
joblib Documentation

Release 0.11.1.dev0

Gael Varoquaux

Feb 07, 2018

Contents

1	Introduction	1
1.1	Vision	1
1.2	Main features	2
2	User manual	3
2.1	Why joblib: project goals	3
2.2	Installing joblib	4
2.3	On demand recomputing: the <i>Memory</i> class	5
2.4	Embarrassingly parallel for loops	13
2.5	Persistence	24
2.6	Examples	26
2.7	Development	37
3	Module reference	51
3.1	joblib.Memory	51
3.2	joblib.Parallel	52
3.3	joblib.dump	56
3.4	joblib.load	57
3.5	joblib.hash	58
	Python Module Index	59

Joblib is a set of tools to provide **lightweight pipelining in Python**. In particular:

1. transparent disk-caching of functions and lazy re-evaluation (memoize pattern)
2. easy simple parallel computing

Joblib is optimized to be **fast** and **robust** in particular on large data and has specific optimizations for *numpy* arrays. It is **BSD-licensed**.

Documentation:	http://pythonhosted.org/joblib
Download:	http://pypi.python.org/pypi/joblib#downloads
Source code:	http://github.com/joblib/joblib
Report issues:	http://github.com/joblib/joblib/issues

1.1 Vision

The vision is to provide tools to easily achieve better performance and reproducibility when working with long running jobs.

- **Avoid computing twice the same thing:** code is rerun over and over, for instance when prototyping computational-heavy jobs (as in scientific development), but hand-crafted solution to alleviate this issue is error-prone and often leads to unreproducible results
- **Persist to disk transparently:** persisting in an efficient way arbitrary objects containing large data is hard. Using joblib's caching mechanism avoids hand-written persistence and implicitly links the file on disk to the execution context of the original Python object. As a result, joblib's persistence is good for resuming an application status or computational job, eg after a crash.

Joblib addresses these problems while **leaving your code and your flow control as unmodified as possible** (no framework, no new paradigms).

1.2 Main features

1. **Transparent and fast disk-caching of output value:** a memoize or make-like functionality for Python functions that works well for arbitrary Python objects, including very large numpy arrays. Separate persistence and flow-execution logic from domain logic or algorithmic code by writing the operations as a set of steps with well-defined inputs and outputs: Python functions. Joblib can save their computation to disk and rerun it only if necessary:

```
>>> from joblib import Memory
>>> cachedir = 'your_cache_dir_goes_here'
>>> mem = Memory(cachedir=cachedir)
>>> 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)
>>> # The above call did not trigger an evaluation
```

2. **Embarrassingly parallel helper:** to make it easy to write readable parallel code and debug it quickly:

```
>>> from joblib import Parallel, delayed
>>> from math import sqrt
>>> Parallel(n_jobs=1)(delayed(sqrt)(i**2) for i in range(10))
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

3. **Fast compressed Persistence:** a replacement for pickle to work efficiently on Python objects containing large data (*joblib.dump* & *joblib.load*).

2.1 Why joblib: project goals

2.1.1 Benefits of pipelines

Pipeline processing systems can provide a set of useful features:

Data-flow programming for performance

- **On-demand computing:** in pipeline systems such as labView or VTK, calculations are performed as needed by the outputs and only when inputs change.
- **Transparent parallelization:** a pipeline topology can be inspected to deduce which operations can be run in parallel (it is equivalent to purely functional programming).

Provenance tracking to understand the code

- **Tracking of data and computations:** This enables the reproducibility of a computational experiment.
- **Inspecting data flow:** Inspecting intermediate results helps debugging and understanding.

But pipeline frameworks can get in the way

Joblib's philosophy is to keep the underlying algorithm code unchanged, avoiding framework-style modifications.

2.1.2 Joblib's approach

Functions are the simplest abstraction used by everyone. Pipeline jobs (or tasks) in Joblib are made of decorated functions.

Tracking of parameters in a meaningful way requires specification of data model. Joblib gives up on that and uses hashing for performance and robustness.

2.1.3 Design choices

- No dependencies other than Python
- Robust, well-tested code, at the cost of functionality
- Fast and suitable for scientific computing on big dataset without changing the original code
- Only local imports: **embed joblib in your code by copying it**

2.2 Installing joblib

2.2.1 Using *pip*

You can use *pip* to install joblib:

- For installing for all users, you need to run:

```
pip install joblib
```

You may need to run the above command as administrator

On a unix environment, it is better to install outside of the hierarchy managed by the system:

```
pip install --prefix /usr/local joblib
```

- Installing only for a specific user is easy if you use Python 2.7 or above:

```
pip install --user joblib
```

2.2.2 Using distributions

Joblib is packaged for several linux distribution: archlinux, debian, ubuntu, altlinux, and fedora. For minimum administration overhead, using the package manager is the recommended installation strategy on these systems.

2.2.3 The manual way

To install joblib first download the latest tarball (follow the link on the bottom of <http://pypi.python.org/pypi/joblib>) and expand it.

Installing in a local environment

If you don't need to install for all users, we strongly suggest that you create a local environment and install *joblib* in it. One of the pros of this method is that you never have to become administrator, and thus all the changes are local to your account and easy to clean up. Simply move to the directory created by expanding the *joblib* tarball and run the following command:

```
python setup.py install --user
```

Installing for all users

If you have administrator rights and want to install for all users, all you need to do is to go in directory created by expanding the *joblib* tarball and run the following line:

```
python setup.py install
```

If you are under Unix, we suggest that you install in '/usr/local' in order not to interfere with your system:

```
python setup.py install --prefix /usr/local
```

2.3 On demand recomputing: the *Memory* class

2.3.1 Use case

The *Memory* class defines a context for lazy evaluation of function, by putting the results in a store, by default using a disk, and not re-running the function twice for the same arguments.

It works by explicitly saving the output to a file and it is designed to work with non-hashable and potentially large input and output data types such as numpy arrays.

A simple example:

First, define the cache directory:

```
>>> cachedir = 'your_cache_location_directory'
```

Then, instantiate a memory context that uses this cache directory:

```
>>> from joblib import Memory
>>> memory = Memory(location=cachedir, verbose=0)
```

After these initial steps, just decorate a function to cache its output in this context:

```
>>> @memory.cache
... def f(x):
...     print('Running f(%s)' % x)
...     return x
```

Calling this function twice with the same argument does not execute it the second time, the output is just reloaded from a pickle file in the cache directory:

```
>>> print(f(1))
Running f(1)
1
>>> print(f(1))
1
```

However, calling the function with a different parameter executes it and recomputes the output:

```
>>> print(f(2))
Running f(2)
2
```

Comparison with *memoize*

The *memoize* decorator (<http://code.activestate.com/recipes/52201/>) caches in memory all the inputs and outputs of a function call. It can thus avoid running twice the same function, with a very small overhead. However, it compares input objects with those in cache on each call. As a result, for big objects there is a huge overhead. Moreover this approach does not work with numpy arrays, or other objects subject to non-significant fluctuations. Finally, using *memoize* with large objects will consume all the memory, whereas with *Memory*, objects are persisted to disk, using a persister optimized for speed and memory usage (*joblib.dump()*).

In short, *memoize* is best suited for functions with “small” input and output objects, whereas *Memory* is best suited for functions with complex input and output objects, and aggressive persistence to disk.

2.3.2 Using with *numpy*

The original motivation behind the *Memory* context was to have a memoize-like pattern on numpy arrays. *Memory* uses fast cryptographic hashing of the input arguments to check if they have been computed;

An example

Define two functions: the first with a number as an argument, outputting an array, used by the second one. Both functions are decorated with *Memory.cache*:

```
>>> import numpy as np

>>> @memory.cache
... def g(x):
...     print('A long-running calculation, with parameter %s' % x)
...     return np.hamming(x)

>>> @memory.cache
... def h(x):
...     print('A second long-running calculation, using g(x)')
...     return np.vander(x)
```

If the function *h* is called with the array created by the same call to *g*, *h* is not re-run:

```
>>> a = g(3)
A long-running calculation, with parameter 3
>>> a
array([ 0.08,  1.   ,  0.08])
```

```

>>> g(3)
array([ 0.08,  1.   ,  0.08])
>>> b = h(a)
A second long-running calculation, using g(x)
>>> b2 = h(a)
>>> b2
array([[ 0.0064,  0.08   ,  1.    ],
       [ 1.    ,  1.    ,  1.    ],
       [ 0.0064,  0.08   ,  1.    ]])
>>> np.allclose(b, b2)
True

```

Using memmapping

Memmapping (memory mapping) speeds up cache looking when reloading large numpy arrays:

```

>>> cachedir2 = 'your_cachedir2_location'
>>> memory2 = Memory(location=cachedir2, mmap_mode='r')
>>> square = memory2.cache(np.square)
>>> a = np.vander(np.arange(3)).astype(np.float)
>>> square(a)

```

```

↪_____
[Memory] Calling square...
square(array([[ 0.,  0.,  1.],
              [ 1.,  1.,  1.],
              [ 4.,  2.,  1.])))
square - 0.0s, 0.

```

```

↪0min
memmap([[ 0.,  0.,  1.],
        [ 1.,  1.,  1.],
        [ 16.,  4.,  1.]])

```

Note: Notice the debug mode used in the above example. It is useful for tracing of what is being reexecuted, and where the time is spent.

If the *square* function is called with the same input argument, its return value is loaded from the disk using memmapping:

```

>>> res = square(a)
>>> print(repr(res))
memmap([[ 0.,  0.,  1.],
        [ 1.,  1.,  1.],
        [ 16.,  4.,  1.]])

```

The memmap file must be closed to avoid file locking on Windows; closing numpy.memmap objects is done with `del`, which flushes changes to the disk

```
>>> del res
```

Note: If the memory mapping mode used was 'r', as in the above example, the array will be read only, and will be impossible to modified in place.

On the other hand, using 'r+' or 'w+' will enable modification of the array, but will propagate these modification to the disk, which will corrupt the cache. If you want modification of the array in memory, we suggest you use the 'c' mode: copy on write.

2.3.3 Shelving: using references to cached values

In some cases, it can be useful to get a reference to the cached result, instead of having the result itself. A typical example of this is when a lot of large numpy arrays must be dispatched across several workers: instead of sending the data themselves over the network, send a reference to the joblib cache, and let the workers read the data from a network filesystem, potentially taking advantage of some system-level caching too.

Getting a reference to the cache can be done using the `call_and_shelve` method on the wrapped function:

```
>>> result = g.call_and_shelve(4)
A long-running calculation, with parameter 4
>>> result
MemorizedResult(cachedir="...", func="g...", argument_hash="...")
```

Once computed, the output of `g` is stored on disk, and deleted from memory. Reading the associated value can then be performed with the `get` method:

```
>>> result.get()
array([ 0.08,  0.77,  0.77,  0.08])
```

The cache for this particular value can be cleared using the `clear` method. Its invocation causes the stored value to be erased from disk. Any subsequent call to `get` will cause a `KeyError` exception to be raised:

```
>>> result.clear()
>>> result.get()
Traceback (most recent call last):
...
KeyError: 'Non-existing cache value (may have been cleared).\nFile ... does_
↳not exist'
```

A `MemorizedResult` instance contains all that is necessary to read the cached value. It can be pickled for transmission or storage, and the printed representation can even be copy-pasted to a different python interpreter.

Shelving when cache is disabled

In the case where caching is disabled (e.g. `Memory(cachedir=None)`), the `call_and_shelve` method returns a `NotMemorizedResult` instance, that stores the full function output, instead of just a reference (since there is nothing to point to). All the above remains valid though, except for the copy-pasting feature.

2.3.4 Gotchas

- **Across sessions, function cache is identified by the function's name.** Thus assigning the same name to different functions, their cache will override each-others (e.g. there are 'name collisions'), and unwanted re-run will happen:

```

>>> @memory.cache
... def func(x):
...     print('Running func(%s)' % x)

>>> func2 = func

>>> @memory.cache
... def func(x):
...     print('Running a different func(%s)' % x)

```

As long as the same session is used, there are no collisions (in joblib 0.8 and above), although joblib does warn you that you are doing something dangerous:

```

>>> func(1)
Running a different func(1)

>>> func2(1)
memory.rst:0: JobLibCollisionWarning: Possible name collisions between_
↳functions 'func' (<doctest memory.rst>:...) and 'func' (<doctest_
↳memory.rst>:...)
Running func(1)

>>> func(1) # No recomputation so far
>>> func2(1) # No recomputation so far

```

But suppose the interpreter is exited and then restarted, the cache will not be identified properly, and the functions will be rerun:

```

>>> func(1)
memory.rst:0: JobLibCollisionWarning: Possible name collisions between_
↳functions 'func' (<doctest memory.rst>:...) and 'func' (<doctest_
↳memory.rst>:...)
Running a different func(1)
>>> func2(1)
Running func(1)

```

As long as the same session is used, there are no needless recomputation:

```

>>> func(1) # No recomputation now
>>> func2(1) # No recomputation now

```

- **lambda functions**

Beware that with Python 2.7 lambda functions cannot be separated out:

```

>>> def my_print(x):
...     print(x)

>>> f = memory.cache(lambda : my_print(1))
>>> g = memory.cache(lambda : my_print(2))

>>> f()
1
>>> f()
>>> g()
memory.rst:0: JobLibCollisionWarning: Cannot detect name collisions for_
↳function '<lambda>'
2

```

```
>>> g()
>>> f()
1
```

- **memory cannot be used on some complex objects**, e.g. a callable object with a `__call__` method.

However, it works on numpy ufuncs:

```
>>> sin = memory.cache(np.sin)
>>> print(sin(0))
0.0
```

- **caching methods: memory is designed for pure functions and it is not recommended to use it for methods.** If one wants to use cache inside a class the recommended pattern is to cache a pure function and use the cached function inside your class, i.e. something like this:

```
@mem.cache
def compute_func(arg1, arg2, arg3):
    # long computation
    return result

class Foo(object):
    def __init__(self, args):
        self.data = None

    def compute(self):
        self.data = compute_func(self.arg1, self.arg2, 40)
```

Using `Memory` for methods is not recommended and has some caveats that make it very fragile from a maintenance point of view because it is very easy to forget about these caveats when a software evolves. If this cannot be avoided (we would be interested about your use case by the way), here are a few known caveats:

1. a method cannot be decorated at class definition, because when the class is instantiated, the first argument (`self`) is *bound*, and no longer accessible to the `Memory` object. The following code won't work:

```
class Foo(object):

    @mem.cache # WRONG
    def method(self, args):
        pass
```

The right way to do this is to decorate at instantiation time:

```
class Foo(object):

    def __init__(self, args):
        self.method = mem.cache(self.method)

    def method(self, ...):
        pass
```

2. The cached method will have `self` as one of its arguments. That means that the result will be recomputed if anything with `self` changes. For example if `self.attr` has changed calling `self.method` will recompute the result even if `self.method` does not use `self.attr` in its body. Another example is changing `self` inside the body of `self.method`.

The consequence is that `self.method` will create cache that will not be reused in subsequent calls. To alleviate these problems and if you *know* that the result of `self.method` does not depend on `self` you can use `self.method = mem.cache(self.method, ignore=['self'])`.

2.3.5 Ignoring some arguments

It may be useful not to recalculate a function when certain arguments change, for instance a debug flag. *Memory* provides the *ignore* list:

```
>>> @memory.cache(ignore=['debug'])
... def my_func(x, debug=True):
...     print('Called with x = %s' % x)
>>> my_func(0)
Called with x = 0
>>> my_func(0, debug=False)
>>> my_func(0, debug=True)
>>> # my_func was not reevaluated
```

2.3.6 Reference documentation of the *Memory* class

```
class joblib.memory.Memory (location=None, backend='local', cachedir=None,
                             mmap_mode=None, compress=False, verbose=1,
                             bytes_limit=None, backend_options={})
```

A context object for caching a function's return value each time it is called with the same input arguments.

All values are cached on the filesystem, in a deep directory structure.

see [Reference documentation of the Memory class](#)

```
__init__ (location=None, backend='local', cachedir=None, mmap_mode=None,
          compress=False, verbose=1, bytes_limit=None, backend_options={})
```

Parameters **location: str or None**

The path of the base directory to use as a data store or None. If None is given, no caching is done and the Memory object is completely transparent. This option replaces `cachedir` since version 0.11.

backend: str or 'local'

Type of store backend for reading/writing cache files. Default is 'local'. The 'local' backend is using regular filesystem operations to manipulate data (open, mv, etc) in the backend.

cachedir: str or None

`cachedir` is deprecated since version 0.11 and will be removed in 0.13. Please consider using `location` option instead. The path of the base directory to use as a data store or None. If None is given, no caching is done and the Memory object is completely transparent.

mmap_mode: {None, 'r+', 'r', 'w+', 'c'}, optional

The memmapping mode used when loading from cache numpy arrays. See `numpy.load` for the meaning of the arguments.

compress: boolean, or integer

Whether to zip the stored data on disk. If an integer is given, it should be between 1 and 9, and sets the amount of compression. Note that compressed arrays cannot be read by memmapping.

verbose: int, optional

Verbosity flag, controls the debug messages that are issued as functions are evaluated.

bytes_limit: int, optional

Limit in bytes of the size of the cache.

backend_options: dict, optional

Contains a dictionary of named parameters used to configure the store backend.

cache (*func=None, ignore=None, verbose=None, mmap_mode=False*)

Decorates the given function *func* to only compute its return value for input arguments not cached on disk.

Parameters **func: callable, optional**

The function to be decorated

ignore: list of strings

A list of arguments name to ignore in the hashing

verbose: integer, optional

The verbosity mode of the function. By default that of the memory object is used.

mmap_mode: {None, 'r+', 'r', 'w+', 'c'}, optional

The memmapping mode used when loading from cache numpy arrays. See `numpy.load` for the meaning of the arguments. By default that of the memory object is used.

Returns *decorated_func*: `MemorizedFunc` object

The returned object is a `MemorizedFunc` object, that is callable (behaves like a function), but offers extra methods for cache lookup and management. See the documentation for `joblib.memory.MemorizedFunc`.

clear (*warn=True*)

Erase the complete cache directory.

eval (*func, *args, **kwargs*)

Eval function *func* with arguments **args* and ***kwargs*, in the context of the memory.

This method works similarly to the builtin `apply`, except that the function is called only if the cache is not up to date.

2.3.7 Useful methods of decorated functions

Function decorated by `Memory.cache()` are `MemorizedFunc` objects that, in addition of behaving like normal functions, expose methods useful for cache exploration and management.

class `joblib.memory.MemorizedFunc` (*func, location, backend='local', ignore=None, mmap_mode=None, compress=False, verbose=1, timestamp=None*)

Callable object decorating a function for caching its return value each time it is called.

Methods are provided to inspect the cache or clean it.

Attributes

func: callable	The original, undecorated, function.
location: string	The location of joblib cache. Depends on the store backend used.
backend: str	Type of store backend for reading/writing cache files. Default is 'local', in which case the location is the path to a disk storage.
ignore: list or None	List of variable names to ignore when choosing whether to recompute.
mmap_mode: {None, 'r+', 'r', 'w+', 'c'}	The memmapping mode used when loading from cache numpy arrays. See <code>numpy.load</code> for the meaning of the different values.
compress: boolean, or integer	Whether to zip the stored data on disk. If an integer is given, it should be between 1 and 9, and sets the amount of compression. Note that compressed arrays cannot be read by memmapping.
verbose: int, optional	The verbosity flag, controls messages that are issued as the function is evaluated.

call (*args, **kwargs)

Force the execution of the function with the given arguments and persist the output values.

clear (warn=True)

Empty the function's cache.

2.4 Embarrassingly parallel for loops

2.4.1 Common usage

Joblib provides a simple helper class to write parallel for loops using multiprocessing. The core idea is to write the code to be executed as a generator expression, and convert it to parallel computing:

```
>>> from math import sqrt
>>> [sqrt(i ** 2) for i in range(10)]
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

can be spread over 2 CPUs using the following:

```
>>> from math import sqrt
>>> from joblib import Parallel, delayed
>>> Parallel(n_jobs=2)(delayed(sqrt)(i ** 2) for i in range(10))
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

2.4.2 Thread-based parallelism vs process-based parallelism

By default `Parallel` uses the 'loky' backend module to start separate Python worker processes to execute tasks concurrently on separate CPUs. This is a reasonable default for generic Python programs but can induce a significant overhead as the input and output data need to be serialized in a queue for communication with the worker processes.

When you know that the function you are calling is based on a compiled extension that releases the Python Global Interpreter Lock (GIL) during most of its computation then it is more efficient to use

threads instead of Python processes as concurrent workers. For instance this is the case if you write the CPU intensive part of your code inside a `with nogil` block of a Cython function.

To hint that your code can efficiently use threads, just pass `prefer="threads"` as parameter of the `Parallel` constructor. In this case joblib will automatically use the "threading" backend instead of the default "loky" backend:

```
>>> Parallel(n_jobs=2, prefer="threads")(
...     delayed(sqrt)(i ** 2) for i in range(10))
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

It is also possible to manually select a specific backend implementation with the help of a context manager:

```
>>> from joblib import parallel_backend
>>> with parallel_backend('threading'):
...     Parallel(n_jobs=2)(delayed(sqrt)(i ** 2) for i in range(10))
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

The latter is especially useful when calling a library that uses `joblib.Parallel` internally without exposing backend selection as part of its public API.

Note that the `prefer="threads"` option was introduced in joblib 0.12. In prior versions, the same effect could be achieved by hardcoding a specific backend implementation such as `backend="threading"` in the call to `Parallel` but this is now considered a bad pattern (when done in a library) as it does not make it possible to override that choice with the `parallel_backend` context manager.

2.4.3 Shared-memory semantics

The default backend of joblib will run each function call in isolated Python processes, therefore they cannot mutate a common Python object defined in the main program.

However if the parallel function really needs to rely on the shared memory semantics of threads, it should be made explicit with `require='sharedmem'`, for instance:

```
>>> shared_set = set()
>>> def collect(x):
...     shared_set.add(x)
...
>>> Parallel(n_jobs=2, require='sharedmem')(
...     delayed(collect)(i) for i in range(5))
[None, None, None, None, None]
>>> sorted(shared_set)
[0, 1, 2, 3, 4]
```

Keep in mind that relying on the shared-memory semantics is probably suboptimal from a performance point of view as concurrent access to a shared Python object will suffer from lock contention.

2.4.4 Reusing a pool of workers

Some algorithms require to make several consecutive calls to a parallel function interleaved with processing of the intermediate results. Calling `Parallel` several times in a loop is sub-optimal because it will create and destroy a pool of workers (threads or processes) several times which can cause a significant overhead.

For this case it is more efficient to use the context manager API of the `Parallel` class to re-use the same pool of workers for several calls to the `Parallel` object:

```
>>> with Parallel(n_jobs=2) as parallel:
...     accumulator = 0.
...     n_iter = 0
...     while accumulator < 1000:
...         results = parallel(delayed(sqrt)(accumulator + i ** 2)
...                             for i in range(5))
...         accumulator += sum(results) # synchronization barrier
...         n_iter += 1
...
>>> (accumulator, n_iter)
(1136.596..., 14)
```

2.4.5 Working with numerical data in shared memory (memmapping)

By default the workers of the pool are real Python processes forked using the multiprocessing module of the Python standard library when `n_jobs != 1`. The arguments passed as input to the `Parallel` call are serialized and reallocated in the memory of each worker process.

This can be problematic for large arguments as they will be reallocated `n_jobs` times by the workers.

As this problem can often occur in scientific computing with `numpy` based datastructures, `joblib.Parallel` provides a special handling for large arrays to automatically dump them on the filesystem and pass a reference to the worker to open them as memory map on that file using the `numpy.memmap` subclass of `numpy.ndarray`. This makes it possible to share a segment of data between all the worker processes.

Note: The following only applies with the `"loky"` and `'multiprocessing'` process-backends. If your code can release the GIL, then using a thread-based backend by passing `prefer='threads'` is even more efficient because it makes it possible to avoid the communication overhead of process-based parallelism.

Scientific Python libraries such as `numpy`, `scipy`, `pandas` and `scikit-learn` often release the GIL in performance critical code paths. It is therefore advised to always measure the speed of thread-based parallelism and use it when the scalability is not limited by the GIL.

Automated array to memmap conversion

The automated array to memmap conversion is triggered by a configurable threshold on the size of the array:

```
>>> import numpy as np
>>> from joblib import Parallel, delayed
>>> from joblib.pool import has_shareable_memory

>>> Parallel(n_jobs=2, max_nbytes=1e6) (
...     delayed(has_shareable_memory)(np.ones(int(i)))
...     for i in [1e2, 1e4, 1e6])
[False, False, True]
```

By default the data is dumped to the `/dev/shm` shared-memory partition if it exists and is writable (typically the case under Linux). Otherwise the operating system's temporary folder is used. The location

of the temporary data files can be customized by passing a `temp_folder` argument to the `Parallel` constructor.

Passing `max_nbytes=None` makes it possible to disable the automated array to memmap conversion.

Manual management of memmapped input data

For even finer tuning of the memory usage it is also possible to dump the array as a memmap directly from the parent process to free the memory before forking the worker processes. For instance let's allocate a large array in the memory of the parent process:

```
>>> large_array = np.ones(int(1e6))
```

Dump it to a local file for memmapping:

```
>>> import tempfile
>>> import os
>>> from joblib import load, dump

>>> temp_folder = tempfile.mkdtemp()
>>> filename = os.path.join(temp_folder, 'joblib_test.memmap')
>>> if os.path.exists(filename): os.unlink(filename)
>>> _ = dump(large_array, filename)
>>> large_memmap = load(filename, mmap_mode='r+')
```

The `large_memmap` variable is pointing to a `numpy.memmap` instance:

```
>>> large_memmap.__class__.__name__, large_array.nbytes, large_array.shape
('memmap', 8000000, (1000000,))

>>> np.allclose(large_array, large_memmap)
True
```

The original array can be freed from the main process memory:

```
>>> del large_array
>>> import gc
>>> _ = gc.collect()
```

It is possible to slice `large_memmap` into a smaller memmap:

```
>>> small_memmap = large_memmap[2:5]
>>> small_memmap.__class__.__name__, small_memmap.nbytes, small_memmap.shape
('memmap', 24, (3,))
```

Finally a `np.ndarray` view backed on that same memory mapped file can be used:

```
>>> small_array = np.asarray(small_memmap)
>>> small_array.__class__.__name__, small_array.nbytes, small_array.shape
('ndarray', 24, (3,))
```

All those three datastructures point to the same memory buffer and this same buffer will also be reused directly by the worker processes of a `Parallel` call:

```
>>> Parallel(n_jobs=2, max_nbytes=None)(
...     delayed(has_shareable_memory)(a)
...     for a in [large_memmap, small_memmap, small_array])
[True, True, True]
```

Note that here `max_nbytes=None` is used to disable the auto-dumping feature of `Parallel`. `small_array` is still in shared memory in the worker processes because it was already backed by shared memory in the parent process. The pickling machinery of `Parallel` multiprocessing queues are able to detect this situation and optimize it on the fly to limit the number of memory copies.

Writing parallel computation results in shared memory

If data are opened using the `w+` or `r+` mode in the main program, the worker will get `r+` mode access. Thus the worker will be able to write its results directly to the original data, alleviating the need of the serialization to send back the results to the parent process.

Here is an example script on parallel processing with preallocated `numpy.memmap` datastructures *NumPy memmap in joblib.Parallel*.

Warning: Having concurrent workers write on overlapping shared memory data segments, for instance by using inplace operators and assignments on a `numpy.memmap` instance, can lead to data corruption as `numpy` does not offer atomic operations. The previous example does not risk that issue as each task is updating an exclusive segment of the shared result array.

Some C/C++ compilers offer lock-free atomic primitives such as add-and-fetch or compare-and-swap that could be exposed to Python via `CFFI` for instance. However providing `numpy`-aware atomic constructs is outside of the scope of the `joblib` project.

A final note: don't forget to clean up any temporary folder when you are done with the computation:

```
>>> import shutil
>>> try:
...     shutil.rmtree(temp_folder)
... except OSError:
...     pass # this can sometimes fail under Windows
```

Note that the `'loky'` backend now used by default for process-based parallelism automatically tries to maintain and reuse a pool of workers by it-self even for calls without the context manager.

2.4.6 Custom backend API (experimental)

New in version 0.10.

Warning: The custom backend API is experimental and subject to change without going through a deprecation cycle.

User can provide their own implementation of a parallel processing backend in addition to the `'loky'`, `'threading'`, `'multiprocessing'` backends provided by default. A backend is registered with the `joblib.register_parallel_backend()` function by passing a name and a backend factory.

The backend factory can be any callable that returns an instance of `ParallelBackendBase`. Please refer to the [default backends source code](#) as a reference if you want to implement your own custom backend.

Note that it is possible to register a backend class that has some mandatory constructor parameters such as the network address and connection credentials for a remote cluster computing service:

```

class MyCustomBackend(ParallelBackendBase):

    def __init__(self, endpoint, api_key):
        self.endpoint = endpoint
        self.api_key = api_key

    ...
    # Do something with self.endpoint and self.api_key somewhere in
    # one of the method of the class

register_parallel_backend('custom', MyCustomBackend)

```

The connection parameters can then be passed to the `joblib.parallel_backend()` context manager:

```

with parallel_backend('custom', endpoint='http://compute', api_key='42'):
    Parallel()(delayed(some_function)(i) for i in range(10))

```

Using the context manager can be helpful when using a third-party library that uses `joblib.Parallel` internally while not exposing the backend argument in its own API.

2.4.7 Old multiprocessing backend

Prior to version 0.12, joblib used the 'multiprocessing' backend as default backend instead of 'loky'.

This backend creates an instance of `multiprocessing.Pool` that forks the Python interpreter in multiple processes to execute each of the items of the list. The `delayed` function is a simple trick to be able to create a tuple (`function, args, kwargs`) with a function-call syntax.

Warning: Under Windows, the use of `multiprocessing.Pool` requires to protect the main loop of code to avoid recursive spawning of subprocesses when using `joblib.Parallel`. In other words, you should be writing code like this when using the 'multiprocessing' backend:

```

import ....

def function1(...):
    ...

def function2(...):
    ...

...
if __name__ == '__main__':
    # do stuff with imports and functions defined about
    ...

```

No code should *run* outside of the "if `__name__ == '__main__'`" blocks, only imports and definitions.

The 'loky' backend used by default in joblib 0.12 and later does not impose this anymore.

2.4.8 Bad interaction of multiprocessing and third-party libraries

Using the 'multiprocessing' backend can cause a crash when using third party libraries that manage their own native thread-pool if the library is first used in the main process and subsequently called again in a worker process (inside the `Parallel` call).

Joblib version 0.12 and later are no longer subject to this problem thanks to the use of `loky` as the new default backend for process-based parallelism.

Prior to Python 3.4 the 'multiprocessing' backend of joblib can only use the `fork` strategy to create worker processes under non-Windows systems. This can cause some third-party libraries to crash or freeze. Such libraries include Apple `vecLib` / `Accelerate` (used by `NumPy` under OSX), some old version of `OpenBLAS` (prior to 0.2.10) or the `OpenMP` runtime implementation from `GCC` which is used internally by third-party libraries such as `XGBoost`, `spaCy`, `OpenCV`...

The best way to avoid this problem is to use the 'loky' backend instead of the `multiprocessing` backend. Prior to joblib 0.12, it is also possible to get `joblib.Parallel` configured to use the 'forkserver' start method on Python 3.4 and later. The start method has to be configured by setting the `JOBLIB_START_METHOD` environment variable to 'forkserver' instead of the default 'fork' start method. However the user should be aware that using the 'forkserver' method prevents `joblib.Parallel` to call function interactively defined in a shell session.

You can read more on this topic in the [multiprocessing documentation](#).

Under Windows the `fork` system call does not exist at all so this problem does not exist (but multiprocessing has more overhead).

2.4.9 Parallel reference documentation

```
class joblib.Parallel(n_jobs=None, backend=None, verbose=0, time-
out=None, pre_dispatch='2 * n_jobs', batch_size='auto',
temp_folder=None, max_nbytes='1M', mmap_mode='r', pre-
fer=None, require=None)
```

Helper class for readable parallel mapping.

Parameters `n_jobs`: int, default: None

The maximum number of concurrently running jobs, such as the number of Python worker processes when `backend="multiprocessing"` or the size of the thread-pool when `backend="threading"`. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For `n_jobs` below -1, $(n_cpus + 1 + n_jobs)$ are used. Thus for `n_jobs = -2`, all CPUs but one are used. None is a marker for 'unset' that will be interpreted as `n_jobs=1` (sequential execution) unless the call is performed under a `parallel_backend` context manager that sets another value for `n_jobs`.

backend: str, ParallelBackendBase instance or None, default: 'loky'

Specify the parallelization backend implementation. Supported backends are:

- "loky" used by default, can induce some communication and memory overhead when exchanging input and output data with the worker Python processes.
- "multiprocessing" previous process-based backend based on `multiprocessing.Pool`. Less robust than `loky`.
- "threading" is a very low-overhead backend but it suffers from the Python Global Interpreter Lock if the called function relies a lot on Python objects.

“threading” is mostly useful when the execution bottleneck is a compiled extension that explicitly releases the GIL (for instance a Cython loop wrapped in a “with nogil” block or an expensive call to a library such as NumPy).

- finally, you can register backends by calling `register_parallel_backend`. This will allow you to implement a backend of your liking.

It is not recommended to hard-code the backend name in a call to `Parallel` in a library. Instead it is recommended to set soft hints (prefer) or hard constraints (require) so as to make it possible for library users to change the backend from the outside using the `parallel_backend` context manager.

prefer: str in {'processes', 'threads'} or None, default: None

Soft hint to choose the default backend if no specific backend was selected with the `parallel_backend` context manager. The default process-based backend is ‘loky’ and the default thread-based backend is ‘threading’.

require: 'sharedmem' or None, default None

Hard constraint to select the backend. If set to ‘sharedmem’, the selected backend will be single-host and thread-based even if the user asked for a non-thread based backend with `parallel_backend`.

verbose: int, optional

The verbosity level: if non zero, progress messages are printed. Above 50, the output is sent to stdout. The frequency of the messages increases with the verbosity level. If it more than 10, all iterations are reported.

timeout: float, optional

Timeout limit for each task to complete. If any task takes longer a `TimeoutError` will be raised. Only applied when `n_jobs != 1`

pre_dispatch: {'all', integer, or expression, as in '3*n_jobs'}

The number of batches (of tasks) to be pre-dispatched. Default is ‘2*n_jobs’. When `batch_size="auto"` this is reasonable default and the workers should never starve.

batch_size: int or 'auto', default: 'auto'

The number of atomic tasks to dispatch at once to each worker. When individual evaluations are very fast, dispatching calls to workers can be slower than sequential computation because of the overhead. Batching fast computations together can mitigate this. The ‘auto’ strategy keeps track of the time it takes for a batch to complete, and dynamically adjusts the batch size to keep the time on the order of half a second, using a heuristic. The initial batch size is 1. `batch_size="auto" with backend="threading"` will dispatch batches of a single task at a time as the threading backend has very little overhead and using larger batch size has not proved to bring any gain in that case.

temp_folder: str, optional

Folder to be used by the pool for memmapping large arrays for sharing memory with worker processes. If None, this will try in order:

- a folder pointed by the `JOBLIB_TEMP_FOLDER` environment variable,
- `/dev/shm` if the folder exists and is writable: this is a RAM disk filesystem available by default on modern Linux distributions,

- the default system temporary folder that can be overridden with TMP, TMPDIR or TEMP environment variables, typically /tmp under Unix operating systems.

Only active when backend="loky" or "multiprocessing".

max_nbytes int, str, or None, optional, 1M by default

Threshold on the size of arrays passed to the workers that triggers automated memory mapping in temp_folder. Can be an int in Bytes, or a human-readable string, e.g., '1M' for 1 megabyte. Use None to disable memmapping of large arrays. Only active when backend="loky" or "multiprocessing".

mmap_mode: {None, 'r+', 'r', 'w+', 'c'}

Memmapping mode for numpy arrays passed to workers. See 'max_nbytes' parameter documentation for more details.

Notes

This object uses workers to compute in parallel the application of a function to many different arguments. The main functionality it brings in addition to using the raw multiprocessing or concurrent.futures API are (see examples for details):

- More readable code, in particular since it avoids constructing list of arguments.
- **Easier debugging:**
 - informative tracebacks even when the error happens on the client side
 - using 'n_jobs=1' enables to turn off parallel computing for debugging without changing the codepath
 - early capture of pickling errors
- An optional progress meter.
- Interruption of multiprocesses jobs with 'Ctrl-C'
- Flexible pickling control for the communication to and from the worker processes.
- Ability to use shared memory efficiently with worker processes for large numpy-based data-structures.

Examples

A simple example:

```
>>> from math import sqrt
>>> from joblib import Parallel, delayed
>>> Parallel(n_jobs=1)(delayed(sqrt)(i**2) for i in range(10))
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

Reshaping the output when the function has several return values:

```
>>> from math import modf
>>> from joblib import Parallel, delayed
>>> r = Parallel(n_jobs=1)(delayed(modf)(i/2.) for i in range(10))
>>> res, i = zip(*r)
>>> res
(0.0, 0.5, 0.0, 0.5, 0.0, 0.5, 0.0, 0.5, 0.0, 0.5)
```

```
>>> i
(0.0, 0.0, 1.0, 1.0, 2.0, 2.0, 3.0, 3.0, 4.0, 4.0)
```

The progress meter: the higher the value of *verbose*, the more messages:

```
>>> from time import sleep
>>> from joblib import Parallel, delayed
>>> r = Parallel(n_jobs=2, verbose=10)(delayed(sleep)(.2) for _ in
↳ range(10))
[Parallel(n_jobs=2)]: Done 1 tasks | elapsed: 0.6s
[Parallel(n_jobs=2)]: Done 4 tasks | elapsed: 0.8s
[Parallel(n_jobs=2)]: Done 10 out of 10 | elapsed: 1.4s finished
```

