CQSim Reinforcement Learning Module implementation using OpenAi Gym Environment

This manual discusses RL enhancements to CQSim, a trace-based HPC scheduling simulator. The goal of these enhancements is to reduce the complexity and cost of implementing a RL agent for HPC scheduling and open up this topic for further analysis. Here we detail the implementations of these enhancements along with some supplemental findings and future work.

Requirements:

Python version - 3.7.x
Python dependencies:

- gym==0.18.0
- h5py==2.10.0
- Keras==2.0.6
- matplotlib==3.4.1
- numpy==1.19.5
- pandas==1.2.3
- tensorflow==1.14.0

Coupling CQSim with OpenAi Gym Environment:

This section discusses the approach used to couple CqSim Simulator with OpenAi Gym Interface to implement a Custom Gym Environment which can enable training of Reinforcement Learning Models using Libraries such as Tensorflow.

The notion behind the implementation is similar to the Producer-Consumer problem where the CqSim behaves as Consumer and Gym.Env.Step provides the Producer. Both CqSim and Gym.Env run on 2 separate threads and interact using shared variables.

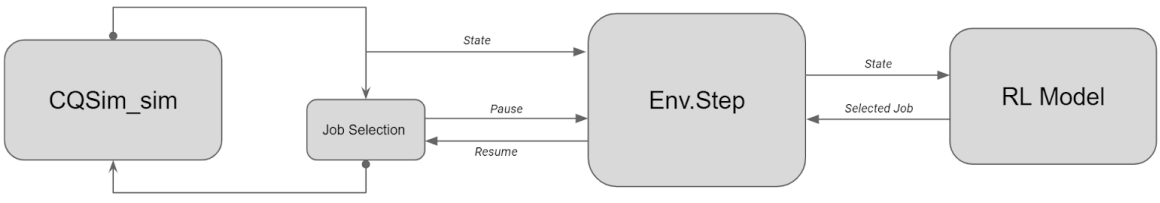

Fig. 2 Overview of the flow for the multithreading approach.

Code Structure:

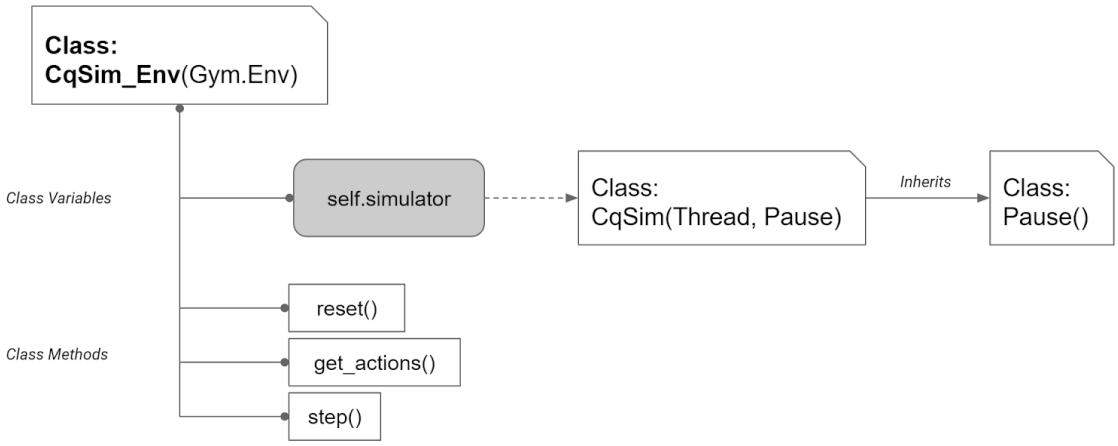

Fig. 3 CqSim_Env Class structure.

