DPDispatcher

Deep Modeling

Jul 06, 2022

CONTENTS:

DPDispatcher is a Python package used to generate HPC (High Performance Computing) scheduler systems (Slurm/PBS/LSF/dpcloudserver) jobs input scripts and submit these scripts to HPC systems and poke until they finish.

DPDispatcher will monitor (poke) until these jobs finish and download the results files (if these jobs is running on remote systems connected by SSH).

CHAPTER

ONE

INSTALL DPDISPATCHER

DPDispatcher can installed by `pip`:

```
pip install dpdispatcher
```

CHAPTER

TWO

GETTING STARTED

DPDispatcher provides the following classes:

- *Task* class, which represents a command to be run on batch job system, as well as the essential files need by the command.
- *Submission* class, which represents a collection of jobs defined by the HPC system. And there may be common files to be uploaded by them. DPDispatcher will create and submit these jobs when a `submission` instance execute *run_submission* method. This method will poke until the jobs finish and return.
- *Job* class, a class used by *Submission* class, which represents a job on the HPC system. *Submission* will generate jobs' submitting scripts used by HPC systems automatically with the *Task* and *Resources*
- *Resources* class, which represents the computing resources for each job within a `submission`.

You can use DPDispatcher in a Python script to submit five tasks:

```
from dpdispatcher import Machine, Resources, Task, Submission

machine = Machine.load_from_json('machine.json')
resources = Resources.load_from_json('resources.json')

task0 = Task.load_from_json('task.json')

task1 = Task(command='cat example.txt', task_work_path='dir1/', forward_files=['example.
↪txt'], backward_files=['out.txt'], outlog='out.txt')
task2 = Task(command='cat example.txt', task_work_path='dir2/', forward_files=['example.
↪txt'], backward_files=['out.txt'], outlog='out.txt')
task3 = Task(command='cat example.txt', task_work_path='dir3/', forward_files=['example.
↪txt'], backward_files=['out.txt'], outlog='out.txt')
task4 = Task(command='cat example.txt', task_work_path='dir4/', forward_files=['example.
↪txt'], backward_files=['out.txt'], outlog='out.txt')

task_list = [task0, task1, task2, task3, task4]

submission = Submission(work_base='lammps_md_300K_5GPa/',
    machine=machine,
    resources=resources,
    task_list=task_list,
    forward_common_files=['graph.pb'],
    backward_common_files=[]
)

submission.run_submission()
```

where `machine.json` is

```
{
    "batch_type": "Slurm",
    "context_type": "SSHContext",
    "local_root" : "/home/user123/workplace/22_new_project/",
    "remote_root": "/home/user123/dpdispatcher_work_dir/",
    "remote_profile":{
        "hostname": "39.106.xx.xxx",
        "username": "user123",
        "port": 22,
        "timeout": 10
    }
}
```

`resources.json` is

```
{
    "number_node": 1,
    "cpu_per_node": 4,
    "gpu_per_node": 1,
    "queue_name": "GPUV100",
    "group_size": 5
}
```

and `task.json` is

```
{
    "command": "lmp -i input.lammps",
    "task_work_path": "bct-0/",
    "forward_files": [
        "conf.lmp",
        "input.lammps"
    ],
    "backward_files": [
        "log.lammps"
    ],
    "outlog": "log",
    "errlog": "err",
}
```

You may also submit mutiple GPU jobs: complex resources example

```
resources = Resources(
    number_node=1,
    cpu_per_node=4,
    gpu_per_node=2,
    queue_name="GPU_2080Ti",
    group_size=4,
    custom_flags=[
        "#SBATCH --nice=100",
        "#SBATCH --time=24:00:00"
    ],
    strategy={
```

(continues on next page)

(continued from previous page)

```
        # used when you want to add CUDA_VISIBLE_DIVECES automatically
        "if_cuda_multi_devices": True
    },
    para_deg=1,
    # will unload these modules before running tasks
    module_unload_list=["singularity"],
    # will load these modules before running tasks
    module_list=["singularity/3.0.0"],
    # will source the environment files before running tasks
    source_list=["./slurm_test.env"],
    # the envs option is used to export environment variables
    # And it will generate a line like below.
    # export DP_DISPATCHER_EXPORT=test_foo_bar_baz
    envs={"DP_DISPATCHER_EXPORT": "test_foo_bar_baz"},
)
```

The details of parameters can be found in *Machine Parameters*, *Resources Parameters*, and *Task Parameters*.

CHAPTER

THREE

MACHINE PARAMETERS

Note: One can load, modify, and export the input file by using our effective web-based tool DP-GUI. All parameters below can be set in DP-GUI. By clicking "SAVE JSON", one can download the input file.

machine:

type: `dict`
argument path: `machine`

batch_type:

type: `str`
argument path: `machine/batch_type`

The batch job system type. Option: PBS, SlurmJobArray, DpCloudServer, Lebesgue, LSF, DistributedShell, Torque, Shell, Slurm

local_root:

type: `NoneType | str`
argument path: `machine/local_root`

The dir where the tasks and relating files locate. Typically the project dir.

remote_root:

type: `str`, optional
argument path: `machine/remote_root`

The dir where the tasks are executed on the remote machine. Only needed when context is not lazy-local.

clean_asynchronously:

type: `bool`, optional, default: `False`
argument path: `machine/clean_asynchronously`

Clean the remote directory asynchronously after the job finishes.

Depending on the value of *context_type*, different sub args are accepted.

context_type:

type: `str` (flag key)
argument path: `machine/context_type`
possible choices: *LocalContext*, *DpCloudServerContext*, *HDFSContext*, *LebesgueContext*, *SSHContext*, *LazyLocalContext*

The connection used to remote machine. Option: LocalContext, LebesgueContext, SSHContext, HDFSContext, LazyLocalContext, DpCloudServerContext

When context_type is set to `LocalContext` (or its aliases `localcontext`, `Local`, `local`):

remote_profile:

type: `dict`, optional
argument path: `machine[LocalContext]/remote_profile`

The information used to maintain the connection with remote machine. This field is empty for this context.

When context_type is set to `DpCloudServerContext` (or its aliases `dpcloudservercontext`, `DpCloudServer`, `dpcloudserver`):

remote_profile:

type: `dict`
argument path: `machine[DpCloudServerContext]/remote_profile`

The information used to maintain the connection with remote machine.

email:

type: `str`
argument path: `machine[DpCloudServerContext]/remote_profile/email`

Email

password:

type: `str`
argument path: `machine[DpCloudServerContext]/remote_profile/password`

Password

program_id:

type: `int`
argument path: `machine[DpCloudServerContext]/remote_profile/program_id`

Program ID

input_data:

type: `dict`
argument path: `machine[DpCloudServerContext]/remote_profile/input_data`

Configuration of job

When context_type is set to `HDFSContext` (or its aliases `hdfscontext`, `HDFS`, `hdfs`):

remote_profile:

type: dict, optional
argument path: machine[HDFSContext]/remote_profile

The information used to maintain the connection with remote machine. This field is empty for this context.