Traceback example, note how the line of the error is indicated as well as the values of the parameter passed to the function that triggered the exception, even though the traceback happens in the child process:

```
>>> from heapq import nlargest
>>> from joblib import Parallel, delayed
>>> Parallel(n_jobs=2)(delayed(nlargest)(2, n) for n in (range(4), 'abcde
↳ ', 3))
#...
-----
↳ --
Sub-process traceback:
-----
↳ --
TypeError                               Mon Nov 12 11:37:46
↳ 2012
PID: 12934                               Python 2.7.3: /usr/bin/
↳ python
.....
↳ ..
/usr/lib/python2.7/heapq.pyc in nlargest(n=2, iterable=3, key=None)
    419         if n >= size:
    420             return sorted(iterable, key=key, reverse=True)[:n]
    421
    422         # When key is none, use simpler decoration
    423         if key is None:
--> 424             it = izip(iterable, count(0,-1))           #
↳ decorate
    425             result = _nlargest(n, it)
    426             return map(itemgetter(0), result)           #
↳ undecorate
    427
    428         # General case, slowest method
TypeError: izip argument #1 must support iteration
↳ _
```

Using `pre_dispatch` in a producer/consumer situation, where the data is generated on the fly. Note how the producer is first called 3 times before the parallel loop is initiated, and then called to generate new data on the fly:

```
>>> from math import sqrt
>>> from joblib import Parallel, delayed
>>> def producer():
...     for i in range(6):
```

```

...     print('Produced %s' % i)
...     yield i
>>> out = Parallel(n_jobs=2, verbose=100, pre_dispatch='1.5*n_jobs')(
...     delayed(sqrt)(i) for i in producer())
Produced 0
Produced 1
Produced 2
[Parallel(n_jobs=2)]: Done 1 jobs      | elapsed: 0.0s
Produced 3
[Parallel(n_jobs=2)]: Done 2 jobs      | elapsed: 0.0s
Produced 4
[Parallel(n_jobs=2)]: Done 3 jobs      | elapsed: 0.0s
Produced 5
[Parallel(n_jobs=2)]: Done 4 jobs      | elapsed: 0.0s
[Parallel(n_jobs=2)]: Done 6 out of 6 | elapsed: 0.0s remaining: 0.0s
[Parallel(n_jobs=2)]: Done 6 out of 6 | elapsed: 0.0s finished

```

`joblib.delayed` (*function, check_pickle=None*)

Decorator used to capture the arguments of a function.

`joblib.register_parallel_backend` (*name, factory, make_default=False*)

Register a new Parallel backend factory.

The new backend can then be selected by passing its name as the backend argument to the Parallel class. Moreover, the default backend can be overwritten globally by setting `make_default=True`.

The factory can be any callable that takes no argument and return an instance of `ParallelBackendBase`.

Warning: this function is experimental and subject to change in a future version of joblib.

New in version 0.10.

`joblib.parallel_backend` (**args, **kws*)

Change the default backend used by Parallel inside a with block.

If backend is a string it must match a previously registered implementation using the `register_parallel_backend` function.

Alternatively backend can be passed directly as an instance.

By default all available workers will be used (`n_jobs=-1`) unless the caller passes an explicit value for the `n_jobs` parameter.

This is an alternative to passing a `backend='backend_name'` argument to the Parallel class constructor. It is particularly useful when calling into library code that uses joblib internally but does not expose the backend argument in its own API.

```

>>> from operator import neg
>>> with parallel_backend('threading'):
...     print(Parallel()(delayed(neg)(i + 1) for i in range(5)))
...
[-1, -2, -3, -4, -5]

```

Warning: this function is experimental and subject to change in a future version of joblib.

New in version 0.10.

2.5 Persistence

2.5.1 Use case

`joblib.dump()` and `joblib.load()` provide a replacement for pickle to work efficiently on arbitrary Python objects containing large data, in particular large numpy arrays.

2.5.2 A simple example

First create a temporary directory:

```
>>> from tempfile import mkdtemp
>>> savedir = mkdtemp()
>>> import os
>>> filename = os.path.join(savedir, 'test.pkl')
```

Then create an object to be persisted:

```
>>> import numpy as np
>>> to_persist = [('a', [1, 2, 3]), ('b', np.arange(10))]
```

which is saved into `filename`:

```
>>> import joblib
>>> joblib.dump(to_persist, filename)
['...test.pkl']
```

The object can then be reloaded from the file:

```
>>> joblib.load(filename)
[('a', [1, 2, 3]), ('b', array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]))]
```

2.5.3 Persistence in file objects

Instead of filenames, `dump` and `load` functions also accept file objects:

```
>>> with open(filename, 'wb') as fo:
...     joblib.dump(to_persist, fo)
>>> with open(filename, 'rb') as fo:
...     joblib.load(fo)
[('a', [1, 2, 3]), ('b', array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]))]
```

2.5.4 Compressed joblib pickles

Setting the `compress` argument to `True` in `joblib.dump()` will allow to save space on disk:

```
>>> joblib.dump(to_persist, filename + '.compressed', compress=True)
['...test.pkl.compressed']
```

If the filename extension corresponds to one of the supported compression methods, the compressor will be used automatically:

```
>>> joblib.dump(to_persist, filename + '.z')
['...test.pkl.z']
```

By default, *joblib.dump* uses the zlib compression method as it gives the best tradeoff between speed and disk space. The other supported compression methods are 'gzip', 'bz2', 'lzma' and 'xz':

```
>>> # Dumping in a gzip compressed file using a compress level of 3.
>>> joblib.dump(to_persist, filename + '.gz', compress=('gzip', 3))
['...test.pkl.gz']
>>> joblib.load(filename + '.gz')
[('a', [1, 2, 3]), ('b', array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]))]
>>> joblib.dump(to_persist, filename + '.bz2', compress=('bz2', 3))
['...test.pkl.bz2']
>>> joblib.load(filename + '.bz2')
[('a', [1, 2, 3]), ('b', array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]))]
```

Note: Lzma and Xz compression methods are only available for python versions >= 3.3.

Compressor files provided by the python standard library can also be used to compress pickle, e.g `gzip.GzipFile`, `bz2.BZ2File`, `lzma.LZMAFile`:

```
>>> # Dumping in a gzip.GzipFile object using a compression level of 3.
>>> import gzip
>>> with gzip.GzipFile(filename + '.gz', 'wb', compresslevel=3) as fo:
...     joblib.dump(to_persist, fo)
>>> with gzip.GzipFile(filename + '.gz', 'rb') as fo:
...     joblib.load(fo)
[('a', [1, 2, 3]), ('b', array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]))]
```

More details can be found in the *joblib.dump()* and *joblib.load()* documentation.

Compatibility across python versions

Compatibility of joblib pickles across python versions is not fully supported. Note that, for a very restricted set of objects, this may appear to work when saving a pickle with python 2 and loading it with python 3 but relying on it is strongly discouraged.

If you are switching between python versions, you will need to save a different joblib pickle for each python version.

Here are a few examples or exceptions:

- Saving joblib pickle with python 2, trying to load it with python 3:

```
Traceback (most recent call last):
  File "/home/lesteve/dev/joblib/joblib/numpy_pickle.py", line 453, in _
↪load
    obj = unpickler.load()
  File "/home/lesteve/miniconda3/lib/python3.4/pickle.py", line 1038, in _
↪load
    dispatch[key[0]](self)
  File "/home/lesteve/miniconda3/lib/python3.4/pickle.py", line 1176, in _
↪load_binstring
    self.append(self._decode_string(data))
  File "/home/lesteve/miniconda3/lib/python3.4/pickle.py", line 1158, in _
↪_decode_string
```

```

    return value.decode(self.encoding, self.errors)
UnicodeDecodeError: 'ascii' codec can't decode byte 0x80 in position_
↳1024: ordinal not in range(128)

Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "/home/lesteve/dev/joblib/joblib/numpy_pickle.py", line 462, in_
↳load
    raise new_exc
  ValueError: You may be trying to read with python 3 a joblib pickle_
↳generated with python 2. This is not feature supported by joblib.

```

- Saving joblib pickle with python 3, trying to load it with python 2:

```

Traceback (most recent call last):
  File "<string>", line 1, in <module>
  File "joblib/numpy_pickle.py", line 453, in load
    obj = unpickler.load()
  File "/home/lesteve/miniconda3/envs/py27/lib/python2.7/pickle.py",_
↳line 858, in load
    dispatch[key](self)
  File "/home/lesteve/miniconda3/envs/py27/lib/python2.7/pickle.py",_
↳line 886, in load_proto
    raise ValueError, "unsupported pickle protocol: %d" % proto
ValueError: unsupported pickle protocol: 3

```

2.6 Examples

2.6.1 General examples

General-purpose and introductory examples for joblib.

Checkpoint using `joblib.Memory` and `joblib.Parallel`

This example illustrates how to cache intermediate computing results using `joblib.Memory` within `joblib.Parallel`.

Embed caching within parallel processing

It is possible to cache a computationally expensive function executed during a parallel process. `costly_compute` emulates such time consuming function.

```

import time

def costly_compute(data, column):
    """Emulate a costly function by sleeping and returning a column."""
    time.sleep(2)
    return data[column]

def data_processing_mean(data, column):

```

```
"""Compute the mean of a column."""
return costly_compute(data, column).mean()
```

Create some data. The random seed is fixed to generate deterministic data across Python session. Note that this is not necessary for this specific example since the memory cache is cleared at the end of the session.

```
import numpy as np
rng = np.random.RandomState(42)
data = rng.randn(int(1e4), 4)
```

It is first possible to make the processing without caching or parallel processing.

```
start = time.time()
results = [data_processing_mean(data, col) for col in range(data.shape[1])]
stop = time.time()

print('\nSequential processing')
print('Elapsed time for the entire processing: {:.2f} s'
      .format(stop - start))
```

Out:

```
Sequential processing
Elapsed time for the entire processing: 8.01 s
```

`costly_compute` is expensive to compute and it is used as an intermediate step in `data_processing_mean`. Therefore, it is interesting to store the intermediate results from `costly_compute` using `joblib.Memory`.

```
from joblib import Memory

location = './cachedir'
memory = Memory(location=location, verbose=0)
costly_compute_cached = memory.cache(costly_compute)
```

Then, we execute the same processing in parallel and caching the intermediate results.

```
from joblib import Parallel, delayed

start = time.time()
results = Parallel(n_jobs=2)(delayed(
    data_processing_mean)(data, col) for col in range(data.shape[1]))
stop = time.time()

print('\nFirst round - caching the data')
print('Elapsed time for the entire processing: {:.2f} s'
      .format(stop - start))
```

Out:

```
First round - caching the data
Elapsed time for the entire processing: 4.93 s
```

By using 2 workers, the parallel processing gives a x2 speed-up compared to the sequential case. By executing again the same process, the intermediate results obtained by calling `costly_compute_cached` will be loaded from the cache instead of executing the function.

```
start = time.time()
results = Parallel(n_jobs=2)(delayed(
    data_processing_mean)(data, col) for col in range(data.shape[1]))
stop = time.time()

print('\nSecond round - reloading from the cache')
print('Elapsed time for the entire processing: {:.2f} s'
      .format(stop - start))
```

Out:

```
Second round - reloading from the cache
Elapsed time for the entire processing: 4.19 s
```

Reuse intermediate checkpoints

Having cached the intermediate results of the `costly_compute_cached` function, they are reusable by calling the function. We define a new processing which will take the maximum of the array returned by `costly_compute_cached` instead of previously the mean.

```
def data_processing_max(data, column):
    """Compute the max of a column."""
    return costly_compute_cached(data, column).max()

start = time.time()
results = Parallel(n_jobs=2)(delayed(
    data_processing_max)(data, col) for col in range(data.shape[1]))
stop = time.time()

print('\nReusing intermediate checkpoints')
print('Elapsed time for the entire processing: {:.2f} s'
      .format(stop - start))
```

Out:

```
Reusing intermediate checkpoints
Elapsed time for the entire processing: 4.23 s
```

The processing time only corresponds to the execution of the `max` function. The internal call to `costly_compute_cached` is reloading the results from the cache.

Clean-up the cache folder

```
memory.clear()
```

Total running time of the script: (0 minutes 21.376 seconds)

Random state within `joblib.Parallel`

Randomness is affected by parallel execution differently by the different backends.

