Python

Karol Krizka, Zach Marshall

Jan 10, 2022

CONTENTS

PyTaskFarmer is a simple task farmer written in Python for parallelizing multiple serial jobs at NERSC. It is flexible enough to run on other systems. It is very loosely based on the concept of Shane Canon's TaskFarmer.

More complex task list definitions and setup environments are implemented through the concepts of tasklist handers and runners.

CHAPTER

ONE

WHAT IT DOES (10 MINUTE VERSION)

Details on what PyTaskFarmer does behind the scenes. This might help in case you see unexpected behavior or want to know what the script is capable of.

PyTaskFarmer uses a series of files to track the progress of processing a tasklist. The usage of a shared file system for communication between multiple instances of PyTaskFarmer removes the need for an omniresent scheduler process.

All files are stored inside the working directory (*workdir*). The access to them is protected using file locking mechanisms to prevent race conditions among workers or multiple instances of PyTaskFarmer. Locks are written to `SCRATCH` as other file systems at NERSC do not support file locking. In case the system does not define *SCRATCH*, a *lock* file inside *workdir* is used instead.

The PyTaskFamer program does the following:

1. The tasklist is parsed by a tasklist handler to create the list of commands to execute (*tasks*). If the *toprocess* file does not exist, then the tasks are stored there. If the file already exists, the assumption is that you are re-starting the task farmer and it should continue from where the last farmer left off. This also allows for multiple farmers (on multiple nodes) processing the same same task list. Such parallel farmers do not compete with each other, but share the tasks.
2. The requested number of workers (`multiprocess.Pool`) is intitiated.
3. The workers are assigned a list of jobs that matches the total number of input tasks to process in parallel. Each job fetches the next available task. If no more tasks are available, then the job finishes quickly. This guarantees that enough jobs are spawened to process all tasks, but does not waste resources on completed tasks.
4. Each task is passed to a runner that then executes it. The runner can perform additional steps like setup the environment or execute the task inside a container (ie: shifter).
5. The output/error stream of each task is stored in a logfile at `logs/log_N`, where *N* is the task ID. The task ID corresponds to the order that the command is written in the original *toprocess* file starting at 0. Note that the tasks might not finish in this order.
6. The task's exit code is used to put it into the *finished* or *failed* file upon completion. Exit code 0 indicates success.
7. If the farmer catches either a timeout or a `SIGUSR1`, then the worker pool is immediately killed in a clean fasion. Any tasks that are being executed are added back to the *toprocess* list.

Note

- The workers don't know (or care) what command they run. That means if your single-line commands use 4 threads at a time, then you can ask PyTaskFarmer to run 64/4=16 processes and it will have 16 four-thread processes running at a time.
- If your program can fully utilize a node (64 threads on Cori Haswell), then you can ask the farmer to run one process at a time. This is equivalent to running the commands in your text file in order, but with support for checkpointing the per-file progress.

CHAPTER

TWO

RUNNERS

Runners define the execution environment in which the tasks are execute. They can be also used globally across multiple tasklists. This reduces the amount of clutter in each task definition and makes tasklists portable across multiple environments.

The desired runner is selected using the `--runner` option to the PyTaskFarmer program.

The `BasicRunner` is always available under the name `default`. See *Provided Runners* for the list of runners shipped with PyTaskFarmer.

2.1 Defining Runners

Custom runners can be defined inside the *~/.pytaskfarmer/runners.d* directory or the current working directory as INI files. All files ending in *.ini* are loaded. There can be multiple runners defined in a single file.

The format of a single runner definition is

```
[runnername]
Runner = runner.python.class
Arg0 = value0
Arg1 = value1
```

where *runnername* is the name of the runner and *Runner* is the Python class (along with package and module) of the implementation. The remaining key:value pairs are passed to the `runner.python.class` constructor as keyword arguments.

2.2 Provided Runners

class taskfarmer.runners.**BasicRunner**
Simple runner that runs a command.

Executes the command as it is given to it. It uses *subprocess.Popen* to execute the task in the same environment as the worker.

__init__()

class taskfarmer.runners.**ShifterRunner**(*image*, *setup=''*, *volumes=''*, *modules=''*, *tempdir=False*)
Executes each task inside a Shifter container. This can be preferable over starting PyTaskFarmer inside Shifter as it does not require a recent version of Python in the image. Shifter itself is started using `subprocess` module with the following command.

```
shifter --image image -- /bin/bash -c "setup && task"
```

The `setup` is user-configurable set of commands to setup the environment (ie: source ALRB) in Shifter.

