adaptive-scheduler

Release 0.9.7.dev2+gf5a8204.dirty

Bas Nijholt

Oct 27, 2020
Run many adaptive.Learners on many cores (>10k) using mpi4py.futures, ipyparallel, or distributed.
The Adaptive scheduler solves the following problem, you need to run more learners than you can run with a single runner and/or can use >1k cores.

ipyparallel and distributed provide very powerful engines for interactive sessions. However, when you want to connect to >1k cores it starts to struggle. Besides that, on a shared cluster there is often the problem of starting an interactive session with ample space available.

Our approach is to schedule a different job for each adaptive.Learner. The creation and running of these jobs are managed by adaptive-scheduler. This means that your calculation will definitely run, even though the cluster might be fully occupied at the moment. Because of this approach, there is almost no limit to how many cores you want to use. You can either use 10 nodes for 1 job (learner) or 1 core for 1 job (learner) while scheduling hundreds of jobs.

Everything is written such that the computation is maximally local. This means that if one of the jobs crashes, there is no problem and it will automatically schedule a new one and continue the calculation where it left off (because of Adaptive’s periodic saving functionality). Even if the central “job manager” dies, the jobs will continue to run (although no new jobs will be scheduled.)
DESIGN GOALS

1. Needs to be able to run on efficiently >30k cores
2. Works seamlessly with the Adaptive package
3. Minimal load on the file system
4. Removes all boilerplate of working with a scheduler
   1. writes job script
   2. (re)submits job scripts
5. Handles random crashes (or node evictions) with minimal data loss
6. Preserves Python kernel and variables inside a job (in contrast to submitting jobs for every parameter)
7. Separates the simulation definition code from the code that runs the simulation
8. Maximizes computation locality, jobs continue to run when the main process dies
CHAPTER
THREE

HOW DOES IT WORK?

You create a bunch of learners and corresponding fnames such that they can be loaded, like:

```python
import adaptive
from functools import partial

def h(x, pow, a):
    return a * x**pow

combos = adaptive.utils.named_product(
    pow=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
    a=[0.1, 0.5],
)  # returns list of dicts, cartesian product of all values

learners = [adaptive.Learner1D(partial(h, **combo),
                               bounds=(-1, 1)) for combo in combos]

fnames = [f"data/{combo}" for combo in combos]
```

Then you start a process that creates and submits as many job-scripts as there are learners. Like:

```python
import adaptive_scheduler

def goal(learner):
    return learner.npoints > 200

scheduler = adaptive_scheduler.scheduler.SLURM(cores=10)  # every learner get this much cores

run_manager = adaptive_scheduler.server_support.RunManager(
    scheduler,
    learners,
    fnames,
    goal=goal,
    log_interval=30,  # write info such as npoints, cpu_usage, time, etc. to the job log file
    save_interval=300,  # save the data every 300 seconds
)
run_manager.start()
```

That's it! You can run `run_manager.info()` which will display an interactive ipywidget that shows the amount of running, pending, and finished jobs, buttons to cancel your job, and other useful information.
BUT HOW DOES IT REALLY WORK?

The RunManager basically does the following. So, you need to create N learners and fnames (like in the section above). Then a “job manager” writes and submits max(N, max_simultaneous_jobs) job scripts but doesn’t know which learner it is going to run! This is the responsibility of the “database manager”, which keeps a database of job_id <-> learner. The job script starts a Python file run_learner.py in which the learner is run.

In a Jupyter notebook we can start the “job manager” and the “database manager”, and create the run_learner.py like:

```python
import adaptive_scheduler
from adaptive_scheduler import server_support

# create a scheduler
scheduler = adaptive_scheduler.scheduler.SLURM(cores=10, run_script="run_learner.py",)

# create a new database that keeps track of job <-> learner
db_fname = "running.json"
url = (server_support.get_allowed_url())

database_manager = server_support.DatabaseManager(
    url, scheduler, db_fname, learners, fnames
)
database_manager.start()

# create the Python script that runs a learner (run_learner.py)
server_support._make_default_run_script(
    url=url,
    save_interval=300,
    log_interval=30,
    goal=None,
    executor_type=scheduler.executor_type,
    run_script_fname=scheduler.run_script,
)

# create unique names for the jobs
n_jobs = len(learners)
job_names = [f"test-job-{i}" for i in range(n_jobs)]

job_manager = server_support.JobManager(job_names, database_manager, scheduler)
job_manager.start()
```

Then when the job have been running for a while you can check server_support.parse_log_files(job_names, database_manager, scheduler).

And use scheduler.cancel(job_names) to cancel the jobs.
You don’t actually ever have to leave the Jupyter notebook, take a look at the example notebook.
CHAPTER FIVE

JUPYTER NOTEBOOK EXAMPLE

See example.ipynb.
WARNING: This is still the pre-alpha development stage.