When context_type is set to LebesgueContext (or its aliases lebesguecontext, Lebesgue, lebesgue):

remote_profile:

type: dict
argument path: machine[LebesgueContext]/remote_profile

The information used to maintain the connection with remote machine.

email:

type: str
argument path: machine[LebesgueContext]/remote_profile/email

Email

password:

type: str
argument path: machine[LebesgueContext]/remote_profile/password

Password

program_id:

type: int
argument path: machine[LebesgueContext]/remote_profile/program_id

Program ID

input_data:

type: dict
argument path: machine[LebesgueContext]/remote_profile/input_data

Configuration of job

When context_type is set to SSHContext (or its aliases sshcontext, SSH, ssh):

remote_profile:

type: dict
argument path: machine[SSHContext]/remote_profile

The information used to maintain the connection with remote machine.

hostname:

type: str
argument path: machine[SSHContext]/remote_profile/hostname

hostname or ip of ssh connection.

username:

type: `str`
argument path: `machine[SSHContext]/remote_profile/username`

username of target linux system

password:

type: `str`, optional
argument path: `machine[SSHContext]/remote_profile/password`

(deprecated) password of linux system. Please use SSH keys instead to improve security.

port:

type: `int`, optional, default: `22`
argument path: `machine[SSHContext]/remote_profile/port`

ssh connection port.

key_filename:

type: `NoneType | str`, optional, default: `None`
argument path: `machine[SSHContext]/remote_profile/key_filename`

key filename used by ssh connection. If left None, find key in ~/.ssh or use password for login

passphrase:

type: `NoneType | str`, optional, default: `None`
argument path: `machine[SSHContext]/remote_profile/passphrase`

passphrase of key used by ssh connection

timeout:

type: `int`, optional, default: `10`
argument path: `machine[SSHContext]/remote_profile/timeout`

timeout of ssh connection

totp_secret:

type: `NoneType | str`, optional, default: `None`
argument path: `machine[SSHContext]/remote_profile/totp_secret`

Time-based one time password secret. It should be a base32-encoded string extracted from the 2D code.

When context_type is set to `LazyLocalContext` (or its aliases `lazylocalcontext`, `LazyLocal`, `lazylocal`):

remote_profile:

type: `dict`, optional

argument path: `machine[LazyLocalContext]/remote_profile`

The information used to maintain the connection with remote machine. This field is empty for this context.

CHAPTER

FOUR

RESOURCES PARAMETERS

Note: One can load, modify, and export the input file by using our effective web-based tool DP-GUI. All parameters below can be set in DP-GUI. By clicking "SAVE JSON", one can download the input file for.

resources:

type: `dict`
argument path: `resources`

number_node:

type: `int`
argument path: `resources/number_node`

The number of node need for each *job*

cpu_per_node:

type: `int`
argument path: `resources/cpu_per_node`

cpu numbers of each node assigned to each job.

gpu_per_node:

type: `int`
argument path: `resources/gpu_per_node`

gpu numbers of each node assigned to each job.

queue_name:

type: `str`
argument path: `resources/queue_name`

The queue name of batch job scheduler system.

group_size:

type: `int`
argument path: `resources/group_size`

The number of *tasks* in a *job*. 0 means infinity.

custom_flags:

type: `list`, optional
argument path: `resources/custom_flags`

The extra lines pass to job submitting script header

strategy:

type: `dict`, optional
argument path: `resources/strategy`

strategies we use to generation job submitting scripts.

if_cuda_multi_devices:

type: `bool`, optional, default: `False`
argument path: `resources/strategy/if_cuda_multi_devices`

ratio_unfinished:

type: `float`, optional, default: `0.0`
argument path: `resources/strategy/ratio_unfinished`

para_deg:

type: `int`, optional, default: `1`
argument path: `resources/para_deg`

Decide how many tasks will be run in parallel.

source_list:

type: `list`, optional, default: `[]`
argument path: `resources/source_list`

The env file to be sourced before the command execution.

module_purge:

type: `bool`, optional, default: `False`
argument path: `resources/module_purge`

Remove all modules on HPC system before module load (module_list)

module_unload_list:

type: `list`, optional, default: `[]`
argument path: `resources/module_unload_list`

The modules to be unloaded on HPC system before submitting jobs

module_list:

type: `list`, optional, default: `[]`
argument path: `resources/module_list`

The modules to be loaded on HPC system before submitting jobs

envs:

type: `dict`, optional, default: `{}`
argument path: `resources/envs`

The environment variables to be exported on before submitting jobs

wait_time:

type: `int` | `float`, optional, default: `0`
argument path: `resources/wait_time`

The waitting time in second after a single *task* submitted

Depending on the value of *batch_type*, different sub args are accepted.

batch_type:

type: `str` (flag key)
argument path: `resources/batch_type`
possible choices: *PBS*, *DpCloudServer*, *Lebesgue*, *DistributedShell*, *Shell*, *LSF*, *Slurm*, *SlurmJobArray*, *Torque*

The batch job system type loaded from machine/batch_type.

When batch_type is set to `PBS` (or its alias `pbs`):

kwargs:

type: `dict`, optional
argument path: `resources[PBS]/kwargs`

This field is empty for this batch.

When batch_type is set to `DpCloudServer` (or its alias `dpcloudserver`):

kwargs:

type: `dict`, optional
argument path: `resources[DpCloudServer]/kwargs`

This field is empty for this batch.

When batch_type is set to `Lebesgue` (or its alias `lebesgue`):

kwargs:

type: `dict`, optional
argument path: `resources[Lebesgue]/kwargs`

This field is empty for this batch.

When batch_type is set to `DistributedShell` (or its alias `distributedshell`):

kwargs:

type: `dict`, optional
argument path: `resources[DistributedShell]/kwargs`

This field is empty for this batch.

When batch_type is set to `Shell` (or its alias `shell`):

kwargs:

type: `dict`, optional
argument path: `resources[Shell]/kwargs`

This field is empty for this batch.

When batch_type is set to `LSF` (or its alias `lsf`):

kwargs:

type: `dict`
argument path: `resources[LSF]/kwargs`

Extra arguments.

gpu_usage:

type: `bool`, optional, default: `False`
argument path: `resources[LSF]/kwargs/gpu_usage`

Choosing if GPU is used in the calculation step.

gpu_new_syntax:

type: `bool`, optional, default: `False`
argument path: `resources[LSF]/kwargs/gpu_new_syntax`

For LFS >= 10.1.0.3, new option -gpu for #BSUB could be used. If False, and old syntax would be used.

gpu_exclusive:

type: `bool`, optional, default: `True`
argument path: `resources[LSF]/kwargs/gpu_exclusive`

Only take effect when new syntax enabled. Control whether submit tasks in exclusive way for GPU.

custom_gpu_line:

type: `NoneType` | `str`, optional, default: `None`
argument path: `resources[LSF]/kwargs/custom_gpu_line`

Custom GPU configuration, starting with #BSUB

When batch_type is set to `Slurm` (or its alias `slurm`):

kwargs:

type: `dict`, optional
argument path: `resources[Slurm]/kwargs`

Extra arguments.

custom_gpu_line:

type: `NoneType` | `str`, optional, default: `None`
argument path: `resources[Slurm]/kwargs/custom_gpu_line`

Custom GPU configuration, starting with #SBATCH

When batch_type is set to `SlurmJobArray` (or its alias `slurmjobarray`):

kwargs:

type: `dict`, optional
argument path: `resources[SlurmJobArray]/kwargs`

Extra arguments.

custom_gpu_line:

type: `NoneType` | `str`, optional, default: `None`
argument path: `resources[SlurmJobArray]/kwargs/custom_gpu_line`

Custom GPU configuration, starting with #SBATCH

When batch_type is set to `Torque` (or its alias `torque`):

kwargs:

type: `dict`, optional
argument path: `resources[Torque]/kwargs`

This field is empty for this batch.

