Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Intelligent HTC for Committor Analysis

Miłosz Białczak[a], Alan O'Cais[b], David Swenson[b], Mariusz Uchronski[a], Adam Włodarczyk[a]

[a]Wroclaw Centre of Networking and Supercomputing (WCSS), Wroclaw University of Science and Technology
{milosz.bialczak, mariusz.uchronski, adam.wlodarczyk}@pwr.edu.pl
[b]E-CAM HPC Centre of Excellence
a.ocais@fz-juelich.de, dwhs@hyperblazer.net

Abstract

Committor analysis is a powerful, but computationally expensive, tool to study reaction mechanisms in complex systems. The committor can also be used to generate initial trajectories for transition path sampling, a less-expensive technique to study reaction mechanisms. The main goal of the project was to facilitate an implementation of committor analysis in the software application OpenPathSampling (`http://openpathsampling.org/`) that is performance portable across a range of HPC hardware and hosting sites. We do this by the use of hardware-enabled MD engines in OpenPathSampling coupled with a custom library extension to the data analytics framework Dask (`https://dask.org/`) that allows for the execution of MPI-enabled tasks in a steerable High Throughput Computing workflow. The software developed here is being used to generate initial trajectories to study a conformational change in the main protease of the SARS-CoV-2 virus, which causes COVID-19. This conformational change may regulate the accessibility of the active site of the main protease, and a better understanding of its mechanism could aid drug design.

Project ID: 2010PA5236

1. Introduction

The main goal of this project is to facilitate an implementation of committor analysis in the software application OpenPathSampling (OPS) [1, 2] that is performance portable across a range of HPC hardware and hosting sites.

Committor analysis is essentially an ensemble calculation that maps straightforwardly to a High Throughput Computing (HTC) workflow, where typical individual tasks have moderate scalability and indefinite duration. Since this workflow requires dynamic and resilient scalability within the HTC framework, OPS is coupled to a custom HTC library [3] that leverages the Dask [4, 5] data analytics framework and implements support for the management of MPI-aware tasks.

The committor can also be used to generate initial trajectories for transition path sampling, a less-expensive technique to study reaction mechanisms. The software developed here is being used to generate initial trajectories to study a conformational change in the main protease of the SARS-CoV-2 virus, which causes COVID-19. This conformational change may regulate the accessibility of the active site of the main protease, and a better understanding of its mechanism could aid drug design.

The project targets porting the custom HTC library to a wider variety of HPC platforms (specifically additional resource managers and MPI runtimes) and stress testing the framework for very large task counts. Both OPS and jobqueue_features are Python-based and we have developed more sophisticated support for Python-based MPI-enabled tasks, including direct access to the memory space of executed tasks (which allows us to avoid the use of the filesystem for information transfer between tasks).

We have also investigated the use of the UCX protocol for communication to reduce the overall overhead of the HTC framework.

2. Preparing OpenPathSampling for the HTC framework

Leveraging the Dask framework involved significant software development within OPS. Two main problems had to be addressed: OPS had no mechanism to gather results reported from parallel workers into a single result set, and tasks based on OPS could not be serialised by Dask.

Both of these problems were solved by building atop an ongoing overhaul of the OPS storage and serialisation subsystem. These have been combined with a new implementation of the committor simulation to make a usable parallel committor simulation for OPS.

The overall approach to interfacing the HTC framework with OPS was to allow the HTC framework to effectively act as a drop-in replacement for objects from the Dask scheduler. This required significant effort to prepare OPS to support Dask in general, but then little additional effort to support the HTC framework. In particular, OPS could not trivially support Dask because:

- Some objects in OPS could not be serialised by Cloudpickle (the default serialisation in Dask).
- OPS caches results of some calculations (functions referred to as "collective variables") in memory and also stores them to disk. Maintaining this behaviour in a parallel scheme required new objects to replace the existing collective variables.

A new storage subsystem was already in development for OPS for reasons of performance, extensibility, and maintainability. We completed the core functionality of that storage subsystem and extended it to address issues with Dask compatibility. The incompatibility with Cloudpickle was overcome by using the new storage subsystem's serialization capabilities to serialise the data into a memory-based database, which could then be again serialized by Cloudpickle. Deserialising the memory-based database then becomes part of the task.