The class **CqSim_Env** inherits the Gym.Env interface and implements all the abstract methods. It holds the **CqSim.Cqsim_sim** class object as its variable - **self.simulator**. From the original implementation in CQSim Code, the class Cqsim.Cqsim_sim is altered in the following ways :

- Cqsim_sim class inherits the classes **Thread** and **Pause.**
- **Thread** class enables implementation of the object to be run on a parallel thread.
- The **Thread.run** method from the parent class is overwritten and executes the class method - **Cqsim_sim.cqsim_sim().** Which is the main function that coordinates the different components of the simulator.
- Class **Pause()** is implemented to control the synchronization between the **Cqsim_Env** main thread and **Cqsim_Env.simulator** thread.

Implementation:

- Once **Cqsim_Env** is initialized, it initiates the thread for **Cqsim_sim.**
- **Cqsim_sim** thread runs through the simulation steps :
    - Imports submit event.
    - Reaches event_job().
    - Initiates start_scan().
- At this point the **Cqsim_Env** thread is notified and **Cqsim_sim** thread gets paused and waits for the *notification* for resuming from **Cqsim_Env.step()**
- **Cqsim_Env** thread is initially paused and waits for the notification from Cqsim_sim thread.

- Once **Cqsim_Env** is resumed, it takes the current state of **Cqsim_sim** using the shared variable **Cqsim_Env.simulator.simulator_state.**
- This state is passed to the Model for making the selection of the job, and the shared variable is updated.
- At this point the **Cqsim_Env** thread is paused and the simulator thread resumes and reaches to the next event or next state.

Thread Pause:

The **Pause** class helps implement the pausing and resuming of the Cqsim_Env(Producer) and Cqsim_sim(Consumer) threads and synchronize their communication.

The **Pause** class makes use of python Conditional Variables. It maintains 2 conditional variables - **Pause.prod_cv** and **Pause.cons_cv.**

In order to implement the Producer-Consumer approach, when one condition variable is paused the other one is resumed and vice-versa.

The **Pause** class also maintains a special variable - intial_check (boolean).
This variable is used to pause the Cqsim_env.get_state() method to the point when the Cqsim_sim thread is initiated and the first set of jobs is loaded.

Step Function Components:

- Gym State:
    - A separate class has been created to maintain the necessary variables for managing a complete state information required for the RL model. An object of this class is instantiated at each event when a decision is required to be made by the RL model.
- State Reward:
    - In order to compute the reward a for the reward at a particular step, the function ‘get_reward(action)’ is defined as part of the GymState class. The function expects as an argument the action suggested by the RL model and then using the state information calculates the reward.
- Render Info:
    - An important feat of this implementation is to provide a visual representation of the current state. The render function accesses the current state information and using the graphing library Matplotlib, is able to provide a comprehensible set of plots describing the important

parameters which can be visually examined during the run time of the RL Model training. This can help the user understand how effectively the model is being trained and can also analyze the updates on the CQsimulator state, environment and RL model parameters during runtime.

