payu Documentation

Release 0+untagged.126.gd7fdc5a.dirty

Marshall Ward

Jan 23, 2024

CONTENTS

Current release: 0+untagged.126.gd7fdc5a.dirty

Payu is a workflow management tool for running numerical models in supercomputing environments.

Payu was designed to allow users to start running climate models immediately, without having to re-learn the nuances of countless runscripts across countless models. Running a model like the MOM ocean model should only require a few commands:

```
mkdir new_expt; cd new_expt
payu init -m mom
payu run
```

Currently, payu is very highly customised for users of NCI computing environments, with a very strong dependence on environment modules, the PBS scheduler, and an MPI runtime environment. Using payu on other machines will require, at a minimum, the installation of these services, as well as a potentially significant modification of the codebase.

CHAPTER

ONE

USER GUIDE

Contents:

1.1 Installation

Payu is currently only supported for users on the NCI computing systems, such as Raijin. If you wish to use payu on other systems, see the notes at the end of this document.

1.1.1 NCI Users

Payu is available for all users on Raijin.

To load payu, load the environment module:

```
module load payu
```

Local installation

If you want to use the latest version of payu, then you can install it locally from the codebase:

```
git clone https://github.com/marshallward/payu
cd payu
python setup.py install --user
```

Payu depends on the following modules:

- f90nml
- PyYAML

If you have a recent version of `setuptools` installed, then these will be installed automatically. But older installations, such as based on `distutils`, will require a manual installation.

1.1.2 General Use

Payu is not supported for general use, and it would be a tremendous surprise if it even worked on other machines. In particular, the following services are presumed to be available:

- Environment Modules: Not only do we assume support for environment modules, but we also assume the existence of certain modules, such as an OpenMPI module and particular versions of Python.
- PBS Scheduler: Payu relies on executables that are provided with most PBS implementations, such as Torque or PBSPro. Most of the argument flags are currently based around PBSPro conventions.
- MPI: Jobs are submitted via `mpirun` and most of the argument flags are based on the OpenMPI implementation. We also rely on Raijin's internal preprocessing scripts for a few tasks.

There are also some additional assumptions based on the architecture of Raijin.

Despite these rather strict requirements, there is opportunity for generalising payu for other platforms, such as through new drivers for alternative schedulers and parallelisation platforms. Please contact the mailing list if you are interested in porting payu to your machine.

1.2 Usage

This document outlines the basic procedure to setup and run an experiment with payu.

1.2.1 Overview

The general layout of a payu-supported experiment consists of two directories:

- The *laboratory*, which contains the executable, input files, actively running experiments, and archived model output, and the
- The *control directory*, where the experiment is configured and run.

This separation allows us to run multiple self-resubmitting experiments simultaneously that can share common executables and input data. It also allows the flexibility to have the relatively small control directories in a location that is continuously backed up.

Using a git repository for the experiment

It is recommended to use the git version control system for the payu *control directory*. This allows the experiment to be easily copied via cloning. There is inbuilt support in payu for an experiment runlog which uses git to track changes to configuration files between experiment runs. There are payu commands for creating and moving between git branches so multiple related experiments can be run from the same control directory.

1.2.2 Setting up the laboratory

Before running an experiment, you must first set up the *laboratory* for the associated numerical model if it does not already exist.

First, check the list of supported models:

```
payu list
```

This shows the keyword for each supported model.

Automatic setup

To initialise the model laboratory, type:

```
payu init -m model
```

where `model` is the model name from `payu list`. This will create the laboratory directory tree.

Automatic compilation of models is no longer supported.

Manual setup

If the automated approach does not work you will have to set up the laboratory manually.

1. Create a directory for the laboratory to reside. The default directory path is shown below:

   ```
   mkdir -p /scratch/${PROJECT}/${USER}/${MODEL}
   ```

   where `${MODEL}` is from the list of supported models. For example, if your username is `abc123` and your default project is `v45`, then the default laboratory directory for the MOM ocean model would be `/scratch/v45/abc123/mom`.

2. Create subdirectories for the model binaries and input fields:

   ```
   cd /scratch/${PROJECT}/${USER}/${MODEL}
   mkdir bin input
   ```

Populate laboratory directories

1. Compile a model and copy its executable into the `bin` directory in the laboratory:

   ```
   cp /path/to/exec bin/exec
   ```

   You will want to give the executable a unique name.

2. Create or gather any input data files into an subdirectory in the input directory in the laboratory:

   ```
   mkdir input/my_data
   cp /path/to/data input/my_data/
   ```

   You will want a unique name for each input directory.

