Supplementary Information (SI) for Digital Discovery.

Electronic Supporting Information: PerQueue: Managing Complex and Dynamic Workflows

Benjamin H. Sjølin, William S. Hansen, Armando A. Morin-Martinez, Martin H. Petersen, Laura H. Rieger, Tejs Vegge, Juan M. García-Lastra, and Ivano E. Castelli‡

Department of Energy Storage and Conversion, Technical University of Denmark, Anker Engelunds Vej 301, DK-2800 Kongens Lyngby, Denmark.

‡Corresponding author email: ivca@dtu.dk

1 Submission Example Codes

Below are the code scripts for the workflow presented in Lst. 1 in the main text. The workflow uses a Monte Carlo method to estimate π and uses this value to calculate the proposed diameter of a perfect sphere made of silicon-28 weighing exactly 1 kilogram.

1.1 Estimation of π

This code uses a Monte Carlo method to estimate the value of π. This is done by randomly generating points in the 2D plane such that the x- and y-coordinate is in the range $[0; 1[$. Dividing the fraction of points where $x^2 + y^2 \leq 1$ by the total number of generated points and multiplying by 4 gives the estimated value of π.

This script estimates π using `N_samples` generated points and returns the estimate for each order of magnitude of points.

```python
from typing import Tuple

import numpy as np


def main(N_samples: int = 1, **kwargs) -> Tuple[bool, dict]:
    """Compute an estimate of pi using Monte Carlo sampling

    Parameters
    ----------
    N_samples : int
        The number of samples to generate for the estimate
    """
    # Seed the random number generator
    np.random.seed(1023)

    # Generate N points in the x-y plane
    X = np.random.random_sample((2, N_samples))

    # Calculate the squared distance from origo of each point and determine
    # which points are inside the unit circle
    inside_mask = (X**2).sum(axis=0) <= 1

    # Calculate the cumulative number of points inside the unit circle
    cs = np.cumsum(inside_mask)

    # Calculate the estimate of pi in logaritmic intervals
    estimates = np.array([
        4 * cs[10**i - 1] / 10**i
        for i in range(np.log10(N_samples).astype(int) + 1)
    ])

    return True, {"pi_estimates": estimates, 'N_samples': N_samples}
```

1.2 Plotting of the Absolute Error to the Value of π

This script finds the absolute error between the estimates of π and the established value of π. It then generates a log-log plot of the error versus the number of points. Depending on the value of the parameter `savefig_filename`, the plot is either saved to file or shown to the user.

```python
from typing import List, Tuple, Union

import matplotlib.pyplot as plt
import numpy as np

# Increase resolution of the saved figure
plt.rcParams["savefig.dpi"] = 300


def main(
```

```
    pi_estimates: Union[List[float], np.ndarray] = None, N_samples: int = 1,
    savefig_filename: str = None, **kwargs
) -> Tuple[bool, dict]:
    """Plot the absolute error of the given estimates of pi

    Compute the absolute error between the given values of pi and the value
    given by ‘numpy.pi‘.

    Parameters
    ----------
    pi_estimates : list of float or 1D ‘numpy.ndarray‘
        A list of estimates for the value of the constant pi
    N_samples : int
        The number of samples used to generate the last estimate
    savefig_filename : str, optional
        If this parameter is not ‘None‘, the generated graph is saved to the
        filename and path given. Otherwise, the graph is shown as an
        interactive window.
    """
    if pi_estimates is None or len(pi_estimates) == 0:
        raise ValueError("Empty list of pi estimates given")

    # Compute the x-tick locations
    n = [10**i for i in range(np.log10(N_samples).astype(int) + 1)]

    # Create figure and plot the absolute error as a log-log plot
    fig, ax = plt.subplots()
    ax.loglog(n, np.abs(np.array(pi_estimates) - np.pi))
    ax.set(
        xlabel="Number of generated points",
        ylabel="Absolute error from value of \"numpy.pi\""
    )
    ax.grid()

    # Either save or show the graph
    if savefig_filename is not None:
        fig.savefig(savefig_filename)
    else:
        plt.show()

    return True, None
```

