8.3. Parallelism, resource management, and configuration

8.3.1. Parallelism

Some scikit-learn estimators and utilities parallelize costly operations using multiple CPU cores.

Depending on the type of estimator and sometimes the values of the constructor parameters, this is either done:

- with higher-level parallelism via joblib.
- with lower-level parallelism via OpenMP, used in C or Cython code.
- with lower-level parallelism via BLAS, used by NumPy and SciPy for generic operations on arrays.

The `n_jobs` parameters of estimators always controls the amount of parallelism managed by joblib (processes or threads depending on the joblib backend). The thread-level parallelism managed by OpenMP in scikit-learn's own Cython code or by BLAS & LAPACK libraries used by NumPy and SciPy operations used in scikit-learn is always controlled by environment variables or `threadpoolctl` as explained below. Note that some estimators can leverage all three kinds of parallelism at different points of their training and prediction methods.

We describe these 3 types of parallelism in the following subsections in more details.

8.3.1.1. Higher-level parallelism with joblib

When the underlying implementation uses joblib, the number of workers (threads or processes) that are spawned in parallel can be controlled via the `n_jobs` parameter.

Note: Where (and how) parallelization happens in the estimators using joblib by specifying `n_jobs` is currently poorly documented. Please help us by improving our docs and tackle issue 14228!

Joblib is able to support both multi-processing and multi-threading. Whether joblib chooses to spawn a thread or a process depends on the **backend** that it's using.

scikit-learn generally relies on the `loky` backend, which is joblib's default backend. Loky is a multi-processing backend. When doing multi-processing, in order to avoid duplicating the memory in each process (which isn't reasonable with big datasets), joblib will create a memmap that all processes can share, when the data is bigger than 1MB.

In some specific cases (when the code that is run in parallel releases the GIL), scikit-learn will indicate to `joblib` that a multi-threading backend is preferable.

As a user, you may control the backend that joblib will use (regardless of what scikit-learn recommends) by using a context manager:

```
from joblib import parallel_backend

with parallel_backend('threading', n_jobs=2):
    # Your scikit-learn code here
```

Please refer to the joblib's docs for more details.

In practice, whether parallelism is helpful at improving runtime depends on many factors. It is usually a good idea to experiment rather than assuming that increasing the number of workers is always a good thing. In some cases it can be highly detrimental to performance to run multiple copies of some estimators or functions in parallel (see oversubscription below).

8.3.1.2. Lower-level parallelism with OpenMP

OpenMP is used to parallelize code written in Cython or C, relying on multi-threading exclusively. By default, the implementations using OpenMP will use as many threads as possible, i.e. as many threads as logical cores.

You can control the exact number of threads that are used either:

- via the `OMP_NUM_THREADS` environment variable, for instance when: running a python script:

```
$ OMP_NUM_THREADS=4 python my_script.py
```

- or via `threadpoolctl` as explained by this piece of documentation.

Toggle Menu

8.3.1.3. Parallel NumPy and SciPy routines from numerical libraries

scikit-learn relies heavily on NumPy and SciPy, which internally call multi-threaded linear algebra routines (BLAS & LAPACK) implemented in libraries such as MKL, OpenBLAS or BLIS.

You can control the exact number of threads used by BLAS for each library using environment variables, namely:

- `MKL_NUM_THREADS` sets the number of thread MKL uses,
- `OPENBLAS_NUM_THREADS` sets the number of threads OpenBLAS uses
- `BLIS_NUM_THREADS` sets the number of threads BLIS uses

Note that BLAS & LAPACK implementations can also be impacted by `OMP_NUM_THREADS`. To check whether this is the case in your environment, you can inspect how the number of threads effectively used by those libraries is affected when running the the following command in a bash or zsh terminal for different values of `OMP_NUM_THREADS`:

```
.. prompt:: bash $
```

```
OMP_NUM_THREADS=2 python -m threadpoolctl -i numpy scipy
```

Note: At the time of writing (2022), NumPy and SciPy packages which are distributed on pypi.org (i.e. the ones installed via `pip install`) and on the conda-forge channel (i.e. the ones installed via `conda install --channel conda-forge`) are linked with OpenBLAS, while NumPy and SciPy packages packages shipped on the `defaults` conda channel from Anaconda.org (i.e. the ones installed via `conda install`) are linked by default with MKL.

8.3.1.4. Oversubscription: spawning too many threads

It is generally recommended to avoid using significantly more processes or threads than the number of CPUs on a machine. Over-subscription happens when a program is running too many threads at the same time.

Suppose you have a machine with 8 CPUs. Consider a case where you're running a `GridSearchCV` (parallelized with joblib) with `n_jobs=8` over a `HistGradientBoostingClassifier` (parallelized with OpenMP). Each instance of `HistGradientBoostingClassifier` will spawn 8 threads (since you have 8 CPUs). That's a total of `8 * 8 = 64` threads, which leads to oversubscription of threads for physical CPU resources and thus to scheduling overhead.

Oversubscription can arise in the exact same fashion with parallelized routines from MKL, OpenBLAS or BLIS that are nested in joblib calls.

Starting from `joblib >= 0.14`, when the `loky` backend is used (which is the default), joblib will tell its child **processes** to limit the number of threads they can use, so as to avoid oversubscription. In practice the heuristic that joblib uses is to tell the processes to use `max_threads = n_cpus // n_jobs`, via their corresponding environment variable. Back to our example from above, since the joblib backend of `GridSearchCV` is `loky`, each process will only be able to use 1 thread instead of 8, thus mitigating the oversubscription issue.