In particular, when using multiple processes, the random sequence can be the same in all processes. This example illustrates the problem and shows how to work around it.

```
import numpy as np
from joblib import Parallel, delayed
```

A utility function for the example

```
def print_vector(vector, backend):
    """Helper function to print the generated vector with a given backend."""
    print('\nThe different generated vectors using the {} backend are:\n {}'.format(backend, np.array(vector)))
```

Sequential behavior

`stochastic_function` will generate five random integers. When calling the function several times, we are expecting to obtain different vectors. For instance, we will call the function five times in a sequential manner, we can check that the generated vectors are all different.

```
def stochastic_function(max_value):
    """Randomly generate integer up to a maximum value."""
    return np.random.randint(max_value, size=5)

n_vectors = 5
random_vector = [stochastic_function(10) for _ in range(n_vectors)]
print('\nThe different generated vectors in a sequential manner are:\n {}'.format(np.array(random_vector)))
```

Out:

```
The different generated vectors in a sequential manner are:
[[2 6 9 1 0]
 [7 6 0 9 0]
 [4 7 6 5 0]
 [2 7 2 9 7]
 [8 6 5 8 2]]
```

Parallel behavior

Joblib provides three different backend: `loky` (default), `threading`, and `multiprocessing`.

```
backend = 'loky'
random_vector = Parallel(n_jobs=2, backend=backend)(delayed(
    stochastic_function)(10) for _ in range(n_vectors))
print_vector(random_vector, backend)
```

Out:

```
The different generated vectors using the loky backend are:
[[9 6 6 9 9]
 [3 3 5 9 6]
 [2 7 4 3 6]]
```

```
[3 4 0 9 0]
[4 7 7 5 3]]
```

```
backend = 'threading'
random_vector = Parallel(n_jobs=2, backend=backend)(delayed(
    stochastic_function)(10) for _ in range(n_vectors))
print_vector(random_vector, backend)
```

Out:

```
The different generated vectors using the threading backend are:
[[4 5 3 8 3]
 [5 8 9 3 4]
 [1 4 8 1 4]
 [2 5 0 3 8]
 [8 9 7 5 8]]
```

Loky and the threading backends behave exactly as in the sequential case and do not require more care. However, this is not the case regarding the multiprocessing backend.

```
backend = 'multiprocessing'
random_vector = Parallel(n_jobs=2, backend=backend)(delayed(
    stochastic_function)(10) for _ in range(n_vectors))
print_vector(random_vector, backend)
```

Out:

```
The different generated vectors using the multiprocessing backend are:
[[5 6 1 8 3]
 [3 6 8 8 5]
 [6 1 1 6 6]
 [5 6 1 8 3]
 [3 6 8 8 5]]
```

Some of the generated vectors are exactly the same, which can be a problem for the application.

Technically, the reason is that all forked Python processes share the same exact random seed. As a result, we obtain twice the same randomly generated vectors because we are using `n_jobs=2`. A solution is to set the random state within the function which is passed to `joblib.Parallel`.

```
def stochastic_function_seeded(max_value, random_state):
    rng = np.random.RandomState(random_state)
    return rng.randint(max_value, size=5)
```

`stochastic_function_seeded` accepts as argument a random seed. We can reset this seed by passing `None` at every function call. In this case, we see that the generated vectors are all different.

```
random_vector = Parallel(n_jobs=2, backend=backend)(delayed(
    stochastic_function_seeded)(10, None) for _ in range(n_vectors))
print_vector(random_vector, backend)
```

Out:

```
The different generated vectors using the multiprocessing backend are:
[[3 8 6 9 6]
 [6 7 8 2 7]
 [5 4 3 4 5]]
```

```
[8 7 4 6 7]
[0 5 6 5 6]]
```

Fixing the random state to obtain deterministic results

The pattern of `stochastic_function_seeded` has another advantage: it allows to control the `random_state` by passing a known seed. So for instance, we can replicate the same generation of vectors by passing a fixed state as follows.

```
random_state = np.random.randint(np.iinfo(np.int32).max, size=n_vectors)

random_vector = Parallel(n_jobs=2, backend=backend)(delayed(
    stochastic_function_seeded)(10, rng) for rng in random_state)
print_vector(random_vector, backend)

random_vector = Parallel(n_jobs=2, backend=backend)(delayed(
    stochastic_function_seeded)(10, rng) for rng in random_state)
print_vector(random_vector, backend)
```

Out:

```
The different generated vectors using the multiprocessing backend are:
[[4 3 6 1 7]
 [8 2 9 2 0]
 [6 6 5 9 2]
 [3 2 2 5 7]
 [4 9 0 7 4]]

The different generated vectors using the multiprocessing backend are:
[[4 3 6 1 7]
 [8 2 9 2 0]
 [6 6 5 9 2]
 [3 2 2 5 7]
 [4 9 0 7 4]]
```

Total running time of the script: (0 minutes 1.238 seconds)

How to use `joblib.Memory`

This example illustrates the usage of `joblib.Memory` with both functions and methods.

Without `joblib.Memory`

`costly_compute` emulates a computationally expensive process which later will benefit from caching using `joblib.Memory`.

```
import time
import numpy as np

def costly_compute(data, column_index=0):
    """Simulate an expensive computation"""
```

```
time.sleep(5)
return data[column_index]
```

Be sure to set the random seed to generate deterministic data. Indeed, if the data is not deterministic, the `joblib.Memory` instance will not be able to reuse the cache from one run to another.

```
rng = np.random.RandomState(42)
data = rng.randn(int(1e5), 10)
start = time.time()
data_trans = costly_compute(data)
end = time.time()

print('\nThe function took {:.2f} s to compute.'.format(end - start))
print('\nThe transformed data are:\n {}'.format(data_trans))
```

Out:

```
The function took 5.00 s to compute.

The transformed data are:
 [ 0.49671415 -0.1382643  0.64768854  1.52302986 -0.23415337 -0.23413696
  1.57921282  0.76743473 -0.46947439  0.54256004]
```

Caching the result of a function to avoid recomputing

If we need to call our function several time with the same input data, it is beneficial to avoid recomputing the same results over and over since it is expensive. `joblib.Memory` enables to cache results from a function into a specific location.

```
from joblib import Memory
location = './cachedir'
memory = Memory(location=location, verbose=0)

def costly_compute_cached(data, column_index=0):
    """Simulate an expensive computation"""
    time.sleep(5)
    return data[column_index]

costly_compute_cached = memory.cache(costly_compute_cached)
start = time.time()
data_trans = costly_compute_cached(data)
end = time.time()

print('\nThe function took {:.2f} s to compute.'.format(end - start))
print('\nThe transformed data are:\n {}'.format(data_trans))
```

Out:

```
The function took 5.13 s to compute.

The transformed data are:
 [ 0.49671415 -0.1382643  0.64768854  1.52302986 -0.23415337 -0.23413696
  1.57921282  0.76743473 -0.46947439  0.54256004]
```

At the first call, the results will be cached. Therefore, the computation time corresponds to the time to compute the results plus the time to dump the results into the disk.

```
start = time.time()
data_trans = costly_compute_cached(data)
end = time.time()

print('\nThe function took {:.2f} s to compute.'.format(end - start))
print('\nThe transformed data are:\n {}'.format(data_trans))
```

Out:

```
The function took 0.03 s to compute.

The transformed data are:
[ 0.49671415 -0.1382643  0.64768854  1.52302986 -0.23415337 -0.23413696
 1.57921282  0.76743473 -0.46947439  0.54256004]
```

At the second call, the computation time is largely reduced since the results are obtained by loading the data previously dumped to the disk instead of recomputing the results.

Using `joblib.Memory` with a method

`joblib.Memory` is designed to work with functions with no side effects. When dealing with class, the computationally expensive part of a method has to be moved to a function and decorated in the class method.

```
def _costly_compute_cached(data, column):
    time.sleep(5)
    return data[column]

class Algorithm(object):
    """A class which is using the previous function."""

    def __init__(self, column=0):
        self.column = column

    def transform(self, data):
        costly_compute = memory.cache(_costly_compute_cached)
        return costly_compute(data, self.column)

transformer = Algorithm()
start = time.time()
data_trans = transformer.transform(data)
end = time.time()

print('\nThe function took {:.2f} s to compute.'.format(end - start))
print('\nThe transformed data are:\n {}'.format(data_trans))
```

Out:

```
The function took 5.12 s to compute.

The transformed data are:
```

```
[ 0.49671415 -0.1382643  0.64768854  1.52302986 -0.23415337 -0.23413696
 1.57921282  0.76743473 -0.46947439  0.54256004]
```

```
start = time.time()
data_trans = transformer.transform(data)
end = time.time()

print('\nThe function took {:.2f} s to compute.'.format(end - start))
print('\nThe transformed data are:\n {}'.format(data_trans))
```

Out:

```
The function took 0.05 s to compute.

The transformed data are:
[ 0.49671415 -0.1382643  0.64768854  1.52302986 -0.23415337 -0.23413696
 1.57921282  0.76743473 -0.46947439  0.54256004]
```

As expected, the second call to the `transform` method load the results which have been cached.

Clean up cache directory

```
memory.clear()
```

Total running time of the script: (0 minutes 15.502 seconds)

NumPy memmap in `joblib.Parallel`

This example illustrates some features enabled by using a memory map (`numpy.memmap`) within `joblib.Parallel`. First, we show that dumping a huge data array ahead of passing it to `joblib.Parallel` speeds up computation. Then, we show the possibility to provide write access to original data.

Speed up processing of a large data array

We create a large data array for which the average is computed for several slices.

```
import numpy as np

data = np.random.random((int(1e7),))
window_size = int(5e5)
slices = [slice(start, start + window_size)
          for start in range(0, data.size - window_size, int(1e5))]
```

The `slow_mean` function introduces a `time.sleep()` call to simulate a more expensive computation cost for which parallel computing is beneficial. Parallel may not be beneficial for very fast operation, due to extra overhead (workers creations, communication, etc.).

```
import time

def slow_mean(data, sl):
    """Simulate a time consuming processing."""
```

```
time.sleep(0.01)
return data[s1].mean()
```

First, we will evaluate the sequential computing on our problem.

```
tic = time.time()
results = [slow_mean(data, s1) for s1 in slices]
toc = time.time()
print('\nElapsed time computing the average of couple of slices {:.2f} s'
      .format(toc - tic))
```

Out:

```
Elapsed time computing the average of couple of slices 1.01 s
```

`joblib.Parallel` is used to compute in parallel the average of all slices using 2 workers.

```
from joblib import Parallel, delayed

tic = time.time()
results = Parallel(n_jobs=2)(delayed(slow_mean)(data, s1) for s1 in slices)
toc = time.time()
print('\nElapsed time computing the average of couple of slices {:.2f} s'
      .format(toc - tic))
```

Out:

```
Elapsed time computing the average of couple of slices 1.98 s
```

Parallel processing is already faster than the sequential processing. It is also possible to remove a bit of overhead by dumping the data array to a memmap and pass the memmap to `joblib.Parallel`.

```
import os
from joblib import dump, load

folder = './joblib_memmap'
try:
    os.mkdir(folder)
except FileExistsError:
    pass

data_filename_memmap = os.path.join(folder, 'data_memmap')
dump(data, data_filename_memmap)
data = load(data_filename_memmap, mmap_mode='r')

tic = time.time()
results = Parallel(n_jobs=2)(delayed(slow_mean)(data, s1) for s1 in slices)
toc = time.time()
print('\nElapsed time computing the average of couple of slices {:.2f} s\n'
      .format(toc - tic))
```

Out:

```
Elapsed time computing the average of couple of slices 0.70 s
```

Therefore, dumping large data array ahead of calling `joblib.Parallel` can speed up the processing by removing some overhead.

