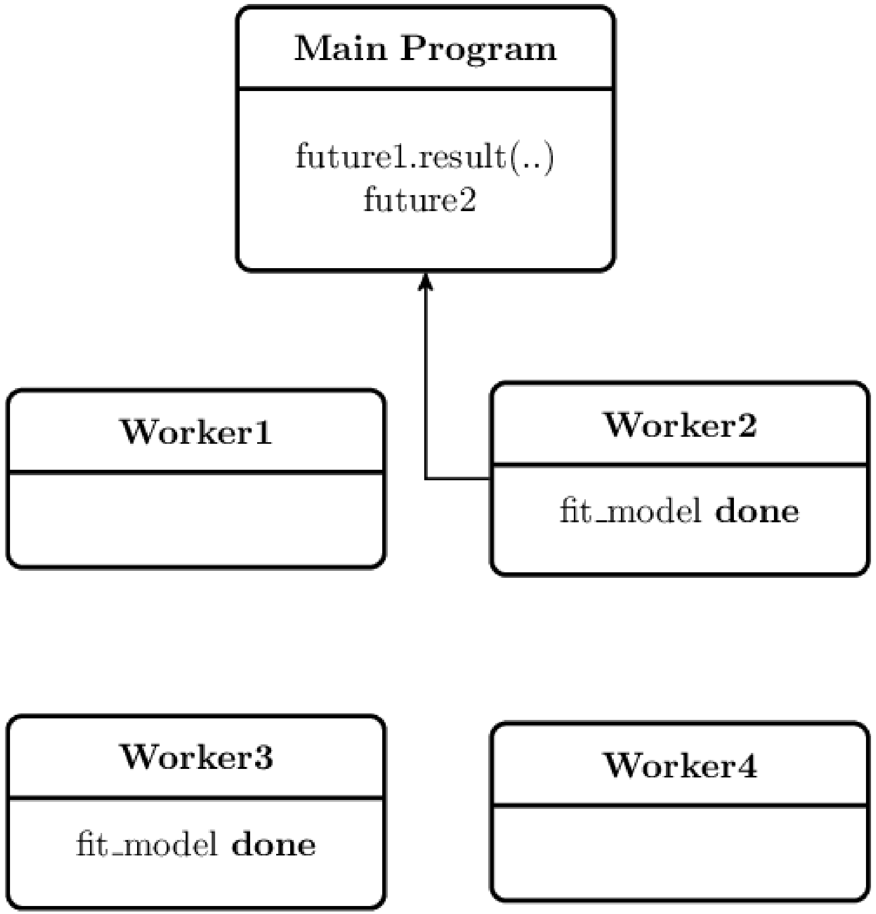
The `Executor`: a worker pool

```
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:

    # Submit an asynchronous job and return a Future
    future1 = executor.submit(fit_model, param1)

    # Submit other job
    future2 = executor.submit(fit_model, param2)

    # Run other computation
    ...

    # Blocking call, wait and return the result
    model1 = future1.result(timeout=None)
    model2 = future2.result(timeout=None)

# The ressources have been cleaned up
print(model1, model2)
```



