Joblib
Toward efficient computing
From laptop to cloud

Alexandre Abadie

PyData Paris 2016

INVENTORS FOR THE DIGITAL WORLD

Overview of Joblib

Recent major improvements

What's next

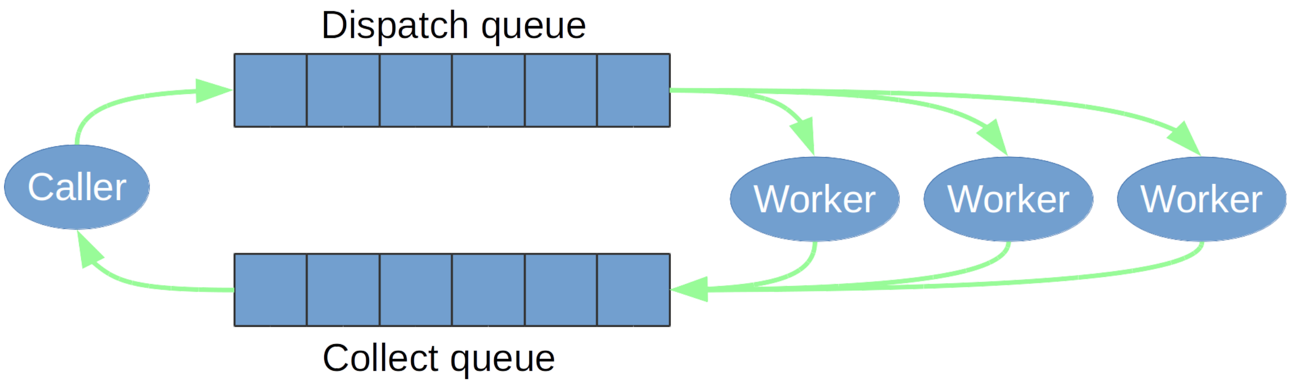
Overview of Joblib

- Embarrassingly Parallel computing helper
- Efficient disk caching to avoid recomputation
- Fast I/O persistence
- No dependencies, optimized for numpy arrays

Joblib is the parallel backend used by Scikit-Learn

https://pythonhosted.org/joblib/

Parallel helper



Available backends: **threading** and **multiprocessing** (default)

```
>>> from joblib import Parallel, delayed
>>> from math import sqrt
>>> Parallel(n_jobs=3, verbose=50)(delayed(sqrt)(i**2) for i in range(6))
[Parallel(n_jobs=3)]: Done   1 tasks      | elapsed:    0.0s
[...]
[Parallel(n_jobs=3)]: Done   6 out of   6 | elapsed:    0.0s finished
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0]
```

Caching on disk

- Use a **memoize** pattern with the **Memory** object

```
>>> from joblib import Memory
>>> mem = Memory(cachedir='/tmp/joblib')
>>> import numpy as np
>>> a = np.vander(np.arange(3)).astype(np.float)
>>> square = mem.cache(np.square)
>>> b = square(a)
________________________________________________________________________________
[Memory] Calling square...
square(array([[ 0.,  0.,  1.],
       [ 1.,  1.,  1.],
       [ 4.,  2.,  1.]]))
___________________________________________________________square - 0...s, 0.0min

>>> c = square(a) # no recomputation
```

- Use **md5** hash of input parameters
- Results are persisted on disk

Persistence

- Convert/create **an arbitrary object** into/from a **string of bytes**
- Persistence in Joblib is based on **pickle** and **Pickler/Unpickler** subclasses

```
>>> import numpy as np
>>> import joblib
>>> obj = [('a', [1, 2, 3]), ('b', np.arange(10))]
>>> joblib.dump(obj, '/tmp/test.pkl')
['/tmp/test.pkl', '/tmp/test.pkl_01.npy']
>>> joblib.load('/tmp/test.pkl')
[('a', [1, 2, 3]), ('b', array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]))]
```

- Use compression for fast I/O

```
>>> joblib.dump(obj, '/tmp/test.pkl', compress=True, cache_size=0)
['/tmp/test.pkl', '/tmp/test.pkl_01.npy.z']
>>> joblib.load('/tmp/test.pkl')
```

- Access numpy arrays with **`np.memmap`** for **out-of-core computing** or for sharing between multiple workers