Clone experiment

Cloning is the best way to copy an experiment as it guarantees that only the required files are copied to a new control directory, and maintains a link to the original experiment through the shared git history. To clone the repository, you can use `payu clone`. This is a wrapper around `git clone` which additionally creates or updates the metadata file which gets copied to the experiment archive directory (see *Metadata and Related Experiments*). For example:

```
mkdir -p ${HOME}/${MODEL}
cd ${HOME}/${MODEL}
payu clone ${REPOSITORY} my_expt
cd my_expt
```

Where `${REPOSITORY}` is the git URL or path of the repository to clone from, for example, https://github.com/payu-org/mom-example.git.

To clone and checkout an existing git branch, use the `--branch` flag and specify the branch name:

```
payu clone --branch ${EXISTING_BRANCH} ${REPOSITORY} my_expt
```

To create and checkout a new git branch use `--new-branch` and specify a new branch name:

> payu clone –new-branch ${NEW_BRANCH} ${REPOSITORY} my_expt

To see more configuration options for `payu clone`, run:

```
payu clone --help
```

As an alternative to creating and checking out branches with `payu clone`, `payu checkout` can be used instead (see *Metadata and Related Experiments*).

Create experiment

If a suitable experiment does not already exist it will have to be created manually:

1. Return to the home directory and create a *control directory*:

   ```
   mkdir -p ${HOME}/${MODEL}/my_expt
   cd ${HOME}/${MODEL}/my_expt
   ```

   Although the example control directory here is in the user's home directory, they can be placed anywhere and there is no predefined location.

2. Populate the control directory.

   Copy any input text files in the control directory:

   ```
   cp /path/to/configs ${HOME}/${MODEL}/my_expt
   ```

   Configure the experiment in a `config.yaml` file, such as the one shown below for MOM:

   ```
   # Scheduler settings
   queue: normal
   ncpus: 1
   walltime: 10:00
   jobname: bowl1

   # Model settings
   model: mom
   shortpath: /scratch/v45
   exe: fms_MOM_solo.x
   input: bowl1

   # Postprocessing
   collate:
       walltime: 10:00
       mem: 1GB
   ```

   See the *Configuring your experiment* section for more details.

1.2.3 Running your experiment

Once the laboratory has been created and the experiment has been configured, as an optional step you can check that the paths have been correctly specified by running:

```
payu  setup
```

This creates the temporary `work` directory and is done automatically when the model is run. If there any errors in the configuration, such as incorrect or missing paths, these can be fixed. `payu` will not run the model if there is an existing `work` directory, so this must be removed (see *Cleaning up*).

The `setup` command will also generate manifest files in the `manifest` directory. The manifest files track the executable, input and restart files used in each run. When running at NCI the manifest file must be present as it is scanned for storage points in order to correctly specify the argument to the ``-l storage=`` option when submitting a PBS job.

It is possible to create an experiment configuration such that the input and executable manifests are correct if the experiment is run on the same system. In such a case the `manifest` options need to be set correctly to always reuse those manifests and it should be possible to run the experiment immediately.

Once you are satisfied the configuration is correct, and there is no existing ``work`` directory, run the experiment by typing the following:

```
payu run
```

This will run the model once and store the output in the ``archive`` directory.

Optionally if there is an existing `work` directory the `-f/--force` flag will automatically `sweep` any existing `work` directory:

```
payu run -f
```

To continue the simulation from its last point, type `payu run` again.

In order to schedule `N` successive runs, use the `-n` flag:

```
payu run -n N
```

If there are no archived runs, then the model will initialise itself. If the model has been run `K` times, then it will continue from this point and run `N` more jobs.

If you need to run (or re-run) the `K`th job, rather than the most recent run, use the `-i` flag:

```
payu run -i K
```

Note that job numbering is 0-based, so that the first run is 0, the second run is 1, and so on.

Running jobs are stored in laboratory's `work` subdirectory, and completed runs are stored in the `archive` subdirectory.

If you have instructed `payu` to run for a number of resubmits but for some reason need to stop a run after the current run has completed create a file called `stop_run` in the control directory.

It is possible to require that a run reproduce an existing run using the `-r/--reproduce` flag:

> payu run -r

When this invoked all the manifests are read in and hashes checked for consistency and only if all executables, inputs and restart files are unchanged will the run proceed. As the restart files are read directly from the manifests which are written before the previous run completed, by definition a restart run will not look for or use any restart files that are more recent.

The reproduce option can be useful to be able to re-run a simulation for the purposes of checking reproducibility when compute infrastructure changes, or when spinning off a perturbation run to ensure consistency with a control run before applying modifications.

To run from an existing model run, also called a warm start, set the `restart` option to point to the folder containing the restart files from a previous matching experiment.

