eCoSimHPC

bwRSE4HPC Project Report

Assigned RSEs:	Glen Hunter[1], Thomas Isensee[2]
Project Partners:	Hüseyin Kemâl Çakmak[3], Alexander Kocher[4], Maurice Müller[5]
Scientific field:	Engineering
Subfield:	Energy Systems
Start:	2025-10-01
Duration:	6 months
Type of Work:	Feature development, Parallelization, HPC Porting
License:	TBD
Link:	Not available
Programming Language:	Python
Technologies:	Dask, MPI
Target Cluster:	bwUniCluster 3.0

1. Executive Summary

Problem Statement: eCoSim is a framework designed for controlling and executing simulators within co-simulations for energy systems. The program has been under development since 2019 and currently support concurrent execution through Python multithreading. Currently, eCoSim makes use of socket-based communication between systems to distribute simulators and computation. This works for small systems, such as a small village model with 30-40 simulators, but simulating the sector coupled energy system of an entire city can require thousands of simulators. To provide sufficient computational resources for such simulations, eCoSim requires the resources of an HPC system.

Solution: We implemented a Dask-based communication protocol as an optional communication backend. Dask is a Python package designed for distributing tasks across multiple processes. By using eCoSim's existing abstractions, the newly implemented backend allows for simulators to be executed on different processors and, with dask-mpi, on different nodes. In addition to the new communication backend, a new user interface (UI) was created to allow for all simulators to be started with a single Python command.

Impact: With the introduction of Dask to eCoSim, the energy system co-simulations conducted by the project partners can be performed across multiple nodes on HPC systems. Additionally, the new UI simplifies launching co-simulations and makes the code easier for new users to use.

[1] SCC, Karlsruhe Institute of Technology, Karlsruhe, Germany; glen.hunter@kit.edu
[2] SSC, Heidelberg University, Heidelberg, Germany; thomas.isensee@iwr.uni-heidelberg.de
[3] IAI, Karlsruhe Institute of Technology, Eggenstein-Leopoldshafen, Germany; hueseyin.cakmak@kit.edu
[4] IAI, Karlsruhe Insitute of Technology, Eggenstein-Leopoldshafen, Germany; alexander.kocher@kit.edu
[5] Bit Raven, Remchingen, Germany; maurice.mueller@bitraven.de

This work is licensed under CC BY 4.0.

2. Description of Work

2.1. Initial Problem

The Energy System Analysis (ESA) group at the Institute for Automation and Applied Informatics (IAI) make use of eCoSim to simulate different kinds of energy systems, such as systems present in buildings and entire energy grids. Initially developed by Dr. Çakmak in 2019, eCoSim establishes a co-simulation framework to simulate energy systems [1]. This allows individual simulators to execute independently, whilst eCoSim coordinates their execution and transfers between them.

eCoSim was originally written in Java and has been ported to Python over the past few years. Like the Java version, the current version of eCoSim can communicate between simulators on one computer using multithreading, or distributed over multiple computers over a network using the standard-library `socket` module. This allows for external users to run simulators on their own systems and only send and receive necessary data to the other simulators without needing to provide the simulators with embedded models to the project partners. This is practical for collaborators who have simulators that are confidential allowing for data privacy to be maintained [2].

When expanding the size of the energy systems simulated, more computational power is inevitably required. Currently, the simulations that are being conducted are small enough that they can be distributed across a few laptops and workstations. However, the project partners plan to conduct city scale simulations requiring thousands of simulators to be controlled and executed in parallel. Such scales require the use of High-Performance Computing (HPC) systems in order to avoid unneeded hardware costs.

Establishing socket-based connections becomes more challenging when computation is distributed across multiple nodes because node IP addresses and ports are not known prior to job submission. A new backend is thus required in order to simplify distributing the simulators across processors and nodes, and establishing the connection between them.

2.2. Solution Design

Parallel tasks in Python can be managed through a futures executor. This allows for tasks to be submitted to a scheduler and distributes the tasks across available processors. The results can then be retrieved from the processors via the scheduler.

Python comes with a futures package installed called `concurrent.futures`. Although the package works well for the distributing tasks within a node, two alternatives were evaluated for multi-node, distributed computing: `dask.distributed` and `mpi4py.Futures`. Both packages follow a very similar syntax to `concurrent.futures` by having a `submit` function to launch tasks. We opted to implement `dask.distributed` into the code base for the following reasons:

- Support for distributed synchronization and communication primitives, including locks, queues and events.
- Access to the Dask dashboard for live interactive profiling.
- The ability to swap the communication backend of Dask easily. A user can make use of the standard `tornado` backend or can swap to `dask-mpi` for use on multiple nodes.

eCoSim makes use of abstraction methods within its code base. Abstraction allows for a parent class to be defined and contains class methods which can be overridden by a class method of the same name in a child class. The abstraction method for the communication exists within `ecosim_common.communication`, which contains both the base `channel` abstraction as well as `socket` for the socket-based communication. Functionality for the new Dask backend was created here, building off of the `channel` abstraction and using the methods present in `socket` as a guide.

eCoSim's client-server communication model has implications on its execution. Currently the server and each client is launched from the command line. As the number of simulators increase, more clients need to be started via command-line. At larger scales, starting each client manually becomes impractical. To circumvent this, a new UI was developed such that both the server and simulators can be launched by single command-line prompt.