Recent major improvements

Persistence

Custom parallel backends

arriving in version 0.10.0

Persistence refactoring

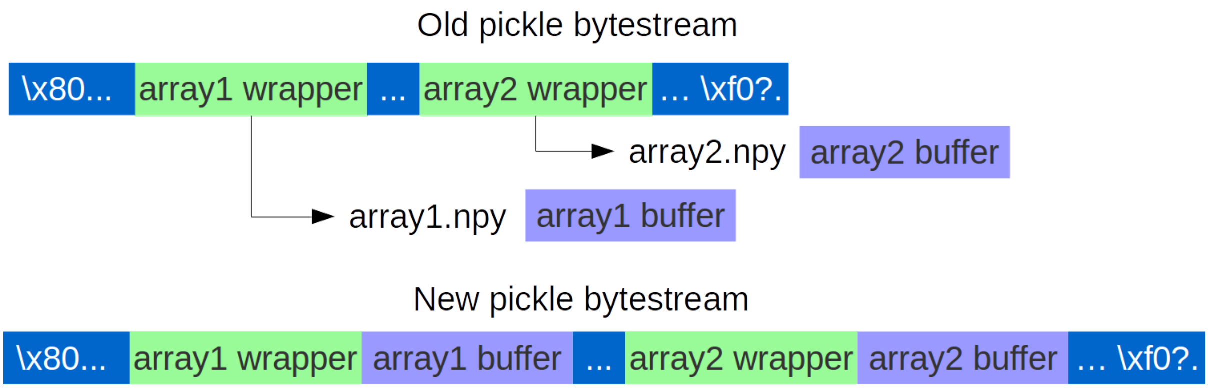
- *Until 0.9.4:*
    - An object with **multiple arrays** is persisted in **multiple files**
    - **Only zlib** compression available
    - **Memory copies** with compression

Persistence refactoring

- *Until 0.9.4:*
  - An object with **multiple arrays** is persisted in **multiple files**
  - **Only zlib** compression available
  - **Memory copies** with compression
- *In 0.10.0 (not released yet):*
  - An object with **multiple arrays** goes in a **single file**
  - Support of **all compression methods** provided by the python standard library
  - **No memory copies** with compression