If restart pruning configuration has changed, there may be warnings if many restarts will be pruned as a result. If this is desired, at the next run use `-F/--force-prune-restarts` flag:

> payu run –force-prune-restarts

1.2.4 Cleaning up

If you experiment crashes or fails for any reason, then payu will usually abort and keep any remaining files in the `work` and control directories.

To clean up a failed job and prepare it for resubmission, use the `sweep` command:

```
payu sweep
```

This will delete the contents of `work` and move any model and scheduler logs into a `pbs_logs` directory. Any model output in `archive` will not be deleted.

Deleting an experiment archive

If you also want to delete all runs from an experiment in the `archive`, use the `--hard` flag:

```
payu sweep --hard
```

This will delete your runs and can potentially erase months of work, so use it with caution.

Hard sweeps will only delete the run output for your particular experiment. Other experiment runs will not be harmed by this command.

1.2.5 Postprocessing

Model output in parallel jobs is sometimes divided across several files, which can be inconvenient for analysis. Payu offers a `collate` subcommand to collate these separated files into a single file. This is only necessary, and supported, for some models.

For most jobs, collation is called automatically. But if you need to manually collate output from run `K`, type the following:

```
payu collate -i K
```

This will also collate restart `K-1` if `restart:  true` in the `collate` section of the configuration file.

Alternatively you can directly specify a directory name:

```
payu collate -d dir_name
```

This is useful when the data files have been moved out of the payu directory structure, or if you need to collate restart files, which is necessary when changing processor layout.

To manually sync experiment output files to a remote archive, firstly ensure that `path` in the `sync` namespace in `config.yaml`, is correctly configured as it may overwrite any pre-exisiting outputs. Then run:

```
payu sync
```

By default `payu sync` will not sync the latest restarts that may be pruned at a later date. To sync all restarts including the latest restarts, use the `--sync-restarts` flag:

```
payu sync  --sync-restarts
```

1.2.6 Metadata and Related Experiments

Metadata files

Each experiment has a metadata file, called `metadata.yaml` in the *control directory*. This contains high-level metadata about the experiment and uses the ACCESS-NRI experiment schema. An important field is the `experiment_uuid` which uniquely identifies the experiment. Payu generates a new UUID when:

- Using payu to clone a pre-existing git repository of the *control directory*
- Using payu to create and checkout a new git branch in the *control directory*
- Or, when setting up an experiment run if there is not a pre-existing metadata file, UUID, or experiment `archive` directory.

For new experiments, payu may generate some additional metadata fields. This includes an experiment name, creation date, contact, and email if defined in the git configuration. This also includes parent experiment UUID if starting from restarts and the experiment UUID is defined in metadata of the parent directory containing the restart.

Once a metadata file is created or updated, it is copied to the directory that stores the archived experiment outputs.

Experiment names

An experiment name is used to identify the experiment inside the `work` and `archive` sub-directories inside the *laboratory*.

The experiment name historically would default to the name of the *control directory*. This is still supported for experiments with pre-existing archived outputs. To support git branches and ensure uniqueness in shared archives, the new default behaviour is to add the branch name and a short version of the experiment UUID to the name of the *control directory* when creating experiment names.

For example, given a control directory named `my_expt` and a UUID of `416af8c6-d299-4ee6-9d77-4aefa8a9ebcb`, the experiment name would be:

- `my_expt-perturb-416af8c6` - if running an experiment on a branch named `perturb`.
- `my_expt-416af8c6` - if the control directory was not a git repository or experiment was run from the `main` or `master` git branch.

To preserve backwards compatibility, if there's a pre-existing archive under the *control directory* name, this will remain the experiment name (e.g. `my_expt` in the above example). Similarly, if the `experiment` value is configured (see *Configuring your experiment*), this will be used for the experiment name.

Switching between related experiments

To be able to run related experiments from the same control directory using git branches, you can use `payu checkout` which is a wrapper around `git checkout`. Creating new branches will generate a new UUID, update metadata files, and create a branch-UUID-aware experiment name in `archive`. Switching branches will change `work` and `archive` symlinks in the control directory to point to directories in *laboratory* if they exist.

To create a git branch for a new experiment, use the `-b` flag. For example, to create and checkout a new branch called `perturb1`, run:

```
payu checkout -b perturb1
```

To create a new experiment from an existing branch, specify the branch name or a commit hash after the new branch name. For example, the following creates a new experiment branch called `perturb2` that starts from `perturb1`:

```
payu checkout -b perturb2 perturb1
```

To specify a restart path to start from, use the `--restart`/ `-r` flag, for example:

```
payu checkout -b perturb --restart path/to/restart
```

Note: This can also be achieved by configuring `restart` (see *Configuring your experiment*).