CHAPTER

FIVE

TASK PARAMETERS

Note: One can load, modify, and export the input file by using our effective web-based tool DP-GUI. All parameters below can be set in DP-GUI. By clicking "SAVE JSON", one can download the input file.

task:

type: `dict`
argument path: `task`

command:

type: `str`
argument path: `task/command`

A command to be executed of this task. The expected return code is 0.

task_work_path:

type: `str`
argument path: `task/task_work_path`

The dir where the command to be executed.

forward_files:

type: `list`
argument path: `task/forward_files`

The files to be uploaded in task_work_path before the task exectued.

backward_files:

type: `list`
argument path: `task/backward_files`

The files to be download to local_root in task_work_path after the task finished

outlog:

type: `NoneType | str`
argument path: `task/outlog`

The out log file name. redirect from stdout

errlog:

type: `NoneType` | `str`
argument path: `task/errlog`

The err log file name. redirect from stderr

CHAPTER

SIX

DPDISPATCHER API

6.1 dpdispatcher package

dpdispatcher.**info**()

6.1.1 Subpackages

dpdispatcher.dpcloudserver package

Submodules

dpdispatcher.dpcloudserver.api module

class dpdispatcher.dpcloudserver.api.**API**(*email*, *password*)

Bases: object

Methods

check_job_has_uploaded	
download	
download_from_url	
get	
get_job_result_url	
get_jobs	
get_tasks	
get_tasks_list	
job_create	
post	
refresh_token	
upload	

check_job_has_uploaded(*job_id*)

download(*oss_file*, *save_file*, *endpoint*, *bucket_name*)

download_from_url(*url*, *save_file*)

get(*url*, *params*, *retry=0*)

get_job_result_url(*job_id*)

get_jobs(*page=1*, *per_page=10*)

get_tasks(*job_id*, *group_id*, *page=1*, *per_page=10*)

get_tasks_list(*group_id*, *per_page=30*)

job_create(*job_type*, *oss_path*, *input_data*, *program_id=None*, *group_id=None*)

post(*url*, *params*, *retry=0*)

refresh_token()

upload(*oss_task_zip*, *zip_task_file*, *endpoint*, *bucket_name*)

dpdispatcher.dpcloudserver.config module

dpdispatcher.dpcloudserver.retcode module

class dpdispatcher.dpcloudserver.retcode.**RETCODE**

Bases: object

DATAERR = '2002'

DBERR = '2000'

IOERR = '2003'

NODATA = '2300'

OK = '0000'

PARAMERR = '2101'

PWDERR = '2104'

REQERR = '2200'

ROLEERR = '2103'

THIRDERR = '2001'

TOKENINVALID = '2100'

UNDERDEBUG = '2301'

UNKOWNERR = '2400'

USERERR = '2102'

VERIFYERR = '2105'

dpdispatcher.dpcloudserver.temp_test module

dpdispatcher.dpcloudserver.zip_file module

dpdispatcher.dpcloudserver.zip_file.**unzip_file**(*zip_file*, *out_dir='./'*)

dpdispatcher.dpcloudserver.zip_file.**zip_file_list**(*root_path*, *zip_filename*, *file_list=[]*)

6.1.2 Submodules

6.1.3 dpdispatcher.JobStatus module

class dpdispatcher.JobStatus.**JobStatus**(*value*)

Bases: IntEnum

An enumeration.

completing = 6

finished = 5

running = 3

terminated = 4

unknown = 100

unsubmitted = 1

waiting = 2

6.1.4 dpdispatcher.arginfo module

6.1.5 dpdispatcher.base_context module

class dpdispatcher.base_context.**BaseContext**(**args*, ***kwargs*)

Bases: object

Methods

machine_arginfo()	Generate the machine arginfo.
machine_subfields()	Generate the machine subfields.

bind_submission	
check_finish	
clean	
download	
kill	
load_from_dict	
read_file	
upload	
write_file	

bind_submission(*submission*)

check_finish(*proc*)

abstract clean()

abstract download(*submission*, *check_exists=False*, *mark_failure=True*, *back_error=False*)

kill(*proc*)

classmethod load_from_dict(*context_dict*)

classmethod machine_arginfo() → Argument

Generate the machine arginfo.

Returns

Argument
machine arginfo

classmethod machine_subfields() → List[Argument]

Generate the machine subfields.

Returns

list[Argument]
machine subfields

options = {'DpCloudServerContext', 'HDFSContext', 'LazyLocalContext', 'LebesgueContext', 'LocalContext', 'SSHContext'}

abstract read_file(*fname*)

subclasses_dict = {'DpCloudServer': <class
'dpdispatcher.dp_cloud_server_context.DpCloudServerContext'>,
'DpCloudServerContext': <class
'dpdispatcher.dp_cloud_server_context.DpCloudServerContext'>, 'HDFS': <class
'dpdispatcher.hdfs_context.HDFSContext'>, 'HDFSContext': <class
'dpdispatcher.hdfs_context.HDFSContext'>, 'LazyLocal': <class
'dpdispatcher.lazy_local_context.LazyLocalContext'>, 'LazyLocalContext': <class
'dpdispatcher.lazy_local_context.LazyLocalContext'>, 'Lebesgue': <class
'dpdispatcher.dp_cloud_server_context.LebesgueContext'>, 'LebesgueContext': <class
'dpdispatcher.dp_cloud_server_context.LebesgueContext'>, 'Local': <class
'dpdispatcher.local_context.LocalContext'>, 'LocalContext': <class
'dpdispatcher.local_context.LocalContext'>, 'SSH': <class
'dpdispatcher.ssh_context.SSHContext'>, 'SSHContext': <class
'dpdispatcher.ssh_context.SSHContext'>, 'dpcloudserver': <class
'dpdispatcher.dp_cloud_server_context.DpCloudServerContext'>,
'dpcloudservercontext': <class
'dpdispatcher.dp_cloud_server_context.DpCloudServerContext'>, 'hdfs': <class
'dpdispatcher.hdfs_context.HDFSContext'>, 'hdfscontext': <class
'dpdispatcher.hdfs_context.HDFSContext'>, 'lazylocal': <class
'dpdispatcher.lazy_local_context.LazyLocalContext'>, 'lazylocalcontext': <class
'dpdispatcher.lazy_local_context.LazyLocalContext'>, 'lebesgue': <class
'dpdispatcher.dp_cloud_server_context.LebesgueContext'>, 'lebesguecontext': <class
'dpdispatcher.dp_cloud_server_context.LebesgueContext'>, 'local': <class
'dpdispatcher.local_context.LocalContext'>, 'localcontext': <class
'dpdispatcher.local_context.LocalContext'>, 'ssh': <class
'dpdispatcher.ssh_context.SSHContext'>, 'sshcontext': <class
'dpdispatcher.ssh_context.SSHContext'>}

abstract upload(*submission*)

abstract write_file(*fname*, *write_str*)