Install the latest stable version from conda with (recommended)

```
conda install adaptive-scheduler
```

or from PyPI with

```
pip install adaptive_scheduler
```

or install master with

```
pip install -U https://github.com/basnijholt/adaptive-scheduler/archive/master.zip
```

or clone the repository and do a dev install (recommended for dev)

```
git clone git@github.com:basnijholt/adaptive-scheduler.git
cd adaptive-scheduler
pip install -e .
```
In order to not pollute the history with the output of the notebooks, please setup the git filter by executing

```
python ipynb_filter.py
```

in the repository.

We also use pre-commit, so `pip install pre_commit` and run

```
pre-commit install
```

in the repository.
Right now adaptive_scheduler is only working for SLURM and PBS, however only a class like adaptive_scheduler/scheduler.py would have to be implemented for another type of scheduler. Also there are no tests at all!

8.1 FAQ

Here is a list of questions we have either been asked by users or potential pitfalls we hope to help users avoid:

8.1.1 Q: It doesn’t work, what now?

A: Check the log-files that are created and look for an error message. If you suspect a bug in adaptive_scheduler check out run_manager.task_status() and if that doesn’t reveal anything, open an issue on GitHub.

8.1.2 Q: What if I have more learners than cores?

A: You can distribute all learners in a certain amount of adaptive.BalancingLearners. Like so

```python
from functools import partial
import adaptive
import numpy as np
from adaptive_scheduler.utils import split_in_balancing_learners, shuffle_list

def jacobi(x, n, alpha, beta):
    from scipy.special import eval_jacobi
    return eval_jacobi(n, alpha, beta, x)

combos = adaptive.utils.named_product(
    n=[1, 2, 4, 8],
    alpha=np.linspace(0, 2, 21),
    beta=np.linspace(0, 1, 21),
)

learners = [adaptive.Learner1D(partial(jacobi, **combo), bounds=(0, 1)) for combo in combos]
fnames = [f"data/jacobi/{combo}" for combo in combos]

# shuffle the learners (and fnames in the same order) because
# some learners might be slower than others (not in this example).
```

(continues on next page)
unshuffled = learners, fnames  # to have a handle to the unshuffled list
learners, fnames = shuffle_list(*unshuffled)

# split in many new BalancingLearners
# `learners` will be a list of BalancingLearners
# `fnames` will be a list of lists with fnames
learners, fnames = split_in_balancing_learners(
    learners,
    fnames,
    n_parts=100,  # split into 100 BalancingLearners
    strategy="npoints"
)

8.1.3 Q: Why aren’t my jobs dying when I cancel the job manager?  
A: The job manager just starts the jobs and you want the job to keep running in case the job manager somehow dies. So you still need to scancel or qdel them in case you want to really cancel the jobs or call adaptive_scheduler.cancel_jobs with job_names from your Python environment.

8.1.4 Q: How do I set extra SBATCH/PBS arguments or environment variables in my job script?
A: You can change this in the scheduler object. For example modifying a job script for SLURM:

```python
from adaptive_scheduler.scheduler import SLURM
scheduler = SLURM(
    cores=10,
    extra_scheduler=['--exclusive=user', '--time=1'],
    extra_env_vars=['TMPDIR=/scratch', 'PYTHONPATH=my_dir:$PYTHONPATH'],
    mpiexec_executable='srun --mpi=pmi2',
)  # pass this to `server_support.start_job_manager` or `RunManager`  

# see the job script with
print(scheduler.job_script('this_will_be_the_job_name'))
```

8.1.5 Q: My code uses MPI so the MPIToolExecutor won’t work for me, I want to use ipyparallel, how?
A: You just have to pass executor_type="ipyparallel" to SLURM or PBS. For example:

```python
from adaptive_scheduler.scheduler import SLURM

scheduler = SLURM(
    cores=48,
    executor_type="ipyparallel",
)

run_manager = adaptive_scheduler.server_support.RunManager(
    scheduler=scheduler,
    learners=learners,
    fnames=fnames,
)
```

(continues on next page)


8.1.6 Q: *ipyparallel* doesn’t work for me, I want to use *process-pool*, how?

A: Sometimes *ipyparallel* doesn’t import modules correctly on its workers. In this case you can use *process-pool*. You just have to pass `executor_type="process-pool"` to SLURM or PBS. Note the *process-pool* uses Python’s `ProcessPoolExecutor` for parallelism and cannot be used beyond a single machine (for one learner).

8.1.7 Q: Cool! What else should I check out?

A: There are a bunch of things that are not present in the example notebook, I recommend to take a look at:

- `adaptive_scheduler.utils.combo_to_fname`
- `adaptive_scheduler.server_support.cleanup`
- `adaptive_scheduler.server_support.parse_log_files`
- `adaptive_scheduler.utils.load_parallel` and `adaptive_scheduler.utils.save_parallel`

8.2 API documentation

8.2.1 General

`adaptive_scheduler.client_support` module

`adaptive_scheduler.client_support.get_learner`:

```
```

Get a learner from the database running at `url` and this learner’s process will be logged in `log_fname` and running under `job_id`.

Parameters

- `url (str)` – The url of the database manager running via `adaptive_scheduler.server_support.manage_database`.
- `log_fname (str)` – The filename of the log-file. Should be passed in the job-script.
- `job_id (str)` – The job_id of the process the job. Should be passed in the job-script.
- `job_name (str)` – The name of the job. Should be passed in the job-script.