https://github.com/joblib/joblib/pull/260

Persistence refactoring strategy

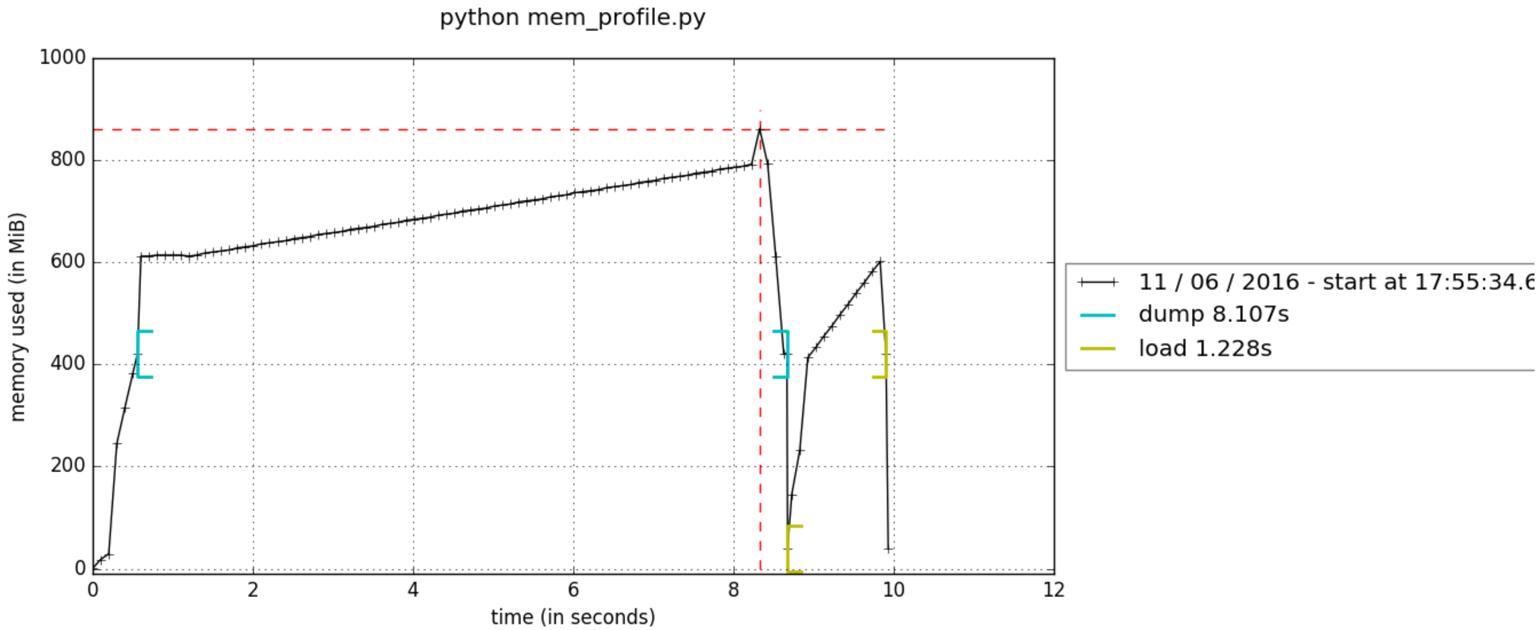


- Write numpy array buffer interleaved in the pickle stream
  ⇒ **all arrays in a single file**

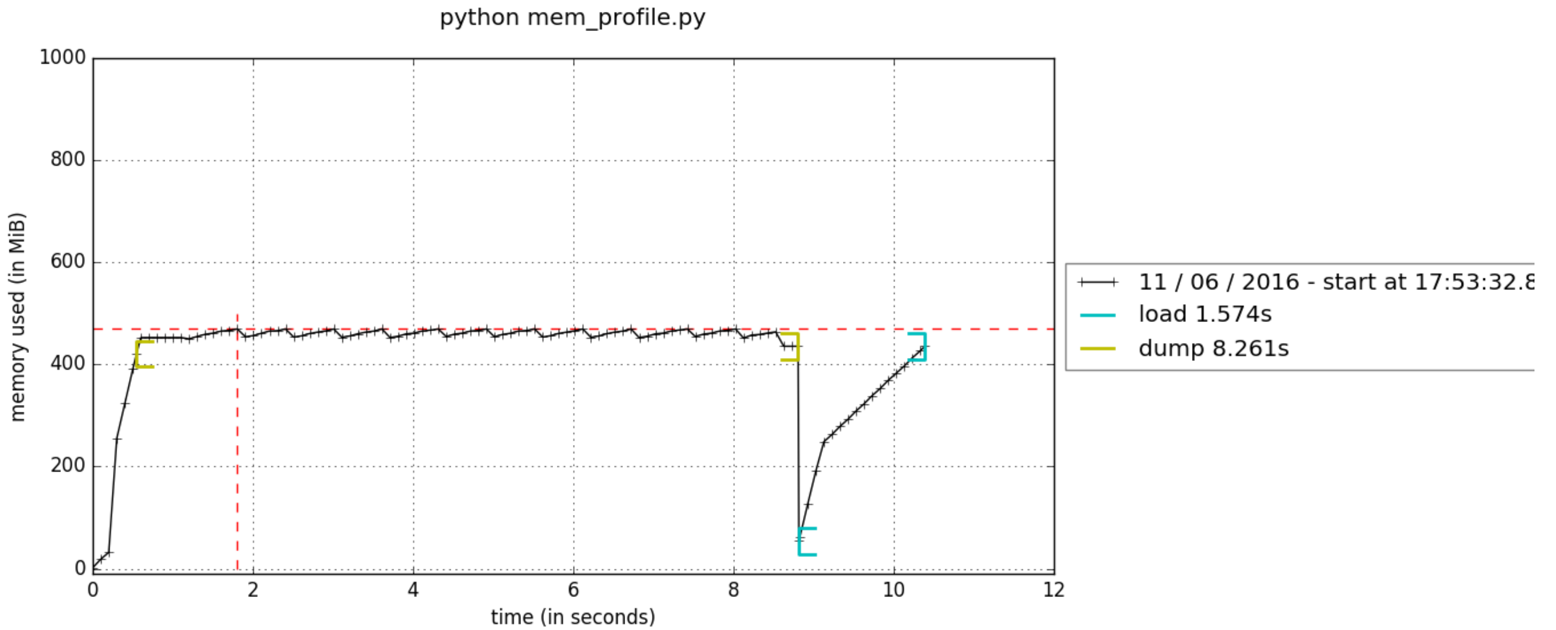
  Caveat: Not compatible with pickle format

- Dump/reconstruct the array by chunks of bytes using numpy functions
  ⇒ **avoid memory copies**