Developing new collective variables was a planned part of the new storage subsystem, and we added functionality to support gathering results into a canonical copy of the collective variable as part of the process of storing results from a task to disk.

In addition, we added support for the OPS GROMACS engine which we use as the scalable, hardware-portable MD engine that drives the OPS tasks.

3. Development and optimization of the `jobqueue_features` HTC library

Building upon [3], we expanded the capabilities of `jobqueue_features` [6] to include the following:

- Support for MPI-enabled tasks (i.e., tasks which can leverage MPI4PY for parallelisation). The framework has the ability to access the memory space of the root MPI process of each task.
- Support for a Portable Batch System (PBS) resource scheduler.
- Continuous integration support for SLURM and PBS. This means that the library features are fully tested (automatically) on both of these schedulers through the use of a set of custom Docker containers.
- Support for OpenMPI, Intel MPI and MPICH MPI runtimes (including runtime process distribution).
- A Docker-based tutorial infrastructure which includes a SLURM scheduler. The infrastructure includes two computation nodes and one head node. On the head node a JupyterLab instance is installed, which allows the user to interact with the tutorial infrastructure (and explore our package) directly through their browser. We intend to create some tutorial Jupyter notebooks to facilitate use of this infrastructure.
- Support for Dask and Dask.distributed 2.x (tested up to version 2.21, released in July 2020).
- Configuration of the HTC library for the PRACE resources JUWELS and Irene. Irene uses both a custom scheduler (based on SLURM) and a custom MPI runtime, however we were able to override the more generic classes of `jobqueue_features` to support these (see Appendix B for details).

MPI-enabled task support is now tested via in a continuous integration setup, and this feature was the main test case used for our scalability studies. For each task, the framework *reuses the MPI environment* created when the worker was initialised on the node. We have tested the framework with PyLAMMPS to show that each task does indeed have access to, and uses, all available resources (this information is something that LAMMPS reports in its output). A minimal example of what the implementation of this looks like in code is provided in Listing 1.

```python
lammps_cluster = CustomSLURMCluster(
    # Request MPI mode and define resources
    mpi_mode=True, nodes=2, ntasks_per_node=12,
    # Set execution environment
    env_extra=["module restore lammps_tutorial",],
)

@mpi_task(cluster_id=lammps_cluster.name)
def lammps_task(input_file, run_steps=100):
    from mpi4py import MPI
    from lammps import PyLammps

    L = PyLammps()      # Initialise LAMMPS
    L.file(input_file)  # Read the input file
    L.run(run_steps)    # Simulate the system
    if MPI.COMM_WORLD.Get_rank()==0:
        return "Potential energy: %s" % L.eval("pe")

@on_cluster(cluster=lammps_cluster)
def my_lammps_job(input_file="in.melt", run_steps=100):
    t1 = lammps_task(input_file, run_steps=run_steps)
    print(t1.result())  # blocking call (t1 is a lazy future object)
```

Listing 1: Example of decorator usage to parallelize computation

The main actions for optimisation/improvement were:

- Porting of the HTC framework to Dask 2.x.
- Porting of the framework to Irene and JUWELS.
- Implementing support for the PBS resource manager and additional MPI runtimes.
- Continuous integration development to cover all library features.
- Investigation of UCX instead of IPoIB as a communication protocol.

There was no particular bottleneck to these developments; a lot of the time was spent resolving diverse issues related to the rapid development of the underlying Dask infrastructure. HPC network infrastructure is one key potential weakness since we currently use IPoIB and require a network connection between the scheduler and the workers. Since the interface used by the scheduler and those of the workers are now independently configurable, we have not (as yet) encountered a situation where issues related to this were not surmountable.

Another issue is that the Dask scheduler is intended to be a (relatively) long running task (with low/moderate resource usage). This is because when the Dask scheduler submits jobs to the resource manager, it cannot know when the jobs will execute nor how long the entire workflow will take. The scheduler would therefore normally run on a login node. When using JupyterLab on the Juelich Supercomputing Centre (JSC) systems (here JUWELS in particular) a Dask scheduler has the potential to run for up to 3 days. On Irene, we also did not encounter limitations to running the scheduler on the login nodes, but we are aware this is likely to be the case on some sites. The scheduler can always be run within a resource manager job context.