Returns

- `learner (adaptive.BaseLearner)` – Learner that is chosen.
- `fname (str)` – The filename of the learner that was chosen.

`adaptive_scheduler.client_support.log_info`:

```
adaptive_scheduler.client_support.log_info(runner: adaptive.runner.AsyncRunner, interval=300) -> _asyncio.Task
```

Log info in the job’s logfile, similar to `runner.live_info`.

8.2. API documentation 19
Parameters

- **runner** (adaptive.Runner instance) –
- **interval** (int, default: 300) – Time in seconds between log entries.

Returns

Return type: asyncio.Task

adaptive_scheduler.client_support.tell_done(url: str, fname: str) → None

Tell the database that the learner has reached it’s goal.

Parameters

- **url** (str) – The url of the database manager running via (adaptive_scheduler.server_support.manage_database).
- **fname** (str) – The filename of the learner that is done.

adaptive_scheduler.server_support module


Bases: adaptive_scheduler.server_support._BaseManager

Database manager.

Parameters

- **url** (str) – The url of the database manager, with the format tcp://ip_of_this_machine:allowed_port. Use get_allowed_url to get a url that will work.
- **scheduler** (BaseScheduler) – A scheduler instance from adaptive_scheduler.scheduler.
- **db_fname** (str) – Filename of the database, e.g. ‘running.json’.
- **learners** (list of adaptive.BaseLearner isinstances) – List of learners corresponding to fnames.
- **fnames** (list) – List of fnames corresponding to learners.
- **overwrite_db** (bool, default: True) – Overwrite the existing database upon starting.

failed

A list of entries that have failed and have been removed from the database.

Type: list

as_dicts() → List[Dict[str, str]]

create_empty_db() → None

Create an empty database that keeps track of fname -> (job_id, is_done, log_fname, job_name).

n_done() → int

20 Chapter 8. Limitations
update (queue: Optional[Dict[str, Dict[str, str]]] = None) → None
If the job_id isn’t running anymore, replace it with None.

class adaptive_scheduler.server_support.JobManager (job_names: List[str],
database_manager: adaptive_scheduler.server_support.DatabaseManager,
scheduler: adaptive_scheduler.scheduler.BaseScheduler,
interval: int = 30, *,
max_simultaneous_jobs: int = 5000, max_fails_per_job: int = 100)

Bases: adaptive_scheduler.server_support._BaseManager

Job manager.

Parameters

• job_names (list) – List of unique names used for the jobs with the same length as learners. Note that a job name does not correspond to a certain specific learner.

• database_manager (DatabaseManager) – A DatabaseManager instance.

• scheduler (BaseScheduler) – A scheduler instance from adaptive_scheduler.scheduler.

• interval (int, default: 30) – Time in seconds between checking and starting jobs.

• max_simultaneous_jobs (int, default: 5000) – Maximum number of simultaneously running jobs. By default no more than 5000 jobs will be running. Keep in mind that if you do not specify a runner.goal, jobs will run forever, resulting in the jobs that were not initially started (because of this max_simultaneous_jobs condition) to not ever start.

• max_fails_per_job (int, default: 40) – Maximum number of times that a job can fail. This is here as a fail switch because a job might fail instantly because of a bug inside run_script. The job manager will stop when n_jobs * total_number_of_jobs_failed > max_fails_per_job is true.

n_started
Total number of jobs started by the JobManager.

Type int

property max_job_starts
Equivalent to self.max_fails_per_job * len(self.job_names)

class adaptive_scheduler.server_support.KillManager (scheduler: adaptive_scheduler.scheduler.BaseScheduler,
database_manager: adaptive_scheduler.server_support.DatabaseManager,
error: Union[str, Callable[[List[str]], bool]] = 'srun: error:', interval: int = 600, max_cancel_tries: int = 5,
move_to: Optional[str] = None)

Bases: adaptive_scheduler.server_support._BaseManager

Kill manager.

Automatically cancel jobs that contain an error (or other condition) in the log files.
Parameters

- **scheduler** *(BaseScheduler)* – A scheduler instance from adaptive_scheduler.scheduler.

- **database_manager** *(DatabaseManager)* – A DatabaseManager instance.

- **error** *(str or callable, default: "srun: error:")* – If error is a string and is found in the log files, the job will be cancelled and restarted. If it is a callable, it is applied to the log text. Must take a single argument, a list of strings, and return True if the job has to be killed, or False if not.

- **interval** *(int, default: 600)* – Time in seconds between checking for the condition.

- **max_cancel_tries** *(int, default: 5)* – Try maximum max_cancel_tries times to cancel a job.

- **move_to** *(str, optional)* – If a job is cancelled the log is either removed (if move_to=None) or moved to a folder (e.g. if move_to='old_logs').