See the constructor for the list of available options.

Example (ATLAS Athena Release 22):

```
[reco22]
Runner = taskfarmer.runners.ShifterRunner
image = zlmarshall/atlas-grid-centos7:20191110
setup = source ${ATLAS_LOCAL_ROOT_BASE}/user/atlasLocalSetup.sh && source $
↪{AtlasSetup}/scripts/asetup.sh Athena,22.2,latest
modules = cvmfs
tempdir = True
```

__init__(*image*, *setup=''*, *volumes=''*, *modules=''*, *tempdir=False*)

Parameters

image [str] Name of Shfter image.

setup [str, optional] Setup command to run before executing task.

volumes [str, optional] List of volume bindings as a space separated string.

module [str, optional] List of modules as a space separated string.

tempdir [bool, optional] Each task should be run in own temporary directory.

CHAPTER

THREE

TASKLIST HANDLERS

TaskList Handlers are responsible for parsing the contents of a tasklist into a set of commands to execute (tasks). This reduces the amount of clutter in each task definition and common tasks (ie: athena job options) reusable across tasklists.

The desired tasklist handler is selected using the `--tasklist` option to the PyTaskFarmer program.

The `ListTaskList` handler is always available under the name `default`. See *Provided TaskList Handlers* for the list of tasklists handlers shipped with PyTaskFarmer.

3.1 Defining TaskList Handlers

TaskList Handler definitions are loaded from *pytaskfarmer/tasklists.d* and the current working directory. All files ending in *.ini* are loaded and are expected to be the INI format.

The following scheme is expected:

```
[tasklisthandlername]
TaskList = tasklist.python.class
Arg0 = value0
Arg1 = value1
```

The extra arguments are passed to the *TaskList* constructor as keyword arguments.

3.2 Provided TaskList Handlers

All TaskList Hander contructions take `path` and `workdir` as the two positional arguments. They are automatically set by the PyTaskFarmer program and should not be specified by the user.

3.2.1 Generic Handlers

class taskfarmer.task.**ListTaskList**(*path*, *workdir*)

A list of tasks is defined using a file containing a task per line, with supporting status files defined using a suffix. The task ID is defined as the line number (starting at 0) inside the main task file.

Subclasses can implement the `__getitem__` function to futher modify the task definitions. The original line/task content is stored in the *tasks* member variable. By default, the *tasks[taskid]* is returned unmodified.

All supporting status files are stored inside the workdir. The used files are:

- *toprocess*: List of tasks that still need to be processed. The format is `taskID task`.

- *finished*: List of tasks that succesfully finished (return code 0). The format is `taskID task`.
- *failed*: List of tasks that finished unsuccesfully (return code not 0). The format is `taskID task`.

The list and corresponding operations are defined in a process-safe manner using the supporting files to synchronize the state. This means that multiple *ListTaskLists* can be created for a single tasklist (even on multiple machines with a shared filesystem).

__init__(*path*, *workdir*)

Parameters

path [str] Path to tasklist.

workdir [str] Path to work directory.

3.2.2 ATLAS Handlers

class taskfarmer.atlas.**TransformTaskList**(*path*, *workdir*, *transform*, *input*, *output*, *maxEventsPerJob=None*, ***kwargs*)

Run an ATLAS transform on input ROOT files.

See the `__init__` function on details how to configure this tasklist handler. A simple example for running no pileup digitization is below.

```
[digi]
TaskList = taskfarmer.atlas.TransformTaskList
transform = Reco_tf.py
input = HITS
output = RDO
autoConfiguration = everything
digiSteeringConf = StandardInTimeOnlyTruth
conditionsTag = default:OFLCOND-MC16-SDR-RUN2-06
geometryVersion = default:ATLAS-R2-2016-01-00-01
postInclude = default:PyJobTransforms/UseFrontier.py
preInclude = HITtoRDO:Campaigns/MC16NoPileUp.py
preExec = all:from ParticleBuilderOptions.AODFlags import AODFlags; AODFlags.
↪ThinGeantTruth.set_Value_and_Lock(False);' 'HITtoRDO:from Digitization.
↪DigitizationFlags import digitizationFlags; digitizationFlags.OldBeamSpotZSize =␣
↪42
```

The `TransformTaskList` supports splitting each input file into multiple tasks, based on a maximum number of events. However, when practical, it is recommeded to use AthenaMP for parallelizing event processing. This has a reduced memory footprint. AthenaMP can enabled by including the following in your tasklist handler defintion.

```
athenaopt = all:--nprocs=64
```

or by setting the `ATHENA_PROC_NUMBER` environmental variable.