6.1.6 dpdispatcher.distributed_shell module

class dpdispatcher.distributed_shell.**DistributedShell**(**args*, ***kwargs*)

Bases: *Machine*

Methods

do_submit(job)	submit th job to yarn using distributed shell
resources_arginfo()	Generate the resources arginfo.
resources_subfields()	Generate the resources subfields.

arginfo
bind_context
check_finish_tag
check_if_recover
check_status
default_resources
deserialize
gen_command_env_cuda_devices
gen_script
gen_script_command
gen_script_custom_flags_lines
gen_script_end
gen_script_env
gen_script_header
gen_script_wait
load_from_dict
load_from_json
serialize
sub_script_cmd
sub_script_head

check_finish_tag(*job*)

check_status(*job*)

do_submit(*job*)

submit th job to yarn using distributed shell

Parameters

job
[Job class instance] job to be submitted

Returns

job_id: string
submit process id

gen_script_end(*job*)

gen_script_env(*job*)

gen_script_header(*job*)

6.1.7 dpdispatcher.dp_cloud_server module

class dpdispatcher.dp_cloud_server.**DpCloudServer**(**args*, ***kwargs*)

Bases: *Machine*

Methods

do_submit(job)	submit a single job, assuming that no job is running there.
resources_arginfo()	Generate the resources arginfo.
resources_subfields()	Generate the resources subfields.

arginfo
bind_context
check_finish_tag
check_if_recover
check_status
default_resources
deserialize
gen_command_env_cuda_devices
gen_local_script
gen_script
gen_script_command
gen_script_custom_flags_lines
gen_script_end
gen_script_env
gen_script_header
gen_script_wait
load_from_dict
load_from_json
map_dp_job_state
serialize
sub_script_cmd
sub_script_head

check_finish_tag(*job*)

check_if_recover(*submission*)

check_status(*job*)

do_submit(*job*)

submit a single job, assuming that no job is running there.

gen_local_script(*job*)

gen_script(*job*)

gen_script_header(*job*)

static **map_dp_job_state**(*status*)

class dpdispatcher.dp_cloud_server.**Lebesgue**(**args*, ***kwargs*)

Bases: *DpCloudServer*

Methods

do_submit(job)	submit a single job, assuming that no job is running there.
resources_arginfo()	Generate the resources arginfo.
resources_subfields()	Generate the resources subfields.

arginfo
bind_context
check_finish_tag
check_if_recover
check_status
default_resources
deserialize
gen_command_env_cuda_devices
gen_local_script
gen_script
gen_script_command
gen_script_custom_flags_lines
gen_script_end
gen_script_env
gen_script_header
gen_script_wait
load_from_dict
load_from_json
map_dp_job_state
serialize
sub_script_cmd
sub_script_head

6.1.8 dpdispatcher.dp_cloud_server_context module

class dpdispatcher.dp_cloud_server_context.**DpCloudServerContext**(**args*, ***kwargs*)

Bases: *BaseContext*

Methods

machine_arginfo()	Generate the machine arginfo.
machine_subfields()	Generate the machine subfields.

bind_submission	
check_file_exists	
check_finish	
check_home_file_exits	
clean	
download	
kill	
load_from_dict	
read_file	
read_home_file	
upload	
write_file	
write_home_file	
write_local_file	

bind_submission(*submission*)

check_file_exists(*fname*)

check_home_file_exits(*fname*)

clean()

download(*submission*)

kill(*cmd_pipes*)

classmethod load_from_dict(*context_dict*)

classmethod machine_subfields() → List[Argument]

Generate the machine subfields.

Returns

list[Argument]
machine subfields

read_file(*fname*)

read_home_file(*fname*)

upload(*submission*)

write_file(*fname*, *write_str*)

write_home_file(*fname*, *write_str*)

write_local_file(*fname*, *write_str*)

class dpdispatcher.dp_cloud_server_context.**LebesgueContext**(**args*, ***kwargs*)

Bases: *DpCloudServerContext*

Methods

machine_arginfo()	Generate the machine arginfo.
machine_subfields()	Generate the machine subfields.

bind_submission
check_file_exists
check_finish
check_home_file_exits
clean
download
kill
load_from_dict
read_file
read_home_file
upload
write_file
write_home_file
write_local_file

6.1.9 dpdispatcher.dpdisp module

dpdispatcher.dpdisp.**main**()

6.1.10 dpdispatcher.hdfs_cli module

class dpdispatcher.hdfs_cli.**HDFS**

Bases: object

Fundamental class for HDFS basic manipulation

Methods

copy_from_local(local_path, to_uri)	Returns: True on success Raises: on unexpected error
exists(uri)	Check existence of hdfs uri Returns: True on exists Raises: RuntimeError
mkdir(uri)	Make new hdfs directory Returns: True on success Raises: RuntimeError
remove(uri)	Check existence of hdfs uri Returns: True on exists Raises: RuntimeError

copy_to_local
move
read_hdfs_file

static copy_from_local(*local_path*, *to_uri*)

Returns: True on success Raises: on unexpected error

static copy_to_local(*from_uri*, *local_path*)

static exists(*uri*)

Check existence of hdfs uri Returns: True on exists Raises: RuntimeError

static mkdir(*uri*)

Make new hdfs directory Returns: True on success Raises: RuntimeError

static move(*from_uri*, *to_uri*)

static read_hdfs_file(*uri*)

static remove(*uri*)

Check existence of hdfs uri Returns: True on exists Raises: RuntimeError

6.1.11 dpdispatcher.hdfs_context module

class dpdispatcher.hdfs_context.**HDFSContext**(**args*, ***kwargs*)

Bases: *BaseContext*

Methods

download(submission[, check_exists, ...])	download backward files from HDFS root dir
machine_arginfo()	Generate the machine arginfo.
machine_subfields()	Generate the machine subfields.
upload(submission[, dereference])	upload forward files and forward command files to HDFS root dir

bind_submission	
check_file_exists	
check_finish	
clean	
get_job_root	
kill	
load_from_dict	
read_file	
write_file	

bind_submission(*submission*)

check_file_exists(*fname*)

clean()

download(*submission*, *check_exists=False*, *mark_failure=True*, *back_error=False*)

download backward files from HDFS root dir

Parameters

submission
[Submission class instance] represents a collection of tasks, such as backward file names

Returns

none

`get_job_root`()

`kill`(*job_id*)

`classmethod load_from_dict`(*context_dict*)

`read_file`(*fname*)

`upload`(*submission*, *dereference=True*)
upload forward files and forward command files to HDFS root dir

Parameters

submission
[Submission class instance] represents a collection of tasks, such as forward file names

Returns

none

`write_file`(*fname*, *write_str*)