4. Experiments and results

4.1. Scalability of `jobqueue_features`

In an HTC use case, strong and weak scalability are open to definition. To give an initial definition, one could say that strong scaling would correspond to the ability of the individual tasks to scale to larger resource usage. Such a measure as this is entirely dependent on the scalability of the applications used within the tasks themselves rather than related to the HTC framework. In the case of OPS, it uses an underlying community application engine to achieve scalability (GROMACS for the use case described here, but this is configurable). We have tested the framework with both GROMACS (as used within OPS) and LAMMPS (in particular PyLAMMPS). Neither of these applications are directly under the remit of this project and both applications are known to scale well on HPC resources.

Another argument that could be made is that strong scaling could be defined as the number of simultaneous workers that the framework can utilise. Since each worker is an individual job, the limitation here does not come from the framework, it is the number of simultaneous jobs that can be allocated by the resource manager for a single user (which is usually of $O(100)$ but varies from site to site). Since job submissions are ultimately handled by the library dependency `dask_jobqueue` [7], we must wait for support of job arrays [8] to be able to reliably perform any such analysis (and this was not in the remit of this project).

Our interest in this project lies in the scalability of the HTC framework itself with respect to task counts. We would argue that is somewhat equivalent to a weak scaling analysis when considering an HTC workload: higher task count is equivalent to larger workload and triggers the dynamic provisioning of additional workers by the HTC framework.

`jobqueue_features` had not really been stress-tested for the number of tasks it was capable of supporting. Previous efforts had looked at up to 2000 tasks, in this case we scaled out to 1M tasks on all available architectures, with each individual task using 2 nodes worth of resources. Each set of 2 nodes forms a worker and the workers are reused by queued tasks. The number of workers that the framework can use is entirely dependent on the maximum number of simultaneous jobs allowed by a specific site. The package can simultaneously use workers of different resource types (CPU, KNL, GPU,...).

Tasks	JUWELS	IRENE (KNL)	IRENE (SKL)
10	0.8899	2.7884	0.6899
100	0.0199	0.2896	0.0699
1000	0.0039	0.0347	0.0069
10000	0.0053	0.0548	0.0017
20000	0.0026	0.2076	0.0168
50000	0.0008	0.0181	0.0092
100000	0.0008	0.0133	0.0011
200000	0.0007	0.0175	0.0010
1000000	0.0007	0.0152	0.0010

Table 1: Overhead per task (seconds, excluding node configuration) on different clusters. A maximum of 5 workers were available for each task count.

As we can see from Figure 1 (and Table 1), we were able to easily scale our task workload out to 1M tasks. The overhead of the framework is negligible at 1ms per task for SkyLake (SKL) architecture and 10ms