1.3 Calculation of the Diameter of the Silicon Sphere

The script below calculates the diameter of a perfect sphere made of silicon-28 using the estimated values of π. The sphere is assumed to weigh 1 kilogram. It returns both the diameters that use the estimates of π and one that uses the established value of π.

```
from typing import List, Tuple, Union

import numpy as np


def main(
    pi_estimates: Union[List[float], np.ndarray] = None, **kwargs
) -> Tuple[bool, dict]:
    """Compute the diameter of the newest protoype for the kilogram

    Using the estimates for pi that we've computed earlier, we want to compute
    the diameter of the silicon sphere that works as a prototype for the
    kilogram after the 2018 redefinition of the SI-system. Given that we know
    the weight of the sphere, we can compute its diameter from the basic
    definitions and constants of the revised SI-system.

    We shall assume that the silicon in the sphere is only silicon-28, but the
    amount of other impurities in the bulk and on the surface are still
```

```python
    included. These values and the method for computation is taken from
    https://doi.org/10.1016/j.crhy.2018.12.005.

    Parameters
    ----------
    pi_estimates : list of float or 1D `numpy.ndarray`
        A list of estimates for the value of the constant pi
    """
    if pi_estimates is None or len(pi_estimates) == 0:
        raise ValueError("Empty list of pi estimates given")

    # Define phsical constants
    planck = 6.62607015e-34  # Planck's constant [J/Hz]
    Rydberg = 10973731.568157  # Rydberg constant [1/m]
    fine_structure = 7.2973525643e-3  # fine structure constant [-]
    c = 299792458  # speed of light [m/s]
    Ar_Si_28 = 27.9769265  # Relative molecular weight of Si-28 [-]
    Ar_e = 5.489e-4  # Relative molecular weight of an electron [-]
    lattice_parameter = 5.43102051e-10  # Si-28 lattice parameter [m]

    # We know the target mass of the silicon sphere.
    target_mass = 1.0  # Target mass of the sphere [kg]

    # Define the mass deficits of the bulk and surface layer
    m_SL = (7.1 + 12.0 + 58.2) * 1e-9  # Surface layer mass
    m_deficit = (17.1 - 2.3 - 0.5 + 6.0) * 1e-9  # Bulk mass

    # Compute the rest energy of an electron
    m_e = 2*planck*Rydberg / (c*fine_structure**2)

    # Compute the core volume of the silicon sphere
    V = (target_mass - m_SL + m_deficit) * Ar_e * \
        lattice_parameter**3 / (m_e * Ar_Si_28 * 8)  # [m^3]

    # Add the `numpy` value for pi to the end of the array
    pi_x = np.array([*pi_estimates, np.pi])

    # Compute the core diameter of the sphere using the estimates of pi
    # Final sphere diameters [mm]
    sphere_diameters = np.cbrt(6 * V / np.array(pi_x)) * 1e3

    return True, {
        "sphere_diameters": sphere_diameters[:-1],
        "actual_sphere_diameter": sphere_diameters[-1]
    }
```

2 Use Case Workflow Codes

Here, we present the submission scripts used for the four use cases presented in Section 3. The contents of the task scripts are not presented herein, but they are available upon reasonable request. As a consequence of this, the arguments to the tasks herein have generally been replaced by {...} for brevity.

2.1 High-Throughput Screening

The following submission script shows how the high-throughput screening workflow, described in the main paper, is set up in PerQueue terms. It should be noted that due to the size of the screening (more than 6000 instances of the workflow) the submission script takes two inputs, j and k. These limit the index into the search space for decorations, allowing the user to submit a subset of the full workflow, such that other projects can use resources in parallel to this study.

Running the central part of the workflow in three parallel sub-workflows is achieved by wrapping tasks `t2-t6` in a `StaticWidthGroup` with a `width` of 3.

Since the convex hull and the band gap are both post-processing steps in this study they are fast to compute and are run on `local`.

```
from sys import argv

from perqueue import PersistentQueue, StaticWidthGroup, Task, Workflow

j = int(argv[1])
k = int(argv[2])
WIDTH = 3

# Defining the tasks for a specific range of entries.
for i in range(j, k):
    # Define tasks
    t1 = Task("generation.py", {'index': i}, "local:10m")

    t2 = Task("relax.py", None, "40:xeon40:1:50h")
    t3 = Task("convex_hull.py", None, "local:5m")
    t4 = Task("band_gap.py", None, "local:5m")
    t5 = Task("preneb.py", {...}, "112:xeon56:1:50h")
    t6 = Task("neb.py", {...}, "168:xeon56:1:50h")

    # Define the subworkflow as a list of tasks - each depends on the previous
    swf = Workflow([t2, t3, t4, t5, t6])

    # Wrap subworkflow in width group to get parallel workflows
    swg = StaticWidthGroup(swf, width=WIDTH)

    # Define final workflow layer
    wf = Workflow([t1, swg])

    # Submit entire workflow through PerQueue
    with PersistentQueue() as pq:
        pq.submit(wf)
```

2.2 Active Learning for MLIPs

The script presented below is that used for submitting the workflow for training machine-learned interactomic potentials using active learning. As shown, the complex workflow is distilled down into four distinct tasks, and the dynamic nature arises from wrapping these in sub-workflows inside `Static-/DynamicWidthGroups` and a `CyclicalGroup`.

