Colmena

ExaLearn and Parsl Teams

May 17, 2024

CONTENTS:

Colmena is a Python library for building applications that combine AI and simulation workflows on HPC. Its core feature is a communication library that simplifies tools for intelligently steering large ensemble simulations.

Colmena open-source and available on GitHub: https://github.com/exalearn/colmena

CHAPTER

ONE

WHAT DOES COLMENA DO?

The core concept of Colmena is a "thinking" application. The Thinking application is responsible for intelligently responding to new data, such as by updating a machine learning model or selecting a new simulation with Bayesian optimization.

Colmena provides a few main components to enable building thinking applications:

1. An extensible base class for building thinking applications with a dataflow-like programming model
2. A "Task Server" that provides a simplified interface to HPC-ready workflow systems
3. A high-performance queuing system communicating to tasks servers from thinking applications

The demo applications illustrate how to implement different thinking applications that solve optimization problems.

1.1 Installation

Colmena is available via PyPi and you can install it via Pip:

```
pip install colmena
```

Some of the features require Redis to facilitate communication between different parts of an application. Wwe recommend Anaconda or your system's package manager to install it.

1.1.1 Installation on Windows

We recommend installing the Windows Subsystem for Linux in order to use Colmena from a Windows system.

1.1.2 Development Environment

The Anaconda environment in the `dev` folder provides an environment capable of running the examples.

1.2 Quickstart

A short introduction to building a Colmena app.

For this exercise, our goal is to find a point that minimizes $f(x) = x^2$ using a simple search algorithm: pick a new point within ± 0.5 from minimum so far.

This tutorial will explain our multi-agent-thinker.py demo application.

1.2.1 0. Write functions

Translating our target function, *f(x)*, and search algorithm into Python yields:

```
def target_function(x: float) -> float:
    return x ** 2


def task_generator(best_to_date: float) -> float:
    from random import random
    return best_to_date + random() - 0.5
```

1.2.2 1. Define Communication

Colmena applications are split into a "thinker" application generates tasks that are executed on remote resources. The easiest way to connect them is using Python's native interprocess communication via our *PipeQueues*:

```
queues = PipeQueues(keep_inputs=True, topics=['generate', 'simulate'])
```

This command creates separates queue for simulation and task generation results, and ensures that the task inputs will get sent back to the client with the result.

Using ProxyStore

Colmena can use ProxyStore to efficiently transfer large objects, typically on the order of 100KB or larger, between the thinker and workers directly. Enable ProxyStore by initializing a Store then passing the name (`proxystore_name`) and threshold size (`proxystore_threshold`) for the store to `make_queue_pairs`. Any input/output object of a target function larger than `proxystore_threshold` will be automatically passed via ProxyStore.

For example, a common use case is to initialize ProxyStore to use a Redis server to communicate data directly to workers

```
from proxystore.connectors.redis import RedisConnector
from proxystore.store import Store
from proxystore.store import register_store
from colmena.queue import PipeQueues

store = Store('redis', RedisConnector('localhost', 6379))
register_store(store)

queue = PipeQueues(
```

(continues on next page)

(continued from previous page)

```
    proxystore_name='redis',
    proxystore_threshold=100000
)
```

Any object larger than 100kB will get sent via Redis, reducing the communication costs of your application.

Learn more about ProxyStore and find the "Getting Started" guides at docs.proxystore.dev.

1.2.3 2. Build a task server

The "task server" in Colmena distributes request to run functions across distributed resources. We create one by defining a list of functions and the resources to run them across.

Colmena uses Parsl to manage executing tasks. Parsl can scale to 1000s of parallel workers and run on HPC or cloud, but we will define it to only use up to 4 processes on a single machine:

```
config = Config(executors=[HighThroughputExecutor(max_workers=4)])
```

Build a task server by providing a list of methods and resources:

```
doer = ParslTaskServer([target_function, task_generator], queues, config)
```

1.2.4 3. Write the thinker

Colmena provides a "BaseThinker" class to create steering applications. These applications run multiple operations (called agents) that send tasks and receive results from the task server.

Our example thinker has two agents that each are class methods marked with the `@agent` decorator:

```
class Thinker(BaseThinker):

    def __init__(self, queue):
        super().__init__(queue)
        self.remaining_guesses = 10
        self.parallel_guesses = 4
        self.best_guess = 10
        self.best_result = inf

    @agent
    def consumer(self):
        for _ in range(self.remaining_guesses):
            # Update the current guess with the
            result = self.queues.get_result(topic='simulate')
            if result.value < self.best_result:
                self.best_result = result.value
                self.best_guess = result.args[0]

    @agent
    def producer(self):
        while not self.done.is_set():
            # Make a new guess
            self.queues.send_inputs(self.best_guess, method='task_generator', topic=
```

(continues on next page)

(continued from previous page)

```
↪'generate')

            # Get the result, push new task to queue
            result = self.queues.get_result(topic='generate')
            self.logger.info(f'Created a new guess: {result.value:.2f}')
            self.queues.send_inputs(result.value, method='target_function', topic=
↪'simulate')
```

"Producer" creates new tasks by calling the "task_generator" method (defined with the task server) and then using that new task as input to the "target_function."

"Consumer" retrieves completed tasks and determines whether to update the best result so far.

A few things to note:

1. Tasks are run as threads and share class attributes (e.g., `self.best_guess`)
2. The queue takes arguments, method name and topic name as inputs to send a task
3. The `self.done` attribute tracks if any thread has completed.
4. The thinker class provides a logger: `self.logger`

1.2.5 4. Launching the application

The task server and thinker objects are run asynchronously. Accordingly, we call their `.start()` methods to launch them.

```
try:
    # Launch the servers
    doer.start()
    thinker.start()
    logging.info('Launched the servers')

    # Wait for the task generator to complete
    thinker.join()
    logging.info('Task generator has completed')
finally:
    queues.send_kill_signal()

# Wait for the task server to complete
doer.join()
```

1.2.6 5. Running the application

Launch the Colmena application by running it with Python: `python multi-agent-thinker.py`

The application will produce log messages from many components, including:

1. Log items from the thinker that mark the agent which wrote them:

   ```
   ... - thinker.producer - INFO - Created a new guess:  9.51
   ```

2. Messages from the Colmena queue or task server

```
... - colmena.queue.base - INFO - Client received a task_generator result
with topic generate`
```

3. Parsl workflow engine status messages

```
... - parsl.dataflow.dflow - INFO - Task 45 completed
```

1.2.7 6. Learning more

We recommend reading more from our how-to guide next. With that knowledge in hand, try improving the optimization algorithm from this example.

1.3 Design

Colmena is a library for building applications that steer ensembles of simulations running on distributed computing resources.

1.3.1 Key Concepts

Applications based on Colmena have two parts: a "Thinker" and "Doer". The Thinker determines which computations to perform and delegates them to the Doer.

"Thinker": Planning Agent

The "Thinker" defines the strategy for a computational campaign. The strategy is expressed by a series of "agents" that identify which computations to run and adapt to their results. As demonstrated in our optimization examples, complex strategies are simple if broken into many agents.

"Doer": Task Server

The "Doer" server accepts tasks specification from the Thinker, deploys tasks on remote services and sends results back to the Thinker. Doers are interfaces to workflow engines, such as Parsl or Globus Compute.

1.3.2 Implementation

The Thinker and Doer from Colmena run as separate Python processes that interact over queues.

Client

The "Thinker" process is a Python program that runs a separate thread for each agent.

Agents are functions that define which computations to run by sending *task requests* to a task server or reading *results* from a queue. Results are returned in the order they are completed.

A simple "run a large batch in parallel" can be defined with a single agent:

```
class Thinker(BaseThinker)

    # ...

    @agent
    def run_batch(self):
        # Submit computations
        for x in self.to_run:
            self.queues.send_inputs(x, method='f')

        # Collect results
        results = [self.queues.get_result() for _ in range(len(self.to_run))]

        # Find best
        best_ind = np.argmin([r.value for r in results])
        print(f'Best result: {results[best_in].args}')
```

We provide a Python API for the message format, *Result*, which provides utility operations for tasks that include accessing the positional or keyword arguments for a task and serializing the inputs and results.

Task Server

We support Task Servers that use different workflow engines, but all follow the same pattern. Each are defined by registering computations (often expressed as Python functions) to be run, a set of available computational resources, and a queue to communicate with the client.