To checkout and switch to an existing branch and experiment, omit the `-b` flag. For example, the following checks out the `perturb1` branch:

```
payu checkout perturb1
```

To see more `payu checkout` options, run:

```
payu checkout --help
```

For more information on git branches that exist in the control directory repository, run:

```
payu branch # Display local branches UUIDs
payu branch --verbose # Display local branches metadata
payu branch --remote # Display remote branches UUIDs
```

1.3 Configuring your experiment

This section outlines the configuration of an individual experiment, which is stored in the `config.yaml` file.

Configuration files are written in the YAML file format. YAML allows us to store and configure both individual fields as well as higher data structures, such as lists and dictionaries. Data can also be encapsulated in larger structures. This is indicated by whitespace in the file, which is significant in YAML.

1.3.1 Configuration Settings

Scheduler

These settings are primarily used by the PBS scheduler.

`queue` (*Default:* `normal`)
: The PBS queue to submit your job. Equivalent to `qsub -q queue`.

`project` (*Default:* `$PROJECT`)
: The project from which to submit the model (and deduct CPU hours). Equivalent to `qsub -P PROJECT`. The default value is the current `$PROJECT` environment variable. Note that `project` is used as part of the default configuration for various laboratory filepaths.

`jobname` (*Default: Control directory name*)
: The name of the job as it appears in the PBS queue. If no name is provided, then it uses the name of the experiment's control directory.

`ncpus`
: The number of CPUs used during model simulation. Usually equivalent to `qsub -l ncpus=N`. This is the number passed on to `mpirun` during model execution.

  Although it usually matches the CPU request, the actual request may be larger if `npernode` is being used.

`ncpureq`
: Hard override for the number of cpus used in the PBS submit. This is useful when the number of CPUs used in the `mpirun` command is not the same as the number of cpus required. For example, when running an OpenMP only model like `qgcm`, set `ncpus=1`, and then set `ncpureq` to the number of threads required to run the model.

`npernode`
: The number of CPUs used per node. This settings is passed on to `mpirun` during model execution. In most cases, this is converted into an equivalent `npersocket` configuration.

  This setting may be needed in cases where a node is unable to efficiently use all of its CPUs, such as performance issues related to NUMA.

`mem` (*Default: 192GiB per node*)
: Amount of memory required for the job. Equivalent to `qsub -l mem=MEM`. The default value requests (almost) all of the nodes' memory for jobs using multiple nodes.

  In general, it is good practice to keep this number as low as possible.

`platform`

  Set platform specific defaults. Available sub options:

  `nodemem`
  : Override default memory per node. Used when memory not specified to calculate memory request

  `nodesize`
  : Override default ncpus per node. Used to calculate ncpus to fully utilise nodes regardless of requested number of cpus

`walltime`
: The amount of time required to run an individual job, specified as `hh:mm:ss`. Equivalent to `qsub -l walltime=TIME`.

  Jobs with shorter walltimes will generally be prioritised ahead of jobs with longer walltimes.

`priority`
: Job priority setting. Equivalent to `qsub -p PRIORITY`.

`umask` (***Default: 027***)
: Default permission mask ("umask") for new files created during model execution. Nonzero values will disable specific permissions, following standard octal notation.

  The first digit should be a zero when using standard octal format.

`qsub_flags`
: This is a generic configuration marker for any unsupported qsub flags. This setting is applied to any `qsub` calls.

`storage`
: On the NCI system gadi all storage mount points must be specified, except `/home` and `/scratch/$PROJECT`. By default payu will scan all relevant configuration paths and manifests for filepaths that are stored on mounts that begin with `/scratch` or `/g/data`, and add the correct storage flags to the `qsub` submission. In cases where payu cannot determine all the required storage points automatically they can be specified using the `storage` option. Each key is a storage mount point descriptor, and contains an array of project code values:

```
storage:
      gdata:
            - x00
            - a15
      scratch:
            - zz3
```

Model

These settings are part of general model execution, including OpenMPI configuration.

`model` (***Default: Parent directory of control directory***)
: The model (or coupled model configuration) used in the experiment. This model name must be one of the supported models shown in `payu list`.

  If no model name is provided, then it will attempt to infer the model based on the parent directory of the experiment. For example, if we run our experiment in `~/mom/bowl1`, then `mom` will be used as the model type. However, it is generally better to specify the model type.

`shortpath` (***Default:*** `/scratch/${PROJECT}`)
: The top-level directory for general scratch space, where laboratories and model output are stored. Users who run from multiple projects will generally want to set this explicitly.

