ADVANCED MATERIALS INTERFACES

Supporting Information

for *Adv. Mater. Interfaces,* DOI: 10.1002/admi.202101987

Enabling Modular Autonomous Feedback-Loops in Materials Science through Hierarchical Experimental Laboratory Automation and Orchestration

Fuzhan Rahmanian, Jackson Flowers, Dan Guevarra, Matthias Richter, Maximilian Fichtner, Phillip Donnely, John M. Gregoire, and Helge S. Stein**

Enabling modular autonomous feedback-loops in materials science through hierarchical experimental laboratory automation and orchestration

Fuzhan Rahmanian[1,2], Jackson Flowers[1,2], Dan Guevarra[4], Matthias Richter[4], Maximilian Fichtner[1,3], John M. Gregoire[4,*], Helge S. Stein[1,2,*]

[1] Helmholtz Institute Ulm (HIU), Helmholtzstr. 11, 89081 Ulm, Germany
[2] Karlsruhe Institute of Technology (KIT), Institute of Physical Chemistry (IPC),Fritz-Haber-Weg 2, 76131 Karlsruhe, Germany
[3] Karlsruhe Institute of Technology (KIT), Institute of Nanotechnology (INT), P.O. Box 3640, D-76021 Karlsruhe, Germany
[4] Division of Engineering and Applied Science, and Liquid Sunlight Alliance (LiSA), California Institute of Technology (Caltech), Pasadena, California, USA

Address correspondence to: helge.stein@kit.edu, gregoire@caltech.edu

In this section, we are going to demonstrate the steps required for running a process by our proposed HELAO framework. The experiment in question will be maximization of a scaled Schwefel function, virtually sampled by moving a pair of motorized probes in parallel to different (x,y) positions on the function. In the proposed demo, two devices and two auxiliary servers including lang motor, force sensor, analyze and active learning respectively are required to be instantiated.

As described in the main section of the paper, we operate the orchestrator by sending it lists of actions to perform, called “experiments”. Generally, one experiment will comprise all the actions we need to perform to complete a measurement at a single point on a substrate, but this formalism is not strict, and can easily be adapted for experiments which do not use substrates at all. While plans are to build a more user-friendly graphical interface in the near future, we currently construct these experiments directly as python dictionaries. An experiment dictionary has three keys: “soe” (sequence of experiments), which contains a list of all the actions to be performed in the experiment in order, “params”, which contains a dictionary which has the actions to be performed as keys and dictionaries of their parameters as values under those keys, and “meta”, which accepts a dictionary of arbitrary metadata.

Below, the first experiment dictionary in our parallel active learning demonstration process is given as an example of this structure. As can be seen under the “soe” key of this dictionary, each action name has three components: the name of the server performing the action (with an optional colon and integer to distinguish between multiple instances of the same server), these instances are separated by a “/” from the name of the action, which is optionally separated by an “_” from an indexing integer. The latter distinguishes between multiple actions of the same type in a single experiment, or between the actions of multiple experiments in a process.
This particular experiment begins with “orchestrator/start”, which, as described in the paper, primes the orchestrator to begin collecting data. It is then followed by several motor movements such as “lang:2/moveWaste_0”, which would normally be necessary to handle pumped liquids in our electrochemistry experiments, but are merely ornamental in this demo representation. After these, “lang:2/moveAbs_0" moves to an arbitrary sample point (dx0, dy0) taken from the defined search space which is required for initializing the process. “lang:2/moveDown_0” uses feedback from the force sensor to safely move the probe into contact with the substrate, as we would in an actual electrochemical experiment, and “measure:2/schwefelFunction_0” takes the value of the Schwefel function at this point. “analysis/dummy_0” performs a placeholder analysis.
Looking under the “params” key, we have a dictionary of parameters for each of our actions. Most of these are relatively straightforward, but there is some value in explaining the parameter structure for “dummy_0”. Active learning normally requires analyzing our data as it is acquired. Data analysis functions in our active learning routines must be able to operate on data as it is collected. From the structure of our action return statements, we are able to predict under what headings each data value will be stored within our hdf5 files. The parameters of this analysis function are partial addresses to these values within the hdf5 file.

```
add_process(dict(soe=['orchestrator/start','lang:2/moveWaste_0', 'lang:2/RemoveDroplet_0',
'lang:2/moveSample_0','lang:2/moveAbs_0','lang:2/moveDown_0','measure:2/schwefelFunction_0','analysis/dummy
_0'],
            params={'start': {'collectionkey' : 'al_parallel'},
                    'moveWaste_0':{'x': 0, 'y':0, 'z':0},
                    'RemoveDroplet_0': {'x':0, 'y':0, 'z':0},
                    'moveSample_0': {'x':0, 'y':0, 'z':0},
                    'moveAbs_0': {'dx':dx0, 'dy':dy0, 'dz':dz},
                    'moveDown_0': {'dz':0.12, 'steps':4, 'maxForce':1.4, 'threshold': 0.13},
                    'schwefelFunction_0':{'x':dx0,'y':dy0},
                    'dummy_0':{'x_address':'experiment_0:0/schwefelFunction_0/data/parameters/x',
                               'y_address':'experiment_0:0/schwefelFunction_0/data/parameters/y',
                               'schwefel_address':'experiment_0:0/schwefelFunction_0/data/data/key_y'}},
            meta=dict()))
```