Note that:

- Manually setting one of the environment variables (`OMP_NUM_THREADS`, `MKL_NUM_THREADS`, `OPENBLAS_NUM_THREADS`, or `BLIS_NUM_THREADS`) will take precedence over what joblib tries to do. The total number of threads will be `n_jobs * <LIB>_NUM_THREADS`. Note that setting this limit will also impact your computations in the main process, which will only use `<LIB>_NUM_THREADS`. Joblib exposes a context manager for finer control over the number of threads in its workers (see joblib docs linked below).
- When joblib is configured to use the `threading` backend, there is no mechanism to avoid oversubscriptions when calling into parallel native libraries in the joblib-managed threads.
- All scikit-learn estimators that explicitly rely on OpenMP in their Cython code always use `threadpoolctl` internally to automatically adapt the numbers of threads used by OpenMP and potentially nested BLAS calls so as to avoid oversubscription.

You will find additional details about joblib mitigation of oversubscription in joblib documentation.

You will find additional details about parallelism in numerical python libraries in this document from Thomas J. Fan.

8.3.2. Configuration switches

8.3.2.1. Python API

`sklearn.set_config` and `sklearn.config_context` can be used to change parameters of the configuration which control aspect of parallelism.

8.3.2.2. Environment variables

These environment variables should be set before importing scikit-learn.

`SUME_FINITE`

Toggle Menu

Sets the default value for the `assume_finite` argument of **`sklearn.set_config`**.

`SKLEARN_WORKING_MEMORY`

Sets the default value for the `working_memory` argument of **`sklearn.set_config`**.

`SKLEARN_SEED`

Sets the seed of the global random generator when running the tests, for reproducibility.

Note that scikit-learn tests are expected to run deterministically with explicit seeding of their own independent RNG instances instead of relying on the numpy or Python standard library RNG singletons to make sure that test results are independent of the test execution order. However some tests might forget to use explicit seeding and this variable is a way to control the initial state of the aforementioned singletons.

`SKLEARN_TESTS_GLOBAL_RANDOM_SEED`

Controls the seeding of the random number generator used in tests that rely on the `global_random_seed` fixture.

All tests that use this fixture accept the contract that they should deterministically pass for any seed value from 0 to 99 included.

If the `SKLEARN_TESTS_GLOBAL_RANDOM_SEED` environment variable is set to `"any"` (which should be the case on nightly builds on the CI), the fixture will choose an arbitrary seed in the above range (based on the BUILD_NUMBER or the current day) and all fixtured tests will run for that specific seed. The goal is to ensure that, over time, our CI will run all tests with different seeds while keeping the test duration of a single run of the full test suite limited. This will check that the assertions of tests written to use this fixture are not dependent on a specific seed value.

The range of admissible seed values is limited to [0, 99] because it is often not possible to write a test that can work for any possible seed and we want to avoid having tests that randomly fail on the CI.

Valid values for `SKLEARN_TESTS_GLOBAL_RANDOM_SEED`:

- `SKLEARN_TESTS_GLOBAL_RANDOM_SEED="42"`: run tests with a fixed seed of 42
- `SKLEARN_TESTS_GLOBAL_RANDOM_SEED="40-42"`: run the tests with all seeds between 40 and 42 included
- `SKLEARN_TESTS_GLOBAL_RANDOM_SEED="any"`: run the tests with an arbitrary seed selected between 0 and 99 included
- `SKLEARN_TESTS_GLOBAL_RANDOM_SEED="all"`: run the tests with all seeds between 0 and 99 included. This can take a long time: only use for individual tests, not the full test suite!

If the variable is not set, then 42 is used as the global seed in a deterministic manner. This ensures that, by default, the scikit-learn test suite is as deterministic as possible to avoid disrupting our friendly third-party package maintainers. Similarly, this variable should not be set in the CI config of pull-requests to make sure that our friendly contributors are not the first people to encounter a seed-sensitivity regression in a test unrelated to the changes of their own PR. Only the scikit-learn maintainers who watch the results of the nightly builds are expected to be annoyed by this.

When writing a new test function that uses this fixture, please use the following command to make sure that it passes deterministically for all admissible seeds on your local machine:

```
$ SKLEARN_TESTS_GLOBAL_RANDOM_SEED="all" pytest -v -k test_your_test_name
```

`SKLEARN_SKIP_NETWORK_TESTS`

When this environment variable is set to a non zero value, the tests that need network access are skipped. When this environment variable is not set then network tests are skipped.

`SKLEARN_RUN_FLOAT32_TESTS`

When this environment variable is set to '1', the tests using the `global_dtype` fixture are also run on float32 data. When this environment variable is not set, the tests are only run on float64 data.

`SKLEARN_ENABLE_DEBUG_CYTHON_DIRECTIVES`

When this environment variable is set to a non zero value, the `Cython` derivative, `boundscheck` is set to `True`. This is useful for finding segfaults.

`SKLEARN_BUILD_ENABLE_DEBUG_SYMBOLS`

When this environment variable is set to a non zero value, the debug symbols will be included in the compiled C extensions. Only debug symbols for POSIX systems is configured.

`SKLEARN_PAIRWISE_DIST_CHUNK_SIZE`

This sets the size of chunk to be used by the underlying `PairwiseDistancesReductions` implementations. The default value is `256` which has been showed to be adequate on most machines.

Users looking for the best performance might want to tune this variable using powers of 2 so as to get the best parallelism behavior for their hardware, with respect to their caches' sizes.