`input`
: Listing of the directories containing model input fields, linked to the experiment during setup. This can either be the name of a directory in the laboratory's `input` directory:

```
input: core_inputs
```

  the absolute path of an external directory:

```
input: /scratch/v45/core_input/iaf/
```

  or a list of input directories:

```
input:
   - year_100_restarts
   - core_inputs
   - /scratch/v45/core_input/iaf/
```

  If there are files in each directory with the same name, then the earlier directory of the list takes precedence.

exe
Binary executable for the model. This can either be a filename in the laboratory's `bin` directory, or an absolute filepath. Various model drivers typically define their own default executable names.

submodels
If one is running a coupled model containing several submodels, then each model is configured individually within a `submodel` namespace, such as in the example below for the ACCESS driver:

```
model: access
submodels:
   atmosphere:
      model: matm
      exe: matm_MPI1_nt62.exe
      input: iaf_matm_simon
      ncpus: 1
   ocean:
      model: mom
      exe: fms_MOM_ACCESS_kate.x
      input: iaf_mom
      ncpus: 120
   ice:
      model: cice
      exe: cice_MPI1_6p.exe
      input: iaf_cice
      ncpus: 6
   coupler:
      model: oasis
      input: iaf_oasis
      ncpus: 0
```

restart_freq (*Default:* `5`)
Specifies the rate of saved restart files. This rate can be either an integer or date-based. For the default rate of 5, we keep the restart files for every fifth run (`restart000`, `restart005`, `restart010`, etc.). To save all restart files, set `restart_freq: 1`.

If `restart_history` is not configured, intermediate restarts are not deleted until a permanently archived restart has been produced. For example, if we have just completed run 11, then we keep `restart000`, `restart005`, `restart010`, and `restart011`. Restarts 11 through 14 are not deleted until `restart015` has been saved.

To use a date-based restart frequency, specify a number with a time unit. The supported time units are `YS` - year-start, `MS` - month-start, `W` - week, `D` - day, `H` - hour, `T` - minute and `S` - second. For example, `restart_freq: 10YS` would save earliest restart of the year, 10 years from the last permanently archived restart's datetime.

Please note that currently, only ACCESS-OM2, MOM5 and MOM6 models support date-based restart frequency, as it depends on the payu model driver being able to parse restarts files for a datetime.

restart_history
Specifies how many of the most recent restart files to retain regardless of `restart_freq`.

The following model-based tags are typically not configured

user (*Default:* `${USER}`)
The username used to construct the laboratory paths. It is generally recommended that laboratories be stored under username, so this setting is usually not necessary (nor recommended).

laboratory (*Default:* `/scratch/${PROJECT}/${USER}/${MODEL}`)
The top-level directory for the model laboratory, where the codebase, model executables, input fields, running jobs, and archived output are stored.

`control` (*Default: current directory*)
: The control path for the experiment. The default setting is the path of the current working directory.

`experiment`
: The experiment name used for archival. This will override the experiment name generated using metadata and existing archives (see *Metadata and Related Experiments*).

Manifests

payu automatically generates and updates manifest files. See *Manifests* section for details.

`reproduce`
: These options allow fine-grained control of manifest checking to enable reproducible experiments. The default value is the value of the global `reproduce` flag, which is set using a command line argument and defaults to *False*. These options **override** the global `reproduce` flag. If set to *True* payu will refuse to run if the hashes in the relevant manifest do not match.

  `exe` (*Default: global reproduce flag*)
  : Enforce executable reproducibility. If set to *True* will refuse to run if hashes do not match.

  `input` (*Default: global reproduce flag*)
  : Enforce input file reproducibility. If set to *True* will refuse to run if hashes do no match. Will not search for any new files.

  `restart` (*Default: global reproduce flag*)
  : Enforce restart file reproducibility.

`scaninputs` (*Default: True*)
: Scan input directories for new files. Set to *False* when reproduce input is *True*.

  If a manifest file is complete and it is desirable to not add spurious files to the manifest but allow existing files to change, setting this option to *False* would allow that behaviour.

`ignore` (*Default: .):**
: List of `glob` patterns which match files to ignore when scanning input directories. This is an array, so multiple patterns can be specified on multiple lines. The default is .* which ignores all hidden files on a POSIX filesystem.

Collation

Collation scheduling can be configured independently of model runs. Not all models support, or indeed require, collation. Collation is currently supported for MITgcm and any of the FMS based models (MOM, GOLD, SIS).

The collate process joins a number of smaller files which contain different parts of the model grid together into target output files.