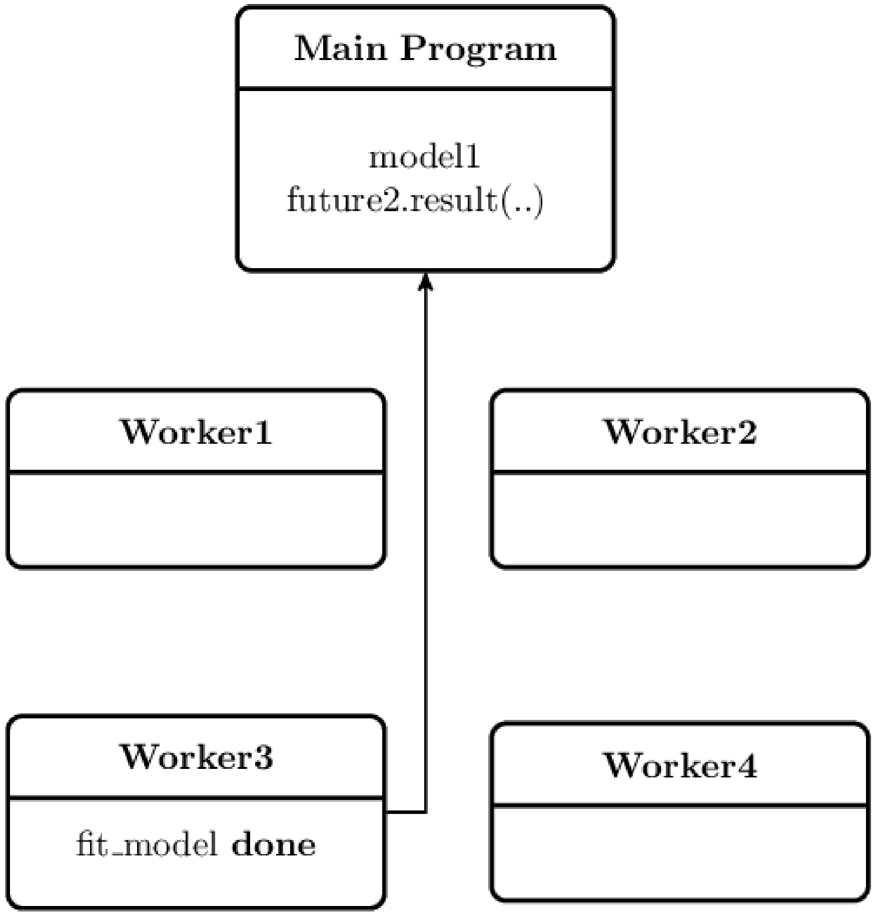
The `Executor`: a worker pool

```python
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:

    # Submit an asynchronous job and return a Future
    future1 = executor.submit(fit_model, param1)

    # Submit other job
    future2 = executor.submit(fit_model, param2)

    # Run other computation
    ...

    # Blocking call, wait and return the result
    model1 = future1.result(timeout=None)
    model2 = future2.result(timeout=None)

# The ressources have been cleaned up
print(model1, model2)
```



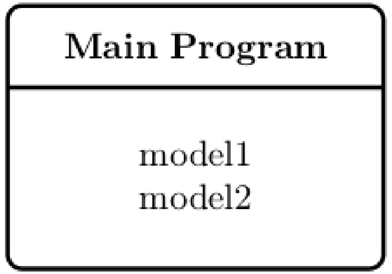
The `Executor`: a worker pool

```
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:

    # Submit an asynchronous job and return a Future
    future1 = executor.submit(fit_model, param1)

    # Submit other job
    future2 = executor.submit(fit_model, param2)

    # Run other computation
    ...

    # Blocking call, wait and return the result
    model1 = future1.result(timeout=None)
    model2 = future2.result(timeout=None)

# The ressources have been cleaned up
print(model1, model2)
```



The `Executor`: a worker pool

```python
from concurrent.futures import ThreadPoolExecutor

def fit_model(params):
    # Heavy computation
    return model

# Create an executor with 4 threads
with ThreadPoolExecutor(max_workers=4) as executor:

    # Submit an asynchronous job and return a Future
    future1 = executor.submit(fit_model, param1)

    # Submit other job
    future2 = executor.submit(fit_model, param2)

    # Run other computation
    ...

    # Blocking call, wait and return the result
    model1 = future1.result(timeout=None)
    model2 = future2.result(timeout=None)

# The ressources have been cleaned up
print(model1, model2)
```



Choosing the type of worker: `Thread` or `Process` ?

Running on multiple cores

Python GIL

The internal implementation of python interpreter relies on a "Global Interpreter Lock" **(GIL)**, protecting the concurrent access to python objects:

- Only one thread can acquire it.
- Not designed for efficient multicore computing.

Global lock everytime we access a python object.

Released when performing long I/O operations or by some libraries. (*e.g.* numpy, openMP,..)

Thread

- Real system thread:
  - pthread
  - windows thread
- All the computation are done with a **single** interpreter.

Advantages:

- Fast spawning
- Reduced memory overhead
- No communication overhead (shared python objects)

Thread

- Real system thread:
  - pthread
  - windows thread
- All the computation are done with a **single** interpreter.

Advantages:

- Fast spawning
- Reduced memory overhead
- No communication overhead (shared python objects)

Wait... shared python objects and a single interpreter?!?

Thread

- Real system thread:
  - pthread
  - windows thread
- All the computation are done with a **single** interpreter.