The best Task Server to start with is Parsl, *ParslTaskServer*. Having it run tasks locally can be achieved by

```
# Function
def f(x):
    return x ** 2 - 3

# Compute configuration
from parsl.configs.htex_local import config

# Communicator
queues = PipeQueues()

# Doer
doer = ParslTaskServer([f], queues, config)
```

Communication

Task requests and results are communicated between Thinker and Doer via queues. Thinkers submit a task request to one queue and receive results in a second as soon it completes. Users can also denote tasks with a "topic" to separate tasks used by different agents.

The easiest-to-configure queue, *PipeQueues*, is based on Python's multiprocessing Pipes. Creating it requires no other services or configuration beyond the topics:

```
queues = PipeQueues(topics=['steer', 'simulate'])
queues.send_inputs(1, method='expensive_func', topic='simulation')
result = queue.get_result(topic='simulation')
```

Task inputs are serialized using Pickle (we support most Python objects this way), and task information is communicated over queues as JSON-serialized objects.

Other implementations of the queue, such as a Redis-backed version (`RedisQueue`) are available.

1.3.3 Life-Cycle of a Task

We describe the life-cycle of a task to illustrate how all of the components of Colmena work together by illustrating a typical *Result* object.

```
{
    "inputs": [[1, 1], {"operator": "add"}],
    "serialization_method": "pickle",
    "method": "reduce",
    "value": 2,
    "success": true,
    "timestamps": {
        "created": 1593498015.132477,
        "input_received": 1593498015.13357,
        "compute_started": 1593498018.856764,
        "result_sent": 1593498018.858268,
        "result_received": 1593498018.860002
    },
    "time": {
        "running": 1.8e-05,
        "serialize_inputs": 4.07e-05,
        "deserialize_inputs": 4.28-05,
        "serialize_results": 3.32e-05,
        "deserialize_results": 3.30e-05
    }
}
```

Launching Tasks: A client creates a task request at `timestamp.created` and adds the the input specification (`method` and `inputs`) to an "outbound" Redis queue. The task request is formatted in the JSON format defined above with only the `method`, `inputs` and `timestamp.created` fields populated. The task inputs are then serialized (`time.serialize_inputs` records the execution time) and passed via the queue to the Task Server.

Task Routing: The task server reads the task request from the outbound queue at `timestamp.input_received` and submits the task to the distributed workflow engine. The method definitions in the task server denote on which resources they can run, and Parsl chooses when and to which resource to submit tasks.

Computation: A Parsl worker starts a task at `timestamp.compute_started`. The task inputs are deserialized

(`time.deserialize_inputs`), the requested work is executed (`time.running`), and the results serialized (`time.serialize_results`).

Result Communication: The task server adds the result to the task specification (`value`) and sends it back to the client in an "inbound" queue at (`timestamp.result_sent`).

Result Retrieval: The client retrieves the message from the inbound queue. The result is deserialized (`time_deserialize_result`) and returned back to the client at `timestamp.result_received`.

The overall efficiency of the task system can be approximated by comparing the `time.running`, which denotes the actual time spent executing the task on the workers, to the difference between the `timestamp.created` and `timestamp.result_returned` (i.e., the round-trip time).

The overhead specific to Colmena (i.e., and not Parsl) can be measured by assessing the communication time for each step. For example, the inbound queue can be assessed by comparing the `timestamp.created` and `timestamp.input_received`. The communication times for Parsl can be measured through the differences between `timestamp.inputs_received` and `timestamp.compute_started`, provided the task does not wait for a worker to become available. The communication times related to serialization are also stored (e.g., `time.serialize_result`).

1.4 Building a Colmena Application

Creating a new application with Colmena involves defining a "task server" that deploys expensive functions and a "thinker" application that decides which tasks to submit.

Note: See Design for details on Colmena architecture.

1.4.1 Configuring a Task Server

The task server for Colmena is configured with the list of methods, a list available computational resources and a mapping of which methods can use each resource.

We describe the *`ParslTaskServer`* in this document, although more are available.

Defining Methods

Methods in Colmena are defined as Python functions. Any Python function can be served by Colmena, but there are several limitations in practice:

1. *Functions must be serializable.* We recommend defining functions in Python in the script that creates the task server or in modules that are accessible from the Python Path (e.g., part of packages installed with `pip`)
2. *Inputs must be serializable.* Parsl makes a best effort to serialize function inputs with JSON, Pickle and other serialization libraries but some object types (e.g., thread locks) cannot be serialized.
3. *Functions must be pure.* Colmena is designed with the assumption that the order in which you execute tasks does not change the outcomes.

See more details about the task types, especially how to include non-Python, in the Method model documentation.

Specifying Computational Resources

Colmena uses Parsl's resource configuration to define available resources for Colmena methods. We use an complex example that specifies running a mix of single-node and multi-node tasks on Theta to illustrate:

```
from parsl.addresses import address_by_hostname
from parsl.config import Config
from parsl.executors import HighThroughputExecutor, ThreadPoolExecutor
from parsl.launchers import AprunLauncher, SimpleLauncher
from parsl.providers import LocalProvider


example_config = Config(
    executors=[
        ThreadPoolExecutor(
            label="multi_node",
            max_threads=8
        ),
        HighThroughputExecutor(
            address=address_by_hostname(),
            label="single_node",
            max_workers=2,
            provider=LocalProvider(
                nodes_per_block=2,
                init_blocks=1,
                max_blocks=1,
                launcher=AprunLauncher('-d 64 --cc depth'),  # Places worker on compute␣
↪node
                worker_init='''
module load miniconda-3
export PATH=~/software/psi4/bin:$PATH
conda activate /lus/theta-fs0/projects/CSC249ADCD08/colmena/env
''',
            ),
        )
    ],
    strategy=None,
)
```

The overall configuration is broken into two types of "executors:"

multi_node
: The `multi_node` executor provides resources for applications that use multiple nodes. We use the `ThreadPoolExecutor` to run the pre- and post-processing Python code on the same Python process as the task server, which can save significant computational resources. The maximum number of tasks being run on this resource is defined by `max_workers`. Colmena users are responsible for providing the appropriate `mpirun` invocation in methods deployed on this resource and for controlling the number of nodes used for each task.

single_node
: The `single_node` executor handles tasks that do not require inter-node communication. Parsl places workers on two nodes (see the `nodes_per_block` setting) with the `aprun` launcher, as required by Theta. Each node spawns 2 workers and can perform two tasks concurrently.

Note that we use `LocalProvider` classes to define how Parsl accesses resources. The `LocalProvider` class assumes that resources are already accessible to the application in contrast to providers like `CobaltProvider` that request resources on behalf of the application (e.g., from an HPC job scheduler).

Mapping Methods to Resources

The constructor of `ParslTaskServer` takes a list of Python function objects as an input. Internally, the task server converts these to Parsl "apps" by calling `python_app()` function from Parsl. You can pass the keyword arguments for this function along with each function to map functions to specific resources.

For example, the following code will place requests for the "launch_mpi_application" method to the "multi_node" resource and the ML task to the "single_node" resource:

```
server = ParslTaskServer([
    (launch_mpi_application, {'executor': 'multi_node'}),
    (generate_designs_with_ml, {'executor': 'single_node'})
])
```

1.4.2 Creating a "Thinker" Application

Colmena is designed to support many different algorithms for creating tasks and responding to results. Such "thinking" applications take the form of threads that send and receive results to/from the task server(s) using queues. Colmena provides as *BaseThinker* class to simplify creating multi-threaded applications.

Working with `BaseThinker`

Creating a new `BaseThinker` subclass involves defining different "agents" that interact with each other and the task server. The class itself provides a template for defining information shared between agents and a mechanism for launching them as separate threads.

A minimal Thinker is as follows:

```
class Thinker(BaseThinker):

    @agent
    def operation(self):
        self.queues.send_inputs(4, method='simulate')
        result = self.queues.get_result()
        self.output = result.value

thinker = Thinker(queues)
thinker.run()
print(f'Simulation result {result.value}')
```

The example shows us a few key concepts:

1. You communicate with the task server using `self.queues`
2. Operations within the a Thinker are marked with the `@agent` decorator.
3. Calling `thinker.run()` launches all agent threads within that class and runs until all complete.

Submitting Tasks

`ColmenaQueues` provides communication to the task server and is available as the `self.queues` class attribute.