2.3. Contributions

2.3.1. Prior work

As mentioned in Section 2.1, eCoSim was developed in 2019 in Java by Dr. Çakmak in order to conduct co-simulations. By making use of the Functional Mock-up Interface (FMI) standard to pre-compile models, the Functional Mock-up Units (FMUs) generated are used as individual simulators for these co-simulations. In 2024, Maurice Müller ported the code into Python, creating the abstraction based structure that eCoSim utilizes now and implementing multithreading to enable concurrency. Matlab support was provided in the same year by Dr. Çakmak and Alexander Kocher [3]. Support for Python simulators via the Pyro5 package was developed by Alexander Kocher in 2025.

2.3.2. Structure

The components of the code follow a client-server styled approach to communication between the simulators and the eCoSim server. This structure is illustrated in Figure 1. This configuration assumes that the server and each simulator are running on separate systems. For smaller co-simulations, this configuration can run on separate threads and communicate via localhost.

The eCoSim server host, executed by `ecosim_server`, triggers the execution of the simulators, controls them and saves all the corresponding data from the simulators at each timestep. `ecosim_server` waits for all simulators to connect and then sends a message to them to start the co-simulation. During each iteration, `ecosim_server` sends the updated timestep along with relevant data from other simulators to each simulator and receives corresponding data back from the simulator. This data is passed to `DBConnection` which stores simulation data in a PostgreSQL database using SQLAlchemy.

Each simulators interfaces with its corresponding client type, for example an FMU V2 simulator interfaces with `ecosim_client_fmu_v2`. The client handles the execution of it's corresponding simulator, passing the data it receives from the server to the simulator and sends back the output for that time step. The clients connect to `ecosim_server` endpoints by sending data through their corresponding channels and transporting the data across the internet via Python's `socket` package.