Advantages:

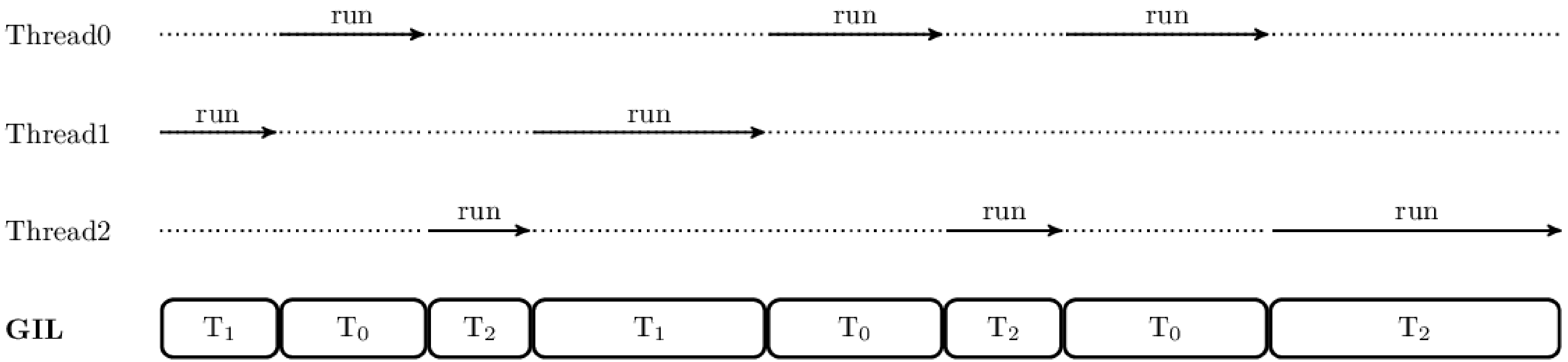
- Fast spawning
- Reduced memory overhead
- No communication overhead (shared python objects)

Wait... shared python objects and a single interpreter?!?

There is only one GIL!

Thread

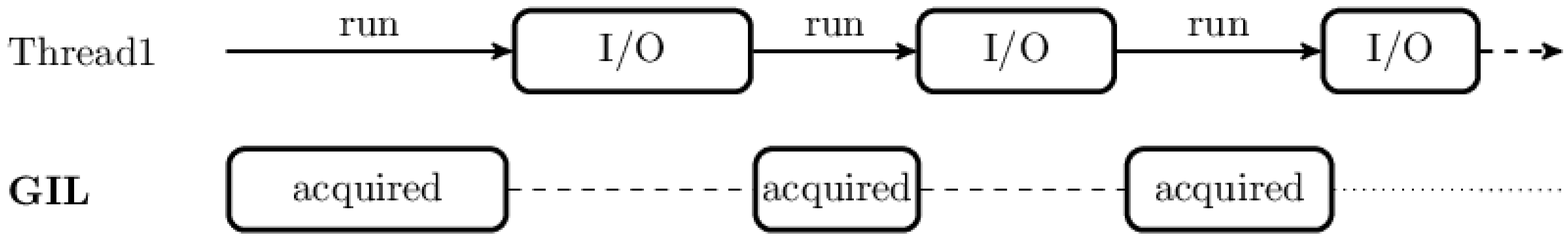
Multiple threads running python code:



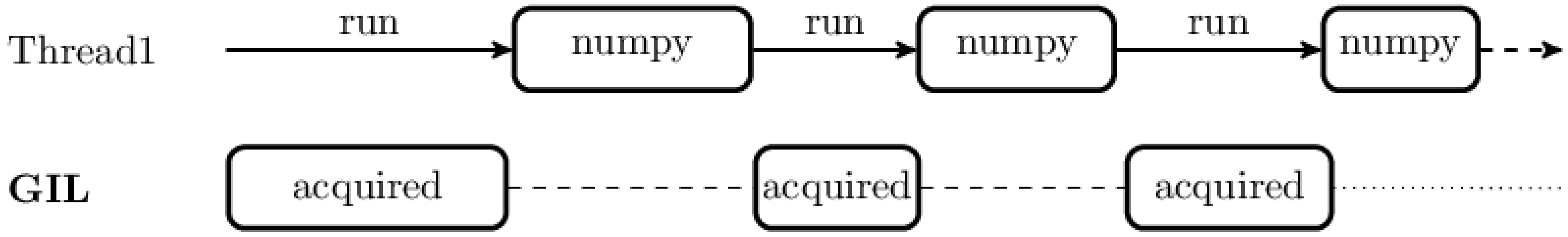
This is not quicker than sequential even on a multicore machine.