Submit requests to the task server with the `send_inputs` function. Besides the input arguments and method name, the function also accepts a "topic" for the method queue used when filtering the output results.

```
client_queue.send_inputs(
    1,
    input_kwargs={'operation': "+"},
    method='f',
    topic='simulation',
    task_info={'key': 'value},
    resources={'node_count': 2}
)
```

The `get_result` function retrieves the next result from the task server as a *`Result`* object. The `Result` object contains the output task and the performance information (e.g., how long communication to the client required). `get_result` accepts a "topic" to only pull tasks sent with a certain topic to the queue.

See the Queue documentation for the available queues.

Inter-agent Communication

Agents in a thinking application are run as separate Python threads. Accordingly, you can share objects between agents. We recommend versing yourself in Python's rich library of threading objects and queue objects to communicate information between agents.

Example Applications

We will describe a few example explanations to illustrate how to make a Thinker applications that implement degrees of overlap between performing simulations and selecting the next simulation.

For all of these cases, we provide a simple demonstration application in the demo applications.

Batch Optimizer

Source code: batch.py

A batch optimization process repeats two steps sequentially: select a batch of simulations and then perform every simulation in the batch. Batch optimization, while simple to implement, can lead to poor utilization if there is a large variation between task completion times (see discussion by Wozniak et al.).

The core logic for each loop can be expressed using a single thread communicating with a single task queue:

```
while not stop_condition:
    # Use the current state of the optimizer to choose new tasks
    tasks = generate_tasks(database, batch_size)

    # Send out tasks on the input queue
    for task in tasks:
        queues.send_inputs(task, method="simulate")
```

(continues on next page)

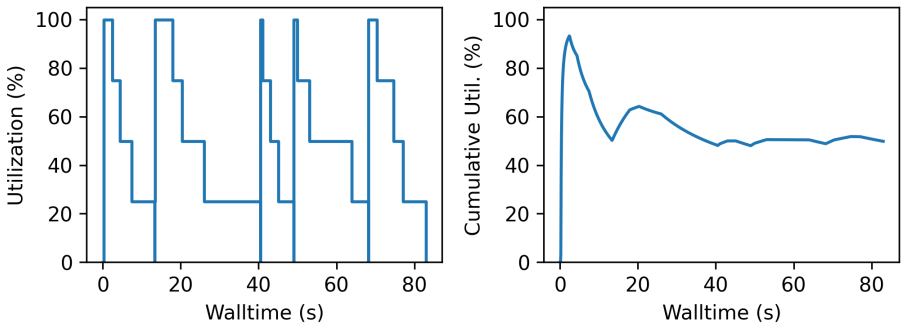

Fig. 1: Resources remain unused while waiting for all members of a batch to complete.

(continued from previous page)

```
    # Collect the tasks, and update the database
    for _ in range(batch_size):
        result = queues.get_result()

        # Save the inputs (args) and output (value)
        database.append((results.args, results.value))
```

Streaming Optimizer

Source code: streaming.py

A streaming or "on-line" optimizer selects a new task immediately after any task completes. The streaming optimizer is particularly beneficial when the time to select a new task is much shorter than the rate at which new tasks complete. As evidenced by codes such as Rocketsled, streaming optimizers are an excellent choice for lengthy tasks run with modest batch sizes. However, the utilization of a computational resource can break down when the rate of task completion becomes comparable to the rate at which new tasks can be generated.

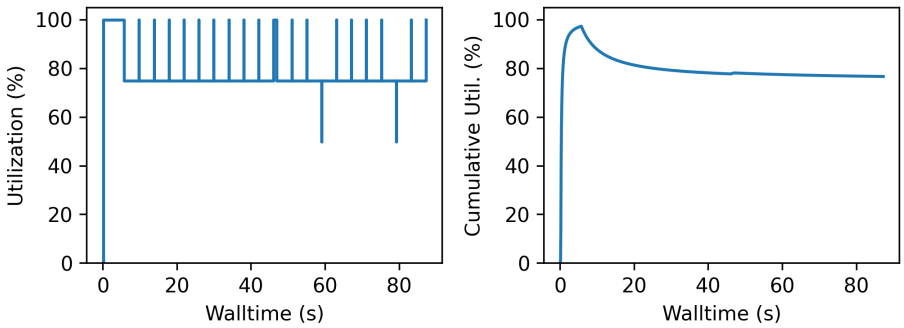

Fig. 2: Utilization limited by task generation rate

A streaming optimizer can also be realized by a single Thinker process and a single task queue.

```
# Create as many parallel tasks as worker slots
tasks = generate_tasks(database, batch_size)
for task in tasks:
```

(continues on next page)

(continued from previous page)

```
    queues.send_inputs(task, method="simulate")

# As new tasks complete immediately generate a single new task
while not stop_condition:
    # Wait until a task completes, pull it from queue
    result = queues.get_result()

    # Add it to the database
    database.append((results.args, results.value))

    # Generate a new task, using the latest results
    task = generate_tasks(database, 1)[0]

    # Sent new task to the queue
    queues.send_inputs(task, method="simulate")
```

Interleaved Optimizer

Source code: interleaved.py

An "interleaved" optimizer continually updates a queue of next simulations while new simulations are running. A new task is started from a task queue as soon as a simulation task completes. The task queue is maintained by a separate thread that continually updates the task generator and re-prioritizes the task queue. Full system utilization can be achieved as long as the task queue is sufficiently long. The challenge instead is to minimize the time between new data received and the task queue being updated with this new data.

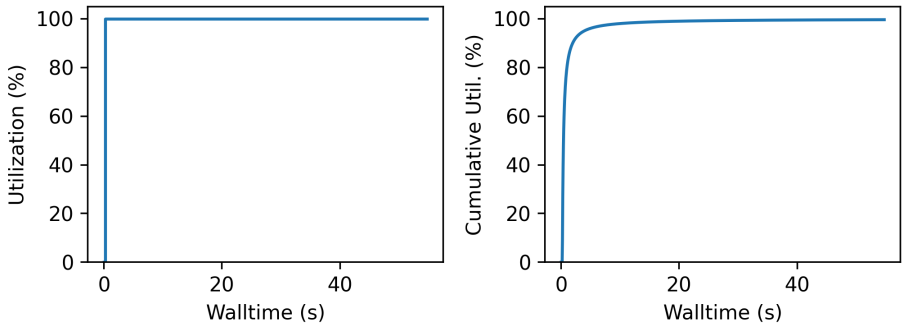

Fig. 3: Caching a prioritized list of tasks prevents under-utilization

Creating an interleaved optimizer in Colmena can be achieved best using two separate threads that each use their own task queues.

The first thread is a simulation dispatcher. It shares a task list, result database, and a Lock with the other thread. We use an Event, `done`, to signal both threads that the optimization loop has completed. We denote tasks associated the simulation dispatcher with the topic "doer."

```
# Send out the initial tasks
for _ in range(batch_size):
    queues.send_inputs(task_queue.pop(), method='simulate', topic='doer')
```

(continues on next page)

(continued from previous page)

```
# Pull and re-submit
while not done.is_set():
    # Get a result
    result = queues.get_result(topic='doer')

    # Immediately send out a new task
    with queue_lock:
        queues.send_inputs(task_queue.pop(), method='simulate', topic='doer')

    # Add the old task to the database
    database.append((result.args, result.value))
```

The second thread is a task generator and prioritizer. Its tasks are labeled with the "thinker" topic.

```
# Create some tasks
tasks = generate_tasks(database, queue_length)

while not done.is_set():
    # Send out an update task, which generates
    #  a new priority order for the tasks
    with queue_lock:
        queues.send_inputs(database, tasks,
                           method='reprioritize_queue',
                           topic='thinker')

    # Wait until it is complete
    result = queues.get_result(topic='thinker')
    new_order = result.value

    # Update the queue (requires locking)
    with queue_lock:
        # Copy out the old values
        current_queue = task_queue.copy()
        task_queue.clear()

        # Note how many of the tasks have been started
        num_started = len(new_order) - len(current_queue)

        # Compute the new position of tasks
        #  Noting that the first items in the queue are gone
        new_order -= num_started

        # Re-submit tasks to the queue
        for i in new_order:
            if i < 0:  # Task has already been sent out
                continue
            task_queue.append(current_queue[i])
```

1.4.3 Creating a `main.py`

The script used to launch a Colmena application must create the queues and launch the task server and thinking application.