---

**exception adaptive_scheduler.server_support.MaxRestartsReached**

**Bases:** Exception

Jobs can fail instantly because of an error in your Python code which results jobs being started indefinitely.

class adaptive_scheduler.server_support.RunManager(scheduler: adaptive_scheduler.scheduler.BaseScheduler,
learners: List[adaptive.learner.base_learner.BaseLearner],
fnames: List[str], goal: Optional[Callable[[adaptive.learner.base_learner.BaseLearner], bool]] = None,
db_fname: Optional[str] = None, overwrite_db: bool = True, job_manager_kwargs: Optional[Dict[str, Any]] = None, kill_manager_kwargs: Optional[Dict[str, Any]] = None)

**Bases:** adaptive_scheduler.server_support._BaseManager

A convenience tool that starts the job, database, and kill manager.

**Parameters**

- **scheduler** *(BaseScheduler)* – A scheduler instance from adaptive_scheduler.scheduler.
• **learners** (list of adaptive.BaseLearner instances) – List of learners corresponding to fnames.

• **fnames** (list) – List of fnames corresponding to learners.

• **goal** (callable, default: None) – The goal passed to the adaptive.Runner. Note that this function will be serialized and pasted in the run_script.

• **check_goal_on_start** (bool, default: True) – Checks whether a learner is already done. Only works if the learner is loaded.

• **runner_kwargs** (dict, default: None) – Extra keyword argument to pass to the adaptive.Runner. Note that this dict will be serialized and pasted in the run_script.

• **url** (str, default: None) – The url of the database manager, with the format tcp://ip_of_this_machine:allowed_port. If None, a correct url will be chosen.

• **save_interval** (int, default: 300) – Time in seconds between saving of the learners.

• **log_interval** (int, default: 300) – Time in seconds between log entries.

• **job_name** (str, default: "adaptive-scheduler") – From this string the job names will be created, e.g. ["adaptive-scheduler-1", "adaptive-scheduler-2", ...].

• **job_manager_interval** (int, default: 60) – Time in seconds between checking and starting jobs.

• **kill_interval** (int, default: 60) – Check for kill_on_error string inside the log-files every kill_interval seconds.

• **kill_on_error** (str or callable, default: "srun: error:") – If error is a string and is found in the log files, the job will be cancelled and restarted. If it is a callable, it is applied to the log text. Must take a single argument, a list of strings, and return True if the job has to be killed, or False if not. Set to None if no KillManager is needed.

• **move_old_logs_to** (str, default: "old_logs") – Move logs of killed jobs to this directory. If None the logs will be deleted.

• **db_fname** (str, default: "running.json") – Filename of the database, e.g. `running.json`.

• **overwrite_db** (bool, default: True) – Overwrite the existing database.

• **job_manager_kwargs** (dict, default: None) – Keyword arguments for the JobManager function that aren’t set in __init__ here.

• **kill_manager_kwargs** (dict, default: None) – Keyword arguments for the KillManager function that aren’t set in __init__ here.

**job_names**
List of job_names. Generated with self.job_name.

Type list

database_manager
The database manager.

Type DatabaseManager
job_manager
The job manager.

Type JobManager

kill_manager
The kill manager.

Type KillManager or None

start_time
Time at which self.start() is called.

Type float or None

end_time
Time at which the jobs are all done or at which self.cancel() is called.

Type float or None

Examples

Here is an example of using the RunManager with a modified job_script_function.

```python
>>> import adaptive_scheduler
>>> scheduler = adaptive_scheduler.scheduler.DefaultScheduler(cores=10)
>>> run_manager = adaptive_scheduler.server_support.RunManager(
...     scheduler=scheduler)
...).start()
```

Or an example using ipyparallel.Client.

```python
>>> from functools import partial
>>> import adaptive_scheduler
>>> scheduler = adaptive_scheduler.scheduler.DefaultScheduler(
...     cores=10, executor_type="ipyparallel",
...)
>>> def goal(learner):
...     return learner.npoints > 2000
>>> run_manager = adaptive_scheduler.server_support.RunManager(
...     scheduler=scheduler,
...     goal=goal,
...     log_interval=30,
...     save_interval=30,
...)
>>> run_manager.start()
```

cancel() → None
Cancel the manager tasks and the jobs in the queue.

cleanup(remove_old_logs_folder=False) → None
Cleanup the log and batch files.

If the RunManager is not running, the run_script.py file will also be removed.

elapsed_time() → float
Total time elapsed since the RunManager was started.

get_database() → List[Dict[str, Any]]
Get the database as a pandas.DataFrame.
info()

load_learners() → None
Load the learners in parallel using adaptive_scheduler.utils.load_parallel.

parse_log_files(only_last: bool = True) → pandas.core.frame.DataFrame
Parse the log-files and convert it to a DataFrame.

Parameters
- **only_last** (bool, default: True) – Only look use the last printed status message.

Returns df

Return type DataFrame

status() → str
Return the current status of the RunManager.

task_status() → None
Print the stack of the asyncio.Tasks.

adaptive_scheduler.server_support.cleanup(job_names: List[str], scheduler: adaptive_scheduler.scheduler.BaseScheduler, with_progress_bar: bool = True, move_to: Optional[str] = None) → None
Cleanup the scheduler log-files files.

