# clusterjob Documentation

***Release 2.0.0***

**Michael Goerz**

**Mar 10, 2017**

# Contents


**1 Introduction** **3**

**2 Model** **5**
2.1 Parallelization Paradigms . . . . . 6

**3 Alternatives** **9**

**4 The clusterjob API** **11**

**5 Tutorial** **13**

**6 Using Configuration Files** **15**

**7 Adding Backends** **17**

**8 Testing** **19**

**9 Indices and tables** **21**

clusterjob is a Python library to manage workflows on traditional HPC cluster systems.

Contents:

# CHAPTER 1

# Introduction

Workflows for scientific computing center around writing scripts for a job scheduler, such as SLURM or TORQUE/PBS, on a high-performance-computing (HPC) cluster. The clusterjob package moves this paradigm into the Python ecosystem. It provides an abstraction of the *common model* underlying the various different scheduling systems. From inside a Python script, cluster jobs can be defined in a way that is agnostic about the specific cluster or scheduling system that the job will be run under. The job can then be submitted to a scheduler (either locally or remotely), and the state of the job can be tracked asynchronously.

The goals of the clusterjob package are *reproducibility*, *robustness*, and *flexibility*:

- Allow defining a complete computing workflow from within Python. By scripting all interactions with the scheduler instead of submitting jobs "manually", reproducibility is ensured.
- Keep calculations together with data pre-/post-processing, analysis and plotting, leveraging the entire scientific Python stack. The Jupyter notebook is a great environment for tying together these different aspects of a project.
- Robustness against any kind of crash or network disconnect. By caching information about submitted jobs, a workflow script can be aborted and rerun at any time, continuing where it left off. The intent is to manage a long-running set of calculations on a cluster from e.g. a laptop computer.
- Aid in separating the calculation workflow from the specifics of a particular cluster/scheduling system. The clusterjob package can read all backend information and resource requirements from *INI-style text files*. This allows to easily port an existing computing workflow to a different cluster/scheduling system.

CHAPTER 2

# Model

The *clusterjob* package is based on a simplified model that generalizes the concepts of various HPC scheduling systems (such as SLURM, PBS, or LSF).

**Job Script** A job script is a shell script with an associated set of properties, which include resource requirements, and backend (scheduler) information. In the *clusterjob* package, job scripts are represented by the `clusterjob.JobScript` class. The package encourages the separation of a computational workflow from the specifics of a scheduler, via the following three mechanisms.

First, while the properties of a jobs script can in general be arbitrary, scheduler-specific keywords, there is a set of common resource properties. The backend will automatically translate these properties to appropriate options for a given scheduling system. This is reflected especially in the simplified *parallelization model* that the *clusterjob* package uses.

Second, the body of the job script may commonly refer to environment variables that are set by the scheduler as it runs the job script. Different schedulers usually have similar variables, but use different names. For example, the job ID or a running job is available as `$SLURM_JOB_ID` when using a SLURM scheduler, and `$PBS_JOBID` when using a PBS scheduler. The body of a *clusterjob* job script may instead use a core set of environment variables (e.g. `$CLUSTERJOB_ID` for the job ID). The backend will replace these variables with the equivalent variable for the given scheduler.

Third, when rendering the job script for execution (`clusterjob.JobScript.render_script()`), placeholders in the body of the job script will be replaced using the properties of the job script. This allows for a high degree of flexibility, as (1) arbitrary properties may be attached to a job script and (2) all properties can be kept separate from the code in *INI files* and thus easily be adapted to new or changing cluster environments. Job scripts can have associated prologue and epilogue scripts for local pre- and post-processing, as well as auxiliary scripts, which are all rendered through the above mechanism.

**Backend** A backend is the collective information required to submit a job script to a *specific* scheduler. This includes information about which commands must be used to submit and manage job scripts, how resource requirements must be encoded, and what environment variables are defined by the scheduler.

Backends are implemented as instances of `ClusterjobBackend`, with subclasses for different scheduling systems. By further sub-classing, it is easy to create more specialized backends for specific clusters, translating the simplified model encoded in the job script properties to arbitrarily complex specifications.

**Scheduler** A scheduler is a software running on a cluster login node that accepts job script submissions and runs the job script on some compute nodes, taking into account resource constraints. Schedulers that *clusterjob* can interact with (provided the appropriate backend has been implemented) must meet the following requirements:

- The scheduler can express all the resource requirements expressed in the job script properties, usually through resource specifications in the header of a submission script or through the command line options of a submission command.
- The scheduler generates a job ID as soon as a job script is submitted. The job IDs must be unique within the uptime of the scheduler.
- The scheduler has a command line utilities for submitting jobs, querying their status (given a job ID), and canceling running jobs
- The scheduler should define environment variables equivalent to the clusterjob core environment variables.

**Run** A Run is the result of submitting a job script to a specific scheduler. In the *clusterjob* package, a run is represented by the `AsyncResult` class. This class provides a superset of the interface in `multiprocessing.pool.AsyncResult`. It is also deliberately similar to the `ipyparallel.AsyncResult` class.

The `AsyncResult` instance maintains all the required information to communicate with the scheduler about the status of the job. The *clusterjob* package can be configured to automatically cache all `AsyncResult` do hard disk. This allows to recover from an interruption of the Python scripts, and prevents submitting the same job multiple times. If a job is submitted for which there exists a cache file, the cached information is loaded and returned, instead of re-submitting.