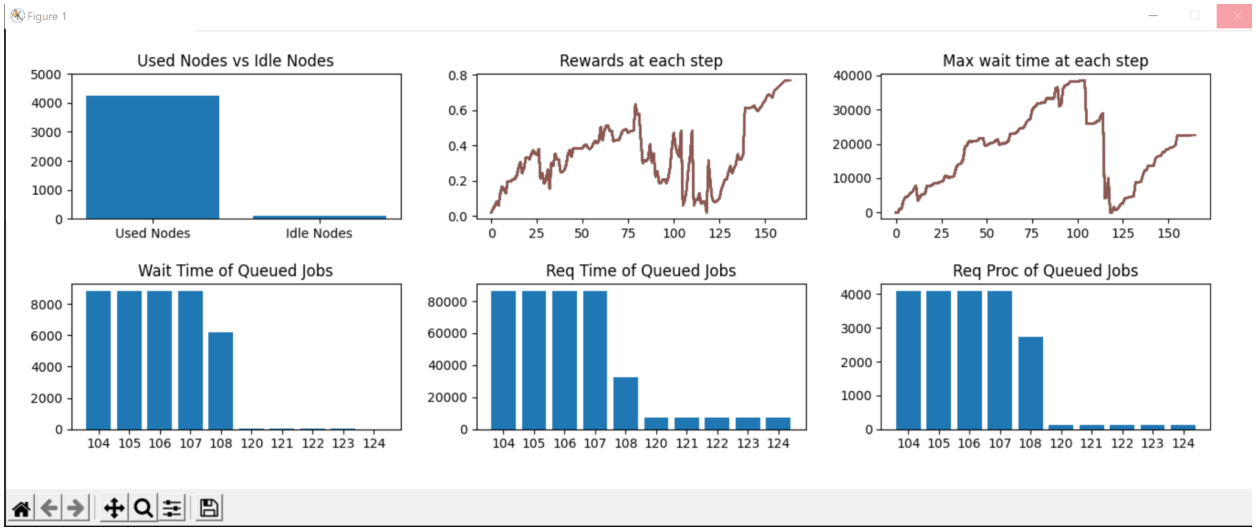

Fig 2: A screenshot of the Rendered Info at a random step while running GymEnv.

Modifications and new Components Added to CQSim:

Initiation Phase :

- **config_n.set <Modification>:**
  - Parameter Added :
    - ***path_weights*** : [str] folder path where the weights to be loaded in the model are saved. Just like *path_fmt* and *path_out* , this path is relative to *path_data.*

- **config_sys.set <Modification>:**
  - Parameter Added :
    - ***input_dim*** : [int] Same as DRAS PG
    - ***job_info_siz*** : [int] Same as DRAS PG
    - ***is_training*** : [int] 1 - if the model should be trained, 0 - otherwise.

        - ***input_weight_file*** : [str] Name of the weights file to be loaded. Just as in DRAS PG, only the common part of the 2 weights file is required. "_policy_.h5" and "_predict_.h5" are concatenated automatically. Weights are not loaded if the parameter is empty or not provided.

        - ***output_weight_file****=new_weights* : [str] Name of the weights file to be saved. Just as in DRAS PG, only the common part of the 2 weights file is required. "_policy_.h5" and "_predict_.h5" are concatenated automatically. Weights are not saved if the parameter is empty or not provided.

- **cqsim.py <Modification> :**
    - Code is modified to load the added config parameters in to *inputPara dict.*

- **cqsim_main.py <Modification> :**
    - # *Cqsim Simulator* section is modified.
    - Parameters Added :
        - *job_cols* : [int] Same as DRAS PG.
        - *window_size* : [int] Input size for the model.
        - *is_training* : [boolean] As defined above.
        - *Input_weight_file* : [str] Complete relative path for the input weight file name. Suffix for individual model is added later.
        - *Output_weight_file* : [str] Complete relative path for the output weight file name. Suffix for individual model is added later.
    - Function Call Modified:
        - Instead of call to simulator directly :
            module_sim = Class_Cqsim_sim.Cqsim_sim
            The call is now made to Trainer.PG_Trainer.model_engine()
            Discussed in detail below.

Architecture Phase:

The goal of this implementation is to provide modularity for easier and faster development of multiple and diverse Reinforcement Learning Models. Hence the following architecture is adapted.

- **CqGym <Addition> :**
  - Main objective of this module is to provide a gym environment as standardised by OpenAI Gym interface. This module connects with the CQSim_sim.py as discussed above.

- **CqGym.Gym.py :**
  - This file provides the class CqGym.Gym.*CqsimEnv* which inherits and implements the necessary functionalities for the OpenAi Gym environment:
  - render() : The rendering functionality is complex task by itself and hence has been completely shifted to another class - *GymGraphics*. Discussed below.
  - get_state(): This function creates a GymState Object for maintaining the current state of the Simulator.
  - step(): Receive param *action* [int] as the simulator_wait_que index for the selected job. Places the selected job in the beginning of the simulator_wait_que. Returns :
    - next GymState.
    - Boolean - if the simulation is complete.
    - Reward for the current action.

- **CqGym.GymGraphics.py :**
  - This class helps render visual graphics to view the current progress in GymStates at each step. The Class gets initiated with the following 2 variables:
    - render_interval : Size of the interval after which the Graphics should be presented.
    - render_pause : Time period for which the current rendering should be paused for displaying.