Parameters
- **job_names** (list) – List of job names.
- **scheduler** (BaseScheduler) – A scheduler instance from adaptive_scheduler.scheduler.
- **with_progress_bar** (bool, default: True) – Display a progress bar using tqdm.
- **move_to** (str, default: None) – Move the file to a different directory. If None the file is removed.
- **log_file_folder** (str, default: '') – The folder in which to delete the log-files.

adaptive_scheduler.server_support.get_allowed_url() → str
Get an allowed url for the database manager.

Returns url – An url that can be used for the database manager, with the format tcp:// ip_of_this_machine:allowed_port..

Return type str

adaptive_scheduler.server_support.logs_with_string_or_condition(error: Union[str, Callable[[List[str]], bool]], database_manager: adaptive_scheduler.server_support.Database) → List[Tuple[str, List[str]]]
Get jobs that have string (or apply a callable) inside their log-file.

Either use string or error.
Parameters

- **error** *(str or callable)* – String that is searched for or callable that is applied to the log text. Must take a single argument, a list of strings, and return True if the job has to be killed, or False if not.

- **database_manager** *(DatabaseManager)* – A `DatabaseManager` instance.

Returns **has_string** – A list `(job_name, fnames)`, which have the string inside their log-file.

Return type `dict`

```python
adaptive_scheduler.server_support.parse_log_files
(job_names: List[str],
database_manager: adaptive_scheduler.server_support.DatabaseManager,
scheduler, only_last: bool = True)
→ pandas.core.frame.DataFrame
```

Parse the log-files and convert it to a DataFrame.

This only works if you use `adaptive_scheduler.client_support.log_info` inside your run_script.

Parameters

- **job_names** *(list)* – List of job names.

- **database_manager** *(DatabaseManager)* – A `DatabaseManager` instance.

- **scheduler** *(BaseScheduler)* – A scheduler instance from `adaptive_scheduler.scheduler`.

- **only_last** *(bool, default: True)* – Only look use the last printed status message.

Returns

Return type `DataFrame`

```python
adaptive_scheduler.server_support.periodically_clean_ipython_profiles
(scheduler, interval: int = 600)
```

Periodically remove old IPython profiles.

In the `RunManager.cleanup` method the profiles will be removed. However, one might want to remove them earlier.

Parameters

- **scheduler** *(BaseScheduler)* – A scheduler instance from `adaptive_scheduler.scheduler`.

- **interval** *(int, default: 600)* – The interval at which to remove old profiles.

Returns

Return type `asyncio.Task`
8.2. API documentation

**adaptive_scheduler.utils module**

**class adaptive_scheduler.utils.LRUCachedCallable(*args, **kwds)**

Bases: collections.abc.Callable, typing.Generic

Wraps a function to become cached.

**Parameters**

- **function** (Callable[\*, \*]) –
- **max_size** (int, optional) – Cache size of the LRU cache, by default 128.

**property cache_dict**

Returns a copy of the cache.

**adaptive_scheduler.utils.add_constant_to_fname**

(combo: Dict[str, Any], constant: Dict[str, Any], folder: Optional[Union[str, pathlib.Path]] = None, ext: Optional[str] = '.pickle', sig_figs: int = 8, dry_run: bool = True)

**adaptive_scheduler.utils.combine_sequence_learners**

(learners: List[adaptive.learner.sequence_learner.SequenceLearner], big_learner: Optional[adaptive.learner.sequence_learner.SequenceLearner] = None) → adaptive.learner.sequence_learner.SequenceLearner

Combine several SequenceLearners into a single SequenceLearner any copy over the data.

**Parameters**

- **learners** (List[SequenceLearner]) – List of SequenceLearners.
- **big_learner** (Optional[SequenceLearner]) – A learner to load, if None, a new learner will be generated.

**Returns** Big SequenceLearner with data from learners.

**Return type** adaptive.SequenceLearner

**adaptive_scheduler.utils.combo2fname**

(combo: Dict[str, Any], folder: Optional[str] = None, ext: Optional[str] = '.pickle') → str

Converts a dict into a human readable filename.

Improved version of combo_to_fname.

**adaptive_scheduler.utils.combo_to_fname**

(combo: Dict[str, Any], folder: Optional[str] = None, ext: Optional[str] = '.pickle') → str

Converts a dict into a human readable filename.

**adaptive_scheduler.utils.connect_to_ipyparallel**

(n: int, profile: str, timeout: int = 300, folder: Optional[str] = None, client_kwargs=None)

Connect to an ipcluster on the cluster headnode.

**Parameters**

- **n** (int) – Number of engines to be started.
- **profile** (str) – Profile name of IPython profile.
- **timeout** (*int*) – Time for which we try to connect to get all the engines.

- **folder** (*str, optional*) – Folder that is added to the path of the engines, e.g. "/~/.Work/my_current_project".

**Returns** `client` – An IPyParallel client.

**Return type** `ipyparallel.Client` object

```python
adaptive_scheduler.utils.copy_from_sequence_learner(learner_from: adaptive.learner.sequence_learner.SequenceLearner, learner_to: adaptive.learner.sequence_learner.SequenceLearner) → None
```

Convenience function to copy the data from a `SequenceLearner` into a different `SequenceLearner`.