6.1.12 dpdispatcher.lazy_local_context module

`class` dpdispatcher.lazy_local_context.`LazyLocalContext`(**args*, ***kwargs*)
Bases: *BaseContext*

Methods

machine_arginfo()	Generate the machine arginfo.
machine_subfields()	Generate the machine subfields.

bind_submission
block_call
block_checkcall
call
check_file_exists
check_finish
clean
download
get_job_root
get_return
kill
load_from_dict
read_file
upload
write_file

bind_submission(*submission*)

block_call(*cmd*)

block_checkcall(*cmd*)

call(*cmd*)

check_file_exists(*fname*)

check_finish(*proc*)

clean()

download(*jobs*, *check_exists=False*, *mark_failure=True*, *back_error=False*)

get_job_root()

get_return(*proc*)

kill(*job_id*)

classmethod load_from_dict(*context_dict*)

read_file(*fname*)

upload(*jobs*, *dereference=True*)

write_file(*fname*, *write_str*)

class dpdispatcher.lazy_local_context.**SPRetObj**(*ret*)

Bases: `object`

Methods

read	
readlines	

read()

readlines()

6.1.13 dpdispatcher.local_context module

class dpdispatcher.local_context.**LocalContext**(**args*, ***kwargs*)

Bases: *BaseContext*

Methods

machine_arginfo()	Generate the machine arginfo.
machine_subfields()	Generate the machine subfields.

bind_submission	
block_call	
block_checkcall	
call	
check_file_exists	
check_finish	
clean	
download	
download_	
get_job_root	
get_return	
kill	
load_from_dict	
read_file	
upload	
upload_	
write_file	

bind_submission(*submission*)

block_call(*cmd*)

block_checkcall(*cmd*)

call(*cmd*)

check_file_exists(*fname*)

check_finish(*proc*)

clean()

download(*submission*, *check_exists=False*, *mark_failure=True*, *back_error=False*)

download_(*job_dirs*, *remote_down_files*, *check_exists=False*, *mark_failure=True*, *back_error=False*)

get_job_root()

get_return(*proc*)

kill(*job_id*)

classmethod load_from_dict(*context_dict*)

read_file(*fname*)

upload(*submission*)

upload_(*job_dirs*, *local_up_files*, *dereference=True*)

write_file(*fname*, *write_str*)

class dpdispatcher.local_context.**SPRetObj**(*ret*)

Bases: object

Methods

read	
readlines	

read()

readlines()

6.1.14 dpdispatcher.lsf module

class dpdispatcher.lsf.**LSF**(**args*, ***kwargs*)

Bases: *Machine*

LSF batch

Methods

default_resources(resources)	
do_submit(job)	submit a single job, assuming that no job is running there.
resources_arginfo()	Generate the resources arginfo.
resources_subfields()	Generate the resources subfields.

arginfo	
bind_context	
check_finish_tag	
check_if_recover	
check_status	
deserialize	
gen_command_env_cuda_devices	
gen_script	
gen_script_command	
gen_script_custom_flags_lines	
gen_script_end	
gen_script_env	
gen_script_header	
gen_script_wait	
load_from_dict	
load_from_json	
serialize	
sub_script_cmd	
sub_script_head	

check_finish_tag(*job*)

check_status(*job*)

default_resources(*resources*)

do_submit(*job*)

submit a single job, assuming that no job is running there.

gen_script(*job*)

gen_script_header(*job*)

classmethod resources_subfields() → List[Argument]

Generate the resources subfields.

Returns

list[Argument]
resources subfields

sub_script_cmd(*res*)

sub_script_head(*res*)

6.1.15 dpdispatcher.machine module

class dpdispatcher.machine.**Machine**(**args*, ***kwargs*)

Bases: object

A machine is used to handle the connection with remote machines.

Parameters

context
[SubClass derived from BaseContext] The context is used to mainatin the connection with remote machine.

Methods

do_submit(job)	submit a single job, assuming that no job is running there.
resources_arginfo()	Generate the resources arginfo.
resources_subfields()	Generate the resources subfields.

arginfo	
bind_context	
check_finish_tag	
check_if_recover	
check_status	
default_resources	
deserialize	
gen_command_env_cuda_devices	
gen_script	
gen_script_command	
gen_script_custom_flags_lines	
gen_script_end	
gen_script_env	
gen_script_header	
gen_script_wait	
load_from_dict	
load_from_json	
serialize	
sub_script_cmd	
sub_script_head	

classmethod arginfo()

bind_context(*context*)

abstract check_finish_tag(***kwargs*)

check_if_recover(*submission*)

abstract check_status(*job*)

default_resources(*res*)

classmethod deserialize(*machine_dict*)

abstract do_submit(*job*)

submit a single job, assuming that no job is running there.

gen_command_env_cuda_devices(*resources*)

gen_script(*job*)

gen_script_command(*job*)

gen_script_custom_flags_lines(*job*)

gen_script_end(*job*)

gen_script_env(*job*)

abstract gen_script_header(*job*)

gen_script_wait(*resources*)

classmethod load_from_dict(*machine_dict*)

classmethod load_from_json(*json_path*)

options = {'DistributedShell', 'DpCloudServer', 'LSF', 'Lebesgue', 'PBS', 'Shell', 'Slurm', 'SlurmJobArray', 'Torque'}

classmethod resources_arginfo() → Argument

Generate the resources arginfo.

Returns

Argument
resources arginfo

classmethod resources_subfields() → List[Argument]

Generate the resources subfields.

Returns

list[Argument]
resources subfields

serialize(*if_empty_remote_profile=False*)

sub_script_cmd(*res*)

sub_script_head(*res*)

subclasses_dict = {'DistributedShell': <class 'dpdispatcher.distributed_shell.DistributedShell'>, 'DpCloudServer': <class 'dpdispatcher.dp_cloud_server.DpCloudServer'>, 'LSF': <class 'dpdispatcher.lsf.LSF'>, 'Lebesgue': <class 'dpdispatcher.dp_cloud_server.Lebesgue'>, 'PBS': <class 'dpdispatcher.pbs.PBS'>, 'Shell': <class 'dpdispatcher.shell.Shell'>, 'Slurm': <class 'dpdispatcher.slurm.Slurm'>, 'SlurmJobArray': <class 'dpdispatcher.slurm.SlurmJobArray'>, 'Torque': <class 'dpdispatcher.pbs.Torque'>, 'distributedshell': <class 'dpdispatcher.distributed_shell.DistributedShell'>, 'dpcloudserver': <class 'dpdispatcher.dp_cloud_server.DpCloudServer'>, 'lebesgue': <class 'dpdispatcher.dp_cloud_server.Lebesgue'>, 'lsf': <class 'dpdispatcher.lsf.LSF'>, 'pbs': <class 'dpdispatcher.pbs.PBS'>, 'shell': <class 'dpdispatcher.shell.Shell'>, 'slurm': <class 'dpdispatcher.slurm.Slurm'>, 'slurmjobarray': <class 'dpdispatcher.slurm.SlurmJobArray'>, 'torque': <class 'dpdispatcher.pbs.Torque'>}