Parallelisation of collation is supported for FMS based models using threaded multiprocessing. Collation time can be reduced if there are multiple target collate files. The magnitude of the collation time reduction depends a great deal on the time taken to collate each target file, the number of such files, and the number of cpus used. It is difficult to say a priori what settings are optimal: some experimentation may be necessary.

There is also experimental support for MPI parallelisation when using mppnccombine-fast

Collate options are specified as sub-options within a separate `collate` namespace:

`enable` (*Default: True*)
: Flag to enable/disable collation

`queue` (*Default:* `copyq`)
: PBS queue used for collation jobs.

`walltime`
: Time required for output collation.

`mem` (*Default:* `2GB`)
: Memory required for output collation.

FMS based model only options:

`ncpus`
: Number of cpus used for collation.

`ignore`
: Ignore these target files during collation. This can either be a single filename or a list of filenames.

`flags`
: Specify the flags passed to the collation program. Defaults depend on value of `mpi` flag

`exe`
: Binary executable for the collate program. This can be either a filename in the laboratory's `bin` directory, or an absolute filepath.

`restart` (*Defaut: False*)
: Collate restart files from previous run.

`mpi`
: Use mpi parallelism and mppnccombine-fast.

`glob`
: When `mpi` is `True` attempt to generate an equivalent glob string for the list of files being collated to avoid issues with limits on the number of arguments for an command being run using MPI

`threads` (*Default:* **1**)
: When `mpi` is `True` it is also possible to still use multiple threads by specifying this option. The number of cpus used for each collation thread is then `ncpus / nthreads`

Postprocessing

`collate` (*Default:* `True`)
: Controls whether or not a collation job is submitted after model execution.

  This is typically `True`, although individual model drivers will often set the default value to `False` if collation is unnecessary.

  See above for specific `collate` options.

`userscripts`
: Namelist to include separate userscripts or subcommands at various stages of a payu submission. Inputs can be either script names (`some_script.sh`) or individual subcommands (`echo "some_data" > input.nml`, `qsub some_script.sh`).

  Specific scripts are defined below:

  `init`
  : User-defined command to be called after experiment initialization, but before model setup.

  `setup`
  : User-defined command to be called after model setup, but prior to model execution.

  `run`
  : User-defined command to be called after model execution but prior to model output archive.

`archive`
User-defined command to be called after model archival, but prior to any postprocessing operations, such as `payu collate`.

`error`
User-defined command to be called if model does not run correctly and returns an error code. Useful for automatic error postmortem.

`sync`
User-defined command to be called at the start of the `sync` pbs job. This is useful for any post-processing before syncing files to a remote archive.

`postscript`
This is an older, less user-friendly, method to submit a script after `payu collate` has completed. Unlike the `userscripts`, it does not support user commands. These scripts are always re-submitted via `qsub`.

`sync`
Sync archive to a remote directory using rsync. Make sure that the configured path to sync output to, i.e. `path`, is the correct location before enabling automatic syncing or before running `payu sync`.

If postscript is also configured, the latest output and restart files will not be automatically synced after a run.

`enable` (*Default:* `False`):
Controls whether or not a sync job is submitted either after the archive or collation job, if collation is enabled.

`queue` (*Default:* `copyq`)
PBS queue used to submit the sync job.

`walltime` (*Default:* `10:00:00`)
Time required to run the job.

`mem` (*Default:* `2GB`)
Memory required for the job.

`ncpus` (*Default:* `1`)
Number of ncpus required for the job.

`path`
Destination path to sync archive outputs to. This must be a unique absolute path for your experiment, otherwise, outputs will be overwritten.

`restarts` (*Default:* `False`)
Sync permanently archived restarts, which are determined by `restart_freq`.

`rsync_flags` (*Default:* `-vrltoD --safe-links`)
Additional flags to add to rsync commands used for syncing files.

`exclude`
Patterns to exclude from rsync commands. This is equivalent to rsync's `--exclude PATTERN`. This can be a single pattern or a list of patterns. If a pattern includes any special characters, e.g. `.*+?|[]{}()`, it will need to be quoted. For example:

```
exclude:
   - 'iceh.????-??-??.nc'
   - '*-IN-PROGRESS'
```

`exclude_uncollated` (*Default:* **`True` if collation is enabled**)
Flag to exclude uncollated files from being synced. This is equivalent to adding `--exclude *.nc.*`.

`extra_paths`
: List of `glob` patterns which match extra paths to sync to remote archive. This can be a single pattern or a list of patterns. Note that these paths will be protected against any local delete options.

`remove_local_files` (*Default:* `False`)
: Remove local files once they are successfully synced to the remote archive. Files in protected paths will not be deleted. Protected paths include the `extra_paths` (if defined), last output, the last saved restart (determined by `restart_freq`), and any subsequent restarts.