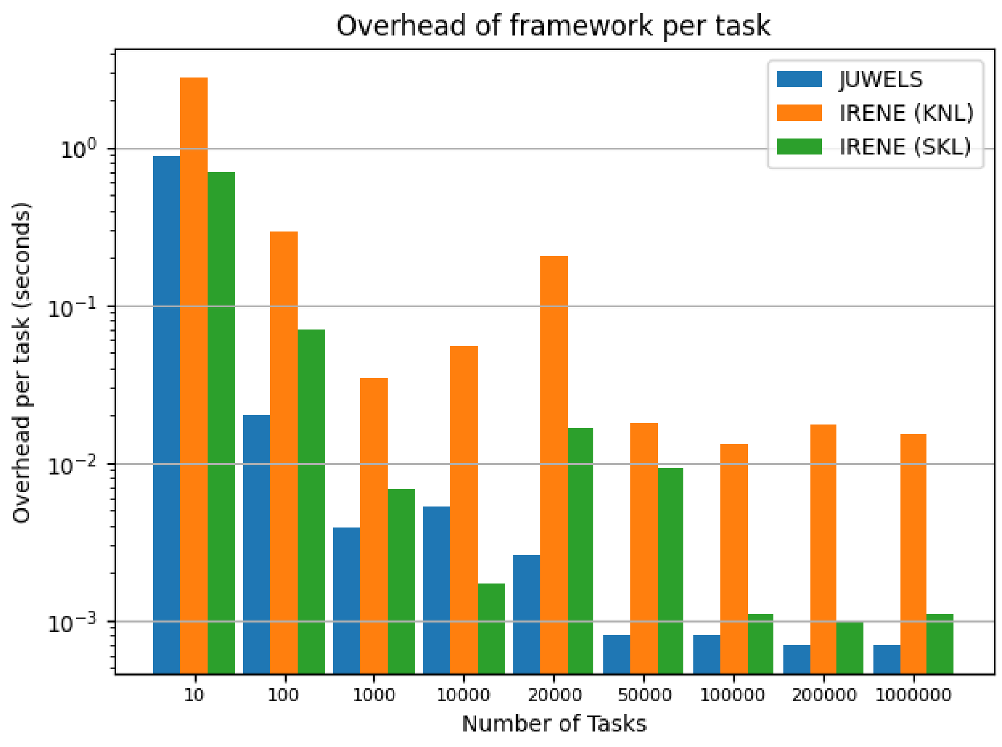

Figure 1: Total overhead of the framework per task in seconds for various numbers of tasks and on various architectures.

	JUWELS	IRENE (KNL)	IRENE (SKL)
Node configuration time (per worker)	10	29	6

Table 2: Node configuration time (seconds, per worker) on different clusters

per task for Knights Landing (KNL) architecture (and is completely independent of the resource usage of the workers themselves). Such overheads are negligible even for short task durations. To put this in context, the node configuration times on JUWELS is about 10s; on Irene it is about 6s for the SKL, and 29s for the KNL (see Table 2). CPU time savings for short task durations are, therefore, potentially substantial when using the framework.

We note that maintaining a functional software environment is non-trivial for the use cases we address here, there are many complications possible due to the software requirements of the framework and the tasks themselves. This is in addition to the complexity due to the potential to simulate simultaneously on a variety of hardware (such as we have done for the CPU, GPU and KNL partitions of JURECA).

4.2. UCX as a communication protocol

To test the efficiency of the UCX protocol in comparison to TCP (using IPoIB in our case), we used several test cases with different kinds of computation, from basic hand written functions to those based on functions provided by Dask.

Size of data in task	TCP(s)	UCX(s)
2000	0.0464	0.0438
5000	0.1598	0.1692
7000	0.2523	0.2600
10000	0.5629	0.5620

Table 3: Data set creation time per task for different protocols in relation to data size

Size of data in task	TCP(s)	UCX(s)
2000	0.1185	0.1016
5000	0.2533	0.2818
7000	0.3535	0.3829
10000	0.7002	0.7239

Table 4: Data set creation computation time per set of tasks for different protocols in relation to data size

Crucial to each test, we need to create data structures for them to use. That is why we test the impact of that creation for both TCP and UCX protocol, shown in Tables [3, 4, 5, 6]. As we can see times of single operations

Number of tasks	TCP(s)	UCX(s)
2	0.1118	0.1240
3	0.1549	0.1623
5	0.2417	0.2624
10	0.5629	0.5620

Table 5: Data set creation time per task for different protocols and numbers of tasks

Number of tasks	TCP(s)	UCX(s)
2	0.1484	0.1732
3	0.2198	0.2277
5	0.3381	0.3721
10	0.7002	0.7239

Table 6: Data set creation computation time per set of tasks for different protocols and numbers of tasks

do not show much differences, which was expected. For whole set of tasks, where most of communication take place, TCP shows better results.