Thread

Threads hold the GIL when running python code.
They release it when blocking for I/O:



Or when using some c library:



Process

- Create a new python interpreter per worker.
- Each worker run in **its own** interpreter.

Inconvenients:

- *Slow* spawning
- Higher memory overhead
- Higher communication overhead.

Process

- Create a new python interpreter per worker.
- Each worker run in **its own** interpreter.

Inconvenients:

- *Slow* spawning
- Higher memory overhead
- Higher communication overhead.

But there is no GIL!
The computation can be done in parallel even for python code.

Process

- Create a new python interpreter per worker.
- Each worker run in **its own** interpreter.

Inconvenients:

- *Slow* spawning
- Higher memory overhead
- Higher communication overhead.

But there is no GIL!
The computation can be done in parallel even for python code.

Method to create a new interpreter: *fork* or *spawn*

Launching a new interpreter: *fork*

Duplicate the current interpreter. (Only available on UNIX)

Advantages:

- Low spawning overhead.
- The interpreter is warm *imported*.

Inconvenient:

- Bad interaction with multithreaded programs
- Does not respect the POSIX specifications

$\Rightarrow$ Some libraries crash: numpy on OSX, openMP, ...

Launching a new interpreter: *spawn*

Create a new interpreter from scratch.

Advantages:

- Safe (respect POSIX)
- Fresh interpreter without extra libraries.

Inconvenient:

- Slower to start.
- Need to reload the libraries, redefine the functions...

Comparison between `Thread` and `Process`

	Thread	**Process** (fork)	**Process** (spawn)
Efficient multicore run	✗	✓	✓
No communication overhead	✓	✗	✗
POSIX safe	✓	✗	✓
No spawning overhead	✓	✓	✗

Comparison between `Thread` and `Process`

	Thread	**Process** (fork)	**Process** (spawn)
Efficient multicore run	✗	✓	✓
No communication overhead	✓	✗	✗
POSIX safe	✓	✗	✓
No spawning overhead	✓	✓	Loky

$\Rightarrow$ Hide the spawning overhead by reusing the pool of processes.

Reusing a `ProcessPoolExecutor`.

Reusing a `ProcessPoolExecutor`.

To Avoid the spawning overhead, reuse a previously started `ProcessPoolExecutor`.

The spawning overhead is only paid once.

Easy using a global pool of process.
Main issue: is that robust?

Managing the state of the executor

Example deadlock

```
>>> from concurrent.futures import ProcessPoolExecutor
>>> with ProcessPoolExecutor(max_workers=4) as e:
...     e.submit(lambda: 1).result()
...
Traceback (most recent call last):
  File "/usr/lib/python3.6/multiprocessing/queues.py", line 241, in _feed
    obj = _ForkingPickler.dumps(obj)
  File "/usr/lib/python3.6/multiprocessing/reduction.py", line 51, in dumps
    cls(buf, protocol).dump(obj)
_pickle.PicklingError: Can't pickle <function <lambda> at 0x7fcff0184d08>:
attribute lookup <lambda> on __main__ failed

^C
```

It can be tricky to know which `submit` call crashed the `Executor`.

Managing the state of the executor

Example deadlock

```
>>> from concurrent.futures import ProcessPoolExecutor
>>> with ProcessPoolExecutor(max_workers=4) as e:
...     e.submit(lambda: 1)
...
Traceback (most recent call last):
  File "/usr/lib/python3.6/multiprocessing/queues.py", line 241, in _feed
    obj = _ForkingPickler.dumps(obj)
  File "/usr/lib/python3.6/multiprocessing/reduction.py", line 51, in dumps
    cls(buf, protocol).dump(obj)
_pickle.PicklingError: Can't pickle <function <lambda> at 0x7f5c787bd488>:
attribute lookup <lambda> on __main__ failed

^C
```

Even worse, shutdown itself is deadlocked.