6.1.16 dpdispatcher.pbs module

class dpdispatcher.pbs.**PBS**(**args*, ***kwargs*)

Bases: *Machine*

Methods

do_submit(job)	submit a single job, assuming that no job is running there.
resources_arginfo()	Generate the resources arginfo.
resources_subfields()	Generate the resources subfields.

arginfo
bind_context
check_finish_tag
check_if_recover
check_status
default_resources
deserialize
gen_command_env_cuda_devices
gen_script
gen_script_command
gen_script_custom_flags_lines
gen_script_end
gen_script_env
gen_script_header
gen_script_wait
load_from_dict
load_from_json
serialize
sub_script_cmd
sub_script_head

check_finish_tag(*job*)

check_status(*job*)

default_resources(*resources*)

do_submit(*job*)

submit a single job, assuming that no job is running there.

gen_script(*job*)

gen_script_header(*job*)

class dpdispatcher.pbs.**Torque**(**args*, ***kwargs*)

Bases: *PBS*

Methods

do_submit(job)	submit a single job, assuming that no job is running there.
resources_arginfo()	Generate the resources arginfo.
resources_subfields()	Generate the resources subfields.

arginfo
bind_context
check_finish_tag
check_if_recover
check_status
default_resources
deserialize
gen_command_env_cuda_devices
gen_script
gen_script_command
gen_script_custom_flags_lines
gen_script_end
gen_script_env
gen_script_header
gen_script_wait
load_from_dict
load_from_json
serialize
sub_script_cmd
sub_script_head

check_status(*job*)

6.1.17 dpdispatcher.shell module

class dpdispatcher.shell.**Shell**(**args*, ***kwargs*)

Bases: *Machine*

Methods

do_submit(job)	submit a single job, assuming that no job is running there.
resources_arginfo()	Generate the resources arginfo.
resources_subfields()	Generate the resources subfields.

arginfo	
bind_context	
check_finish_tag	
check_if_recover	
check_status	
default_resources	
deserialize	
gen_command_env_cuda_devices	
gen_script	
gen_script_command	
gen_script_custom_flags_lines	
gen_script_end	
gen_script_env	
gen_script_header	
gen_script_wait	
load_from_dict	
load_from_json	
serialize	
sub_script_cmd	
sub_script_head	

check_finish_tag(*job*)

check_status(*job*)

default_resources(*resources*)

do_submit(*job*)

submit a single job, assuming that no job is running there.

gen_script(*job*)

gen_script_header(*job*)

6.1.18 dpdispatcher.slurm module

class dpdispatcher.slurm.**Slurm**(**args*, ***kwargs*)

Bases: *Machine*

Methods

do_submit(job[, retry, max_retry])	submit a single job, assuming that no job is running there.
resources_arginfo()	Generate the resources arginfo.
resources_subfields()	Generate the resources subfields.

arginfo	
bind_context	
check_finish_tag	
check_if_recover	
check_status	
default_resources	
deserialize	
gen_command_env_cuda_devices	
gen_script	
gen_script_command	
gen_script_custom_flags_lines	
gen_script_end	
gen_script_env	
gen_script_header	
gen_script_wait	
load_from_dict	
load_from_json	
serialize	
sub_script_cmd	
sub_script_head	

check_finish_tag(*job*)

check_status(*job*, *retry=0*, *max_retry=3*)

default_resources(*resources*)

do_submit(*job*, *retry=0*, *max_retry=3*)

submit a single job, assuming that no job is running there.

gen_script(*job*)

gen_script_header(*job*)

classmethod resources_subfields() → List[Argument]

Generate the resources subfields.

Returns

list[Argument]
resources subfields

class dpdispatcher.slurm.**SlurmJobArray**(**args*, ***kwargs*)

Bases: *Slurm*

Slurm with job array enabled for multiple tasks in a job

Methods

do_submit(job[, retry, max_retry])	submit a single job, assuming that no job is running there.
resources_arginfo()	Generate the resources arginfo.
resources_subfields()	Generate the resources subfields.

arginfo	
bind_context	
check_finish_tag	
check_if_recover	
check_status	
default_resources	
deserialize	
gen_command_env_cuda_devices	
gen_script	
gen_script_command	
gen_script_custom_flags_lines	
gen_script_end	
gen_script_env	
gen_script_header	
gen_script_wait	
load_from_dict	
load_from_json	
serialize	
sub_script_cmd	
sub_script_head	

check_finish_tag(*job*)

check_status(*job*, *retry=0*, *max_retry=3*)

gen_script_command(*job*)

gen_script_end(*job*)

gen_script_header(*job*)

6.1.19 dpdispatcher.ssh_context module

class dpdispatcher.ssh_context.**SSHContext**(**args*, ***kwargs*)

Bases: *BaseContext*

Attributes

sftp
ssh

Methods

block_checkcall(cmd[, asynchronously, ...])	Run command with arguments.
machine_arginfo()	Generate the machine arginfo.
machine_subfields()	Generate the machine subfields.

bind_submission
block_call
call
check_file_exists
check_finish
clean
close
download
get_job_root
get_return
kill
load_from_dict
read_file
upload
write_file

bind_submission(*submission*)

block_call(*cmd*)

block_checkcall(*cmd*, *asynchronously=False*, *stderr_whitelist=None*)

Run command with arguments. Wait for command to complete. If the return code was zero then return, otherwise raise RuntimeError.

Parameters

cmd: str
The command to run.

asynchronously: bool, optional, default=False
Run command asynchronously. If True, *nohup* will be used to run the command.

call(*cmd*)

check_file_exists(*fname*)

check_finish(*cmd_pipes*)

clean()

close()

download(*submission*, *check_exists=False*, *mark_failure=True*, *back_error=False*)

get_job_root()

get_return(*cmd_pipes*)

kill(*cmd_pipes*)

classmethod load_from_dict(*context_dict*)

classmethod machine_subfields() → List[Argument]

Generate the machine subfields.

Returns

list[Argument]
machine subfields

read_file(*fname*)

property sftp

property ssh

upload(*submission*, *dereference=True*)

write_file(*fname*, *write_str*)

class dpdispatcher.ssh_context.**SSHSession**(*hostname*, *username*, *password=None*, *port=22*, *key_filename=None*, *passphrase=None*, *timeout=10*, *totp_secret=None*)

Bases: object

Attributes

remote
rsync_available
sftp
Returns sftp.

Methods

exec_command(cmd[, retry])	Calling self.ssh.exec_command but has an exception check.

arginfo	
close	
ensure_alive	
get	
get_ssh_client	
put	

static arginfo()

close()

ensure_alive(*max_check=10*, *sleep_time=10*)

exec_command(*cmd*, *retry=0*)

Calling self.ssh.exec_command but has an exception check.

get(*from_f*, *to_f*)

get_ssh_client()

put(*from_f*, *to_f*)

property remote: str

property rsync_available: bool

property sftp

Returns sftp. Open a new one if not existing.

6.1.20 dpdispatcher.submission module

class dpdispatcher.submission.**Job**(*job_task_list*, *, *resources*, *machine=None*)

Bases: object

Job is generated by Submission automatically. A job ususally has many tasks and it may request computing resources from job scheduler systems. Each Job can generate a script file to be submitted to the job scheduler system or executed locally.