Writable memmap for shared memory `joblib.Parallel`

`slow_mean_write_output` will compute the mean for some given slices as in the previous example. However, the resulting mean will be directly written on the output array.

```
def slow_mean_write_output(data, sl, output, idx):
    """Simulate a time consuming processing."""
    time.sleep(0.005)
    res_ = data[sl].mean()
    print("[Worker %d] Mean for slice %d is %f" % (os.getpid(), idx, res_))
    output[idx] = res_
```

Prepare the folder where the memmap will be dumped.

```
output_filename_memmap = os.path.join(folder, 'output_memmap')
```

Pre-allocate a writable shared memory map as a container for the results of the parallel computation.

```
output = np.memmap(output_filename_memmap, dtype=data.dtype,
                  shape=len(slices), mode='w+')
```

`data` is replaced by its memory mapped version. Note that the buffer as already been dumped in the previous section.

```
data = load(data_filename_memmap, mmap_mode='r')
```

Fork the worker processes to perform computation concurrently

```
Parallel(n_jobs=2)(delayed(slow_mean_write_output)(data, sl, output, idx)
                  for idx, sl in enumerate(slices))
```

Compare the results from the output buffer with the expected results

```
print("\nExpected means computed in the parent process:\n {}".format(np.array(results)))
print("\nActual means computed by the worker processes:\n {}".format(output))
```

Out:

```
Expected means computed in the parent process:
[0.4997252  0.49985921 0.50004708 0.49976175 0.49985365 0.49999822
 0.49989174 0.49988504 0.50021788 0.50010432 0.50019335 0.50012209
 0.50031476 0.50031334 0.50077802 0.50063581 0.50068191 0.5005552
 0.50039329 0.5003834 0.50016912 0.50013128 0.49993937 0.50012462
 0.49979611 0.49987225 0.5001676 0.50026037 0.49996802 0.5003434
 0.5003142 0.49982489 0.49955003 0.49946761 0.49917905 0.49927771
 0.49983272 0.50006476 0.50021888 0.50014042 0.50003512 0.49972372
 0.49974321 0.49981815 0.49982915 0.50025461 0.50016865 0.50003413
 0.5003633 0.50033162 0.50026832 0.50047937 0.50068879 0.5004074
 0.50055791 0.50032773 0.5002506 0.49999473 0.50003967 0.5000602
 0.5003231 0.50038813 0.50039471 0.50034432 0.50028982 0.50031295
 0.49982593 0.49992445 0.49986043 0.49977769 0.49915871 0.49909062
 0.49905012 0.49917672 0.4995306 0.50005464 0.50044124 0.50024091
 0.50015131 0.49987451 0.49945097 0.49939234 0.49936171 0.49921556
 0.49906003 0.49972036 0.49978499 0.49974487 0.49981996 0.49996701
 0.49965235 0.49992389 0.5000836 0.50024789 0.50026565]
```



```
Actual means computed by the worker processes:
[0.4997252 0.49985921 0.50004708 0.49976175 0.49985365 0.49999822
0.49989174 0.49988504 0.50021788 0.50010432 0.50019335 0.50012209
0.50031476 0.50031334 0.50077802 0.50063581 0.50068191 0.5005552
0.50039329 0.5003834 0.50016912 0.50013128 0.49993937 0.50012462
0.49979611 0.49987225 0.5001676 0.50026037 0.49996802 0.5003434
0.5003142 0.49982489 0.49955003 0.49946761 0.49917905 0.49927771
0.49983272 0.50006476 0.50021888 0.50014042 0.50003512 0.49972372
0.49974321 0.49981815 0.49982915 0.50025461 0.50016865 0.50003413
0.5003633 0.50033162 0.50026832 0.50047937 0.50068879 0.5004074
0.50055791 0.50032773 0.5002506 0.49999473 0.50003967 0.5000602
0.5003231 0.50038813 0.50039471 0.50034432 0.50028982 0.50031295
0.49982593 0.49992445 0.49986043 0.49977769 0.49915871 0.49909062
0.49905012 0.49917672 0.4995306 0.50005464 0.50044124 0.50024091
0.50015131 0.49987451 0.49945097 0.49939234 0.49936171 0.49921556
0.49906003 0.49972036 0.49978499 0.49974487 0.49981996 0.49996701
0.49965235 0.49992389 0.5000836 0.50024789 0.50026565]
```

Clean-up the memmap

Remove the different memmap that we created. It might fail in Windows due to file permissions.

```
import shutil

try:
    shutil.rmtree(folder)
except: # noqa
    print('Could not clean-up automatically.')
```

Total running time of the script: (0 minutes 4.866 seconds)

2.7 Development

The homepage of joblib with user documentation is located on:

<https://pythonhosted.org/joblib/>

2.7.1 Getting the latest code

To get the latest code using git, simply type:

```
git clone git://github.com/joblib/joblib.git
```

If you don't have git installed, you can download a zip or tarball of the latest code: <http://github.com/joblib/joblib/archives/master>

2.7.2 Installing

You can use *pip* to install joblib:

```
pip install joblib
```

from any directory or:

```
python setup.py install
```

from the source directory.

Joblib has no other mandatory dependency than Python (supported versions are 2.7+ and 3.4+). Numpy (at least version 1.6.1) is an optional dependency for array manipulation.

2.7.3 Workflow to contribute

To contribute to joblib, first create an account on [github](#). Once this is done, fork the [joblib repository](#) to have your own repository, clone it using ‘git clone’ on the computers where you want to work. Make your changes in your clone, push them to your github account, test them on several computers, and when you are happy with them, send a pull request to the main repository.

2.7.4 Running the test suite

To run the test suite, you need the pytest (version ≥ 3) and coverage modules. Run the test suite using:

```
pytest joblib
```

from the root of the project.

2.7.5 Building the docs

To build the docs you need to have `setuptools` and `sphinx` (≥ 0.5) installed. Run the command:

```
pip install -U -r .readthedocs-requirements.txt
make doc-html
```

The docs are built in the `build/sphinx/html` directory.

2.7.6 Making a source tarball

To create a source tarball, eg for packaging or distributing, run the following command:

```
python setup.py sdist
```

The tarball will be created in the `dist` directory. This command will compile the docs, and the resulting tarball can be installed with no extra dependencies than the Python standard library. You will need `setuptools` and `sphinx`.

2.7.7 Making a release and uploading it to PyPI

This command is only run by project manager, to make a release, and upload in to PyPI:

```
python setup.py sdist bdist_wheel upload_docs --upload-dir build/sphinx/html
twine upload dist/*
```

2.7.8 Updating the changelog

Changes are listed in the CHANGES.rst file. They must be manually updated but, the following git command may be used to generate the lines:

```
git log --abbrev-commit --date=short --no-merges --sparse
```

Licensing

joblib is **BSD-licenced** (3 clause):

This software is OSI Certified Open Source Software. OSI Certified is a certification mark of the Open Source Initiative.

Copyright (c) 2009-2011, joblib developpers All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Gael Varoquaux. nor the names of other joblib contributors may be used to endorse or promote products derived from this software without specific prior written permission.

This software is provided by the copyright holders and contributors “as is” and any express or implied warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed. In no event shall the copyright owner or contributors be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

2.7.9 Latest changes

Development

Elizabeth Sander

Prevent numpy arrays with the same shape and data from hashing to the same memmap, to prevent jobs with preallocated arrays from writing over each other.

Olivier Grisel

Reduce overhead of automatic memmap by removing the need to hash the array.

Make `Memory.cache` robust to `PermissionError` (`errno 13`) under Windows when run in combination with `Parallel`.

The automatic array memory mapping feature of `Parallel` does no longer use `/dev/shm` if it is too small (less than 2 GB). In particular in docker containers `/dev/shm` is only 64 MB by default which would cause frequent failures when running joblib in Docker containers.

Make it possible to hint for thread-based parallelism with `prefer='threads'` or enforce shared-memory semantics with `require='sharedmem'`.

Release 0.11

Alexandre Abadie

Remove support for python 2.6

Alexandre Abadie

Remove deprecated `format_signature`, `format_call` and `load_output` functions from Memory API.

Loïc Estève

Add initial implementation of LRU cache cleaning. You can specify the size limit of a `Memory` object via the `bytes_limit` parameter and then need to clean explicitly the cache via the `Memory.reduce_size` method.

Olivier Grisel

Make the multiprocessing backend work even when the name of the main thread is not the Python default. Thanks to Roman Yurchak for the suggestion.

Karan Desai

`pytest` is used to run the tests instead of `nosetests`. `python setup.py test` or `python setup.py nosetests` do not work anymore, run `pytest joblib` instead.

Loïc Estève

An instance of `joblib.ParallelBackendBase` can be passed into the `parallel` argument in `joblib.Parallel`.

Loïc Estève

Fix handling of `mmap` objects with offsets greater than `mmap.ALLOCATIONGRANULARITY` in `joblib.Parrallel`. See <https://github.com/joblib/joblib/issues/451> for more details.

Loïc Estève

Fix performance regression in `joblib.Parallel` with `n_jobs=1`. See <https://github.com/joblib/joblib/issues/483> for more details.

Loïc Estève

Fix race condition when a function cached with `joblib.Memory.cache` was used inside a `joblib.Parallel`. See <https://github.com/joblib/joblib/issues/490> for more details.

Release 0.10.3

Loïc Estève

Fix tests when multiprocessing is disabled via the `JOBLIB_MULTIPROCESSING` environment variable.

harishmk

Remove warnings in nested Parallel objects when the inner Parallel has `n_jobs=1`. See <https://github.com/joblib/joblib/pull/406> for more details.

Release 0.10.2

Loïc Estève

FIX a bug in stack formatting when the error happens in a compiled extension. See <https://github.com/joblib/joblib/pull/382> for more details.

Vincent Latrouite

FIX a bug in the constructor of `BinaryZlibFile` that would throw an exception when passing unicode filename (Python 2 only). See <https://github.com/joblib/joblib/pull/384> for more details.

Olivier Grisel

Expose `joblib.parallel.ParallelBackendBase` and `joblib.parallel.AutoBatchingMixin` in the public API to make them officially re-usable by backend implementers.

Release 0.10.0

Alexandre Abadie

ENH: `joblib.dump/load` now accept file-like objects besides filenames. <https://github.com/joblib/joblib/pull/351> for more details.

Niels Zeilemaker and Olivier Grisel

Refactored `joblib.Parallel` to enable the registration of custom computational backends. <https://github.com/joblib/joblib/pull/306> Note the API to register custom backends is considered experimental and subject to change without deprecation.

Alexandre Abadie

Joblib pickle format change: `joblib.dump` always create a single pickle file and `joblib.dump/joblib.save` never do any memory copy when writing/reading pickle files. Reading pickle files generated with joblib versions prior to 0.10 will be supported for a limited amount of time, we advise to regenerate them from scratch when convenient. `joblib.dump` and `joblib.load` also support pickle files compressed using various strategies: `zlib`, `gzip`, `bz2`, `lzma` and `xz`. Note that `lzma` and `xz` are only available with python `>= 3.3`. <https://github.com/joblib/joblib/pull/260> for more details.

Antony Lee

ENH: `joblib.dump/load` now accept `pathlib.Path` objects as filenames. <https://github.com/joblib/joblib/pull/316> for more details.

Olivier Grisel

Workaround for “WindowsError: [Error 5] Access is denied” when trying to terminate a multiprocessing pool under Windows: <https://github.com/joblib/joblib/issues/354>

Release 0.9.4

Olivier Grisel

FIX a race condition that could cause a `joblib.Parallel` to hang when collecting the result of a job that triggers an exception. <https://github.com/joblib/joblib/pull/296>

Olivier Grisel

FIX a bug that caused `joblib.Parallel` to wrongly reuse previously memmapped arrays instead of creating new temporary files. <https://github.com/joblib/joblib/pull/294> for more details.

Loïc Estève

FIX for raising non inheritable exceptions in a `Parallel` call. See <https://github.com/joblib/joblib/issues/269> for more details.

Alexandre Abadie

FIX `joblib.hash` error with mixed types sets and dicts containing mixed types keys when using Python 3. see <https://github.com/joblib/joblib/issues/254>

Loïc Estève

FIX `joblib.dump/load` for big numpy arrays with `dtype=object`. See <https://github.com/joblib/joblib/issues/220> for more details.

Loïc Estève

FIX `joblib.Parallel` hanging when used with an exhausted iterator. See <https://github.com/joblib/joblib/issues/292> for more details.

Release 0.9.3

Olivier Grisel

Revert back to the `fork` start method (instead of `forkserver`) as the latter was found to cause crashes in interactive Python sessions.

Release 0.9.2

Loïc Estève

Joblib hashing now uses the default pickle protocol (2 for Python 2 and 3 for Python 3). This makes it very unlikely to get the same hash for a given object under Python 2 and Python 3.

In particular, for Python 3 users, this means that the output of `joblib.hash` changes when switching from `joblib 0.8.4` to `0.9.2`. We strive to ensure that the output of `joblib.hash` does not change needlessly in future versions of `joblib` but this is not officially guaranteed.

Loïc Estève

Joblib pickles generated with Python 2 can not be loaded with Python 3 and the same applies for `joblib` pickles generated with Python 3 and loaded with Python 2.

During the beta period `0.9.0b2` to `0.9.0b4`, we experimented with a `joblib` serialization that aimed to make pickles serialized with Python 3 loadable under Python 2. Unfortunately this serialization strategy proved to be too fragile as far as the long-term maintenance was concerned (For example see <https://github.com/joblib/joblib/pull/243>). That means that `joblib` pickles generated with `joblib 0.9.0bN` can not be loaded under `joblib 0.9.2`. `Joblib` beta testers, who are the only ones likely to be affected by this, are advised to delete their `joblib` cache when they upgrade from `0.9.0bN` to `0.9.2`.

Arthur Mensch

Fixed a bug with `joblib.hash` that used to return unstable values for strings and `numpy.dtype` instances depending on interning states.

Olivier Grisel

Make joblib use the 'forkserver' start method by default under Python 3.4+ to avoid causing crash with 3rd party libraries (such as Apple vecLib / Accelerate or the GCC OpenMP runtime) that use an internal thread pool that is not not reinitialized when a `fork` system call happens.

Olivier Grisel

New context manager based API (with `block`) to re-use the same pool of workers across consecutive parallel calls.

Vlad Niculae and Olivier Grisel

Automated batching of fast tasks into longer running jobs to hide multiprocessing dispatching overhead when possible.

Olivier Grisel

FIX make it possible to call `joblib.load(filename, mmap_mode='r')` on pickled objects that include a mix of arrays of both memory memmapable dtypes and object dtype.

Release 0.8.4

2014-11-20 Olivier Grisel

OPTIM use the C-optimized pickler under Python 3

This makes it possible to efficiently process parallel jobs that deal with numerous Python objects such as large dictionaries.