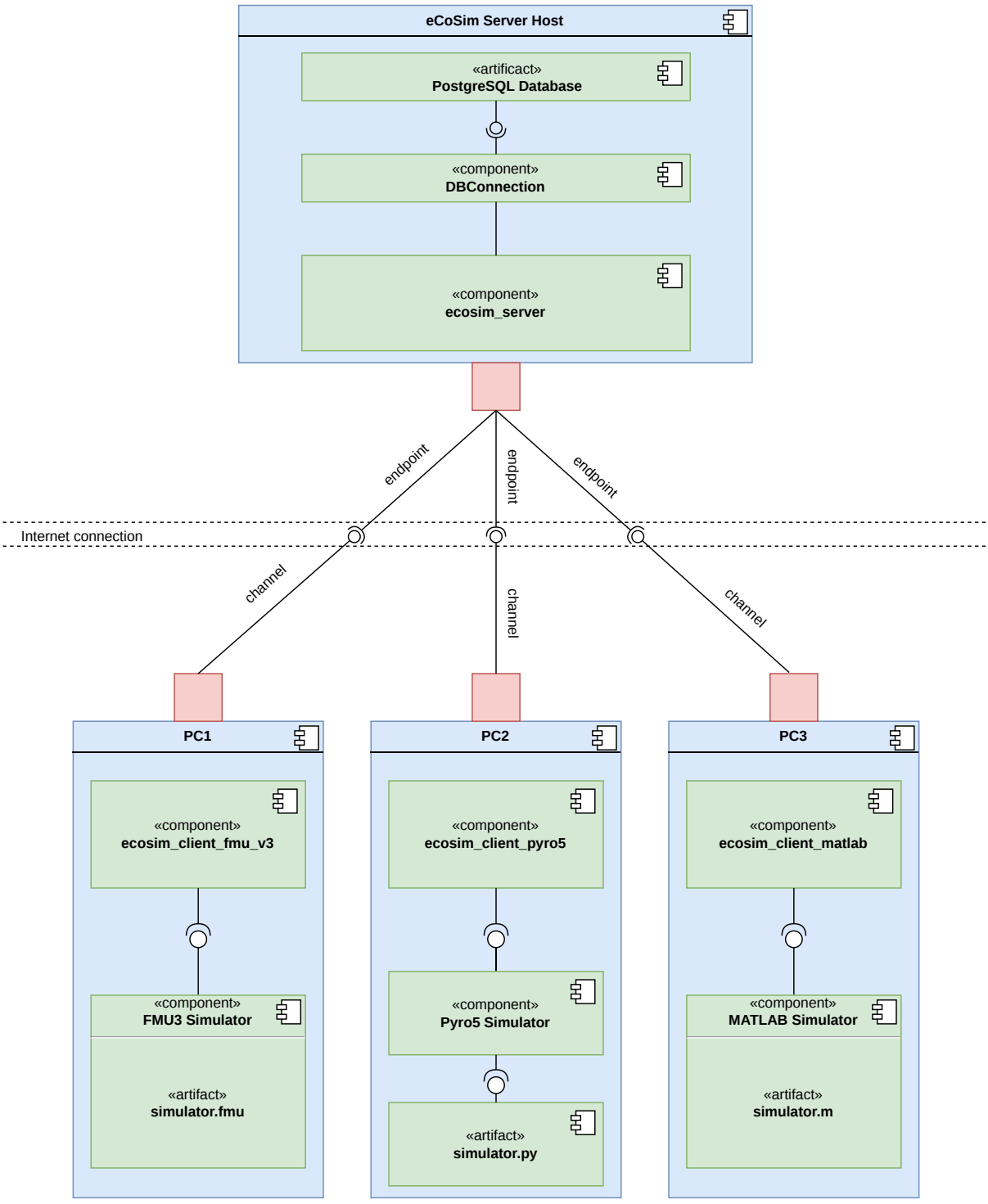


Figure 1: Component diagram of eCoSim detailing how each component communicates within eCoSim. FMU V2 uses the same structure as FMU V3.

We adapt the existing architecture with Dask.distributed in order to enable distribution of the simulators across multiple processors/nodes. This is shown in Figure 2. The main difference is the replacement of the internet connection with Dask.distributed as well as encapsulating both eCoSim server and the clients into a single program, eCoSimHPC. Here, rather than having individual PCs for each client, the simulators are executed on a worker - typically represented by a thread or processor. The details of this implementation are discussed in the next section.

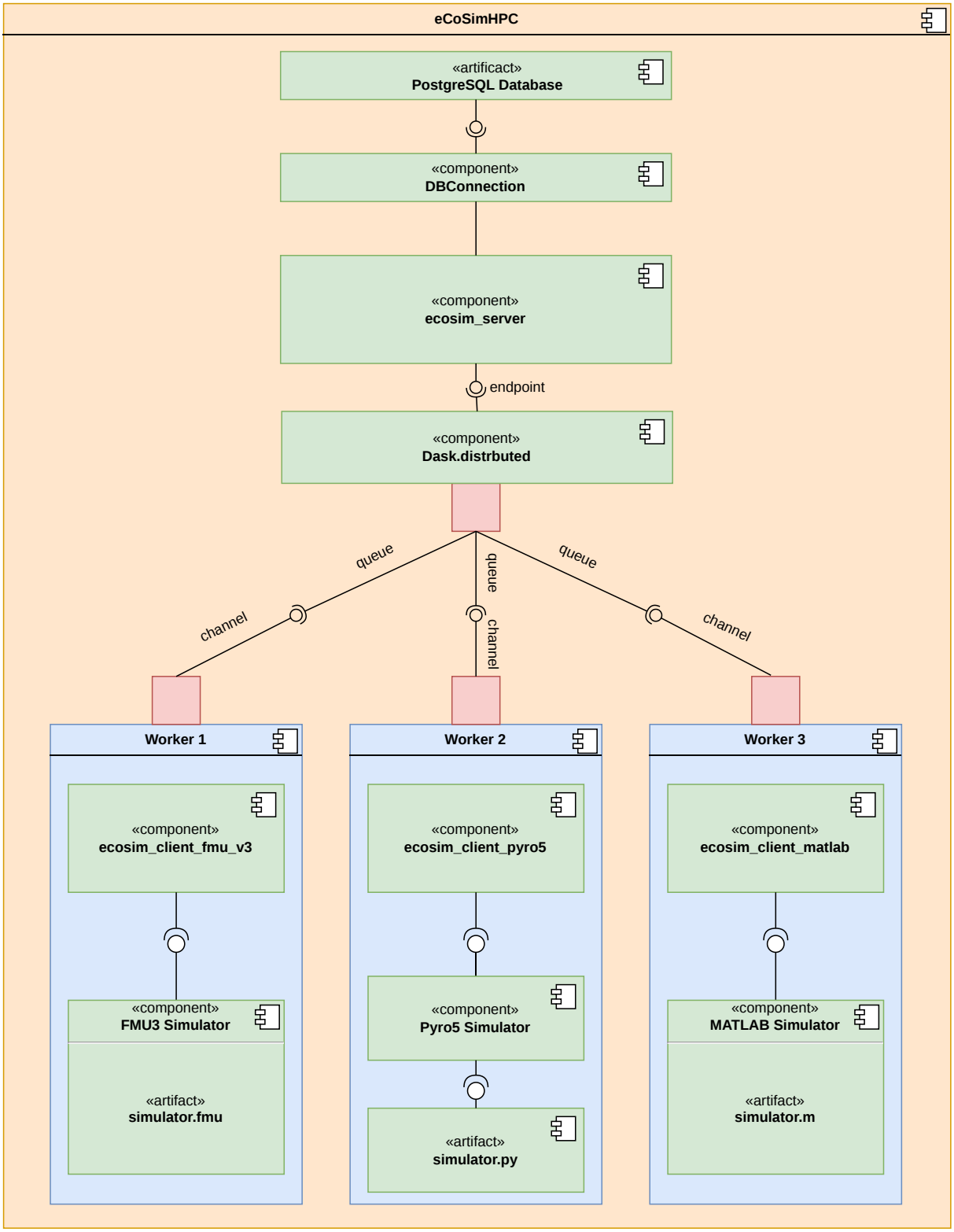


Figure 2: Same as Figure 1 but adapted for eCoSimHPC. Each worker communicates with `dask.distributed` and then subsequently to `ecosim_server`. Workers can be distributed across multiple nodes.