Parameters

job_task_list
[list of Task] the tasks belonging to the job

resources
[Resources] the machine resources. Passed from Submission when it constructs jobs.

machine
[machine] machine object to execute the job. Passed from Submission when it constructs jobs.

Methods

deserialize(job_dict[, machine])	convert the job_dict to a Submission class object
get_job_state()	get the jobs.
serialize([if_static])	convert the Task class instance to a dictionary.

get_hash	
handle_unexpected_job_state	
job_to_json	
register_job_id	
submit_job	

classmethod deserialize(*job_dict*, *machine=None*)

convert the job_dict to a Submission class object

Parameters

submission_dict
[dict] path-like, the base directory of the local tasks

Returns

submission
[Job] the Job class instance converted from the job_dict

`get_hash()`

`get_job_state()`

get the jobs. Usually, this method will query the database of slurm or pbs job scheduler system and get the results.

Notes

this method will not submit or resubmit the jobs if the job is unsubmitted.

`handle_unexpected_job_state()`

`job_to_json()`

`register_job_id(job_id)`

`serialize(if_static=False)`

convert the Task class instance to a dictionary.

Parameters

if_static
[bool] whether dump the job runtime infomation (job_id, job_state, fail_count, job_uuid etc.) to the dictionary.

Returns

task_dict
[dict] the dictionary converted from the Task class instance

`submit_job()`

`class dpdispatcher.submission.Resources(number_node, cpu_per_node, gpu_per_node, queue_name, group_size, *, custom_flags=[], strategy={'if_cuda_multi_devices': False, 'ratio_unfinished': 0.0}, para_deg=1, module_unload_list=[], module_purge=False, module_list=[], source_list=[], envs={}, wait_time=0, **kwargs)`

Bases: `object`

Resources is used to describe the machine resources we need to do calculations.

Parameters

number_node
[int] The number of node need for each *job*.

cpu_per_node
[int] cpu numbers of each node.

gpu_per_node
[int] gpu numbers of each node.

queue_name
[str] The queue name of batch job scheduler system.

group_size
[int] The number of *tasks* in a *job*.

custom_flags
[list of Str] The extra lines pass to job submitting script header

strategy
[dict] strategies we use to generation job submitting scripts. if_cuda_multi_devices : bool

If there are multiple nvidia GPUS on the node, and we want to assign the tasks to different GPUS. If true, dpdispatcher will manually export environment variable CUDA_VISIBLE_DEVICES to different task. Usually, this option will be used with Task.task_need_resources variable simultaneously.

ratio_unfinished
[float] The ratio of *jobs* that can be unfinished.

para_deg
[int] Decide how many tasks will be run in parallel. Usually run with *strategy['if_cuda_multi_devices']*

source_list
[list of Path] The env file to be sourced before the command execution.

wait_time
[int] The waitting time in second after a single task submitted. Default: 0.

Methods

arginfo	
deserialize	
load_from_dict	
load_from_json	
serialize	

`static arginfo`(*detail_kwargs=True*)

`classmethod deserialize`(*resources_dict*)

`classmethod load_from_dict`(*resources_dict*)

`classmethod load_from_json`(*json_file*)

`serialize`()

`class` dpdispatcher.submission.`Submission`(*work_base*, *machine=None*, *resources=None*, *forward_common_files=[]*, *backward_common_files=[]*, *, *task_list=[]*)

Bases: `object`

A submission represents a collection of tasks. These tasks usually locate at a common directory. And these Tasks may share common files to be uploaded and downloaded.

Parameters

work_base
[Path] the base directory of the local tasks. It is usually the dir name of project .

machine
[Machine] machine class object (for example, PBS, Slurm, Shell) to execute the jobs. The machine can still be bound after the instantiation with the bind_submission method.

resources
[Resources] the machine resources (cpu or gpu) used to generate the slurm/pbs script

forward_common_files
[list] the common files to be uploaded to other computers before the jobs begin

backward_common_files
[list] the common files to be downloaded from other computers after the jobs finish

task_list
[list of Task] a list of tasks to be run.

Methods

bind_machine(machine)	bind this submission to a machine.
check_all_finished()	check whether all the jobs in the submission.
deserialize(submission_dict[, machine])	convert the submission_dict to a Submission class object
generate_jobs()	After tasks register to the self.belonging_tasks, This method generate the jobs and add these jobs to self.belonging_jobs.
handle_unexpected_submission_state()	handle unexpected job state of the submission.
run_submission(*[, exit_on_submit, clean])	main method to execute the submission.
serialize([if_static])	convert the Submission class instance to a dictionary.
update_submission_state()	check whether all the jobs in the submission.

check_ratio_unfinished	
clean_jobs	
download_jobs	
get_hash	
register_task	
register_task_list	
remove_unfinished_jobs	
submission_from_json	
submission_to_json	
try_recover_from_json	
upload_jobs	

`bind_machine`(*machine*)

bind this submission to a machine. update the machine's context remote_root and local_root.

Parameters

machine
[Machine] the machine to bind with

`check_all_finished`()

check whether all the jobs in the submission.

Notes

This method will not handle unexpected job state in the submission.

check_ratio_unfinished(*ratio_unfinished*)

clean_jobs()

classmethod deserialize(*submission_dict*, *machine=None*)

convert the submission_dict to a Submission class object

Parameters

submission_dict
[dict] path-like, the base directory of the local tasks

Returns

submission
[Submission] the Submission class instance converted from the submission_dict

download_jobs()

generate_jobs()

After tasks register to the self.belonging_tasks, This method generate the jobs and add these jobs to self.belonging_jobs. The jobs are generated by the tasks randomly, and there are self.resources.group_size tasks in a task. Why we randomly shuffle the tasks is under the consideration of load balance. The random seed is a constant (to be concrete, 42). And this insures that the jobs are equal when we re-run the program.

get_hash()

handle_unexpected_submission_state()

handle unexpected job state of the submission. If the job state is unsubmitted, submit the job. If the job state is terminated (killed unexpectly), resubmit the job. If the job state is unknown, raise an error.

register_task(*task*)

register_task_list(*task_list*)

remove_unfinished_jobs()

run_submission(*, *exit_on_submit=False*, *clean=True*)

main method to execute the submission. First, check whether old Submission exists on the remote machine, and try to recover from it. Second, upload the local files to the remote machine where the tasks to be executed. Third, run the submission defined previously. Forth, wait until the tasks in the submission finished and download the result file to local directory. if exit_on_submit is True, submission will exit.

serialize(*if_static=False*)

convert the Submission class instance to a dictionary.

Parameters

if_static
[bool] whether dump the job runtime infomation (like job_id, job_state, fail_count) to the dictionary.

Returns

submission_dict
[dict] the dictionary converted from the Submission class instance

classmethod submission_from_json(*json_file_name='submission.json'*)

submission_to_json()

try_recover_from_json()

update_submission_state()

check whether all the jobs in the submission.