A common pattern is as follows:

```
from colmena.task_server.parsl import ParslTaskServer
from colmena.queue import PipeQueues

if __name__ == "__main__":
    # [ ... Create the Parsl configuration, list of functions, ... ]

    # Generate the queue pairs
    queues = PipeQueues(keep_inputs=True, serialization_method='json')

    # Instantiate the task server and thinker
    task_server = ParslTaskServer(functions, queues, config)
    thinker = Thinker(queues)

    try:
        # Launch the servers
        doer.start()
        thinker.start()

        # Wait for the thinking application to complete
        thinker.join()
    finally:
        # Send a shutdown signal to the task server
        ueues.send_kill_signal()

    # Wait for the task server to complete
    doer.join()
```

The above script can be run as any other python code (e.g., `python run.py`)

We have described configuration options for task server and thinker applications earlier. The key options to discuss here are those of the communication queues.

The *`PipeQueues()`* object manages communication between Thinker and Task Server. It takes a few options in addition to the topics of tasks, such as

- `serialization_method`: Whether to use JSON or Pickle to serialize inputs and outputs. Either may produce smaller objects or provide faster conversion depending on your data types.
- `keep_inputs`: Whether to retain inputs in the *`Result`* object after task has completed. Removing inputs could speed communication but may complicate steering logic.

1.5 Advanced Features for Thinkers

The core of a Colmena application is a "thinker" process that controls the use of computational resources by submitting new tasks in response to new data being acquired. This portion of the guide delves into the advanced features for building a "Thinker" application and builds the steps described in the previous page of the guide.

1.5.1 Objects Shared by Default

Several objects are available to all "agent" threads available in a Thinker application.

Queues

The `self.queue` attribute of a Thinker class manages communication to the task server. Each agent can use it to submit tasks or wait for results from the task server.

The *ColmenaQueues* object must be provided to the constructor.

Logger

The `self.logger` attribute is a logger unique to each thread. Log messages written with this attribute will be marked with the name of the agent.

Completion Flag

Upon completion, an agent sets the `self.done` flag. All threads may view the status of this event.

Thread-Local Details

Each agent has a `self.local_details` object to store information which should not be altered by other threads. It is derived from Python's `local` object.

Resource Counter

The `self.rec` attribute is used to communicate the availability of compute resources between threads. Threads may release resources to make them available for use by other agents, request resources, or transfer resources between different available pools.

The core actions for the resource counter include reserving nodes for a particular task typ (`.acquire`), releasing them for use by other agents (`.release`), and reallocating between different resource pools (`.reallocate`). All operations are thread-safe.

A *ResourceCounter* that is configured with the proper number of slots and task pools must be provided to the constructor for this feature to be available.

```
from colmena.queue import ColmenaQueues
from colmena.thinker import BaseThinker, ResourceCounter, agent


class ResourceLimited(BaseThinker):
    def __init__(self, queues: ColmenaQueues, nodes: int = 1):
```

(continues on next page)

(continued from previous page)

```
        """
        Args:
            queues: Queues to use to communicate with the task server
            nodes: Number of nodes to available
        """

        super().__init__(queues, resource_counter=ResourceCounter(nodes, task_types=["a",
↪ "b"]))

        # Start with all nodes allocated to "a"
        self.rec.reallocate(None, "a", nodes)

    @agent()
    def give_away(self):
        for i in range(self.rec.allocated_slots("a")):
            self.rec.reallocate("a", "b", 1)

            self.logger.info("Gave 1 node from a to b")
        self.logger.info("Done giving nodes away")

    @agent()
    def receive(self):
        while not self.done.is_set():
            self.rec.acquire("a", 1, cancel_if=self.done)
            self.logger.info("Reserved a node for task type b")
```

See the documentation for *ResourceCounter*.

1.5.2 Configuring General Agents

Agent threads in Colmena take a few different configuration options. For example, the `startup` keyword argument means that the `self.done` event will not be set when this agent completes.

See *agent()* for more details.

1.5.3 Setup and Teardown Logic

Some agents require expensive operations that only need run once per application or ensure that resources are cleaned up after completion. For example, some may connect to a database to store results persistently between runs of an application.

Override the *prepare_agent()* and *tear_down_agent()* to define these methods, and remember to use *thread-local storage* as this function is run by every agent.

1.5.4 Special-Purpose Agents

There are a few common patterns of agents within Colmena, such as agents that wait for results to become available in a queue. We provide decorators that simplify creating agents for such tasks.

The reallocation example application demonstrates all three of these agent types.

Result Processing Agents

The *colmena.thinker.result_processor()* is for agents that respond to results becoming available. It takes a single argument that defines which topic queue to be associated with and must decorate a function that takes Result object as an input.

```
class Thinker(BaseThinker):
    @result_processor(topic='simulation')
    def process(self, result: Result):
        self.database.append(result)
```

The above example runs the `process` function whenever a complete task with a "simulation" topic is received.

Task Submission Agents

Task submission agents execute a function as soon as resources are available. The agent runs a decorated function once resources are acquired from a certain resource pool. Task submission agents are often paired with a *result processor* that receives the result and marks resources as available once a task completes.

```
class Thinker(BaseThinker):
    @task_submitter(task_type="sim", n_slots=1)
    def submit(self):
        task = self.queue.pop(0)
        self.queues.send_inputs(task, method='simulate', topic='simulation')
```

The above function submits a task from the front of a task queue once 1 slot is available from the "sim" resource pool.

Event Responder Agents

The *colmena.thinker.event_responder()* runs a function when a certain event is triggered. The event responder agents can be configured to request resources in a background thread that are then deallocated after the function completes.

```
class Thinker(BaseThinker):
    @event_responder(event_name='retrain_now', reallocate_resources=True,
                     gather_from="sim", gather_to="ml", disperse_to="sim", max_slots=1)
    def reorder(self):
        # Submit a task to re-order task queue given
        self.rec.allocate('ml', 1)  # Blocks until resources are free
        self.queues.send_inputs(self.database, self.queue, method='reorder', topic='plan
↪')

        # Wait for task to complete
        result = self.queues.get_result(topic='plan')
        self.rec.release('ml', 1)  # Mark that resources are unneeded
```

(continues on next page)

(continued from previous page)

```
        # Store the new task queue
        self.queue = result.value
```

The above example performs a task to reorder the task queue when the `retrain_now` event is set. Colmena will automatically re-allocate resources from simulation to machine learning when the event is set and then re-allocate them back to simulation after the function completes. The Thinker class will also reset the flag once all functions triggered by the event complete.

1.6 Types of Colmena Methods

Colmena encapsulates the methods being executed with a wrapper that manages deserializing task data and tracking runtime information.

The task servers automatically determine the correct wrapper for Python tasks, though you will need to instantiate your own wrapper *for non-Python tasks*.

While you thus do not need to know about the wrappers in most cases, we describe what each does and how they can be adjusted here.

Note: Apply the `PythonGeneratorMethod` wrapper before supplying to the Task Server for wrapped generator functions. Automatic detection of *Python generator function* usually fails for wrapped functions.

1.6.1 Basic Python Functions

The "basic" Python function has a single return value.

```
def f(x: int) -> int:
    return x * 2
```

The wrapper for this type of function, *PythonMethod* needs only a reference to the function and a name to know the function by, if different than the function's existing name.

```
from colmena.models.tasks import PythonMethod

wrapper = PythonMethod(
    function=f,
    name='new_name'
)
```

1.6.2 Generator Python Functions

Python generator functions produce a continual series of outputs then, in some cases, a value on completion.

```
def g(x: int) -> Iterable[int]:
    yield from range(x)
    return "done"
```

Like the single-return wrapper, Colmena wraps these functions using a reference to the function and a name. There is an additional option about whether to return the "return" value in addition to the "yield" values. Colmena returns only the yielded values by default.

```
from colmena.models.methods import PythonGeneratorMethod

wrapper = PythonGeneratorMethod(
    function=f,
    store_return_value=True
)
```

Setting `store_return_values` to `True` will return a tuple of results, such as `([1], "done")` for `x=1`, and return only the first element of the tuple if `False`.

Streaming Results

Configure a generator tasks to stream results as soon as they are created by supplying a *ColmenaQueues* when defining the method.

```
queues = RedisQueues()
wrapper = PythonGeneratorMethod(
    function=f,
    store_return_value=True
    streaming_queue=queues
)
```

The Thinker will receive the yielded results over the task queue provied to the function. Each of the yielded result will have the `completed` field of the `Results` set to `False`, wheras the returned value will have a value of `True`.

Note: We recommend using *RedisQueues* with Redis configured to accept connections from other nodes if workers are run on a different node than the Thinker.

1.6.3 Running Executables

All tasks in Colmena require a Python interface to be executed in the workflow and the *ExecutableMethod* as a guiderail for including computations that are performed outside of Python.