`remove_local_dirs` (*Default:* `False`)
: Remove local directories once a directory has been successfully synced. This will delete any files in local directories that were excluded from syncing. Similarly to `remove_local_files`, protected paths will not be deleted.

`runlog` (*Default:* `True`)
: Create or update a bare git repository clone of the run history, called `git-runlog`, in the remote archive directory.

Experiment Tracking

`runlog`
: Automatically commits changes to configuration files and manifests in the *control directory* when the model runs. This creates a git runlog of the history of the experiment.

  `enable` (*Default:* `True`) Flag to enable/disable runlog.

`metadata`
: Generates and updates metadata files and unique experiment IDs (UUIDs). For more details, see *Metadata and Related Experiments*.

  `enable` (*Default:* `True`)
  : Flag to enable/disable creating/updating metadata files and UUIDs. If set to False, the UUID is left out of the experiment name used for archival.

  `model` (*Default: The configured model value*)
  : Model name used when generating metadata for new experiments.

1.3.2 Miscellaneous

`restart`
: Specify the full path to a restart directory from which to start the run. This is known as a “warm start”. This option has no effect if there is an existing restart directory in `archive`, and so does not **have** to be removed for subsequent submissions.

`debug` (*Default:* `False`)
: Enable the debugger for any `mpirun` jobs. Equivalent to `mpirun --debug`. At NCI this defaults to a Totalview session. This will probably only work for interactive sessions.

`mpirun`
: Append any unsupported `mpirun` arguments to the `mpirun` call of the model. This setting supports both single lines and a list of input arguments. Example shown below:

```
mpirun:
   - -mca mpi_preconnect_mpi 1   # Enable preconnecting
   - -mca mtl ^mxm               # Disable MXM acceleration
   - -mca coll ^fca              # Disable FCA acceleration
```

`ompi`
Enable any environment variables required by `mpirun` during execution, such as `OMPI_MCA_coll`. The following example below disables "matching transport layer" and "collective algorithm" components:

```
ompi:
   OMPI_MCA_coll: ''
   OMPI_MCA_mtl: ''
```

`stacksize`
Set the stacksize for each process in kiB. `unlimited` is also a valid setting (and typically required for many models).

Note: `unlimited` *works without any issues, but explicit stacksize values may not be correctly communicated across raijin nodes.*

`runspersub`
Define the maximum number of runs per PBS job submit. The default is 1. The actual number of runs per PBS submit will be the minimum of `runspersub` and the total number of runs set with the `-n` command-line flag.

`repeat`
Ignore any restart files and repeat the initial run upon resubmission. This is generally only used for testing purposes, such as bit reproducibility.

`modules`
Specify lists of environment modules and/or directories to load/use at the start of the PBS job, for example:

```
modules:
   use:
      - /path/to/module/directory
   load:
      - netcdf-c-4.9.0
      - parallel-netcdf-1.12.3
      - xerces-c-3.2.3
```

This is seldom needed, because payu is good at automatically determining the environment modules required by model executables. If the modules require *module use* in order to be found, this command can also be run prior to *payu run* instead of listing the directory under the *use* option, e.g.:

```
module use /path/to/module/directory
payu run
```

1.4 Manifests

1.4.1 Introduction

payu automatically generates and updates manifest files in the `manifest` subdirectory in the control directory. The manifests are stored in YAML format.

There are three manifests: `manifest/exe.yaml` tracks executable files, `manifest/input.yaml` tracks input files and `manifest/restart.yaml` tracks restart files.

Only files in the temporary `work` directory are tracked by manifests. Any files that are directly accessed from other locations in the filesystem within models or other programs **are not tracked**

1.4.2 Manifest contents

The manifests store information about the files contained in the `work` directory of an experiment. In most cases those files are symbolically linked from another location.

An example input manifest is shown below:

```
format: yamanifest
version: 1.0
---
work/INPUT/gotmturb.inp:
    fullpath: /scratch/x00/aaa000/mom/input/bowl1/gotmturb.inp
    hashes:
        binhash: 1730d092cdc5d86e234d3749857ed318
        md5: 3016ea3bccf1acd2c18eefdd6dbf02e9
work/INPUT/grid_spec.nc:
    fullpath: /scratch/x00/aaa000/mom/input/bowl1/grid_spec.nc
    hashes:
        binhash: b79c406507e2b96725a08237e2165314
        md5: f571a0106c4a2eba38e3c407335e8cca
work/INPUT/ocean_temp_salt.res.nc:
    fullpath: /scratch/x00/aaa000/mom/input/bowl1/ocean_temp_salt.res.nc
    hashes:
        binhash: d70322dece2f10aaacf751254a2acee7
        md5: f506e15417ed813fde3516a262ff35e5
```

The first section of the file specifes a format (`yamanifest`) and a version number (`1.0`). The second section has a local path in the `work` directory as the key, and for each of these paths stores the location in the filesystem (`fullpath`) and two hashes, `binhash` and `md5`.