2.3.3. Distributed backend with Dask.distributed

Adapting to abstractions: As mentioned in previous sections, the code base of eCoSim makes use of the abstraction method which allows for alternative functionality based on

existing structure. The abstract base classes Python package (`abc`) provides a helper class, `ABC`, in which a user can create an abstract class by deriving the class from `ABC`. Additionally the package provides a decorator for instance methods that are to be overridden, `@abstractmethod`. Any instance methods with this decorator are defined within the abstract class but do not have any implementation. The implementation is defined within any child classes of the abstract class, tailored for the functionality of a given child class. These modified instance methods can by typed with the `@override` decorator from the `typing_extensions` module. An example of this is illustrated in Code snippet 1 for the `close` method of `DaskChannel` in which `DaskChannel` based on the abstract class `ESPChannel`.

```python
@dataclass
class ESPChannel(ABC):
  @abstractmethod
  def close(self) -> None:
    pass
```

```python
class DaskChannel(ESPChannel):
  @override
  def close(self) -> None:
    self._is_finished.set()
    connection_address = self._internal_dask_client.connection_address()
    self._internal_dask_client.close()
```

Code snippet 1: Sample of how the abstraction is conducted. `ESPChannel` is the parent class with abstract class method `close`. `DaskChannel` is a child class of `ESPChannel` with the `close` method being overridden for functionality within `DaskChannel`.

The bulk of the abstractions used in the process of implementing Dask.distributed for the communication is within `ecosim_common.communication`. Here, there is a dedicated directory/module/submodule called channel containing all the abstract classes for the communication infrastructure. A new sub-package was created in `ecosim_common.communication` called dask that contains all the corresponding child classes from the abstract ones with methods modified for the use of Dask.distributed. A full list of Python files linking the abstracted classes in channel to the derived ones in dask is provided below. Details of individual classes and method can be found in Appendix A.

Table 1: Table detailing which Python files containing the abstracted classes correspond to the files containing the child classes for the use of Dask.distributed.

Functionality	**channel file**	**dask file**
Channel	esp_channel.py	dask_channel.py
Connector	esp_channel_connector.py	dask_connector.py
Endpoint	esp_channel_endpoint.py	endpoint_dask.py
Messages	esp_channel_message.py	dask_messages.py

The Dask client and how it is configured: Dask.distributed provides a cluster and client API for tasks to be submitted and executed in parallel and, optionally, asynchronously. When a cluster is initialized, it establishes a scheduler and set of workers for tasks to be distributed

across. Workers can be threads or processors. The client API handles the submission of tasks to the cluster, not to be confused with the eCoSim clients.

When a client is initialized without an existing cluster, it automatically initializes a `LocalCluster` instance with a number of workers adapted for a given system. This is determined internally by taking the total number of available threads, finding the lowest factor of this number that is also above the square root of this number. This factor is set as the number of workers whilst the corresponding factor pair is set as the number of threads per worker. Optionally, a user can pass parameters for the cluster through the client initialization, such as number of workers. If a cluster is already running, the client connects to it via its IP address and port.

Different types of clusters can be used with Dask.distributed. One such alternative we make use of is `dask-mpi`. This cluster makes use of `mpi4py` and a host system's MPI library (in our case Open MPI 5.0.8) to initialize a Dask cluster and workers, distributed across multiple nodes of a HPC system. This requires a slightly different configuration when initializing. The difference in initializing is shown in Code snippet 2 and Code snippet 3.

```python
import dask.distributed as dd

dask_client = dd.Client(n_workers=16)
```

Code snippet 2: Initialization of a Dask client, and by extension `LocalCluster`, with 16 workers.

```python
import dask.distributed as dd
from dask_mpi import initialize

initialize(interface="ib0",n_threads=2)
dask_client = dd.Client()
```

Code snippet 3: Initialization of a Dask client making use of `dask-mpi` to generate the corresponding Dask cluster.

When initializing the `dask-mpi` cluster, two additional pieces of information are required: internode network connection type (`interface`) and the number of threads per worker (`n_threads`). The connection type can be determined by running `ifconfig` a node of HPC system. This will provide the connection types on the system. Typically these are infiniband (`ib`) or ethernet (`eth`). The shorthand form `ib` and `eth` are followed by numbers and letters and will vary depending on the HPC system, e.g `ib0`, `eth1`, etc. The number of threads per worker in this case has been set to 2 to reflect the number of threads per core of the HPC system used.