The definition of an `ExectuableMethod` is split into three parts:

1. `__init__`: create the shell command needed to launch your code and pass it to the initializer of the base class.
2. `preprocess`: use method arguments to create the input files, command line arguments, or stdin needed to execute the simulation code with the desired settings
3. `postprocess`: extract the desired outputs for the function from any files or the standard out produced when executing the code.

The example code below runs the `simulator` software, which reads inputs from CLI arguments and from a `options.json` file then stores the result in stdout.

```
class Simulation(ExecutableMethod):

    def __init__(self):
        super().__init__(executable=['/path/to/my/simulator'], name='simulator')

    def preprocess(self, run_dir, args, kwargs):
        with open(run_dir / 'option.json', 'w') as fp:
            json.dump(kwargs, fp)  # Write any kwargs to disk
        return [str(args[0])], None  # Uses the args as CLI arguments

    def postprocess(self, run_dir: Path):
        # The stdout of the code is routed to `colmena.stdout`
        with open(run_dir / 'colmena.stdout') as fp:
            return float(fp.read().strip())
```

Some Task Server implements execute the pre- and post-processing step on separate resources from the executable task to make more efficient use of the compute resources.

See the MPI example.

MPI Applications

Message-Passing Interface (MPI) codes are the standard type of application that utilize multiple nodes of a supercomputer for the same task. In addition to defining the path to the executable and processing operations, MPI codes also require a definition of how to launch the executable across many compute nodes.

For most cases, provide these option in the `__init__` method of your executable and set the `mpi` option to `True`.

```
class Simulation(ExecutableMethod):

    def __init__(self):
        super().__init__(
            executable=['/path/to/my/simulator'],
            name='simulator',
            mpi=True,  # Designate this as an MPI application
            mpi_command_string='mpirun -np {total_ranks}',  # Optionally provide the MPI␣
↪invocation template
        )
```

Some workflow tools, like RCT, can supply the `mpi_command_string` information automatically.

Specify the number of nodes and ranks per node for each tasks using the `resources` keyword argument during task submission.

```
client_queue.send_inputs(1,  method='simulator', resources={'node_count': 2})
```

1.7 Task Servers Available for Colmena

Colmena provides multiple "task servers" for executing computations. Here, we detail the available task servers, describe when they are best used, and provide the basics of configuring them.

1.7.1 Parsl

`ParslTaskServer` is the reference implementation for a Colmena task server and is suitable for most use cases. Parsl is a distributed workflow engine written in Python that we chose because tasks are described in Python, workflows can include thousands of concurrent tasks, and Parsl can be used on many different supercomputing systems.

Configuring Parsl

Tasks in Parsl are defined using Python functions and are mapped to specific "executors" that control the resources on which they are run. See our how-to documentation for a thorough walkthrough on how to define tasks. The "executors" describe how many resources to use for each task, how resources are acquired (e.g., how to inteface with the job scheduler), and how each worker communicates with the task server (e.g., address and ports). The Parsl documentation explains how to configure executors.

1.7.2 Globus Compute

The `GlobusComputeTaskServer` uses Globus Compute to run functions on remote computational resources in a way that requires less network configuration than with Parsl. Globus Compute operates by using a cloud-hosted service to facilitate sending function requests to and receiving results from remote "endpoints" that performs the computation. In contrast to our Parsl task server, you need not have direct network access (e.g., via SSH) to that system or set up SSH tunnels to communicate tasks to or from remote compute nodes. The ease of multi-site configuration for Globus Compute comes at the cost of higher communication latencies and limits on the size of inputs or results that are sent over the network.

Configuring Globus Compute

Like Parsl, the task server is defined using a list of methods mapped to the resources on which they are executed. Unlike Parsl, the execution resources are defined using the ID of a Globus Compute endpoint rather than a name of a specific executor. Any configuration for how that endpoint actually provides compute resources (e.g., launching Kubernetes pods, requesting HPC jobs) is provided when setting up the endpoint (see Globus Compute docs).

1.7.3 Python's Native executor

The `LocalTaskServer` is backed by Python's native `Executor` classes. It is useful for developing new Colmena workflows because it runs with minimal configuraiton. `LocalTaskServer` will automatically run workers on as many threads as your computer has processors, though you can configure it to use separate processes and change the number of workers.

1.8 Queues Available for Colmena

Colmena supports multiple backends for queues that pass data between Thinker and Task Server (see Design)

1.8.1 Python Pipes

`PipeQueues` uses Python Pipes transmit data between two Python processes.

Advantages:

- Simple setup (no configuration or other services required)
- Very portable. Uses only Python native libraries

Disadvantages:

- Small message sizes (<32 MiB)
- Thinker and Task Server must be on same system
- Only one Thinker and Task server are allowed

1.8.2 Redis

`RedisQueues` uses Redis, a high-performance in-memory data store.

Advantages:

- Support moderate message sizes (<512 MiB)
- Thinker and Task Server can run on different systems
- Applications can use multiple Thinkers and Task Servers
- Redis server can also serve as a backend for ProxyStore

Disadvantages:

- Redis must run as a second service
- Redis is difficult to install on some OSs or architectures
- Open ports or SSH tunnels may be required if Redis on separate host from Task Server/Thinker

1.9 Examples

Presentations about Colmena, examples of using Colmena in the scientific literature, and in other open-source codes.

1.9.1 Tutorials

Presentations focused on how to use Colmena

- Exascale Computing Project's 2023 Annual Meeting [YouTube]

1.9.2 Publications

Improvements to Colmena

Papers about improving the performance or utility of Colmena

- Ward et al. "Colmena: Scalable Machine-Learning-Based Steering of Ensemble Simulations for High Performance Computing". *2021 IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments (MLHPC)* [paper] [ArXiv] [slides] [YouTube]
- Ward et al. "Cloud Services Enable Efficient AI-Guided Simulation Workflows across Heterogeneous Resources". *Heterogenous Computing Workshop at IPDPS 2023* [ArXiv] [slides] [YouTube]

Applications of Colmena

Papers where Colmena was used

- Guo et al. "Composition-transferable machine learning potential for LiCl-KCl molten salts validated by high-energy x-ray diffraction" *Physical Review B* [paper] [ChemRxiv]
- Zvyagin et al. "GenSLMs: Genome-scale language models reveal SARS-CoV-2 evolutionary dynamics." [bioRxiv]
- Dharuman et al. "Protein Generation via Genome-scale Language Models with Bio-physical Scoring" [SC-W '23]

1.9.3 Open-Source Software

Other codes which use Colmena

- ExaMol: Performs molecule design on HPC by combining AI and quantum chemistry (same devs as Colmena)

1.10 colmena

1.10.1 colmena.task_server

Implementations of the task server

colmena.task_server.parsl

Parsl task server and related utilities

class colmena.task_server.parsl.**ParslTaskServer**(*methods: List[Callable | Tuple[Callable, Dict]]*, *queues:* ColmenaQueues, *config: Config*, *timeout: int | None = None*, *default_executors: str | List[str] = 'all'*)

Bases: *FutureBasedTaskServer*

Task server based on Parsl

Create a Parsl task server by first creating a resource configuration following the recommendations in the Parsl documentation. Then instantiate a task server with a list of Python functions, configurations defining on which Parsl executors each function can run, and the Parsl resource configuration. The executor(s) for each function can be defined with a combination of per method specifications

```
ParslTaskServer([(f, {'executors': ['a']})], queue, config)
```

and also using a default executor

```
ParslTaskServer([f], queue, config, default_executors=['a'])
```

Further configuration options for each method can be defined in the list of methods.

Technical Details

The task server stores each of the supplied methods as Parsl "Apps". Tasks are launched on remote workers by calling these Apps, and results are placed in the result queue by callbacks attached the resultant Parsl Futures.

The behavior of an `ExecutableTask` involves several Apps and callbacks.

1. A `PythonApp` to invoke the "preprocessing" function that is given the `Result`.

   The app produces a path to a temporary run directory containing the input files, content for the standard input of the executable, and an updated copy of the `Result` object containing timing information.

   Note that the `Result` object returned by this app lacks the inputs to reduce communication costs.

   Once complete (successfully or not), it invokes a callback which launches the next two tasks and creates the next callback. In the even of an unsuccessful execution, the callback function returns the failure information to the client and exits.

2. A `BashApp` to run the executable that is given the path to the run directory and the list of resources required for executing the task.

   There is no callback for app.