**Parameters**

- **learner_from** (*adaptive.SequenceLearner*) – Learner to take the data from.

- **learner_to** (*adaptive.SequenceLearner*) – Learner to tell the data to.

```python
adaptive_scheduler.utils.hash_anything(x)
```

```python
```

Load a sequence of learners in parallel.

**Parameters**

- **learners** (sequence of `adaptive.BaseLearner`) – The learners to be loaded.

- **fnames** (sequence of `str`) – A list of filenames corresponding to `learners`.

- **with_progress_bar** (*bool*, default `True`) – Display a progress bar using `tqdm`.

- **max_workers** (*int, optional*) – The maximum number of parallel threads when loading the data. If `None`, use the maximum number of threads that is possible.

```python
adaptive_scheduler.utils.log_exception(log, msg, exception)
```

```python
adaptive_scheduler.utils.maybe_lst(fname: Union[List[str], str])
```

```python
adaptive_scheduler.utils.maybe_round(x: Any, sig_figs: int) → Any
```

```python
adaptive_scheduler.utils.round_sigfigs(num: float, sig_figs: int) → float
```

Round to specified number of sigfigs.


```python
```

Save a sequence of learners in parallel.

**Parameters**

- **learners** (sequence of `adaptive.BaseLearners`) – The learners to be saved.

- **fnames** (sequence of `str`) – A list of filenames corresponding to `learners`.

- **with_progress_bar** (*bool*, default `True`) – Display a progress bar using `tqdm`. 

28 Chapter 8. Limitations
adaptive_scheduler.utils.shared_memory_cache

Create a cache similar to `functools.lru_cache`.

This will actually cache the return values of the function, whereas `functools.lru_cache` will pickle the decorated function each time with an empty cache.

adaptive_scheduler.utils.shuffle_list

Shuffle multiple lists in the same order.

adaptive_scheduler.utils.split

Split up a sequence into n_parts.

Parameters

- `seq (sequence)` – A list or other iterable that has to be split up.
- `n_parts (int)` – The sequence will be split up in this many parts.

Returns

Return type \(\text{iterable of tuples}\)

adaptive_scheduler.utils.split_in_balancing_learners

Split a list of learners and fnames into \(\text{adaptive.BalancingLearner}\)s.

Parameters

- `learners (list)` – List of learners.
- `fnames (list)` – List of filenames.
- `n_parts (int)` – Total number of \(\text{BalancingLearner}\)s.
- `strategy (str)` – Learning strategy of the \(\text{BalancingLearner}\).

Returns

Return type \(\text{new_learners, new_fnames}\)

adaptive_scheduler.utils.split_sequence_in_sequence_learners

Split a sequence into \(\text{adaptive.SequenceLearner}\)s and fnames.

Parameters

- `function (callable)` – Function for \(\text{adaptive.SequenceLearner}\).
- `sequence (sequence)` – The sequence to split into n_learners.
- `n_learners (int)` – Total number of \(\text{SequenceLearner}\)s.
- `folder (pathlib.Path or str)` – Folder to prepend to fnames.
Returns

- **new_learners** (*List[adaptive.SequenceLearner]*) – List of *SequenceLearners*.
- **new_fnames** (*List[Path]*) – List of str based on a hash of the sequence.


Split a single *SequenceLearner* into multiple *adaptive.SequenceLearners* (with the data loaded) and fnames.

See also `split_sequence_in_sequence_learners`.

Parameters

- **big_learner** (*callable*) – A *SequenceLearner* instance
- **n_learners** (*int*) – Total number of *SequenceLearners*.
- **folder** (*pathlib.Path or str*) – Folder to prepend to fnames.

Returns

- **new_learners** (*List[adaptive.SequenceLearner]*) – List of *SequenceLearners*.
- **new_fnames** (*List[Path]*) – List of str based on a hash of the sequence.

### 8.2.2 Scheduler specific

**adaptive_scheduler.scheduler module**

```python
class adaptive_scheduler.scheduler.BaseScheduler(*args, **kwargs)
    Bases: object
    Base object for a Scheduler.
```

Parameters

- **cores** (*int*) – Number of cores per job (so per learner.)
- **run_script** (*str*) – Filename of the script that is run on the nodes. Inside this script we query the database and run the learner.
- **python_executable** (*str*, default: `sys.executable`) – The Python executable that should run the `run_script`. By default it uses the same Python as where this function is called.
- **log_folder** (*str*, default: `""`) – The folder in which to put the log-files.
- **mpiexec_executable** (*str*, optional) – `mpiexec` executable. By default `mpiexec` will be used (so probably from conda).
- **executor_type** (*str*, default: "mpi4py") – The executor that is used, by default `mpi4py.futures.MPIPoolExecutor` is used. One can use "ipyparallel", "dask-mpi", "mpi4py", or "process-pool".
- **num_threads** (*int*, default 1) – `MKL_NUM_THREADS`, `OPENBLAS_NUM_THREADS`, `OMP_NUM_THREADS`, and `NUMEXPR_NUM_THREADS` will be set to this number.
- **extra_scheduler** (*list*, optional) – Extra #SLURM (depending on scheduler type) arguments, e.g. ["--exclusive=user", "--time=1"].
• **extra_env_vars** (list, optional) – Extra environment variables that are exported in the job script. e.g. ["TMPDIR='/scratch'", "PYTHONPATH='my_dir:$PYTHONPATH'"].