Before: memory copies



Now: **no** memory copies



Credits: Memory profiler

Persistence in a single file

```
>>> import numpy as np
>>> import joblib
>>> obj = [np.ones((5000, 5000)), np.random.random((5000, 5000))]

# only 1 file is generated:
>>> joblib.dump(obj, '/tmp/test.pkl', compress=True)
['/tmp/test.pkl']
>>> joblib.load('/tmp/test.pkl')
[array([[ 1.,  1., ...,  1.,  1.]],
 array([[ 0.47006195,  0.5436392 , ...,  0.1218267 ,  0.48592789]])]
```

- useful with scikit-learn estimators
- simpler management of backup files
- robust when using memory map on distributed file systems

Compression formats

- New supported compression formats

  ⇒ **gzip, bz2, lzma and xz**

- Automatic compression based on file extension
- Automatic detection of compression format when loading
- Valid compression file formats
- Slower than **zlib**

Compression formats

- New supported compression formats

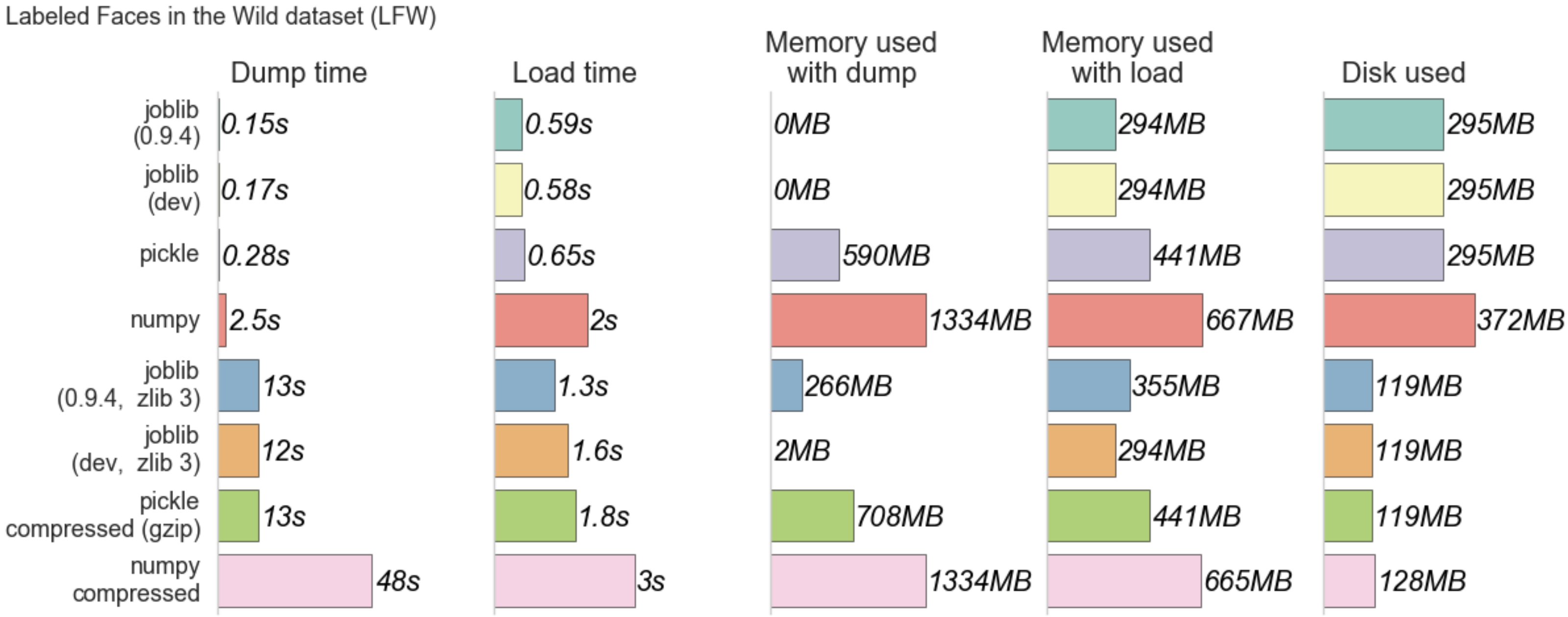
  ⇒ **gzip, bz2, lzma and xz**

- Automatic compression based on file extension
- Automatic detection of compression format when loading
- Valid compression file formats
- Slower than **zlib**