3. A `PythonApp` to store the results of the execution that is given the exit code of the executable (should be 0), a copy of the `Result` object produced by the preprocessing, the path to the run directory, and a serialized version of the inputs to the app.

   The application parses the outputs from the execution, stores them in the `Result` object, and then serializes results for transmission back to the client. The application also re-inserts the inputs if they are required to be sent back to the client.

   The callback for this function submits the outputs, if successful, or any failure information, if not, to the result queue.

Every one of the Apps is run on the remote system as they may involve manipulating files on the remote system.

Parameters

- **methods** (*list*) – List of methods to be served. Each element in the list is either a function or a tuple where the first element is a function and the second is a dictionary of the arguments being used to create the Parsl ParslApp see Parsl documentation.
- **queues** – Queues for the task server
- **config** – Parsl configuration
- **timeout** (*int*) – Timeout, if desired
- **default_executors** – Executor or list of executors to use by default.

colmena.task_server.globus

Task server based on Globus Compute

Globus Compute provides the ability to execute functions on remote "endpoints" that provide access to computational resources (e.g., cloud providers, HPC). Tasks and results are communicated to/from the endpoint through a cloud service secured using Globus Auth.

class colmena.task_server.globus.**GlobusComputeTaskServer**(*methods: Dict[Callable, str]*, *funcx_client: Client*, *queues:* PipeQueues, *timeout: int | None = None*, *batch_size: int = 128*)

Bases: *FutureBasedTaskServer*

Task server that uses Globus Compute to execute tasks on remote systems

Create a task server by providing a dictionary of functions mapped to the endpoint on which each should run. The task server will wrap the provided function in an interface that tracks execution information (e.g., runtime) and registers the wrapped function with Globus Compute. You must also provide a Globus Compute `Client` that the task server will use to authenticate with the web service.

The task server works using Globus Compute's `Executor` to communicate to the web service over a web socket. The functions used by the executor are registered when you create the task server, and the Executor is launched when you start the task server.

Parameters

- **methods** – Map of functions to the endpoint on which it will run
- **funcx_client** – Authenticated Globus Compute client
- **queues** – Queues used to communicate with thinker
- **timeout** – Timeout for requests from the task queue
- **batch_size** – Maximum number of task request to receive before submitting

perform_callback(*future: Future*, *result:* Result, *topic: str*)

Send a completed result back to queue. Used as a callback for complete tasks

Parameters

- **future** – Future for a task
- **result** – Initial result object. Used if the future throws an exception
- **topic** – Topic used to send back to the user

colmena.task_server.local

Use Python's `Executor` to run workers on a local system

class colmena.task_server.local.**LocalTaskServer**(*queues: ColmenaQueues*, *methods: Collection[Callable | ColmenaMethod]*, *threads: bool = True*, *num_workers: int | None = None*)

Bases: *FutureBasedTaskServer*

Use Python's native concurrent libraries to execute tasks

Parameters

- **methods** – Methods to be served
- **queues** – Queues used to communicate with thinker
- **threads** – Use threads instead of workers
- **num_workers** – Number of workers to deploy.

colmena.task_server.base

Base classes for the Task Server and associated functions

class colmena.task_server.base.**BaseTaskServer**(*queues: ColmenaQueues*, *method_names: Collection[str]*, *timeout: int | None = None*)

Bases: `Process`

Abstract class for the Colmena Task Server, which manages the execution of tasks

Start the task server by first instantiating it and then calling `start()` to launch the server in a separate process. Clients submit task requests to the server by pushing them to a Redis queue, and then receive results from a second queue.

The task server can be stopped by pushing a `None` to the task queue, signaling that no new tasks will be incoming. The remaining tasks will continue to be pushed to the output queue.

Implementing a Task Server

Different implementations vary in how the queue is processed.

Each implementation must provide the *process_queue()* function is responsible for executing tasks supplied on the tasks queue and ensuring completed results are written back to the result queue on completion. Tasks must first be wrapped in the `run_and_record_timing()` decorator function to capture the runtime information.

Implementations should also provide a *_cleanup* function that releases any resources reserved by the task server.

Parameters

- **queues** (*TaskServerQueues*) – Queues for the task server
- **timeout** (*int*) – Timeout for reading from the task queue, if desired

listen_and_launch()

abstract process_queue(*topic: str*, *task: Result*)

Execute a single task from the task queue

Parameters

- **topic** – Which task queue this result came from
- **task** – Task description

run() → None

Launch the thread and start running tasks

Blocks until the inputs queue is closed and all tasks have completed

class colmena.task_server.base.**FutureBasedTaskServer**(*queues:* ColmenaQueues, *method_names:* *Collection[str]*, *timeout:* *int* | *None* = *None*)

Bases: *BaseTaskServer*

Base class for workflow engines that use Python's native Future object

Implementations need to specify a function, `_submit()`, that creates the Future and *FutureBasedTaskServer*'s implementation of *process_queue()* will add a callback to submit the output to the result queue. Note that implementations are still responsible for adding the `run_and_record_timing()` decorator.

Parameters

- **queues** (*TaskServerQueues*) – Queues for the task server
- **timeout** (*int*) – Timeout for reading from the task queue, if desired

perform_callback(*future: Future*, *result:* Result, *topic:* *str*)

Send a completed result back to queue. Used as a callback for complete tasks

Parameters

- **future** – Future for a task
- **result** – Initial result object. Used if the future throws an exception
- **topic** – Topic used to send back to the user

process_queue(*topic:* *str*, *task:* Result)

Execute a single task from the task queue

Parameters

- **topic** – Which task queue this result came from
- **task** – Task description

colmena.task_server.base.**convert_to_colmena_method**(*function:* *Callable* | ColmenaMethod) → *ColmenaMethod*

Wrap user-supplied functions in the task model wrapper, if needed

Parameters
function – User-provided function

Returns
Function as appropriate subclasses of Colmena Task wrapper

1.10.2 colmena.queue

Implementations of the task and result queues from Colmena

colmena.queue.python

Queues built on Python's native libraries

class colmena.queue.python.**PipeQueues**(*topics: Collection[str] = ()*, *serialization_method: str | SerializationMethod = SerializationMethod.PICKLE*, *keep_inputs: bool = True*, *proxystore_name: str | Dict[str, str] | None = None*, *proxystore_threshold: int | Dict[str, int] | None = None*)

Bases: *ColmenaQueues*

Queues using Python's implementation of multiprocessing Pipes

Parameters

- **topics** – Names of topics that are known for this queue
- **serialization_method** – Method used to serialize task inputs and results
- **keep_inputs** – Whether to return task inputs with the result object
- **proxystore_name** (*str*, *dict*) – Name of a registered ProxyStore *Store* instance. This can be a single name such that the corresponding *Store* is used for all topics or a mapping of topics to registered *Store* names. If a mapping is provided but a topic is not in the mapping, ProxyStore will not be used.
- **proxystore_threshold** (*int*, *dict*) – Threshold in bytes for using ProxyStore to transfer objects. Optionally can pass a dict mapping topics to threshold to use different threshold values for different topics. None values in the mapping will exclude ProxyStore use with that topic.

flush()

Remove all existing results from the queues

colmena.queue.redis

Queues that use Redis

class colmena.queue.redis.**RedisQueues**(*topics: Collection[str]*, *hostname: str = '127.0.0.1'*, *port: int = 6379*, *prefix: str = UUID('35d080e1-8cc0-4ae9-bad3-34c704143dc9')*, *serialization_method: str | SerializationMethod = SerializationMethod.PICKLE*, *keep_inputs: bool = True*, *proxystore_name: str | Dict[str, str] | None = None*, *proxystore_threshold: int | Dict[str, int] | None = None*)

Bases: *ColmenaQueues*

A basic redis queue for communications used by the task server

A queue is defined by its prefix and a "topic" designation. The full list of available topics is defined when creating the queue, and simplifies writing software that waits for only certain types of messages without needing to manage several "queue" objects. By default, the `get()` methods for the queue listen on all topics and the `put()` method pushes to the default topic. You can put messages into certain "topical" queue and wait for responses that are from a single topic.

The queue only connects when the *connect* method is called to avoid issues with passing an object across processes.