## Parallelization Paradigms

The common resource properties of a `JobScript` instance include *nodes*, *ppn* (processes per node), and *threads* (per process). It is instructive to see how these terms relate to various common parallelization paradigms, and how they compare to the corresponding resource specification for some of the scheduling systems (SLURM, PBS Pro, PBS/TORQUE, LSF). To make the discussion concrete, we assume a cluster that consists of homogeneous nodes with 32 CPU cores each.

**Multi-process parallelization** In the multi-process paradigm, multiple copies of the same program are run as independent processes, exchanging data via message passing (MPI). Each process runs single-threaded. Assuming we want to run a total of 64 processes, the appropriate specification would be:

```
nodes=2, ppn=32, threads=1
```

This maps to the different schedulers as follows:

- SLURM: `--nodes=2 --tasks-per-node=32 --cpus-per-task=1`
- PBS Pro: `-l select=2:ncpus=32:mpiprocs=32:ompthreads=1`
- PBS/TORQUE: `-l nodes=2:ppn=32`
- LSF: `-n64 -R "span[ptile=32]"`

Note that in principle, since MPI processes are fully independent, manually splitting the 64 MPI processes into 2 nodes and 32 processes per node could be seen as overly specific. Therefore, a custom backend could ignore the *nodes* specification, and distribute the total number of processes over an arbitrary number of nodes, based on availability. For example, in SLURM, `--tasks=64 --cpus-per-task=1`, or `-n 64 -c 1` would suffice, and for LSF, `-R "span[ptile=32]"` could be left out. However, a backend using such a mapping would be suitable only for pure MPI, and ignoring the *nodes* specification might interfere with other paradigms.

**Multi-threaded, shared memory parallelization (OpenMP)** In multi-threaded parallelization, there is a single process, running on a single node, but spawning multiple threads (or subprocesses). Communication between the

threads is through the shared memory, using OpenMP. For using 32 threads (i.e. all a node's cores), the specification would be:

```
nodes=1, ppn=1, threads=32
```

For different schedulers, this corresponds to

- SLURM: `-n1 -c32`
- PBS Pro: `-l select=1:ncpus=32:mpiprocs=1:ompthreads=32`
- PBS/TORQUE: `-l nodes=1:ppn=32`
- LSF: `-n32 -R "span[ptile=32]"`, or `-n32 -R "span[hosts=1]"`

In the run script body, the environment variable `$OMP_NUM_THREADS` should be set to `{threads}`. Note that that the *ppn* parameter used in PBS/TORQUE specifies the total number of cores used on the node, and thus differs from the definition of *ppn* in *clusterjob* (where the total number of cores used on a node is always `ppn*threads`).

It depends on the configuration of the scheduler whether *threads* can be less than the number of cores on a physical nodes. The scheduler may require that each job fills complete nodes, or it may assign different jobs to the same physical node if they use less than the full number of cores.

**Hybrid parallelization** MPI-based and OpenMP based parallelization may be combined to run an arbitrary number of MPI processes, each spawning several OpenMP threads. For example, having a total of 8 MPI processes running that start 16 threads each, implies:

```
nodes=4, ppn=2, threads=16
```

For the different schedulers, this corresponds to

- SLURM: `--nodes=4 --tasks-per-node=2 --cpus-per-task=16`, or `-n8 -c16`
- PBS Pro: `-l select=4:ncpus=32:mpiprocs=2:ompthreads=16`
- PBS/TORQUE: `-l nodes=4:ppn=32`
- LSF: `-n 128 -R "span[ptile=32]"`

**Embarrassingly parallel workloads** In the "embarrassingly parallel" paradigm, we run multiple processes of the same program with different parameters, *without* communication between the processes. There are several ways of realizing this:

- Use a MPI-based wrapper script (mpi4py is especially useful for this), request resources as for any MPI job, as described above.
- Use the linux utility `xargs`. See Process Pools in Bash. This is limited to running on a single node. Resources are best requested as for a multi-threaded job.
- Use job arrays, where multiple copies of the same job script are run with an index counter stored in an environment variable.

CHAPTER 3

# Alternatives

- clusterlib
- joblib
- ipyparallel

CHAPTER 4

# The clusterjob API

The `clusterjob` package provides the following two classes:

- **`JobScript`** Encapsulation of a Jobscript
- **`AsyncResult`** Encapsulation of a Run, i.e., a submitted Jobscript

The package contains two sub-modules:

- **`clusterjob.utils`** Collection of utility function
- **`clusterjob.status`** Definition of status codes

The default backends are defined in the `clusterjob.backends` sub-package

CHAPTER 5

# Tutorial

CHAPTER 6

# Using Configuration Files

CHAPTER 7

# Adding Backends

CHAPTER 8

# Testing

CHAPTER 9

# Indices and tables

- genindex
- modindex

# Index


## B

Backend, **5**

## E

Embarrassingly parallel workloads, **7**

## H

Hybrid parallelization, **7**

## J

Job Script, **5**

## M

Multi-process parallelization, **6**
Multi-threaded, shared memory parallelization (OpenMP), **6**

## R

Run, **6**

## S

Scheduler, **6**