The transform output is stored in the current working directory. It is then copied to the *workdir* using rsync. This two stage process is required due to how AthenaMP determines its temporary outputs. The implication is that the runner needs to run the command using bash.

__init__(*path*, *workdir*, *transform*, *input*, *output*, *maxEventsPerJob=None*, ***kwargs*)

The *kwargs* are interpreted as arguments to the transform command. For example, having an kwarg of `kwargs['postInclude']="HITtoRDO:Campaigns/MC16NoPileUp.py"` translates into a transform ar-

gument of `--postInclude='HITtoRDO:Campaigns/MC16NoPileUp.py'`. Note the automatic wrapping of the value string inside singlue quotes. These are automatically by the added by this tasklist handler.

Parameters

path [str] Path to tasklist

workdir [str] Path to work directory

transform [str] Name of transform (ie: `Sim_tf.py`)

input [str] Type of input file (ie: `EVNT`)

output [str] Type of output file (ie: `HITS`)

maxEventsPerJob [str, optional] Maximum number of events per task

kwargs Arguments passed to athena as `--key='value'`.

class taskfarmer.atlas.**AthenaTaskList**(*path*, *workdir*, *jobOptions*, *output*, *maxEventsPerJob=None*, ***kwargs*)

Run an athena job on input ROOT files.

See the `__init__` function on details how to configure this tasklist handler. A simple example for running no pileup digitization is below.

The job options need to use the built-in athena support for input files (ie: `--filesInput`).

The `AthenaTaskList` supports splitting each input file into multiple tasks, based on a maximum number of events. However, when practical, it is recommeded to use AthenaMP for parallelizing event processing. This has a reduced memory footprint. AthenaMP can enabled by including the following in your tasklist handler defintion.

```
nprocs = 64
```

or by setting the `ATHENA_PROC_NUMBER` environmental variable.

The output file name is set as the `output` setting. The handler looks for it in the current working directory and then copies it to the *workdir* using rsync. This two stage process is required due to how AthenaMP determines its temporary outputs. The implication is that the runner needs to run the command using bash.

__init__(*path*, *workdir*, *jobOptions*, *output*, *maxEventsPerJob=None*, ***kwargs*)

The *kwargs* are interpreted as arguments the the athena command. For `kwargs['postInclude']="HITtoRDO:Campaigns/MC16NoPileUp.py"` translates into an athena argument of `--postInclude='HITtoRDO:Campaigns/MC16NoPileUp.py'`. Note the automatic wrapping of the value string inside singlue quotes. These are automatically added by this tasklist handler.

Parameters

path [str] Path to tasklist.

workdir [str] Path to work directory.

jobOptions [str] Name of jobOptions file to execute.

output [str] Expected name of output file.

maxEventsPerJob [str, optional] Maximum number of events per task.

kwargs Arguments passed to athena as `--key='value'`.

CHAPTER

FOUR

EXAMPLES

4.1 Simple Example

Included in the package is a small test file that you can use as an example. Try running

```
pytaskfarmer.py mywork.tasks
```

That will give you a sense of how the thing works. Feel free to kill it and restart it if you wish.

4.2 SLURM Example (Array Jobs)

Example of a batch job for using PyTaskFarmer with SLURM is below. It demonstrates how to correctly handle cleanup.

```
#!/bin/bash
#SBATCH --output=slurm-%j.out
#SBATCH --error=slurm-%j.err
#SBATCH --qos=debug
#SBATCH --tasks-per-node=1
#SBATCH --constraint=haswell
#SBATCH --signal=B:USR1@60
#SBATCH --array=1-5
#SBATCH --time=00:05:00

function handle_signal
{
    echo "$(date) bash is being killed, also kill ${PROCPID}"
    kill -s USR1 ${PROCPID}
    wait ${PROCPID}
}
trap handle_signal INT USR1

if [ ${#} != 1 ]; then
    echo "usage: ${0} tasklist"
    exit 1
fi
tasklist=${1}
logdir=${tasklist}_logs

hostname
```

(continues on next page)

(continued from previous page)

```
uname -a
pwd
echo "tasklist = ${tasklist}"

${HOME}/mcgen/pytaskfarmer/pytaskfarmer.py --logDir ${logdir} --proc 32 ${tasklist} &
export PROCPID=${!}
wait ${PROCPID}
echo "$(date) Finish running!"
```