Size of data in task	TCP(s)	UCX(s)
2000	0.3961	0.3898
5000	0.4974	0.4856
7000	0.6344	0.6389
10000	0.7774	0.7292

Table 7: Bag increment computation time per task for different protocols relative data size

Size of data in task	TCP(s)	UCX(s)
2000	1.1214	1.4356
5000	1.2933	1.5669
7000	1.3628	1.6730
10000	1.5068	1.7856

Table 8: Bag increment computation time per set of tasks for different protocols relative data size

Number of tasks	TCP(s)	UCX(s)
3	0.2126	0.2070
5	0.3361	0.3196
7	0.4569	0.4467
10	0.6344	0.6389

Table 9: Bag increment computation time per task for different protocols and number of tasks

Number of tasks	TCP(s)	UCX(s)
3	0.4157	0.4767
5	0.6764	0.8920
7	0.9809	1.1653
10	1.3628	1.6730

Table 10: Bag increment computation time per set of tasks for different protocols and number of tasks

The most significant differences between Dask related operations and hand written tests can be seen on two specific tests: increment mapped on a Dask bag, and regular tree reduction. This can be seen in Tables [7, 8, 9, 10]. As we can see, UCX working only on Dask bag gives slightly better results then TCP. However, when it comes to also working on multiple tasks, TCP lost less time on communication.

Size of data in task	TCP(s)	UCX(s)
2000	0.3918	0.3934
5000	-	0.8234
7000	0.9206	1.2249
10000	1.2106	1.8574

Table 11: Tree reduction computation time per task for different protocols relative to data size

Size of data in task	TCP(s)	UCX(s)
2000	0.8876	0.8904
5000	-	1.9600
7000	1.7382	2.2385
10000	2.1026	1.7856

Table 12: Tree reduction computation time per set of tasks for different protocols relative to data size

Number of tasks	TCP(s)	UCX(s)
3	-	0.2696
5	0.4998	0.5094
7	0.8136	0.7982
10	1.2106	1.2249

Table 13: Tree reduction computation time per task for different protocols and numbers of tasks

Number of tasks	TCP(s)	UCX(s)
3	-	0.6362
5	1.0168	1.0209
7	1.6587	1.4600
10	2.1026	2.2386

Table 14: Tree reduction computation time per set of tasks for different protocols and numbers of tasks

In Tables [11, 12, 13, 14], we can see that for the case of a fully hand written tree reduction, we have better performance for TCP for both single task, and multiple tasks.

On most of these tests, UCX performance is not very different to TCP performance (where we are utilising IPoIB). In general, UCX does give slightly better results for tasks based on Dask functions and TCP performs better for hand written tasks (most likely due to the fact that Dask can perform optimisations when it has control over the data objects). Critically, UCX has been seen to lead to some errors related to the fact that it is not yet a fully supported technology within Dask. In particular, during tests there were problems in the case of restarting the workers while UCX was in use, which means that we potentially lose resiliency in this scenario. Given the very limited potential for performance improvement and the significant impact of sacrificing resiliency, we would not recommend the use of UCX with `jobqueue_features` (or even Dask) at this time.

5. Conclusions

Our HTC library `jobqueue_features` is resilient and scales extremely well. OPS was expanded to be able to use the Dask framework and integration with the `jobqueue_features` library now requires just 3 lines of code (see Appendix A for details). We have shown that support on other PRACE machines is a relatively straightforward process. This means that OPS can now almost trivially transition from use on a personal laptop to some of the largest HPC sites in Europe.

UCX support was found to be somewhat unstable (currently) and beneficial only in specific scenarios that make heavy use of Dask objects. Several times on closing a cluster we received errors about UCX losing the connection, but results were correctly returned. Problems appear when a cluster attempts to restart, then the connection can not be reestablished. For this reason we say that resiliency is currently compromised when using UCX, as such we would not recommend its use until this issue is resolved.

The parallelised committor simulation capability that the integration between OPS and the HTC library provides allows for the possibility to more easily address committor analysis in a scalable way. Ongoing work will use the tools developed here for a committor simulation of the SARS-CoV-2 main protease. Initial analysis of the stable states is based on a long trajectory provided by D.E. Shaw Research [9]. That trajectory indicates that a loop region of the protein may act as a gate to the active site. Initial frames will be taken from that trajectory to be used as configurations in the committor simulation.

For a given molecular system, the committor simulation can be scaled up in two ways: by taking more initial configurations, which ensures a broader exploration of configuration space, and by running more trajectories per configuration, which leads to a more accurate calculation of the committor value. The results of this simulation will provide insight into the dynamics of the loop region, and the mechanism of its gate-like activity. Furthermore, trajectories generated by the committor simulation can be used an initial conditions for further studies using methods such as transition path sampling.