Notes

this method will not handle unexpected (like resubmit terminated) job state in the submission.

upload_jobs()

class dpdispatcher.submission.**Task**(*command*, *task_work_path*, *forward_files=[]*, *backward_files=[]*, *outlog='log'*, *errlog='err'*)

Bases: `object`

A task is a sequential command to be executed, as well as the files it depends on to transmit forward and backward.

Parameters

command
[Str] the command to be executed.

task_work_path
[Path] the directory of each file where the files are dependent on.

forward_files
[list of Path] the files to be transmitted to remote machine before the command execute.

backward_files
[list of Path] the files to be transmitted from remote machine after the comand finished.

outlog
[Str] the filename to which command redirect stdout

errlog
[Str] the filename to which command redirect stderr

Methods

deserialize(task_dict)	convert the task_dict to a Task class object

arginfo	
get_hash	
load_from_dict	
load_from_json	
serialize	

static arginfo()

classmethod deserialize(*task_dict*)

convert the task_dict to a Task class object

Parameters

task_dict
[dict] the dictionary which contains the task information

Returns

task
[Task] the Task class instance converted from the task_dict

get_hash()

classmethod load_from_dict(*task_dict: dict*) → *Task*

classmethod load_from_json(*json_file*)

serialize()

6.1.21 dpdispatcher.utils module

dpdispatcher.utils.**generate_totp**(*secret: str*, *period: int = 30*, *token_length: int = 6*) → int

Generate time-based one time password (TOTP) from the secret.

Some HPCs use TOTP for two-factor authentication for safety.

Parameters

secret: str
The encoded secret provided by the HPC. It's usually extracted from a 2D code and base32 encoded.

period: int, default=30
Time period where the code is valid in seconds.

token_length: int, default=6
The token length.

Returns

token: int
The generated token.

References

https://github.com/lepture/otpauth/blob/49914d83d36dbcd33c9e26f65002b21ce09a6303/otpauth.py#L143-L160

dpdispatcher.utils.**get_sha256**(*filename*)

Get sha256 of a file.

Parameters

filename: str
The filename.

Returns

sha256: str
The sha256.

dpdispatcher.utils.**rsync**(*from_file: str*, *to_file: str*)

Call rsync to transfer files.

Parameters

from_file: str
SRC

to_file: str
DEST

Raises

RuntimeError
when return code is not 0

dpdispatcher.utils.**run_cmd_with_all_output**(*cmd*, *shell=True*)

CHAPTER

SEVEN

RUNNING THE DEEPMD-KIT ON THE EXPANSE CLUSTER

Expanse is a cluster operated by the San Diego Supercomputer Center. Here we provide an example to run jobs on the expanse.

The machine parameters are provided below. Expanse uses the SLURM workload manager for job scheduling. *remote_root* has been created in advance. It's worth metioned that we do not recommend to use the password, so SSH keys are used instead to improve security.

```
{
  "batch_type": "Slurm",
  "local_root": "./",
  "remote_root": "/expanse/lustre/scratch/njzjz/temp_project/dpgen_workdir",
  "clean_asynchronously": true,
  "context_type": "SSHContext",
  "remote_profile": {
    "hostname": "login.expanse.sdsc.edu",
    "username": "njzjz",
    "port": 22
  }
}
```

Expanse's standard compute nodes are each powered by two 64-core AMD EPYC 7742 processors and contain 256 GB of DDR4 memory. Here, we request one node with 32 cores and 16 GB memory from the `shared` partition. Expanse does not support `--gres=gpu:0` command, so we use *custom_gpu_line* to customize the statement.

```
{
  "number_node": 1,
  "cpu_per_node": 1,
  "gpu_per_node": 0,
  "queue_name": "shared",
  "group_size": 1,
  "custom_flags": [
    "#SBATCH -c 32",
    "#SBATCH --mem=16G",
    "#SBATCH --time=48:00:00",
    "#SBATCH --account=rut149",
    "#SBATCH --requeue"
  ],
  "source_list": [
    "activate /home/njzjz/deepmd-kit"
  ],
  "envs": {
```

(continues on next page)

(continued from previous page)

```
    "OMP_NUM_THREADS": 4,
    "TF_INTRA_OP_PARALLELISM_THREADS": 4,
    "TF_INTER_OP_PARALLELISM_THREADS": 8,
    "DP_AUTO_PARALLELIZATION": 1
  },
  "batch_type": "Slurm",
  "kwargs": {
    "custom_gpu_line": "#SBATCH --gpus=0"
  }
}
```

The following task parameter runs a DeePMD-kit task, forwarding an input file and backwarding graph files. Here, the data set will be used among all the tasks, so it is not included in the *forward_files*. Instead, it should be included in the submission's forward_common_files.

```
{
    "command": "dp train input.json && dp freeze && dp compress",
    "task_work_path": "model1/",
    "forward_files": [
      "input.json"
    ],
    "backward_files": [
      "frozen_model.pb",
      "frozen_model_compressed.pb"
    ],
    "outlog": "log",
    "errlog": "err"
}
```

CHAPTER

EIGHT

RUNNING GAUSSIAN 16 WITH FAILURE ALLOWED

Typically, a task will retry three times if the exit code is not zero. Sometimes, one may allow non-zero code. For example, when running large amounts of Gaussian 16 single-point calculation tasks, some of the Gaussian 16 tasks may throw SCF errors and return a non-zero code. One can append `||:` to the command:

```
{
    "command": "g16 < input > output ||:",
    "task_work_path": "p1/",
    "forward_files": [
      "input"
    ],
    "backward_files": [
      "output"
    ]
}
```

This command ensures the task will always provide zero code.

CHAPTER

NINE

RUNNING MULTIPLE MD TASKS ON A GPU WORKSTATION

In this example, we are going to show how to run multiple MD tasks on a GPU workstation. This workstation does not install any job scheduling packages installed, so we will use `Shell` as *batch_type*.

```
{
  "batch_type": "Shell",
  "local_root": "./",
  "remote_root": "/data2/jinzhe/dpgen_workdir",
  "clean_asynchronously": true,
  "context_type": "SSHContext",
  "remote_profile": {
    "hostname": "mandu.iqb.rutgers.edu",
    "username": "jz748",
    "port": 22
  }
}
```

The workstation has 48 cores of CPUs and 8 RTX3090 cards. Here we hope each card runs 6 tasks at the same time, as each task does not consume too many GPU resources. Thus, *strategy/if_cuda_multi_devices* is set to `true` and *para_deg* is set to 6.

```
{
  "number_node": 1,
  "cpu_per_node": 48,
  "gpu_per_node": 8,
  "queue_name": "shell",
  "group_size": 0,
  "strategy": {
    "if_cuda_multi_devices": true
  },
  "source_list": [
    "activate /home/jz748/deepmd-kit"
  ],
  "envs": {
    "OMP_NUM_THREADS": 1,
    "TF_INTRA_OP_PARALLELISM_THREADS": 1,
    "TF_INTER_OP_PARALLELISM_THREADS": 1
  },
  "para_deg": 6
}
```

Note that *group_size* should be set to `0` (means infinity) to ensure there is only one job and avoid running multiple jobs

at the same time.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

T

U

V

W

Z