• **extra_script** (str, optional) – Extra script that will be executed after any environment variables are set, but before the main scheduler is run.

Returns

Return type `BaseScheduler` object.

**batch_fname** (name: str) → str
The filename of the job script.

**cancel** (job_names: List[str], with_progress_bar: bool = True, max_tries: int = 5) → None
Cancel all jobs in `job_names`.

Parameters

• **job_names** (list) – List of job names.

• **with_progress_bar** (bool, default: True) – Display a progress bar using `tqdm`.

• **max_tries** (int, default: 5) – Maximum number of attempts to cancel a job.

**property ext**
The extension of the job script.

**property extra_env_vars**
Environment variables that need to exist in the job script.

**property extra_scheduler**
Scheduler options that go in the job script.

**property extra_script**
Script that will be run before the main scheduler.

**abstract job_script** (name: str) → str
Get a jobscript in string form.

Returns **job_script** – A job script that can be submitted to the scheduler.

Return type str

**log_fname** (name: str) → str
The filename of the log.

**output_fnames** (name: str) → List[str]

**abstract queue** (me_only: bool) → Dict[str, dict]
Get the current running and pending jobs.

Parameters **me_only** (bool, default: True) – Only see your jobs.

Returns **queue** – Mapping of `job_id` -> `dict` with name and state, for example `{job_id: {"job_name": "TEST_JOB-1", "state": "R" or "Q"}}`.

Return type dict
Notes

This function might return extra information about the job, however this is not used elsewhere in this
package.

required_attributes = ['_ext', '_submit_cmd', '_options_flag', '_cancel_cmd']

static sanitize_job_id(job_id)

start_job(name: str) → None
  Writes a job script and submits it to the scheduler.

property submit_cmd
  Command to start a job, e.g. qsub fname.batch or sbatch fname.sbatch.

write_job_script(name: str) → None

Notes

Currently, there is a problem that this will not properly cleanup. for example ipengine ... & will be
detached and go on, normally a scheduler will take care of this.

queue(me_only: bool = True) → Dict[str, dict]

Get the current running and pending jobs.

Parameters me_only (bool, default: True) – Only see your jobs.

Returns queue – Mapping of job_id -> dict with name and state, for example {job_id:
  ("job_name": "TEST_JOB-1", "state": "R" or "Q")}.

Return type dict
Notes

This function might return extra information about the job, however this is not used elsewhere in this package.

```python
start_job(name: str) → None

Writes a job script and submits it to the scheduler.
```

```python
class adaptive_scheduler.scheduler.PBS(*args, **kwargs)
Bases: adaptive_scheduler.scheduler.BaseScheduler

job_script() → str
Get a jobscript in string form.

Returns job_script – A job script that can be submitted to PBS.

Return type str
```

```python
output_fnames(name: str) → List[str]
```

```python
queue(me_only: bool = True) → Dict[str, dict]

Get the current running and pending jobs.

Parameters me_only (bool, default: True) – Only see your jobs.

Returns queue – Mapping of job_id -> dict with name and state, for example {job_id: 
("job_name": "TEST_JOB-1", "state": "R" or "Q")}.

Return type dict
```

Notes

This function might return extra information about the job, however this is not used elsewhere in this package.

```python
static sanitize_job_id(job_id)
Changes '91722.hpc05.hpc' into '91722'.
```

```python
start_job(name: str) → None

Writes a job script and submits it to the scheduler.
```

```python
class adaptive_scheduler.scheduler.SLURM(*args, **kwargs)
Bases: adaptive_scheduler.scheduler.BaseScheduler

job_script() → str
Get a jobscript in string form.

Returns job_script – A job script that can be submitted to SLURM.

Return type str
```

```python
queue(me_only: bool = True) → Dict[str, Dict[str, str]]

Get the current running and pending jobs.

Parameters me_only (bool, default: True) – Only see your jobs.

Returns queue – Mapping of job_id -> dict with name and state, for example {job_id: 
("job_name": "TEST_JOB-1", "state": "R" or "Q")}.

Return type dict
```
Notes

This function might return extra information about the job, however this is not used elsewhere in this package.

`start_job(name: str) → None`

Writes a job script and submits it to the scheduler.