To run using array jobs:

```
sbatch slurm_test.sh mywork.tasks
```

4.3 SLURM Example (Multi-Node Jobs)

Example of a batch job for using PyTaskFarmer with a SLURM multi-node job is below. It demonstrates how to correctly handle cleanup and launch PyTaskFarmer on multiple nodes using srun.

```
#!/bin/bash
#SBATCH --output=slurm-%j.out
#SBATCH --error=slurm-%j.err
#SBATCH --qos=debug
#SBATCH --tasks-per-node=1
#SBATCH --constraint=haswell
#SBATCH --signal=B:USR1@60
#SBATCH -N5
#SBATCH --time=00:05:00

function handle_signal
{
    echo "$(date) bash is being killed, also kill ${PROCPID}"
    kill -s USR1 ${PROCPID}
    wait ${PROCPID}
}
trap handle_signal INT USR1

if [ ${#} != 1 ]; then
    echo "usage: ${0} tasklist"
    exit 1
fi
tasklist=${1}
logdir=${tasklist}_logs

hostname
uname -a
pwd
echo "tasklist = ${tasklist}"

srun -N${SLURM_JOB_NUM_NODES} \
     ${HOME}/mcgen/pytaskfarmer/pytaskfarmer.py --logDir ${logdir} --proc 32 ${tasklist}␣
↪&
```

(continues on next page)

(continued from previous page)

```
export PROCPID=${!}
wait ${PROCPID}
echo "$(date) Finish running!"
```

To run by requesting multiple nodes at the same time (srun):

```
sbatch srun_test.sh mywork.taskss
```

CHAPTER

FIVE

RUNNING AT NERSC

Tips and tricks for using the PyTaskFarmer on NERSC machines (ie: cori).

You can use PyTaskFarmer a part of your top-level batch script for submissions into the NERSC slurm batch system. There are a variety of examples for running multi-core or multi-node jobs available here.

5.1 Equalize Task Running Time

The farmer likes to have more work than workers, in order to keep those workers busy at all times. That means if you have tasks that might be different lengths (e.g. MC and data, or different size datasets, etc), it is very important to

1. put the longer tasks earlier in the list,
2. have a total run time that is longer than the longest job (preferably by a factor of 2 or more) and
3. request a number of cores that will be kept busy by your jobs.

For example, if you expect to have one 1-hour job and ten 5-minute jobs, you can requests two threads; one thread will process the 1-hour job and the other thread will process all the 5-minute jobs. This relies on your ordering the task list well – if you make the 1-hour job last, then the two threads will work through all your 5-minute jobs in about 25 minutes and then one will process the 1-hour job while the other sit idle (and wastes CPU). This requires some thought and care, but can save us significant numbers of hours, so please do think carefully about what you're running!

5.2 Clean-up In Batch Jobs

The farmer can be used in any queue at NERSC. One of the better options if some work needs doing but is not urgent is to use the flex queue on KNL. When submitting into that queue, one must add `--time-min=01:30:00 --time=10:00:00`, where the first is the minimum time that the farmer should be run (cannot be not be longer than 2 hours) and should be longer than a typical command you need to execute. The second is the total wall time for the job, which must be less than 12 hours. Jobs in this queue will be started, killed, and restarted from checkpoints.

Add to your job script

```
# requeueing the job if reamining time >0 (do not change the following 3 lines )
. /usr/common/software/variable-time-job/setup.sh
requeue_job func_trap USR1
```

in order to have the job automatically re-queued so that it will continue to run. You should also add to your run script

```
#SBATCH --signal=B:USR1@10
```

To give the job 10 seconds to handle the USR1 signal (it should not need that long, but in case there are multiple workers fighing for the same lock). For the check-pointing, please also add these to your job script:

```
# use the following three variables to specify the time limit per job (max_timelimit),
# the amount of time (in seconds) needed for checkpointing,
# and the command to use to do the checkpointing if any (leave blank if none)
max_timelimit=12:00:00   # can match the #SBATCH --time option but don't have to
ckpt_overhead=60         # should match the time in the #SBATCH --signal option
ckpt_command=
```

Note that these are in addition to the usual sbatch specifications, and it is quite important that they match.