Executing a script with `dask-mpi` requires the execution of MPI, for example:
`mpirun --bind-to core --map-by core -np $N_TASKS uv run script.py`
where `$N_TASKS` is the number of processes needed for computation and is typically set the number of allocated cores. For further details with regards to use in the context of eCoSimHPC, see Section 2.3.4.

Asynchronous listening: The listening routines of `dask_channel.py` and `endpoint_dask.py` have been modified from their socket-based counterparts. In the socket version, the listening routines make use of multithreading to run continuously and concurrently with the rest of eCoSim. As such, similar functionality was created with the Dask.distributed framework.

The listening method of `dask_channel.py`, `_listen_blocking`, is primarily used by the clients. It processes any messages received and executes the appropriate function with the simulator according to the message received. Structurally, the code here is similar to its socket counterpart with a main while loop continuously checking for received messages, and exiting when the Dask.distributed event `is_finished` is set. The event is set when shutdown has been called or no message is received.

As for `endpoint_dask.py`, the listening function `_listen` is executed by the server and handles the new connections from clients. The structure of the function differs somewhat between the socket and Dask versions. In both cases the function conducts a while loop that during each iteration checks for a new connection message and passes the connection message to ecosim_server's `handle_client` protocol. The loop only terminates when the `_closed` event is set when the program terminates. The socket based approach makes use of methods provided by the `socket.socket` class to accept a new connection and pass the connection and the address to a new `SocketChannel` for handling messages received from the newly connected client. The Dask based approach does not have access to sockets but instead checks if a new entry is added to the Dask dataset has been added to the Dask client from a connecting eCoSim client. The dataset is a Python dictionary that can be accessed by any Dask client connected to the Dask cluster. When a new connection is established, the client's ID is recorded on the server side at this step for future data transfer by adding the ID as a key to the Dask dataset (see Data transfer with Queues).

For both listening functions of `dask_channel.py` and `endpoint_dask.py`, the functions are submitted to the Dask client asynchronously. When they begin, the tasks are subsequently seceded from the worker pool of the Dask client to free up resources for other tasks.

Data transfer with Queues: Data transfer between server and clients as primarily handled through `esp_dask.py` and the corresponding class `ESPDask`. The class utilizes Dask.distrubted's `Queue` function, a method to send small pieces of data between different processes via the scheduler. A unique queue can be created and accessed by passing an identifier to the `name` parameter of the function.

The following lists the relevant class methods of `ESPDask` and details their functionality:

- `ESPDask.__init__`: Initializes the class. During this step two sets of unique identifiers are created, one hexadecimal ID and one integer ID. These are required for sending and receiving data.
- `ESPDask.on_app_shutdown`: Executes closure of `ESPDask` when the program exits for any reason.
- `ESPDask.connect`: Initializes/connects to the queue by using the hexadecimal unique ID.
- `ESPDask.close`: Closes `ESPDask`, empties the queue and puts one last message to trigger the shutdown of the listeners.

- `ESPDask.send`: Passes a message and relevant IDs to the static method `send_to_dask_client`.
- `ESPDask.receive`: Returns the message obtained from the static method `receive_from_dask_client`.
- `send_to_dask_client`: Submits the message to the queue along with the integer ID, referred to as the origin ID.
- `receive_from_dask_client`: Gets the message from the queue and compares the origin ID sent with the message to the instances origin ID. If the same, the message is put back on the queue as the client would end up receiving its own message that it sent. Otherwise, the message is accepted.

Due to a single queue being accessed by both the server and a given client, it is possible for sent message to be received by the instance it gets sent from. This has resulted in a check of the origin ID to check where the message was sent from.

Resolving serialization issues: When submitting tasks to the Dask client, the tasks and parameters used are serialized, sent to the workers, deserialized and subsequently begin running. Dask.distributed provides a serializer for standard and common Python data structures, including serializers for numpy array. If a data type is not support by the serializer, `pickle` is used instead. This requires the functions and parameters to be pickleable at the very least. If not, the task cannot be distributed to the workers.

During development there have been two main instances where functions cannot be serialized. These are when a function contains any multithreading methods and/or if the function is a class method that accesses other methods of the same class.

Multithreading methods cannot be serialized with Dask.distributed due to two reasons: an executed threaded task is bound to a thread and, as such, the whole running thread would need to be serialized with `pickle`. The multithreading methods present in eCoSim are `threading.Event`, `threading.Lock` and `threading.Thread`. For `threading.Event` and `threading.Lock`, equivalent functions exist in Dask.distributed (`distribtued.Event` & `distributed.Lock`) and are applied across the distributed system. In the case of `threading.Thread`, the equivalent functionality would be using the asynchronous task submission through the Dask client. An example of threading and the Dask client submission equivalent are shown in Code snippet 4 and Code snippet 5 respectively.