8.2.3 Private API
a

adaptive_scheduler.client_support, 19
adaptive_scheduler.scheduler, 30
adaptive_scheduler.server_support, 20
adaptive_scheduler.utils, 27
Python Module Index
get_database() (adaptive_scheduler.server_support.RunManager method), 24
get_learner() (in module adaptive_scheduler.client_support), 19

H
hash_anything() (in module adaptive_scheduler.utils), 28

I
info() (adaptive_scheduler.server_support.RunManager method), 24

J
job_manager (adaptive_scheduler.server_support.RunManager attribute), 23
job_names (adaptive_scheduler.server_support.RunManager attribute), 23
job_script() (adaptive_scheduler.scheduler.BaseScheduler method), 31
job_script() (adaptive_scheduler.scheduler.LocalMockScheduler method), 32
job_script() (adaptive_scheduler.scheduler.PBS method), 33
JobManager (class in adaptive_scheduler.server_support), 21

K
kill_manager (adaptive_scheduler.server_support.RunManager attribute), 24
KillManager (class in adaptive_scheduler.server_support), 21

L
load_learners() (adaptive_scheduler.server_support.RunManager method), 25
load_parallel() (in module adaptive_scheduler.utils), 28
LocalMockScheduler (class in adaptive_scheduler.scheduler), 32
log_exception() (in module adaptive_scheduler.utils), 28
log_fname() (adaptive_scheduler.scheduler.BaseScheduler method), 31
log_info() (in module adaptive_scheduler.client_support), 19
logs_with_string_or_condition() (in module adaptive_scheduler.server_support), 25
LRUCachedCallable (class in adaptive_scheduler.utils), 27

M
max_job_starts() (adaptive_scheduler.server_support.JobManager property), 21
MaxRestartsReached, 22
maybe_lst() (in module adaptive_scheduler.utils), 28
maybe_round() (in module adaptive_scheduler.utils), 28

N
n_done() (adaptive_scheduler.server_support.DatabaseManager method), 20
n_started (adaptive_scheduler.server_support.JobManager attribute), 21
output_fnames() (adaptive_scheduler.scheduler.BaseScheduler method), 31
output_fnames() (adaptive_scheduler.scheduler.PBS method), 33

O
parse_log_files() (adaptive_scheduler.server_support.RunManager method), 25
parse_log_files() (in module adaptive_scheduler.server_support), 26
PBS (class in adaptive_scheduler.scheduler), 33
periodically_clean_ipython_profiles() (in module adaptive_scheduler.server_support), 26

Q
queue() (adaptive_scheduler.scheduler.BaseScheduler method), 31
queue() (adaptive_scheduler.scheduler.LocalMockScheduler method), 32
queue() (adaptive_scheduler.scheduler.PBS method), 33
<table>
<thead>
<tr>
<th>Function/Attribute</th>
<th>Module/Class</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue()</td>
<td>adaptive_scheduler.scheduler.SLURM method</td>
<td>33</td>
</tr>
<tr>
<td>R</td>
<td>required_attributes (adaptive_scheduler.scheduler.BaseScheduler attribute)</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>round_sigfigs() (in module adaptive_scheduler.utils)</td>
<td>28</td>
</tr>
<tr>
<td>RunManager</td>
<td>(class in adaptive_scheduler.server_support)</td>
<td>22</td>
</tr>
<tr>
<td>S</td>
<td>sanatize_job_id() (adaptive_scheduler.scheduler.BaseScheduler static method)</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>save_parallel() (in module adaptive_scheduler.utils)</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>shared_memory_cache() (in module adaptive_scheduler.utils)</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>shuffle_list() (in module adaptive_scheduler.utils)</td>
<td>29</td>
</tr>
<tr>
<td>SLURM class</td>
<td>in adaptive_scheduler.scheduler</td>
<td>33</td>
</tr>
<tr>
<td>split()</td>
<td>(in module adaptive_scheduler.utils)</td>
<td>29</td>
</tr>
<tr>
<td>split_in_balancing_learners()</td>
<td>(in module adaptive_scheduler.utils)</td>
<td>29</td>
</tr>
<tr>
<td>split_sequence_in_sequence_learners()</td>
<td>(in module adaptive_scheduler.utils)</td>
<td>29</td>
</tr>
<tr>
<td>split_sequence_learner()</td>
<td>(in module adaptive_scheduler.utils)</td>
<td>30</td>
</tr>
<tr>
<td>start_job()</td>
<td>(adaptive_scheduler.scheduler.BaseScheduler method)</td>
<td>32</td>
</tr>
<tr>
<td>start_job()</td>
<td>(adaptive_scheduler.scheduler.LocalMockScheduler method)</td>
<td>33</td>
</tr>
<tr>
<td>start_job()</td>
<td>(adaptive_scheduler.scheduler.PBS method)</td>
<td>33</td>
</tr>
<tr>
<td>start_job()</td>
<td>(adaptive_scheduler.scheduler.SLURM method)</td>
<td>34</td>
</tr>
<tr>
<td>start_time</td>
<td>(adaptive_scheduler.server_support.RunManager attribute)</td>
<td>24</td>
</tr>
<tr>
<td>status()</td>
<td>(adaptive_scheduler.server_support.RunManager method)</td>
<td>25</td>
</tr>
<tr>
<td>submit_cmd()</td>
<td>(adaptive_scheduler.scheduler.BaseScheduler property)</td>
<td>32</td>
</tr>
<tr>
<td>T</td>
<td>task_status() (adaptive_scheduler.server_support.RunManager)</td>
<td>39</td>
</tr>
</tbody>
</table>