Release 0.8.3

2014-08-19 Olivier Grisel

FIX disable memmapping for object arrays

2014-08-07 Lars Buitinck

MAINT NumPy 1.10-safe version comparisons

2014-07-11 Olivier Grisel

FIX #146: Heisen test failure caused by thread-unsafe Python lists

This fix uses a `queue.Queue` datastructure in the failing test. This datastructure is thread-safe thanks to an internal `Lock`. This `Lock` instance not picklable hence cause the picklability check of delayed to check fail.

When using the threading backend, picklability is no longer required, hence this PRs give the user the ability to disable it on a case by case basis.

Release 0.8.2

2014-06-30 Olivier Grisel

BUG: use `mmap_mode='r'` by default in `Parallel` and `MemmappingPool`

The former default of `mmap_mode='c'` (copy-on-write) caused problematic use of the paging file under Windows.

2014-06-27 Olivier Grisel

BUG: fix usage of the `/dev/shm` folder under Linux

Release 0.8.1

2014-05-29 Gael Varoquaux

BUG: fix crash with high verbosity

Release 0.8.0

2014-05-14 Olivier Grisel

Fix a bug in exception reporting under Python 3

2014-05-10 Olivier Grisel

Fixed a potential segfault when passing non-contiguous `mmap` instances.

2014-04-22 Gael Varoquaux

ENH: Make memory robust to modification of source files while the interpreter is running. Should lead to less spurious cache flushes and recomputations.

2014-02-24 Philippe Gervais

New `Memory.call_and_shelve` API to handle memoized results by reference instead of by value.

Release 0.8.0a3

2014-01-10 Olivier Grisel & Gael Varoquaux

FIX #105: Race condition in task iterable consumption when `pre_dispatch != 'all'` that could cause crash with error messages “Pools seems closed” and “ValueError: generator already executing”.

2014-01-12 Olivier Grisel

FIX #72: joblib cannot persist “output_dir” keyword argument.

Release 0.8.0a2

2013-12-23 Olivier Grisel

ENH: set default value of `Parallel`'s `max_nbytes` to 100MB

Motivation: avoid introducing disk latency on medium sized parallel workload where memory usage is not an issue.

FIX: properly handle the `JOBLIB_MULTIPROCESSING` env variable

FIX: timeout test failures under windows

Release 0.8.0a

2013-12-19 Olivier Grisel

FIX: support the new Python 3.4 multiprocessing API

2013-12-05 Olivier Grisel

ENH: make Memory respect mmap_mode at first call too

ENH: add a threading based backend to Parallel

This is low overhead alternative backend to the default multiprocessing backend that is suitable when calling compiled extensions that release the GIL.

Author: Dan Stahlke <dan@stahlke.org> Date: 2013-11-08

FIX: use safe_repr to print arg vals in trace

This fixes a problem in which extremely long (and slow) stack traces would be produced when function parameters are large numpy arrays.

2013-09-10 Olivier Grisel

ENH: limit memory copy with Parallel by leveraging numpy.memmap when possible

Release 0.7.1

2013-07-25 Gael Varoquaux

MISC: capture meaningless argument (n_jobs=0) in Parallel

2013-07-09 Lars Buitinck

ENH Handles tuples, sets and Python 3's dict_keys type the same as lists. in pre_dispatch

2013-05-23 Martin Luessi

ENH: fix function caching for IPython

Release 0.7.0

This release drops support for Python 2.5 in favor of support for Python 3.0

2013-02-13 Gael Varoquaux

BUG: fix nasty hash collisions

2012-11-19 Gael Varoquaux

ENH: Parallel: Turn of pre-dispatch for already expanded lists

Gael Varoquaux 2012-11-19

ENH: detect recursive sub-process spawning, as when people do not protect the `__main__` in scripts under Windows, and raise a useful error.

Gael Varoquaux 2012-11-16

ENH: Full python 3 support

Release 0.6.5

2012-09-15 Yannick Schwartz

BUG: make sure that sets and dictionaries give reproducible hashes

2012-07-18 Marek Rudnicki

BUG: make sure that object-dtype numpy array hash correctly

2012-07-12 GaelVaroquaux

BUG: Bad default n_jobs for Parallel

Release 0.6.4

2012-05-07 Vlad Niculae

ENH: controlled randomness in tests and doctest fix

2012-02-21 GaelVaroquaux

ENH: add verbosity in memory

2012-02-21 GaelVaroquaux

BUG: non-reproducible hashing: order of kwargs

The ordering of a dictionary is random. As a result the function hashing was not reproducible.
Pretty hard to test

Release 0.6.3

2012-02-14 GaelVaroquaux

BUG: fix joblib Memory pickling

2012-02-11 GaelVaroquaux

BUG: fix hasher with Python 3

2012-02-09 GaelVaroquaux

API: filter_args: **args, **kwargs -> args, kwargs*

Release 0.6.2

2012-02-06 Gael Varoquaux

BUG: make sure Memory pickles even if cachedir=None

Release 0.6.1

Bugfix release because of a merge error in release 0.6.0

Release 0.6.0

Beta 3

2012-01-11 Gael Varoquaux

BUG: ensure compatibility with old numpy

DOC: update installation instructions

BUG: file semantic to work under Windows

2012-01-10 Yaroslav Halchenko

BUG: a fix toward 2.5 compatibility

Beta 2

2012-01-07 Gael Varoquaux

ENH: hash: bugware to be able to hash objects defined interactively in IPython

2012-01-07 Gael Varoquaux

ENH: Parallel: warn and not fail for nested loops

ENH: Parallel: n_jobs=-2 now uses all CPUs but one

2012-01-01 Juan Manuel Caicedo Carvajal and Gael Varoquaux

ENH: add verbosity levels in Parallel

Release 0.5.7

2011-12-28 Gael varoquaux

API: zipped -> compress

2011-12-26 Gael varoquaux

ENH: Add a zipped option to Memory

API: Memory no longer accepts save_npy

2011-12-22 Kenneth C. Arnold and Gael varoquaux

BUG: fix numpy_pickle for array subclasses

2011-12-21 Gael varoquaux

ENH: add zip-based pickling

2011-12-19 Fabian Pedregosa

Py3k: compatibility fixes. This makes run fine the tests test_disk and test_parallel

Release 0.5.6

2011-12-11 Lars Buitinck

ENH: Replace os.path.exists before makedirs with exception check New disk.mkdir will fail with other errors than EEXIST.

2011-12-10 Bala Subrahmanyam Varanasi

MISC: pep8 compliant

Release 0.5.5

2011-19-10 Fabian Pedregosa

ENH: Make joblib installable under Python 3.X

Release 0.5.4

2011-09-29 Jon Olav Vik

BUG: Make mangling path to filename work on Windows

2011-09-25 Olivier Grisel

FIX: doctest heisenfailure on execution time

2011-08-24 Ralf Gommers

STY: PEP8 cleanup.

Release 0.5.3

2011-06-25 Gael varoquaux

API: All the usefull symbols in the `__init__`

Release 0.5.2

2011-06-25 Gael varoquaux

ENH: Add `cpu_count`

2011-06-06 Gael varoquaux

ENH: Make sure memory hash in a reproducible way

Release 0.5.1

2011-04-12 Gael varoquaux

TEST: Better testing of parallel and `pre_dispatch`

Yaroslav Halchenko 2011-04-12

DOC: quick pass over docs – trailing spaces/spelling

Yaroslav Halchenko 2011-04-11

ENH: `JOBLIB_MULTIPROCESSING` env var to disable multiprocessing from the environment

Alexandre Gramfort 2011-04-08

ENH : adding log message to know how long it takes to load from disk the cache

Release 0.5.0

2011-04-01 Gael varoquaux

BUG: pickling MemoizeFunc does not store timestamp

2011-03-31 Nicolas Pinto

TEST: expose hashing bug with cached method

2011-03-26...2011-03-27 Pietro Berkes

BUG: fix error management in rm_subdirs BUG: fix for race condition during tests in mem.clear()

Gael varoquaux 2011-03-22...2011-03-26

TEST: Improve test coverage and robustness

Gael varoquaux 2011-03-19

BUG: hashing functions with only *var **kwargs

Gael varoquaux 2011-02-01... 2011-03-22

BUG: Many fixes to capture interprocess race condition when mem.cache is used by several processes on the same cache.

Fabian Pedregosa 2011-02-28

First work on Py3K compatibility

Gael varoquaux 2011-02-27

ENH: pre_dispatch in parallel: lazy generation of jobs in parallel for to avoid drowning memory.

GaelVaroquaux 2011-02-24

ENH: Add the option of overloading the arguments of the mother 'Memory' object in the cache method that is doing the decoration.

Gael varoquaux 2010-11-21

ENH: Add a verbosity level for more verbosity

Release 0.4.6

Gael varoquaux 2010-11-15

ENH: Deal with interruption in parallel

Gael varoquaux 2010-11-13

BUG: Exceptions raised by Parallel when n_job=1 are no longer captured.

Gael varoquaux 2010-11-13

BUG: Capture wrong arguments properly (better error message)

Release 0.4.5

Pietro Berkes 2010-09-04

BUG: Fix Windows peculiarities with path separators and file names
BUG: Fix more windows locking bugs

Gael varoquaux 2010-09-03

ENH: Make sure that exceptions raised in Parallel also inherit from the original exception class
ENH: Add a shadow set of exceptions

Fabian Pedregosa 2010-09-01

ENH: Clean up the code for parallel. Thanks to Fabian Pedregosa for the patch.

Release 0.4.4

Gael varoquaux 2010-08-23

BUG: Fix Parallel on computers with only one CPU, for n_jobs=-1.

Gael varoquaux 2010-08-02

BUG: Fix setup.py for extra setuptools args.

Gael varoquaux 2010-07-29

MISC: Silence tests (and hopefully Yaroslav :P)

Release 0.4.3

Gael Varoquaux 2010-07-22

BUG: Fix hashing for function with a side effect modifying their input argument. Thanks to Pietro Berkes for reporting the bug and proving the patch.

Release 0.4.2

Gael Varoquaux 2010-07-16

BUG: Make sure that joblib still works with Python2.5. => release 0.4.2

Release 0.4.1

Module reference

<code>Memory([location, backend, cachedir, ...])</code>	A context object for caching a function's return value each time it is called with the same input arguments.
<code>Parallel([n_jobs, backend, verbose, ...])</code>	Helper class for readable parallel mapping.
<code>dump(value, filename[, compress, protocol, ...])</code>	Persist an arbitrary Python object into one file.
<code>load(filename[, mmap_mode])</code>	Reconstruct a Python object from a file persisted with <code>joblib.dump</code> .
<code>hash(obj[, hash_name, coerce_mmap])</code>	Quick calculation of a hash to identify uniquely Python objects containing numpy arrays.

3.1 `joblib.Memory`

class `joblib.Memory` (*location=None, backend='local', cachedir=None, mmap_mode=None, compress=False, verbose=1, bytes_limit=None, backend_options={}*)

A context object for caching a function's return value each time it is called with the same input arguments.

All values are cached on the filesystem, in a deep directory structure.

see [Reference documentation of the Memory class](#)

__init__ (*location=None, backend='local', cachedir=None, mmap_mode=None, compress=False, verbose=1, bytes_limit=None, backend_options={}*)

Parameters **location:** str or None

The path of the base directory to use as a data store or None. If None is given, no caching is done and the Memory object is completely transparent. This option replaces `cachedir` since version 0.11.

backend: str or 'local'

Type of store backend for reading/writing cache files. Default is 'local'. The 'local' backend is using regular filesystem operations to manipulate data (open, mv, etc) in the backend.

cachedir: str or None

cachedir is deprecated since version 0.11 and will be removed in 0.13. Please consider using location option instead. The path of the base directory to use as a data store or None. If None is given, no caching is done and the Memory object is completely transparent.

mmap_mode: {None, 'r+', 'r', 'w+', 'c'}, optional

The memmapping mode used when loading from cache numpy arrays. See `numpy.load` for the meaning of the arguments.

compress: boolean, or integer

Whether to zip the stored data on disk. If an integer is given, it should be between 1 and 9, and sets the amount of compression. Note that compressed arrays cannot be read by memmapping.

verbose: int, optional

Verbosity flag, controls the debug messages that are issued as functions are evaluated.

bytes_limit: int, optional

Limit in bytes of the size of the cache.

backend_options: dict, optional

Contains a dictionary of named parameters used to configure the store backend.