We often use a “for” loop to define a large number of experiments and, consequently, send those experiments all at once to the orchestrator, to be performed one at a time. Shown below is one such for loop, generating the list of experiments which follows the initial experiment described just above. The “soe” here is identical to the initial experiment, with an additional two actions, “ml/activeLearningParallel” and “orchestrator/modify”, at the start of each experiment. “activeLearningParallel” takes in data from the dummy analysis of the previous experiment, by the same means that the analysis server itself takes in data from the results of other actions, and then stores the data in an active memory, sends that data to

a model, and suggests a new point (x_2, y_2) at which the Schwefel function should be sampled. The active memory of the “ml” action server is central to how we enable multi-threading, as multiple experiment threads in the orchestrator can each access a single “ml” server, and thus can give data to and take suggestions from the same model.
The “modify” action is coded directly into the orchestrator, and is uniquely able to modify the action parameters of an in-progress experiment from values within the previous experimental results. It takes as inputs a set of addresses within the relevant hdf5 file, and pointers to values within the current “params” dictionary. Any value that will be changed by this function must be initialized to “?”, as can be seen in the parameters for “moveAbs” and “schwefelFunction” below.

```
n = 100
for i in range(n):
    add_process(dict(soe=[f'ml/activeLearningParallel_{i}',f'orchestrator/modify_{i}',
                        f'lang:2/moveWaste_{i+1}', f'lang:2/RemoveDroplet_{i+1}',
                        f'lang:2/moveSample_{i+1}', f'lang:2/moveAbs_{i+1}',
                        f'lang:2/moveDown_{i+1}', f'measure:2/schwefelFunction_{i+1}',
                        f'analysis/dummy_{i+1}'],
                  params={f'activeLearningParallel_{i}':{'name': 'sdc_2', 'num': int(i+1),
                           'query': query, 'address':f'experiment_{i}:0/dummy_{i}/data/data'},
                          f'modify_{i}':{'addresses':
                           [f'experiment_{i+1}:0/activeLearningParallel_{i}/data/data/next_x',
                            f'experiment_{i+1}:0/activeLearningParallel_{i}/data/data/next_y',
                            f'experiment_{i+1}:0/activeLearningParallel_{i}/data/data/next_x',
                            f'experiment_{i+1}:0/activeLearningParallel_{i}/data/data/next_y'],
                           'pointers':[f'schwefelFunction_{i+1}/x',f'schwefelFunction_{i+1}/y',
                          f'moveAbs_{i+1}/dx', f'moveAbs_{i+1}/dy']},
                          f'moveWaste_{i+1}':{'x': 0, 'y':0, 'z':0},
                          f'RemoveDroplet_{i+1}': {'x':0, 'y':0, 'z':0},
                          f'moveSample_{i+1}': {'x':0, 'y':0, 'z':0},
                          f'moveAbs_{i+1}': {'dx':'?', 'dy':'?', 'dz':dz},
                          f'moveDown_{i+1}': {'dz':0.12, 'steps':4, 'maxForce':1.4,'threshold':
                                                                                    0.13},
                          f'schwefelFunction_{i+1}':{'x':'?','y':'?'},
                          f'dummy_{i+1}':
                    {'x_address':f'experiment_{i+1}:0/schwefelFunction_{i+1}/data/parameters/x',
                     'y_address':f'experiment_{i+1}:0/schwefelFunction_{i+1}/data/parameters/y',
                     'schwefel_address':f'experiment_{i+1}:0/schwefelFunction_{i+1}/data/data/key_y'}},
                     meta=dict()))

        add_process(dict(soe=['orchestrator/finish'], params={'finish': None}, meta={}))
```

Finally, as shown in the function “add_process”, we use the python package “requests” to send each experiment to be performed to the orchestrator, through the parameters of the orchestrator’s “addExperiment” function, as defined by fastAPI. The “thread” parameter of this function is also visible. While normally experiments are performed in the order in which they are received, it is possible to run multiple such sequences in parallel by assigning different thread values to different experiments. In our parallel active learning experiment, each instrument was run on a different thread, and they communicated only by contributing data to the same model within the memory of the active learning server.

```
def add_process(sequence,thread=0):
    server = 'orchestrator'
    action = 'addExperiment'
    params = dict(experiment=json.dumps(sequence),thread=thread)
```

```
    requests.post("http://{}:{}/{}/{}".format(
        config['servers']['orchestrator']['host'], 13380, server, action), params=params).json()
```

We have included a demo of the process we have just exemplified above in action.
In this video*, we demonstrate how such a parallel run appears to the user.
Two anaconda terminals can be observed in the left column, which correspond to two devices. Once an action is performed, its fastAPI server sends a message to the terminal hosting that server and the subsequent action will be then executed. Note that the lower terminal is substantially more crowded than the upper terminal, as it is hosting the orchestrator, analysis, and machine learning servers in addition to an instrument. In the middle of the frame, an acquisition function taken from the active learning function at every step is depicted. Based on the maximum of the acquisition function, the next subsequent experiment can be selected. Lastly, in the right column, video feeds of the two SDCs are displayed. The operated SDCs are located in the Glovebox (later for battery application) and in our fume hood (for other inorganic experimentations)