References

1. David W. H. Swenson, Jan-Hendrik Prinz, Frank Noe, John D. Chodera, and Peter G. Bolhuis. OpenPathSampling: A Python framework for path sampling simulations. 1. Basics. *Journal of Chemical Theory and Computation*, 15(2):813–836, 2019. PMID: 30336030.
2. David W. H. Swenson, Jan-Hendrik Prinz, Frank Noe, John D. Chodera, and Peter G. Bolhuis. OpenPathSampling: A Python framework for path sampling simulations. 2. Building and customizing path ensembles and sample schemes. *Journal of Chemical Theory and Computation*, 15(2):837–856, 2019.
3. Alan O'Cais, David Swenson, Mariusz Uchronski, and Adam Wlodarczyk. Task Scheduling Library for Optimising Time-Scale Molecular Dynamics Simulations, August 2019. https://doi.org/10.5281/zenodo.3527643.
4. Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016.
5. Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 130 – 136, 2015.
6. Milosz Bialczak, Alan O'Cais, David Swenson, Mariusz Uchronski, and Adam Wlodarczyk. jobqueue_features GitHub repository. `https://github.com/E-CAM/jobqueue_features`, 2020. [Online; accessed 26-Oct-2020].
7. Joe Hamman, Matthew Rocklin, Jim Edwards, Guillaume Eynard-Bontemps, and Loic Esteve. *Dask-Jobqueue*, 2018. https://blog.dask.org/2018/10/08/Dask-Jobqueue.
8. No specific author. Add support for job arrays #196. `https://github.com/dask/dask-jobqueue/issues/196`, 2018. [Online; accessed 12-March-2020].
9. No specific author. Long trajectory provided by D.E. Shaw Research. `https://www.deshawresearch.com/downloads/download_trajectory_sarscov2.cgi/`, 2020. [Online; accessed 22-Oct-2020].

Acknowledgements

This work was financially supported by the PRACE project (funded in part by the EU's Horizon 2020 Research and Innovation programme (2014-2020) under grant agreement 823767), and also the E-CAM Centre of Excellence (funded in the same programme under grant agreement 676531).

A Integration of OPS and `jobqueue_features`

In Listing 2, we show a final proof of principle for the integration of OPS and `jobqueue_features` that can leveraged for a committor calculation. The 3 lines that integrate the two packages are prepended by comments.[1]

```python
import mdtraj as md
import openpathsampling as paths
from openpathsampling.experimental.storage.dask_integration import DaskDistributedScheduler
from openpathsampling.experimental.storage.parallel_committor import NewCommittor
from openpathsampling.experimental.storage.collective_variables import MDTrajFunctionCV
from openpathsampling.engines.external_engine import RandomString
from openpathsampling.experimental.storage.sql_backend import SQLStorageBackend
from openpathsampling.experimental.storage.ops_storage import OPSStorage
from jobqueue_features.clusters import CustomSLURMCluster
from jobqueue_features.mpi_wrapper import SRUN, mpi_wrap

# Create the Dask cluster
custom_cluster = CustomSLURMCluster(
    scheduler_options={'interface': 'ib1'},
    name="mpiCluster", walltime="00:03:00", nodes=2, mpi_mode=True, fork_mpi=True,
    mpi_launcher=SRUN, queue="devel", scale=5,
    env_extra=[
        "module restore ops_test_07_2020",
    ],
)

# Wrap the calls to GROMACS with our MPI runtime
wrap_gmx = mpi_wrap(
    executable="gmx ",
    mpi_launcher=custom_cluster.mpi_launcher,
    mpi_tasks=custom_cluster.mpi_tasks,
    nodes=custom_cluster.nodes,
    cpus_per_task=custom_cluster.cpus_per_task,
    ntasks_per_node=custom_cluster.ntasks_per_node,
    return_wrapped_command=True,
)

engine = paths.engines.gromacs.Engine(
    gro="conf.gro",
    mdp="md.mdp",
    top="topol.top",
    options={'filename_setter': RandomString(),
             'gmx_executable': wrap_gmx,
             'mdrun_args': '-cpt -1'},
    base_dir='.',
    prefix="ad_committor"
).named("AD Committor Engine")

snap = paths.engines.gromacs.snapshot_from_gro("conf.gro")

cv = MDTrajFunctionCV(md.compute_dihedrals,
                      topology=snap.topology,
                      result_type='float',
                      indices=[[4, 6, 8, 12]]).named("angle")

infinity_vol = paths.CVDefinedVolume(cv, -float("inf"), float("inf")).named("all space")


if __name__ == "__main__":
    backend = SQLStorageBackend("ad_committor.db", mode='w')
    storage = OPSStorage.from_backend(backend)

    # Use the cluster client as the Dask scheduler
    sched = DaskDistributedScheduler(custom_cluster.client)

    committor = NewCommittor( infinity_vol, engine, storage=storage, scheduler=sched)

    committor.run([snap], 2)
    storage.close()
```