Reusable pool of workers: `loky`.

loky : a robust pool of workers

```
>>> from loky import ProcessPoolExecutor
>>> class CrashAtPickle(object):
...     """Bad object that triggers a segfault at unpickling time."""
...     def __reduce__(self):
...         raise RuntimeError()
...
>>> with ProcessPoolExecutor(max_workers=4) as e:
...     e.submit(CrashAtPickle()).result()
...
Traceback (most recent call last):
...
RuntimeError
Traceback (most recent call last):
...
BrokenExecutor: The QueueFeederThread was terminated abruptly while feeding a
new job. This can be due to a job pickling error.
>>>
```

- Return and raise a user friendly exception.
- Fix some other deadlocks.

A reusable `ProcessPoolExecutor`

```
>>> from loky import get_reusable_executor
>>> excutor = get_reusable_executor(max_workers=4)
>>> print(excutor.executor_id)
0

>>> excutor.submit(id, 42).result()
139655595838272

>>> excutor = get_reusable_executor(max_workers=4)
>>> print(excutor.executor_id)
0

>>> excutor.submit(CrashAtUnpickle()).result()
Traceback (most recent call last):
...
BrokenExecutorError
>>> excutor = get_reusable_executor(max_workers=4)
>>> print(excutor.executor_id)
1
>>> excutor.submit(id, 42).result()
139655595838272
```

Create a `ProcessPoolExecutor` using the factory function `get_reusable_executor`.

A reusable `ProcessPoolExecutor`

```
>>> from loky import get_reusable_executor
>>> excutor = get_reusable_executor(max_workers=4)
>>> print(excutor.executor_id)
0

>>> excutor.submit(id, 42).result()
139655595838272

>>> excutor = get_reusable_executor(max_workers=4)
>>> print(excutor.executor_id)
0

>>> excutor.submit(CrashAtUnpickle()).result()
Traceback (most recent call last):
...
BrokenExecutorError
>>> excutor = get_reusable_executor(max_workers=4)
>>> print(excutor.executor_id)
1
>>> excutor.submit(id, 42).result()
139655595838272
```

Create a `ProcessPoolExecutor` using the factory function `get_reusable_executor`.

The executor can be used exactly as `ProcessPoolExecutor`.

A reusable `ProcessPoolExecutor`

```
>>> from loky import get_reusable_executor
>>> excutor = get_reusable_executor(max_workers=4)
>>> print(excutor.executor_id)
0

>>> excutor.submit(id, 42).result()
139655595838272

>>> excutor = get_reusable_executor(max_workers=4)
>>> print(excutor.executor_id)
0

>>> excutor.submit(CrashAtUnpickle()).result()
Traceback (most recent call last):
...
BrokenExecutorError
>>> excutor = get_reusable_executor(max_workers=4)
>>> print(excutor.executor_id)
1
>>> excutor.submit(id, 42).result()
139655595838272
```

Create a `ProcessPoolExecutor` using the factory function `get_reusable_executor`.

The executor can be used exactly as `ProcessPoolExecutor`.

When the factory is called elsewhere, reuse the same executor if it is working.

A reusable `ProcessPoolExecutor`

```
>>> from loky import get_reusable_executor
>>> excutor = get_reusable_executor(max_workers=4)
>>> print(excutor.executor_id)
0

>>> excutor.submit(id, 42).result()
139655595838272

>>> excutor = get_reusable_executor(max_workers=4)
>>> print(excutor.executor_id)
0

>>> excutor.submit(CrashAtUnpickle()).result()
Traceback (most recent call last):
...
BrokenExecutorError
>>> excutor = get_reusable_executor(max_workers=4)
>>> print(excutor.executor_id)
1
>>> excutor.submit(id, 42).result()
139655595838272
```

Create a `ProcessPoolExecutor` using the factory function `get_reusable_executor`.

The executor can be used exactly as `ProcessPoolExecutor`.

When the factory is called elsewhere, reuse the same executor if it is working.

When the executor is broken, automatically re-spawn a new one.

Conclusion

Conclusion

- `Thread` can be efficient to run multicore programs if your code releases the **GIL**.
- Else, you should use `Process` with `spawn` and try to reuse the pool of process as much as possible.
- `loky` can help you do that ;).
- Improves the management of a pool of workers in projects such as `joblib`.

Thanks for your attention!

Slides available at tommoral.github.io/pyparis17/

More on the GIL by Dave Beazley : dabeaz.com/python/GIL.pdf

Loky project : github.com/tommoral/loky

@tomamoral