- **CqGym.GymState.py :**
  - This class maintains all the data variables needed to define the state of the simulator at a particular time. The current time is also maintained.
  - *define_state():* Function to process all the raw data collected from the simulator and define state along with the feature such that it can be directly used by RL Model.
  - All the other functions are directly derived from DRAS PG.

- **<u>Models</u> <Addition> :**
    - This Module is supposed to maintain different RL Models which act on the CqGym to provide decisions for the Scheduling of Jobs.

- **<u>Models</u>.PG :**
    - Implementation of PG model. This model is derived from the implementation of DRAS PG. The functions are restructured to connect the RL model with the CqGym environment discussed above.
    - *build_policy()* : Function to construct the RL model developed using Keras. The PG implementation constitutes 2 models - Police and Prediction. Both models are returned from this function.
    - *act()* : Calls prediction of the model for an input vector. No Training.
    - *train()* : To train the model based on the Batch size defined by the user.
    - *save_using_model_name()* : Save weights for the trained model. Expects only the model name. The 2 models are saved using the same name along with additional suffixes for distinguishing between Policy and Prediction.
    - *load_using_model_name()* : Load saved weights for the trained model. Expects only the model name. The 2 models are saved using the same name along with additional suffixes for distinguishing between Policy and Prediction.

- **<u>Trainer</u> <Addition> :**
    - This module acts as a communicator between the Models and CqGym.
    - This modularity can be useful in keeping different components independent of modifications done in them, as long as the communication standards are maintained.

- **<u>Trainer</u>.PG_Trainer :**
    - *model_engine()* : Function to be called for invoking the CqGym and PG model. This function manages the parameters required for initialization the Gym Environment along with CqSim simulator and also for loading RL model - PG. Following parameters are expected :
        - module_list: CQSim Module :- List of attributes for loading CqSim Simulator
        - module_debug: Debug Module :- Module to manage debugging CqSim run.
        - job_cols: [int] :- No. of attributes to define a job.

        - window_size: [int] :- Size of the input window for the DeepLearning (RL) Model.
        - is_training: [boolean] :- If the weights trained need to be saved.
        - weights_file: [str] :- Existing Weights file path.
        - output_file: [str] :- File path if the where the new weights will be saved.

    - *model_training()* : Function creates Tensorflow session and executes model run using CqGym Environment.
        - env.render() is also invoked here which can be kept optional depending on the requirement.

CQSim Modifications:

As discussed above, using the multi-threaded approach, CqSim code is only minorly modified with almost no update done for any of the functionalities. Following are the modification details :

- *Cqsim_sim()* :
    - __init__() <Modification>: Inorder to implement multithreading and synchronization with CqGym, this class inherits 2 major parent classes -
        - Thread - In built python class for implementing multithreading.
        - Pause - Class to implement synchronization using conditional variables.
    - It Also initializes 2 variables -
        - simulator_wait_que_indices - To communicate the wait queue updates with the CqGym.
        - Is_simulation_complete - Lets know CqGym if the simulation is complete.
    - Note : Using other existing variables in CqSim, information is retrieved by CqGym. However CqGym can only makes modifications to the flow of simulation of CqSim using the above mentioned 2 variables.
    - *run() <Addition>* : Overwrites Thread.run(). Invoke cqsim_sim() function.
    - *cqsim_sim() <Modification>* :
        - The only modification made to this function is - inside the while as soon as the wait queue is defined, the current thread is paused and the wait queue is updated by the CqGym.
    - *reorder_queue() <Addition>* :

        - This(and only this) function manages thread synchronization and communication with the GymEnvironment i.e. the CqSim thread pause is only caused using this function. It expects the following argument :
            - wait_que: [List[int]] : CqSim WaitQue at current Time.
            - return: [List[int]] : Updated wait_que, with the selected job at the beginning.
    - *backfill() <Modification>* :
        - reorder_queue function is passed as an argument, to be invoked while selecting back-fill jobs.