Listing 2: Integration of `jobqueue_features` and OPS for committor calculation using GROMACS as the MD engine.

[1]Note that this code extract was created to work with a version of OPS at a particular commit in the repository history (`https://github.com/openpathsampling/openpathsampling/tree/5b48708`) and the API may have changed since then.

B Integration with the PRACE Irene Joliot-Curie system

The PRACE Irene Joliot-Curie system uses a the fully customised workload manager as a wrapper for several workload managers, such as SLURM or Moab. The `jobqueue_features` library is configured for standard SLURM and PBS workload managers, and so some integration steps were required. The syntax of the custom workload manager wrapper was far from common but, for the most part, corresponds with an underlying SLURM syntax. This general correspondence allows for a relatively straightforward customization of the `jobqueue_features` library to allow for interaction with the Irene system.

The required steps to achieve such integration were the identification of the specialized commands and syntax of Irene, and mapping those to the classes already available within the `jobqueue_features`. The customizations of the `jobqueue_features` classes are shown in Listing 3.

```python
class IreneJob(CustomSLURMJob):
    submit_command = "ccc_msub"  # Replaces sbatch
    cancel_command = "ccc_mdel"  # Replaces scancel
    mpi_launcher = "ccc_mprun"   # Replaces srun
    parameters_map = {"-p": "-q", "-A": "-A", "-m": "-m", "-t": "-T"}

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        sbatch_prefix = "#SBATCH"
        msub_prefix = "#MSUB"
        header_lines = self.job_header.split("\n")
        modified_lines = []
        slurm_parameters = []
        for line in header_lines:
            if line.startswith(sbatch_prefix):
                stripped = line.replace(f"{sbatch_prefix} ", "")
                for k, v in self.parameters_map.items():
                    if stripped.startswith(k):
                        modified_lines.append(f"{msub_prefix} {stripped.replace(k, v)}")
                        break
                else:
                    slurm_parameters.append(stripped)
            else:
                modified_lines.append(line)

        idx = modified_lines.index("")
        self.job_header = "\n".join(modified_lines[:idx])
        self.job_header += f"\n{msub_prefix} -E '{' '.join(slurm_parameters)}'\n"
        self.job_header += "\n".join(modified_lines[idx:])

        self._command_template = self._command_template.replace("srun", self.mpi_launcher)


class IreneCluster(CustomSLURMCluster):
    job_cls = IreneJob


knl_kwargs = {"queue": "knl", "cores_per_node": 64, "ntasks_per_node": 64, "memory": "96GB"}

custom_cluster = IreneCluster(
    name="mpiCluster",
    project=PROJECT_NAME, walltime="3600",
    interface="", nodes=2, mpi_mode=True, maximum_jobs=10, mpi_launcher=SRUN,
    python="python3",
    job_extra=["-m scratch"],
    env_extra=[
        "module purge",
        "module load python3/3.7.2",
        "export PYTHONPATH=/path/to/python3.7/site-packages/",
    ],
    **knl_kwargs,
)
```

Listing 3: Customisation of `jobqueue_features` classes to work with PRACE Irene Joliot-Curie cluster workload manager