5.3 Extra Memory

If you have serious memory issues, then it is possible to enable swap space when running in a full node queue (e.g. regular; this is not possible in the shared queue). To do so, make a burst-buffer config file like:

```
$ cat bb_swap.conf
#DW jobdw capacity=160GB access_mode=striped type=scratch
#DW swap 150GB
```

This uses the Cray DataWarp configuration format. The second line is the important one here; it provides 150 GB of swap space within the burst buffer. The first line describes the scratch space reservation that your job needs, and may be unnecessary or even problematic depending on where you write your inputs and outputs for the job (think about what it's doing before sending the command off to the queue). You can then add it to your job submission like:

```
salloc ... --bbf=bb_swap.conf
```

This allocates space on the burst buffer (generally pretty fast) to be used for swap space memory for users. Note that swap is quite a bit slower than standard (even main) memory, and so this option should be used with care. It is not, in principle, clever enough to guarantee each job space in the main memory, so as long as swap is being used on a node, all jobs on that node may be slowed down, depending on the memory profile and usage of the offending job.

CHAPTER

SIX

TODO

Lost of possible future work to futher improve PyTaskFarmer.

- At the moment, if the original process file is significantly modified (item added and removed) or contains duplicates, in some cases the process IDs may not be unique. Of course, the output can be re-directed by the user to a log file with a more appropriate name, so the log files created by the farmer may be dummy. If *PROC_NUMBER* is important to your workflow, then please either submit additional farmers for new lists of processes or add a unique (short as you like) comment to the end of the command to make the items distinguishable.
- It would be nice to add some monitoring hooks so that we can watch what users are doing with this script.
- Longer-term, it would be interesting to try to keep all tasks that need to be finished in an sqlite file, including a state (to process, running, finished, failed). Adding an integer identifier would solve the above problem and give us a free way to add jobs mid-way through a run.
- Storing all configuration inside *workdir* to reduce the overhead in restarting PyTaskFarmer. Ideally to restart, one would just have to specify the workdir. The tasklist handler and runner should be picked up from it.

CHAPTER

SEVEN

FEATURES

- Per-task checkpointing.
- Mulitple farmers running on the same tasklist.
- Simple synchronization protocol using the file system.
- Abstract definition of tasklists via tasklist handlers.
- Automatic environment setup (ie: asetup or shifter).
- Analysis of task packing using Perfetto.

CHAPTER

EIGHT

QUICK START

PyTaskFarmer can be installed using *pip*.

```
pip install pytaskfarmer
```

Create a list of tasks that you want to process in parallel. In this simple example, a counter will be echoed.

```
for i in $(seq 0 10); do
  echo ${i} >> mywork.tasks
done
```

Run PyTaskFarmer. The progress will be stored inside the specified workdir.

```
pytaskfarmer.py --proc 8 --workdir myworkdir mywork.tasks
```

CHAPTER

NINE

USAGE

The executable script is:

```
usage: pytaskfarmer.py [-h] [--proc [Processes]] [--timeout TIMEOUT]
                       [--workdir WORKDIR] [--verbose VERB]
                       [--runner RUNNER] [--tasklist TASKLISTHANDLER]
                       tasklist
```

The `tasklist` argument is a simple text file with one task per line. The interpretation of the task is up to the *TASKLISTHANDLER*. By default, the task is treated as a command to run. It is not important how complex the command is.

The `--verbose` flag adds a bit more output (though not much) telling You what the script is doing as it runs.

The `--timeout` option allows you to set a timeout for the script, so that after some number of seconds the tasks will be automatically killed (none by default).

The `--proc` option tells the script how many parallel workers to create (default 8).

The `--workdir` option tells the script where to store the progress (task status, log files..) of a single run (default is *tasklist_workdir*).

The `--runner` options indicates which runner to execute the command with. See the dedicated section on the available runners and how they work.

The `--tasklist` options indicates which tasklist handler to parse the tasklist with. See the dedicated section on the available runners and how they

CHAPTER

TEN

WHAT IT DOES (60 SECOND VERSION)

The basic behavior, with the default runner/handler, is as follows. Each access to a file is protected using a file locking mechanism.

1. The tasklist is read and a *toprocess* file is created in the *workdir* with unprocessed tasks.
2. A number of workers (`multiprocessing.Pool`) are constructed to run on the tasks.
3. When some work is done, the command is placed into a *finished* or *failed* files, depending on the status code.
4. Duration and start times of completed tasks (timeline) are saved into a *timeline.json* file. This can then be opened with Perfetto.
5. The tasks are processed by the workers until 1) the work is completed; 2) the timeout is reached; or 3) a signal is intercepted.

INDEX

Symbols

A

B

L

S

T