For the second serialization issue, the issues of handling class `self` dependent methods required a slightly different approach. The given class method that is to be executed on a worker was developed into static method (a class method without `self` as a parameter) and all the class methods and variables needed for the submitted function are passed as parameters. An example of this would be the listening method of the clients, `_listening_blocking`, which is shown below:

```python
# class method definition
def _listening_blocking(self,on_rec):

#static method definition
```

```
@staticmethod
def _listening_blocking(on_rec, dask_client, event_name):
```

```python
def _handle_new_channel(
    self,
    connection_data: ConnectionData,
) -> None:
    client_thread = threading.Thread(
        target=self._handle_client,
        args=(
            ClientConnection(
                address=connection_data.client_address,
                _channel=connection_data.channel,
            ),
        ),
    )
    client_thread.start()
```

Code snippet 4: Multithreaded version of `_handle_new_channel` within the `Server` class of the eCoSim Server.

```python
def _handle_new_channel(
    self,
    connection_data: ConnectionData,
) -> None:
    dd_client = dd.get_client()

    async def async_handle() -> None:
        future = dd_client.submit(
            self._handle_client,
            ClientConnection(
                address=connection_data.client_address,
                _channel=connection_data.channel,
            ),
            key=uuid.uuid4().hex,
        )
        dd.fire_and_forget(future)
    dd_client.sync(async_handle)
```

Code snippet 5: The Dask.distributed version of `_handle_new_channel` as used by `Dask_Server`.

Further modifications to eCoSim Server: eCoSim Server required additional modifications in order to interface with Dask.distributed. These are detailed as follows:

- A separate server class was created called `Dask_Server` in order to contain the functionality needed for handling new eCoSim clients through the dask client.
- The parameter `server_state` was added to its own queue in order to avoid race conditions. The parameter is made available to all workers via the queue and can be accessed by each eCoSim server connection handler. It contains information such as the state of the clients connected and, importantly, the message buffer. These values are continuously being updated and requires blocking access whilst they are updated. The queue is limited to one value to avoid multiple occurrences of the `server_state` in the queue.
- The simulation executor has many functions and parameters that make use of the server state. Functionality to access the queue for `server_state` as well as a new function called `_update_client_communication` was added. This updates the client states whenever the server state is pulled from the queue.
- The callback handler of the message buffer, `OnMessageCallbackHandler`, makes use of multithreading. An alternative, `OnMessageCallbackHandlerDask`, has been implemented to make use of the the asynchronous task submission through the Dask client. In addition, new callback functions were created in the simulation executor to account for this change.

2.3.4. New user interface

eCoSim has two different user interfaces (UIs) for executing a simulation: one for the server side of eCoSim, `ecosim_server`, and one for the clients, `ecosim_common.client`, with it being an abstraction for each client type is adapted to. The server UI only needs to be executed once for any given simulation, whilst the client UI needs to be executed for each simulator participating in the simulation. This can become cumbersome when there are many simulators. As such, a single executor was created in order to simplify the UI.

In order to handle the distribution and initialization of the clients, a new class called `ClientManager` was created. The method `ClientManager.clients_start` loops through all the directories at a given location and identifies the corresponding file type of a given simulator: `.fmu` for FMU V2 & V3, `.py` for Python simulators via Pyro5, and `.m` for Matlab. From this, the simulator is connected to its corresponding client interface and is submitted through dask client to a free worker.

The new UI is adapted from `ecosim_server/main.py` and is referred to as eCoSimHPC. The new UI takes in the same command line arguments as eCoSim server with two additional arguments:

- `--simulator-dir`: The directory containing all the simulators with each simulator being in a separate subdirectory.
- `--interactive`: a flag to make eCoSimHPC use an established dask cluster. The cluster is normally ran through the command line and requires a scheduler file called `dask-scheduler.json` to be created. See the corresponding documentation to run the cluster in standard mode[6] and with MPI[7].

An example of how eCoSimHPC can be executed with MPI is shown below:

[6] https://docs.dask.org/en/latest/deploying-cli.html
[7] https://mpi.dask.org/en/stable/interactive.html

```bash
mpirun --bind-to core --map-by core -np 64 uv run ecosim_hpc/main.py --server-config ../server/preferences.ini --simulation-config ../server/simulation.json --simulator-dir ../Modules/ --steps 100
```

2.3.5. Adjusting existing CI and tests

The continuous integration (CI) pipeline had to be adjusted to account for the new dependencies as well the new `ecosim-hpc` directory within the repository. Since uv is used as the package manager, all the dependencies of eCoSim are managed via `pyproject.toml` files. A new project file was created for eCoSimHPC whilst others were modified to include the new dependencies introduced by Dask.distributed.

The new infrastructure is mostly test through integration tests due to the connectivity required between server and client. New versions of these have been adapted from existing tests. Notably a new test to test the sending and receiving of a single message, tests to try different configurations of eCoSimHPC (such as multiple clients, different data types, etc.) and a test using an FMU V3 simulator to check the interfacing with the simulators. These new tests required an updated `conftest.py` file in order to enable the Dask cluster to be accessed by the tests.