There are two hashes as `binhash` is fast and size independent designed just to detect if a file has changed. If the calculated binhash is not the same as that stored in the manifest the slower but robust MD5 hash is calculated. Whenever a hash changes the updated value is stored in the manifest file.

1.4.3 Experiment tracking

The manifest files are automatically added to the git repository that tracks changes to the experimental configuration. Each time the model is run the manifest is checked and changed hashes are updated, and any new files found are added to the manifest.

In this way manifests uniquely identify all executables, input and restart files for each model run.

Manifest updates

Each of the manifests is updated in a slightly different way which reflects the way the files are expected to change during an experiment.

The executable manifest is recalculated each time the model is run. Executables are generally fairly small in size and number, so there is very little overhead calculating full MD5 hashes. This also means there is no need to check that exectutable paths are still correct and also any changes to executables are automatically included in the manifest.

The restart manifest is also recalculated for every run as there is no expectation that restart (or pickup) files are ever the same between normal model runs.

The input manifest changes relatively rarely and can often contain a small number of very large files. It is this combination that can cause a significant time overhead if full MD5 hashes have to be computed for every run. By using binhash, a fast change-sensitive hash, these time consuming hashes only need be computed when a change has been detected. So the slow md5 hashes are recalculated as little as possible.

Manifest options

By default manifests just reflect the state of the model, and when files change the update hashes are saved in the manifest. These changes in the manifest files are then tracked with git.

There are some configuration options available to change this default behaviour.

1.5 Design Notes

This section describes miscellaneous information about the design of payu.

1.5.1 Model and Experiment Layout

Laboratory Structure

An experiment requires that the model executable, configuration, and data files be staged in the appropriate directories, outlined below:

Control Path
: Configuration files are stored here, and is also the directory where `payu` is invoked. This is usually the current working directory.

Laboratory Path
: This is the top-level directory for a particular model, and contains the model executables, input data, and model output for all experiments using this model. The default directory is `/scratch/${PROJECT}/${USER}/${MODEL}`.

Herein, `${LAB}` refers to the laboratory path.

Executable Path
: Model executables are stored here. The default is `${LAB}/bin`.

Input Path
: Data files are stored here. The default is `${LAB}/input`.

Codebase Path (*not currently supported*)
: The sourcecode of the current active executable will be stored in this directory. The default is `${LAB}/codebase`.

Archive Path
: Model output is stored in this directory, separated by experiment. For an experiment named `myrun`, the archived output is stored in `${LAB}/archive/myrun/output000`, `output001`, `output002`, etc. and restart information is stored in `restart000`, `restart001`, etc.

Work Path
: Experiments that are actively running are stored in the work path. For an experiment named `myrun`, the default directory is `${LAB}/work/myrun`.

1.5.2 Style Guide

These are various unorganised notes on preferred coding style for payu. While it's unlikely that every file adheres to this style, it should generally be adopted where possible.

1. All files should adhere to PEP8 rules. In particular, no warnings should be reported by `pycodestyle` using default settings.
2. Docstrings should similarly adhere to PEP257, as reported by `pydocstyle`. (Currently conformance to this rule is admittedly very poor.)

   In particular, `help()` should be readable and well-formatted for every module and function.
3. Imports should be one per line (as in PEP8), and ideally alphabetical (as recommended by PyLint). Additionally, we separate these into three groups with a blank line, and in this order:

   a. Future statements
   b. Standard library modules
   c. Dependencies
   d. Modules local to the project

   Example import:

   ```
   from __future__ import print_function

   import os
   import shlex
   import sys

   import requests
   import yaml

   import payu.envmod
   ```
4. Modules should not be renamed. This is bad:

   ```
   import numpy as np
   ```

   This is good:

   ```
   import numpy
   ```

   The reason here is to preserve shorter names for other uses in the code. But, as usual, the HHGP's section on modules explains this better than I can within a bullet point list.

   (Also note that this is another rule with poor conformance.)
5. Multiple equivalence checks should use tuples. This is bad:

   ```
   if x == 'a' or x == 'b':
   ```

   This is good:

   ```
   if x in ('a', 'b'):
   ```

CHAPTER TWO

SUPPORT AND DEVELOPMENT

- Mailing List: https://groups.google.com/group/payu-climate
- Source: https://github.com/marshallward/payu