Parameters

- **hostname** (*str*) – Hostname of the Redis server

- **port** (*int*) – Port on which to access Redis
- **prefix** (*str*) – Name of the Redis queue
- **topics** – Names of topics that are known for this queue
- **serialization_method** – Method used to serialize task inputs and results
- **keep_inputs** – Whether to return task inputs with the result object
- **proxystore_name** (*str, dict*) – Name of ProxyStore backend to use for all topics or a mapping of topic to ProxyStore backend for specifying backends for certain tasks. If a mapping is provided but a topic is not in the mapping, ProxyStore will not be used.
- **proxystore_threshold** (*int, dict*) – Threshold in bytes for using ProxyStore to transfer objects. Optionally can pass a dict mapping topics to threshold to use different threshold values for different topics. None values in the mapping will exclude ProxyStore use with that topic.

connect()

Connect to the Redis server

disconnect()

Disconnect from the server

Useful if sending the connection object to another process

flush(**args*, ***kwargs*) → Any

Remove all existing results from the queues

property is_connected

colmena.queue.base

Base classes for queues and related functions

class colmena.queue.base.**ColmenaQueues**(*topics: Collection[str]*, *serialization_method: str | SerializationMethod = SerializationMethod.JSON*, *keep_inputs: bool = True*, *proxystore_name: str | Dict[str, str] | None = None*, *proxystore_threshold: int | Dict[str, int] | None = None*)

Bases: object

Base class for a queue used in Colmena.

Follows the basic `get` and `put` semantics of most queues, with the addition of a "topic" used by Colmena to separate task requests or objects used for different purposes.

Parameters

- **topics** – Names of topics that are known for this queue
- **serialization_method** – Method used to serialize task inputs and results
- **keep_inputs** – Whether to return task inputs with the result object
- **proxystore_name** (*str, dict*) – Name of a registered ProxyStore *Store* instance. This can be a single name such that the corresponding *Store* is used for all topics or a mapping of topics to registered *Store* names. If a mapping is provided but a topic is not in the mapping, ProxyStore will not be used.

- **proxystore_threshold** (*int, dict*) – Threshold in bytes for using ProxyStore to transfer objects. Optionally can pass a dict mapping topics to threshold to use different threshold values for different topics. None values in the mapping will exclude ProxyStore use with that topic.

property active_count: int

Number of active tasks

abstract flush()

Remove all existing results from the queues

get_result(*topic: str = 'default', timeout: float | None = None*) → *Result* | None

Get a completed result

Parameters

- **topic** – Which topic of task to wait for
- **timeout** – Timeout for waiting for a value

Returns

(Result) Result from a computation

Raises

TimeoutException if the timeout is met –

get_task(*timeout: float | None = None*) → Tuple[str, *Result*]

Get a task object

Parameters

timeout (*float*) – Timeout for waiting for a task

Returns

- (str) Topic of the calculation. Used in defining which queue to use to send the results
- (Result) Task description

Raises

- ***TimeoutException*** – If the timeout on the queue is reached
- ***KillSignalException*** – If the queue receives a kill signal

send_inputs(**input_args: Any, method: str | None = None, input_kwargs: Dict[str, Any] | None = None, keep_inputs: bool | None = None, resources: ResourceRequirements | dict | None = None, topic: str = 'default', task_info: Dict[str, Any] | None = None*) → str

Send a task request

Parameters

- ***input_args** (*Any*) – Positional arguments to a function
- **method** (*str*) – Name of the method to run. Optional
- **input_kwargs** (*dict*) – Any keyword arguments for the function being run
- **keep_inputs** (*bool*) – Whether to override the
- **topic** (*str*) – Topic for the queue, which sets the topic for the result
- **resources** – Suggestions for how many resources to use for the task
- **task_info** (*dict*) – Any information used for task tracking

Returns
Task ID

send_kill_signal()

Send the kill signal to the task server

send_result(*result:* Result)

Send a value to a client

Parameters

- **result** (Result) – Result object to communicate back
- **topic** (*str*) – Topic of the calculation

set_role(*role:* QueueRole | *str*)

Define the role of this queue.

Controls whether users will be warned away from performing actions that are disallowed by a certain queue role, such as sending results from a client or issuing requests from a server

wait_until_done(*timeout: float | None = None*)

Wait until all out-going tasks have completed

Returns
Whether the event was set within the timeout

class colmena.queue.base.**QueueRole**(*value*)

Bases: str, Enum

Role a queue is used for

ANY = 'any'

CLIENT = 'client'

SERVER = 'server'

1.10.3 colmena.models

Models used for different aspects of a Colmena application

class colmena.models.**FailureInformation**(*, *exception: str*, *traceback: str | None = None*)

Bases: BaseModel

Stores information about a task failure

Create a new model by parsing and validating input data from keyword arguments.

Raises ValidationError if the input data cannot be parsed to form a valid model.

exception: str

classmethod from_exception(*exc: BaseException*) → *FailureInformation*

traceback: str | None

class colmena.models.**ResourceRequirements**(*, *node_count: int = 1*, *cpu_processes: int = 1*, *cpu_threads: int = 1*)

Bases: BaseModel

Resource requirements for tasks. Used by some Colmena backends to allocate resources to the task

Follows the naming conventions of RADICAL-Pilot.

Create a new model by parsing and validating input data from keyword arguments.

Raises ValidationError if the input data cannot be parsed to form a valid model.

cpu_processes: int

cpu_threads: int

node_count: int

property total_ranks: int

Total number of MPI ranks

class colmena.models.**Result**(*inputs: Tuple[Tuple[Any], Dict[str, Any]]*, ***, *task_id: str = None*, *value: Any = None*, *method: str | None = None*, *success: bool | None = None*, *complete: bool | None = None*, *task_info: Dict[str, Any] | None = None*, *resources: ResourceRequirements = None*, *failure_info: FailureInformation | None = None*, *worker_info: WorkerInformation | None = None*, *message_sizes: Dict[str, int] = None*, *timestamp: Timestamps = None*, *time: TimeSpans = None*, *serialization_method: SerializationMethod = SerializationMethod.JSON*, *keep_inputs: bool = True*, *proxystore_name: str | None = None*, *proxystore_config: Dict | None = None*, *proxystore_threshold: int | None = None*, *topic: str | None = None*)

Bases: BaseModel

A class which describes the inputs and results of the calculations evaluated by the MethodServer

Each instance of this class stores the inputs and outputs to the function along with some tracking information allowing for performance analysis (e.g., time submitted to Queue, time received by client). All times are listed as Unix timestamps.

The Result class also handles serialization of the data to be transmitted over a RedisQueue

Parameters
inputs (*Any, Dict*) – Inputs to a function. Separated into positional and keyword arguments

property args: Tuple[Any]

complete: bool | None

deserialize() → float

De-serialize the input and value fields

Returns
(float) The time required to deserialize

failure_info: *FailureInformation* | None

classmethod from_args_and_kwargs(*fn_args: Sequence[Any]*, *fn_kwargs: Dict[str, Any] | None = None*, ***kwargs*)

Create a result object form a the arguments and kwargs for the function

Keyword arguments to this function are passed to the initializer for *Result*.

Parameters

- **fn_args** – Positional arguments to the function
- **fn_kwargs** – Keyword arguments to the function

Returns
Result object with the results object

inputs: Tuple[Tuple[Any, ...], Dict[str, Any]] | str

json(**kwargs: *Dict[str, Any]*) → str
Override json encoder to use a custom encoder with proxy support

keep_inputs: bool

property kwargs: Dict[str, Any]

mark_compute_ended()
Mark when the task finished executing

mark_compute_started()
Mark that the compute for a method has started

mark_input_received()
Mark that a task server has received a value

mark_result_received()
Mark that a completed computation was received by a client

mark_result_sent()
Mark when a result is sent from the task server

mark_start_task_submission()
Mark when the Task Server submits a task to the engine

mark_task_received()
Mark when the Task Server receives the task from the engine

message_sizes: Dict[str, int]

method: str | None

proxystore_config: Dict | None

proxystore_name: str | None

proxystore_threshold: int | None

resources: *ResourceRequirements*

serialization_method: *SerializationMethod*

serialize() → Tuple[float, List[Proxy]]
Stores the input and value fields as a pickled objects

Returns

- (float) Time to serialize
- List of any proxies that were created

set_result(*result: Any*, *runtime: float = nan*, *intermediate: bool = False*)

Set the value of this computation

Automatically sets the "time_result_completed" field and, if known, defines the runtime.