Example with **`gzip`** compression:

```
>>> joblib.dump(obj, '/tmp/test.pkl.gz', compress=('gzip', 3))
>>> joblib.load('/tmp/test.pkl.gz')
[array([[ 1.,  1., ...,  1.,  1.]],
 array([[ 0.47006195,  0.5436392 , ...,  0.1218267 ,  0.48592789]])]
>>> # or with file extension detection
>>> joblib.dump(obj, '/tmp/test.pkl.gz')
```

Performance comparison: Memory footprint and Speed



- Joblib persists **faster** and with **low extra memory consumption**
- Performance is data dependent

http://gael-varoquaux.info/programming/new_low-overhead_persistence_in_joblib_for_big_data.html

Parallel backends

Custom parallel backends

- *Until 0.9.4:*
    - Only **threading** and **multiprocessing** backends
    - Not extensible
    - Active backend cannot be changed easily at a high level

Custom parallel backends

- *Until 0.9.4:*
    - Only **threading** and **multiprocessing** backends
    - Not extensible
    - Active backend cannot be changed easily at a high level
- *In 0.10.0:*
    - Common API using **`ParallelBackendBase`** interface
    - Use **`with`** to set the active backend in a context manager
    - New backends for:
        - **distributed** implemented by Matthieu Rocklin
        - **ipyparallel** implemented by Min RK
        - **YARN** implemented by Niels Zielemaker

https://github.com/joblib/joblib/pull/306 contributed by Niels Zielemaker

Principle

1. Subclass **ParallelBackendBase**:

```
class ExampleParallelBackend(ParallelBackendBase):
    """Example of minimum parallel backend."""

def configure(self, n_jobs=1, parallel=None, **backend_args):
    self.n_jobs = self.effective_n_jobs(n_jobs)
    self.parallel = parallel
    return n_jobs

def apply_async(self, func, callback=None):
    """Schedule a func to be run"""
    result = func() # depends on the backend
    if callback:
        callback(result)
    return result
```

2. Register your backend:

```
>>> register_parallel_backend("example_backend", ExampleParallelBackend)
```

IPython parallel backend

Integration for Joblib available in version 5.1

1. Launch a 5 engines cluster:

   ```
   $ ipcontroller &
   $ ipcluster engines -n 5
   ```

2. Run the following script:

```python
import time
import ipyparallel as ipp
from ipyparallel.joblib import register as register_joblib
from joblib import parallel_backend, Parallel, delayed

# Register ipyparallel backend
register_joblib()
# Start the job
with parallel_backend("ipyparallel"):
    Parallel(n_jobs=20, verbose=50)(delayed(time.sleep)(1) for i in range(10))
```

Demo

https://github.com/aabadie/ipyparallel-cloud/blob/master/examples/sklearn_parameter_search_local_ipyparallel.ipynb

What's next

Persistence in file objects

1. With regular file object:

```
>>> with open('/tmp/test.pkl', 'wb') as fo:
...     joblib.dump(obj, fo)
>>> with open('/tmp/test.pkl', 'rb') as fo:
...     joblib.load(fo)
```

Also works with **gzip.GzipFile**, **bz2.BZ2File** and **lzma.LZMAFile**

https://github.com/joblib/joblib/pull/351

Persistence in file objects

1. With regular file object:

```
>>> with open('/tmp/test.pkl', 'wb') as fo:
...     joblib.dump(obj, fo)
>>> with open('/tmp/test.pkl', 'rb') as fo:
...     joblib.load(fo)
```

Also works with **gzip.GzipFile**, **bz2.BZ2File** and **lzma.LZMAFile**
https://github.com/joblib/joblib/pull/351

2. Or with any file-like object, e.g exposing read/write functions:

```
>>> with RemoteStoreObject(hostname, port, obj_id) as rso:
...     joblib.dump(obj, rso)
>>> with RemoteStoreObject(hostname, port, obj_id) as rso:
...     joblib.load(rso)
```

⇒Example: blob databases, remote storage

Joblib in the cloud

- Share persistence files
- Use computing ressources

Conclusion

- Persistence of numpy arrays in **a single file**...

  ... and soon in file objects

- New compression formats: **gzip**, **bz2**, **xz**, **lzma**

  ... extend to faster implementations (**blosc**) ?

- Persists **without memory copies**

  ... manipulate bigger data

- New parallel backends: **distributed**, **ipyparallel**, **YARN**...

  ... new ones to come ?

Thanks !

WENDELIN