Methods

<code>__init__</code> ([location, backend, cachedir, ...])	Parameters
<code>cache</code> ([func, ignore, verbose, mmap_mode])	Decorates the given function <code>func</code> to only compute its return value for input arguments not cached on disk.
<code>clear</code> ([warn])	Erase the complete cache directory.
<code>debug</code> (msg)	
<code>eval</code> (func, *args, **kwargs)	Eval function <code>func</code> with arguments <code>*args</code> and <code>**kwargs</code> , in the context of the memory.
<code>format</code> (obj[, indent])	Return the formatted representation of the object.
<code>reduce_size</code> ()	Remove cache elements to make cache size fit in <code>bytes_limit</code> .
<code>warn</code> (msg)	

3.2 joblib.Parallel

```
class joblib.Parallel(n_jobs=None, backend=None, verbose=0, timeout=None, pre_dispatch='2
* n_jobs', batch_size='auto', temp_folder=None, max_nbytes='1M',
mmap_mode='r', prefer=None, require=None)
```

Helper class for readable parallel mapping.

Parameters n_jobs: int, default: None

The maximum number of concurrently running jobs, such as the number of Python worker processes when `backend="multiprocessing"` or the size of the thread-pool when

backend="threading". If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used. None is a marker for 'unset' that will be interpreted as n_jobs=1 (sequential execution) unless the call is performed under a parallel_backend context manager that sets another value for n_jobs.

backend: str, ParallelBackendBase instance or None, default: 'loky'

Specify the parallelization backend implementation. Supported backends are:

- "loky" used by default, can induce some communication and memory overhead when exchanging input and output data with the worker Python processes.
- "multiprocessing" previous process-based backend based on *multiprocessing.Pool*. Less robust than *loky*.
- "threading" is a very low-overhead backend but it suffers from the Python Global Interpreter Lock if the called function relies a lot on Python objects. "threading" is mostly useful when the execution bottleneck is a compiled extension that explicitly releases the GIL (for instance a Cython loop wrapped in a "with nogil" block or an expensive call to a library such as NumPy).
- finally, you can register backends by calling register_parallel_backend. This will allow you to implement a backend of your liking.

It is not recommended to hard-code the backend name in a call to Parallel in a library. Instead it is recommended to set soft hints (prefer) or hard constraints (require) so as to make it possible for library users to change the backend from the outside using the parallel_backend context manager.

prefer: str in {'processes', 'threads'} or None, default: None

Soft hint to choose the default backend if no specific backend was selected with the parallel_backend context manager. The default process-based backend is 'loky' and the default thread-based backend is 'threading'.

require: 'sharedmem' or None, default None

Hard constraint to select the backend. If set to 'sharedmem', the selected backend will be single-host and thread-based even if the user asked for a non-thread based backend with parallel_backend.

verbose: int, optional

The verbosity level: if non zero, progress messages are printed. Above 50, the output is sent to stdout. The frequency of the messages increases with the verbosity level. If it more than 10, all iterations are reported.

timeout: float, optional

Timeout limit for each task to complete. If any task takes longer a TimeoutError will be raised. Only applied when n_jobs != 1

pre_dispatch: {'all', integer, or expression, as in '3*n_jobs'}

The number of batches (of tasks) to be pre-dispatched. Default is '2*n_jobs'. When batch_size="auto" this is reasonable default and the workers should never starve.

batch_size: int or 'auto', default: 'auto'

The number of atomic tasks to dispatch at once to each worker. When individual evaluations are very fast, dispatching calls to workers can be slower than sequential

computation because of the overhead. Batching fast computations together can mitigate this. The 'auto' strategy keeps track of the time it takes for a batch to complete, and dynamically adjusts the batch size to keep the time on the order of half a second, using a heuristic. The initial batch size is 1. `batch_size="auto"` with `backend="threading"` will dispatch batches of a single task at a time as the threading backend has very little overhead and using larger batch size has not proved to bring any gain in that case.

temp_folder: str, optional

Folder to be used by the pool for memmapping large arrays for sharing memory with worker processes. If None, this will try in order:

- a folder pointed by the `JOBLIB_TEMP_FOLDER` environment variable,
- `/dev/shm` if the folder exists and is writable: this is a RAM disk filesystem available by default on modern Linux distributions,
- the default system temporary folder that can be overridden with `TMP`, `TMPDIR` or `TEMP` environment variables, typically `/tmp` under Unix operating systems.

Only active when `backend="loky"` or `"multiprocessing"`.

max_nbytes int, str, or None, optional, 1M by default

Threshold on the size of arrays passed to the workers that triggers automated memory mapping in `temp_folder`. Can be an int in Bytes, or a human-readable string, e.g., `'1M'` for 1 megabyte. Use None to disable memmapping of large arrays. Only active when `backend="loky"` or `"multiprocessing"`.

mmap_mode: {None, 'r+', 'r', 'w+', 'c'}

Memmapping mode for numpy arrays passed to workers. See `'max_nbytes'` parameter documentation for more details.

Notes

This object uses workers to compute in parallel the application of a function to many different arguments. The main functionality it brings in addition to using the raw multiprocessing or `concurrent.futures` API are (see examples for details):

- More readable code, in particular since it avoids constructing list of arguments.
- **Easier debugging:**
 - informative tracebacks even when the error happens on the client side
 - using `'n_jobs=1'` enables to turn off parallel computing for debugging without changing the code-path
 - early capture of pickling errors
- An optional progress meter.
- Interruption of multiprocesses jobs with `'Ctrl-C'`
- Flexible pickling control for the communication to and from the worker processes.
- Ability to use shared memory efficiently with worker processes for large numpy-based datastructures.

Examples

A simple example:

```
>>> from math import sqrt
>>> from joblib import Parallel, delayed
>>> Parallel(n_jobs=1)(delayed(sqrt)(i**2) for i in range(10))
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

Reshaping the output when the function has several return values:

```
>>> from math import modf
>>> from joblib import Parallel, delayed
>>> r = Parallel(n_jobs=1)(delayed(modf)(i/2.) for i in range(10))
>>> res, i = zip(*r)
>>> res
(0.0, 0.5, 0.0, 0.5, 0.0, 0.5, 0.0, 0.5, 0.0, 0.5)
>>> i
(0.0, 0.0, 1.0, 1.0, 2.0, 2.0, 3.0, 3.0, 4.0, 4.0)
```

The progress meter: the higher the value of *verbose*, the more messages:

```
>>> from time import sleep
>>> from joblib import Parallel, delayed
>>> r = Parallel(n_jobs=2, verbose=10)(delayed(sleep)(.2) for _ in range(10))
[Parallel(n_jobs=2)]: Done 1 tasks | elapsed: 0.6s
[Parallel(n_jobs=2)]: Done 4 tasks | elapsed: 0.8s
[Parallel(n_jobs=2)]: Done 10 out of 10 | elapsed: 1.4s finished
```

Traceback example, note how the line of the error is indicated as well as the values of the parameter passed to the function that triggered the exception, even though the traceback happens in the child process:

```
>>> from heapq import nlargest
>>> from joblib import Parallel, delayed
>>> Parallel(n_jobs=2)(delayed(nlargest)(2, n) for n in (range(4), 'abcde', 3))
#...

-----
Sub-process traceback:
-----

TypeError                               Mon Nov 12 11:37:46 2012
PID: 12934                               Python 2.7.3: /usr/bin/python
.....
/usr/lib/python2.7/heapq.pyc in nlargest(n=2, iterable=3, key=None)
    419         if n >= size:
    420             return sorted(iterable, key=key, reverse=True)[:n]
    421
    422         # When key is none, use simpler decoration
    423         if key is None:
--> 424             it = izip(iterable, count(0,-1))           # decorate
    425                 result = _nlargest(n, it)
    426                 return map(itemgetter(0), result)       # undecorate
    427
    428         # General case, slowest method
TypeError: izip argument #1 must support iteration
```

Using `pre_dispatch` in a producer/consumer situation, where the data is generated on the fly. Note how the producer is first called 3 times before the parallel loop is initiated, and then called to generate new data on the

fly:

```
>>> from math import sqrt
>>> from joblib import Parallel, delayed
>>> def producer():
...     for i in range(6):
...         print('Produced %s' % i)
...         yield i
>>> out = Parallel(n_jobs=2, verbose=100, pre_dispatch='1.5*n_jobs')(
...     delayed(sqrt)(i) for i in producer())
Produced 0
Produced 1
Produced 2
[Parallel(n_jobs=2)]: Done 1 jobs      | elapsed:  0.0s
Produced 3
[Parallel(n_jobs=2)]: Done 2 jobs      | elapsed:  0.0s
Produced 4
[Parallel(n_jobs=2)]: Done 3 jobs      | elapsed:  0.0s
Produced 5
[Parallel(n_jobs=2)]: Done 4 jobs      | elapsed:  0.0s
[Parallel(n_jobs=2)]: Done 6 out of 6 | elapsed:  0.0s remaining:  0.0s
[Parallel(n_jobs=2)]: Done 6 out of 6 | elapsed:  0.0s finished
```

`__init__`(*n_jobs=None, backend=None, verbose=0, timeout=None, pre_dispatch='2 * n_jobs', batch_size='auto', temp_folder=None, max_nbytes='1M', mmap_mode='r', prefer=None, require=None*)

Methods

<code>__init__</code> ([<i>n_jobs, backend, verbose, ...</i>])	
<code>debug</code> (msg)	
<code>dispatch_next</code> ()	Dispatch more data for parallel processing
<code>dispatch_one_batch</code> (iterator)	Prefetch the tasks for the next batch and dispatch them.
<code>format</code> (obj[, indent])	Return the formatted representation of the object.
<code>print_progress</code> ()	Display the process of the parallel execution only a fraction of time, controlled by self.verbose.
<code>retrieve</code> ()	
<code>warn</code> (msg)	

3.3 joblib.dump

`joblib.dump`(*value, filename, compress=0, protocol=None, cache_size=None*)

Persist an arbitrary Python object into one file.

Parameters **value**: any Python object

The object to store to disk.

filename: str, pathlib.Path, or file object.

The file object or path of the file in which it is to be stored. The compression method corresponding to one of the supported filename extensions (‘.z’, ‘.gz’, ‘.bz2’, ‘.xz’ or ‘.lzma’) will be used automatically.

compress: int from 0 to 9 or bool or 2-tuple, optional

Optional compression level for the data. 0 or False is no compression. Higher value means more compression, but also slower read and write times. Using a value of 3 is often a good compromise. See the notes for more details. If `compress` is True, the compression level used is 3. If `compress` is a 2-tuple, the first element must correspond to a string between supported compressors (e.g 'zlib', 'gzip', 'bz2', 'lzma' 'xz'), the second element must be an integer from 0 to 9, corresponding to the compression level.

protocol: int, optional

Pickle protocol, see `pickle.dump` documentation for more details.

cache_size: positive int, optional

This option is deprecated in 0.10 and has no effect.

Returns filenames: list of strings

The list of file names in which the data is stored. If `compress` is false, each array is stored in a different file.

See also:

`joblib.load` corresponding loader

Notes

Memmapping on load cannot be used for compressed files. Thus using compression can significantly slow down loading. In addition, compressed files take extra extra memory during dump and load.

3.4 joblib.load

`joblib.load(filename, mmap_mode=None)`

Reconstruct a Python object from a file persisted with `joblib.dump`.

Parameters `filename`: str, `pathlib.Path`, or file object.

The file object or path of the file from which to load the object

mmap_mode: {None, 'r+', 'r', 'w+', 'c'}, optional

If not None, the arrays are memory-mapped from the disk. This mode has no effect for compressed files. Note that in this case the reconstructed object might no longer match exactly the originally pickled object.

Returns result: any Python object

The object stored in the file.

See also:

`joblib.dump` function to save an object

Notes

This function can load numpy array files saved separately during the dump. If the `mmap_mode` argument is given, it is passed to `np.load` and arrays are loaded as memmaps. As a consequence, the reconstructed object

might not match the original pickled object. Note that if the file was saved with compression, the arrays cannot be memmapped.

3.5 joblib.hash

`joblib.hash(obj, hash_name='md5', coerce_mmap=False)`

Quick calculation of a hash to identify uniquely Python objects containing numpy arrays.

Parameters `hash_name`: 'md5' or 'sha1'

Hashing algorithm used. sha1 is supposedly safer, but md5 is faster.

coerce_mmap: boolean

Make no difference between `np.memmap` and `np.ndarray`

j

joblib, 1

Symbols

`__init__()` (joblib.Memory method), 51
`__init__()` (joblib.Parallel method), 56
`__init__()` (joblib.memory.Memory method), 11

C

`cache()` (joblib.memory.Memory method), 12
`call()` (joblib.memory.MemorizedFunc method), 13
`clear()` (joblib.memory.MemorizedFunc method), 13
`clear()` (joblib.memory.Memory method), 12

D

`delayed()` (in module joblib), 23
`dump()` (in module joblib), 56

E

`eval()` (joblib.memory.Memory method), 12

H

`hash()` (in module joblib), 58

J

joblib (module), 1

L

`load()` (in module joblib), 57

M

MemorizedFunc (class in joblib.memory), 12
Memory (class in joblib), 51
Memory (class in joblib.memory), 11

P

Parallel (class in joblib), 52
`parallel_backend()` (in module joblib), 23

R

`register_parallel_backend()` (in module joblib), 23