DPDispatcher

Deep Modeling

Jun 18, 2021

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=reasources,
    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

machine:

type: `dict`
argument path: `machine`

batch_type:

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

The batch job system type. Option: Slurm, PBS, LSF, Shell, DpCloudServer

context_type:

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

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

local_root:

type: `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.

remote_profile:

type: `dict`
argument path: `machine/remote_profile`

The information used to maintain the connection with remote machine. Only needed when context is ssh.

hostname:

type: `str`
argument path: `machine/remote_profile/hostname`

hostname or ip of ssh connection.

username:

type: str
argument path: machine/remote_profile/username

username of target linux system

password:

type: str, optional
argument path: machine/remote_profile/password

password of linux system

port:

type: int, optional, default: 22
argument path: machine/remote_profile/port

ssh connection port.

key_filename:

type: NoneType | str, optional, default: None
argument path: machine/remote_profile/key_filename

key_filename used by ssh connection

passphrase:

type: NoneType | str, optional, default: None
argument path: machine/remote_profile/passphrase

passphrase used by ssh connection

timeout:

type: int, optional, default: 10
argument path: machine/remote_profile/timeout

timeout of ssh connection

clean_asynchronously:

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

Clean the remote directory asynchronously after the job finishes.

CHAPTER

FOUR

RESOURCES PARAMETERS

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*.

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: `True`
argument path: `resources/strategy/if_cuda_multi_devices`

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_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

CHAPTER

FIVE

TASK PARAMETERS

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

dpdispatcher.**info**()

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

An enumeration.

completing = 6

finished = 5

running = 3

terminated = 4

unknown = 100

unsubmitted = 1

waiting = 2

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

bind_submission(*submission*)

check_finish(*proc*)

clean()

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

kill(*proc*)

classmethod load_from_dict(*context_dict*)

read_file(*fname*)

subclasses_dict = {'LazyLocal': <class 'dpdispatcher.lazy_local_context.LazyLocalContext'>, 'LazyLocalContext': <class 'dpdispatcher.lazy_local_context.LazyLocalContext'>, 'Local': <class 'dpdispatcher.local_context.LocalContext'>, 'LocalContext': <class 'dpdispatcher.local_context.LocalContext'>, 'SSH': <class 'dpdispatcher.ssh_context.SSHContext'>, 'SSHContext': <class 'dpdispatcher.ssh_context.SSHContext'>, 'lazylocal': <class 'dpdispatcher.lazy_local_context.LazyLocalContext'>, 'lazylocalcontext': <class 'dpdispatcher.lazy_local_context.LazyLocalContext'>, 'local': <class 'dpdispatcher.local_context.LocalContext'>, 'localcontext': <class 'dpdispatcher.local_context.LocalContext'>, 'ssh': <class 'dpdispatcher.ssh_context.SSHContext'>, 'sshcontext': <class 'dpdispatcher.ssh_context.SSHContext'>}

upload(*submission*)

write_file(*fname*, *write_str*)

dpdispatcher.dpdisp.**main**()

class dpdispatcher.lazy_local_context.**LazyLocalContext**(**args*, ***kwargs*)

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(*proc*)

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*)

read()

readlines()

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

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(*proc*)

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*)

read()

readlines()

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

LSF batch

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*)

sub_script_cmd(*res*)

sub_script_head(*res*)

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

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

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

static arginfo()

bind_context(*context*)

check_finish_tag(***kwargs*)

check_if_recover(*submission*)

check_status(*job*)

default_resources(*res*)

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*)

gen_script_header(*job*)

gen_script_wait(*resources*)

classmethod load_from_dict(*machine_dict*)

classmethod load_from_json(*json_path*)

sub_script_cmd(*res*)

sub_script_head(*res*)

subclasses_dict = {'LSF': <class 'dpdispatcher.lsf.LSF'>, 'PBS': <class 'dpdispatcher.pbs.PBS'>, 'Shell': <class 'dpdispatcher.shell.Shell'>, 'Slurm': <class 'dpdispatcher.slurm.Slurm'>, 'lsf': <class 'dpdispatcher.lsf.LSF'>, 'pbs': <class 'dpdispatcher.pbs.PBS'>, 'shell': <class 'dpdispatcher.shell.Shell'>, 'slurm': <class 'dpdispatcher.slurm.Slurm'>}

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

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

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

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*)

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

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.

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*)

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*)

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_ssh_client()

property sftp
Returns sftp. Open a new one if not existing.

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

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.

`classmethod deserialize`(*job_dict*, *machine=None*)
convert the job_dict to a Submission class object

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

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.

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.

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

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}*, *para_deg=1*, *module_unload_list=[]*, *module_list=[]*, *source_list=[]*, *envs={}*, ***kwargs*)
Resources is used to describe the machine resources we need to do calculations.

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.

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.

static arginfo()

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=[]*)

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.

work_base [Path] path-like, the base directory of the local tasks

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.

bind_machine(*machine*)
bind this submission to a machine. update the machine's context remote_root and local_root.

machine [Machine] the machine to bind with

check_all_finished()
check whether all the jobs in the submission.

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

clean_jobs()

classmethod deserialize(*submission_dict*, *machine=None*)
convert the submission_dict to a Submission class object

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

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()

get_submission_state()
check whether all the jobs in the submission.

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

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*)

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.

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

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()

upload_jobs()

class dpdispatcher.submission.**Task**(*command*, *task_work_path*, *forward_files=[]*, *backward_files=[]*, *outlog='log'*, *errlog='err'*)
A task is a sequential command to be executed, as well as the files it depends on to transmit forward and backward.

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

static arginfo()

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

task_dict [dict] the dictionary which contains the task information

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

get_hash()

serialize()

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

DATAERR = '2002'

DBERR = '2000'

IOERR = '2003'

LOGINERR = '2100'

NODATA = '2300'

OK = '0000'

PARAMERR = '2101'

PWDERR = '2104'

REQERR = '2200'

ROLEERR = '2103'

THIRDERR = '2001'

UNDERDEBUG = '2301'

UNKOWNERR = '2400'

USERERR = '2102'

VERIFYERR = '2105'

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

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

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z