Additionally, pre-commit configuration was also updated to account for the modifications and new files added to repository.

2.4. Impact

2.4.1. Usage on multiple nodes

The ultimate goal of eCoSimHPC was to enable the use of eCoSim across multiple nodes of a HPC system. With the implementation of Dask.distributed with dask-mpi as a connection interface between eCoSim server and clients, simulators can now be distributed across multiple nodes and can send and receive data through the new Dask-based backend.

2.4.2. Integration test runtimes

The integration tests existing within eCoSim have been adapted for eCoSimHPC. Table 2 shows the runtimes between the standard tests for eCoSim (socket-based) and eCoSimHPC (Dask-based). Here in most cases the tests are faster due to the removal of unneeded sleep statements in eCoSimHPC and the direct passage of data through a Dask queue rather than over a socket. The exception to this speed up is for the codependent tests. In this case, the data types sent during these tests are not standard to the Dask package and additional serialization is likely required for these data types.

Table 2: Runtimes of integrations tests for both the socket-based and Dask-based versions of eCoSim.

Test	Socket duration (s)	Dask duration (s)
Single Client	10.16	5.07
Three Clients	10.05	1.22
Matrix Transfer	10.42	8.0
Codependent	10.5	13.74
Batch steps (range across five tests)	10.03-10.04	1.19-1.57

2.4.3. Scaling

Strong scaling tests were conduct on three different simulation: the Gredler model which consists of 32 interconnected simulators, and two Roegling models - one containing 43 independent simulators and the other containing 83. Here we scale the number of cores N used by $N = 2^n + 1$ where $n = 1, 2, 3...$ One more core is required than for normal strong scaling tests due to one core being dedicated to the scheduler used by the Dask cluster. Each simulation is ran for 50 timesteps. For the large Roegling model, the minimum number of cores needed is 9 and so there are no simulations for 3 and 5 cores.

Figure 3 shows the outcome of the strong scaling test. In all cases, there is speed up by increasing the number of cores from their initial value, however it quickly plateaus. For the small Roegling model, a speed up of factor approximately 7x, the Gredler model has maximum factor 2x, and the larger Roegling model has a maximum factor 1.6x speedup. This indicates that eCoSimHPC does not scale strongly due to the number of processors required for a given number simulators being reached fairly quickly. The are two reasons for the early plateauing. The first is that the resources used become quickly sufficient for the simulations to be executed, causing no benefit to be obtained at higher core counts. The second reason is due to the seceding of the listening routines from worker pool (as mentioned in Section 2.3.3). When a task secedes from the worker pool, the task moves onto an available thread on the system outside of those reserved by the Dask client resulting in the task not being constrained by the resources set by `mpirun`. In this case, only the message processing tasks can be constrained by `mpirun` as they do not secede from the worker pool.