The training, simulation and selection tasks all utilize GPU resources (through the `sm3090` resource), and the labeling runs Density Functional Theory calculations on a 24-core CPU resource.

To control when to break out of the `CyclicalGroup`, the `t_train` task returns the PerQueue constant `CYCLICALGROUP_KEY` with a value of `True` for stopping the loop or `False` for continuing to iterate.

```
from perqueue import CyclicalGroup, DynamicWidthGroup, PersistentQueue,
    StaticWidthGroup, Task, Workflow

# Define tasks
t_train = Task('work_train.py', None, '1:sm3090:30m')
t_sim = Task('work_simulate.py', None, '1:sm3090:30m')
t_select = Task('work_select.py', None, '1:sm3090:30m')
t_label = Task('work_label.py', None, '24:xeon24:10m')

# Define groups for workflow width
swg_train = StaticWidthGroup(t_train, width=2)
dwg_ssl = DynamicWidthGroup([t_sim, t_select, t_label])
dwg_train = DynamicWidthGroup(t_train)

# Wrap width groups in a CyclicalGroup for looping
cg = CyclicalGroup([dwg_ssl, dwg_train], max_tries=10)

# Define and submit workflow
wf = Workflow([swg_train, cg])

# Submit the workflow through PerQueue
with PersistentQueue() as pq:
    pq.submit(wf)
```

2.3 Cluster Expansion

Below, we present the submission script used for the Cluster Expansion workflow explained in the main paper. Not much is different here from the previous use cases, which speaks to the simplicity for setting up workflows in PerQueue. Once the workflow structure is defined, converting it to PerQueue constructs yields rather simple code.

Here, we show the use of the optional `name` parameter to a `Task`, which gives it a different name for visualization and searching.

```
from perqueue import CyclicalGroup, DynamicWidthGroup, PersistentQueue,
    StaticWidthGroup, Task, Workflow

# Define tasks
CE_task_init = Task('CE_model.py', {...}, 'local:10m')
CE_task = Task('CE_model.py', {...}, '24:xeon24:10m', name='train')
relax_task = Task('relaxation.py', {...}, '8:sm3090:3h', name='optimize')
MC_task  = Task('MC.py', {...}, '24:xeon24:10m')
KMC_task = Task('KMC.py', {...}, '24:xeon24:10m')

# Wrap up subworkflow for loop
dwg = DynamicWidthGroup(relax_task)
cg  = CyclicalGroup([dwg, CE_task], max_tries=10)

# Wrap (kinetic) Monte Carlo in width group
swg = StaticWidthGroup([MC_task, KMC_task], width=5)

# Package up workflow
wf = Workflow([CE_task_init, cg, swg])

# Submit the workflow through PerQueue
with PersistentQueue() as pq:
    pq.submit(wf)
```

2.4 Active Learning for Image Segmentation

While the image segmentation workflow should be the hardest to express due to the human-in-the-loop constraint, the submission script is the most simple with only three `Task`s that are connected through a `CyclicalGroup` and wrapped in a workflow. Currently, human-in-the-loop is achieved be purposefully failing the `t3` task after writing output to the human and restarting the `Entry` once data has been written to a pre-specified file for the task to read from. In future versions of PerQueue the intention is for this to be a more graceful maneuver.

The key to starting with the training step and have it be at the end of the `CyclicalGroup` is the ordering of the `Task` objects in line 9.

```
1 from perqueue import CyclicalGroup, PersistentQueue, Task, Workflow
2
3 # Define tasks
4 t1 = Task("train.py", {...}, "1:sm3090:2h")
5 t2 = Task("predict_select.py", None, "1:sm3090:30m")
6 t3 = Task("label.py", None, "1:sm3090:10m")
7
8 # Wrap tasks in CyclicalGroup for looping
9 cg = CyclicalGroup([t2, t3, t1], max_tries=10)
10
11 # Define full workflow
12 wf = Workflow([t1, cg])
13
14 # Submit the workflow through PerQueue
15 with PersistentQueue() as pq:
16     pq.submit(wf)
```