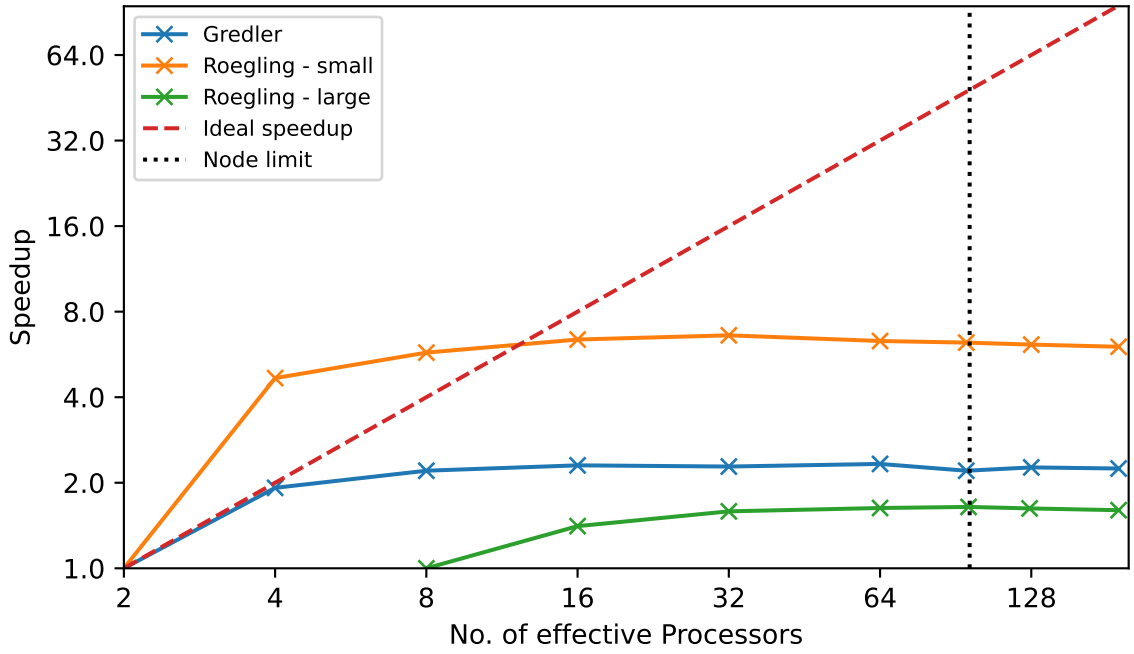


Figure 3: Strong scaling relation for the 3 types of simulations available. Effective cores is $N-1$ where N is the number of cores. This is to account for the single core used for the dask scheduler. The red dashed line represents ideal scaling where doubling the number of cores gives twice the speed-up. The black dotted line is the maximum number of cores of a single node on bwUniCluster 3.0. Left of this line are simulations ran on a single node, right of this line are the simulations ran on two nodes.

Weak scaling could not be measured at this time due to an insufficient number of models. From testing however, it is expected that eCoSimHPC weak scales better than strong scaling. When the project partners have a sufficient number of models, this scaling test can be revisited.

3. Possible Follow-Ups

Since completion of the project, eCoSim and subsequently eCoSimHPC has been refactored allowing for further abstraction of the methods implemented during this project. This allows for ease of maintainability and future development. The further abstraction also gave a factor 1.5x speedup when tested on the smaller Roegling model. Following this, there a couple of potential follow-ups that can be conducted.

One potential follow-up that was discussed early on by the project partners and the RSEs is interfacing eCoSimHPC with the existing graphical user interface (GUI). The existing GUI exists as part of eASiMOV, a closed source interface to eCoSim that allows for visualization of the components of given co-simulation as well as live values of parameters during the co-simulation. For this interface to work, further development is required to establish an interface between the GUI and the Dask-based backend created in this project. Additionally, a connection between the HPC systems running eCoSimHPC and the local system running the GUI would need to be created.

One of the long-term goals for eCoSim is to publish the code as a Python package, with each client type, eCoSim server, eCosim communication, and eCoSimHPC being their own modules. One of the potential follow-ups that can be offered is further consultation into standards needed for publishing eCoSim to PyPI and/or publishing a software paper to a journal, such as the Journal of Open Source Software (JOSS).

Lastly, the Python simulators currently interact with eCoSim via Pyro5. In the context of eCoSimHPC, this provides an extra communication layer, creating additional overhead. This can be circumvented by creating a direct Python client that can be submitted to the dask cluster. Therefore a potential further follow-up is to develop this direct client type to remove this overhead.

A List of abstracted classes in `ecosim_common.communication`

The table below lists the abstracted classes and overridden functions that were used from `ecosim_common.communication.channel` to create the new communication framework with Dask.distributed.

Base abstract class	Child class	Overrides
ESPChannel	DaskChannel	close, listen_async, wait_for_finish, listen_blocking, send
ESPChannelConnector	DaskConnector	connect
ESPChannelEndpoint	EndpointDask	listen_async, close

B Note on using Pyro5 simulators on HPC systems

When using eCoSimHPC on multiple nodes, there is a chance the task corresponding to the Pyro5 client for Python simulators may execute on a different node to the Pyro5 connection itself. To avoid this, the client manager explicitly runs Pyro5 clients on the same node as the Dask scheduler. When executing the Pyro5 connector, the IP address needs to correspond to the same one as the Dask scheduler and needs to be reflected in the `preferences.ini` file for the simulator. This is achieved by adding the following before initializing the daemon thread:

```python
import socket
import configparser
import os
# Other imports

# Pyro5 code

hostip = socket.gethostbyname(socket.gethostname())  # Get IP address
config = configparser.ConfigParser()  # Setup config parser
```

```
file_path = os.path.dirname(os.path.abspath(__file__)) + "/preferences.ini"
config.read(file_path)  # Open preferences.ini
config['pyro']['pyro_host'] = hostip  # Overwrite pyro_host with host IP

with open(file_path,'w') as configfile:
    config.write(configfile)  # Save config

daemon = Pyro5.server.Daemon(host=hostip, port=9090)
...  # Rest of the code
```

Bibliography

[1] H. Çakmak, A. Erdmann, M. Kyesswa, U. Kühnapfel, and V. Hagenmeyer, "Analysis of multimodal energy systems," *at - Automatisierungstechnik*, vol. 67, no. 11, pp. 972–983, 2019, doi: doi:10.1515/auto-2019-0081.

[2] A. Kocher *et al.*, "Privacy Protection in Collaborative Geographically Distributed Co-Simulation of Multimodal Energy Systems," in *Proceedings of the 15th ACM International Conference on Future and Sustainable Energy Systems*, in e-Energy '24. Singapore, Singapore: Association for Computing Machinery, 2024, pp. 639–647. doi: 10.1145/3632775.3661990.

[3] X. Dai *et al.*, "Ensuring data privacy in AC Optimal Power Flow with a distributed co-simulation framework," *Electric Power Systems Research*, vol. 235, p. 110710, 2024, doi: https://doi.org/10.1016/j.epsr.2024.110710.