Will delete the inputs to the function if the user specifies `self.return_inputs == False`. Removing the inputs once the result is known can save communication time

Parameters

- **result** – Result to be stored
- **runtime** – Runtime for the function
- **intermediate** – If this result is not the final one in a workflow

success: `bool | None`

task_id: `str`

task_info: `Dict[str, Any] | None`

time: `TimeSpans`

timestamp: `Timestamps`

topic: `str | None`

value: `Any`

worker_info: `WorkerInformation | None`

class colmena.models.**SerializationMethod**(*value*)

Bases: `str`, `Enum`

Serialization options

JSON = 'json'

PICKLE = 'pickle'

static deserialize(*method: SerializationMethod*, *message: str*) → Any

Deserialize an object

Parameters

- **method** – Method used to serialize
- **message** – Message to deserialize

Returns
Result object

static serialize(*method: SerializationMethod*, *data: Any*) → str

Serialize an object using a specified method

Parameters

- **method** – Method used to serialize the object
- **data** – Object to be serialized

Returns
Serialized data

1.10.4 colmena.models.methods

Base classes used by Colmena to describe functions being executed by a workflow engine

class colmena.models.methods.**ColmenaMethod**

Base wrapper for a Python function run as part of a Colmena workflow

The wrapper handles the parts of running a Colmena task that are beyond running the function, such as serialization, timing, interfaces to ProxyStore.

name: *str*

Name used to identify the function

function(**args*, ***kwargs*) → Any

Function provided by the Colmena user

class colmena.models.methods.**PythonMethod**(*function: Callable*, *name: str | None = None*)

A Python function to be executed on a single worker

Parameters

- **function** – Generator function to be executed
- **name** – Name of the function. Defaults to *function.__name__*

class colmena.models.methods.**PythonGeneratorMethod**(*function: Callable[[...], Generator | Iterable]*, *name: str | None = None*, *store_return_value: bool = False*, *streaming_queue:* ColmenaQueues *| None = None*)

Python function which runs on a single worker and generates results iteratively

Generator functions support streaming each iteration of the generator to the Thinker when a *streaming_queue* is provided.

Parameters

- **function** – Generator function to be executed
- **name** – Name of the function. Defaults to *function.__name__*
- **store_return_value** – Whether to capture the return value of the generator and store it in the Result object.

stream_result(*y: Any*, *result:* Result, *start_time: float*)

Send an intermediate result using the task queue

Parameters

- **y** – Yielded data from the generator function
- **result** – Result package carrying task metadata
- **start_time** – Start time of the algorithm, used to report

function(**args*, *_result:* Result, ***kwargs*) → Any

Run the Colmena task and collect intermediate results to provide as a list

class colmena.models.methods.**ExecutableMethod**(*executable: List[str]*, *name: str | None = None*, *mpi: bool = False*, *mpi_command_string: str | None = None*)

Task that involves running an executable using a system call.

Such tasks often include a "pre-processing" step in Python that prepares inputs for the executable and a "post-processing" step which stores the outputs (either produced from stdout or written to files) as Python objects.

Separating the task into these two functions and a system call for launching the program simplifies development (shorter functions that ar easier to test), and allows some workflow engines to improve performance by running processing and execution tasks separately.

Implement a new ExecutableTask by defining the executable, a preprocessing method (*preprocess()*), and a postprocessing method (*postprocess()*).

Use the ExecutableTask by instantiating a copy of your new class and then passing it to the task server as you would with any other function.

MPI Executables

Launching an MPI executable requires two parts: a path to an executable and a preamble defining how to launch it. Defining an MPI application using the instructions described above and then set the *mpi* attribute to `True`. This will tell the Colmena task server to look for a "preamble" for how to launch the application.

You may need to supply an MPI command invocation recipe for your particular cluster, depending on your choice of task server. Supply a template as the `mpi_command_string` field, which will be converted by Python's string format function to produce a version of the command with the specific resource requirements of your task by the *render_mpi_launch()* method. The attributes of this class (e.g., `node_count`, `total_ranks`) will be used as arguments to *format*. For example, a template of `aprun -N {total_ranks} -n {cpu_process}` will produce `aprun -N 6 -n 3` if you specify `node_count=2` and `cpu_processes=3`.

Parameters

- **executable** – List of executable arguments
- **name** – Name used for the task. Defaults to `executable[0]`
- **mpi** – Whether to use MPI to launch the exectuable
- **mpi_command_string** – Template for MPI launcher. See *mpi_command_string*.

executable: `List[str]`

Command used to launch the executable

mpi: `bool = False`

Whether this is an MPI executable

mpi_command_string: `str | None = None`

Template string defining how to launch this application using MPI. Should include placeholders named after the fields in ResourceRequirements marked using {}'s. Example: *mpirun -np {total_ranks}*

render_mpi_launch(*resources:* ResourceRequirements) → str

Create an MPI launch command given the configuration

Returns
MPI launch configuration

preprocess(*run_dir: Path*, *args: Tuple[Any]*, *kwargs: Dict[str, Any]*) → Tuple[List[str], str | None]

Perform preprocessing steps necessary to prepare for executable to be started.

These may include writing files to the local directory, creating CLI arguments, or standard input to be passed to the executable

Parameters

- **run_dir** – Path to a directory in which to write files used by an executable
- **args** – Arguments to the task, control how the run is set up
- **kwargs** – Keyword arguments to the function

Returns

- Options to be passed as command line arguments to the executable
- Values to pass to the standard in of the executable

execute(*run_dir: Path*, *arguments: List[str]*, *stdin: str | None*, *resources:* ResourceRequirements | *None* = *None*) → float

Run an executable

Parameters

- **run_dir** – Directory in which to execute the code
- **arguments** – Command line arguments
- **stdin** – Content to pass in via standard in
- **resources** – Amount of resources to use for the application

Returns

s)

Return type
Runtime (unit

assemble_shell_cmd(*arguments: List[str]*, *resources:* ResourceRequirements) → List[str]

Assemble the shell command to be launched

Parameters

- **arguments** – Command line arguments
- **resources** – Resource requirements

Returns
Components of the shell command

postprocess(*run_dir: Path*) → Any

Extract results after execution completes

Parameters
run_dir – Run directory for the executable. Stdout will be written to *run_dir/colmena.stdout* and stderr to *run_dir/colmena.stderr*

function(**args*, *_resources:* ResourceRequirements | *None* = *None*, ***kwargs*)

Execute the function

Parameters

- **args** – Positional arguments
- **kwargs** – Keyword arguments
- **_resources** – Resources available. Optional. Only used for MPI tasks.

1.10.5 colmena.thinker

Base classes for 'thinking' applications that respond to tasks completing

class colmena.thinker.**BaseThinker**(*queue: ColmenaQueues*, *resource_counter: ResourceCounter | None = None*, *daemon: bool = True*, *logger_name: str | None = None*, ***kwargs*)

Bases: Thread

Base class for dataflow program that steers a Colmena application

The intent of this class is to simplify writing an dataflow programs using Colmena. When implementing a subclass, write each operation in the program as class method. Each method should take no inputs and produce no output, and could be thought of as an "operation" or "agent" that will run as a thread.

Each agent communicates with others via queues or other threading objects and the Colmena task server via the `ClientQueues`. The only communication method available by default is a class attribute named `done` that is used to signal that the program should terminate.

Denote each of these agents with the *agent()* decorator, as in:

The decorator will tell Colmena to launch that method as a separate thread when the "Thinker" thread is started. Colmena will also create a distinct logger for each of the agents to that is accessible as the *logger()* property.

Start the thinker by calling `.start()`

Parameters

- **queue** – Queue wrapper used to communicate with task server
- **resource_counter** – Utility to used track resource utilization
- **daemon** – Whether to launch this as a daemon thread
- **logger_name** – An optional name to give to the root logger for this thinker
- ****kwargs** – Options passed to `Thread`

property agent_name

Name of the agent

classmethod list_agents() → List[Callable]

List all functions that map to operations within the thinker application

Returns

List of methods that define agent threads

property logger: Logger

Get the logger for the active thread

make_logger(*name: str | None = None*)

Make a sub-logger for our application

Parameters

name – Name to use for the sub-logger

Returns

Logger with an appropriate name

prepare_agent()

Logic ran before launching an agent.

Override to define how to set up an agent. Consider using `local_details()` to store any agent-specific objects

run()

Launch all operation threads and wait until all complete

Sets the done flag when a thread completes, then waits for all other flags to finish.

Does not raise exceptions if a thread exits with an exception. Exception and traceback information are printed using logging at the WARNING level.

tear_down_agent()

Logic ran after an agent completes.

Override to define any tear down logic.

colmena.thinker.**agent**(*func: Callable | None = None*, *startup: bool = False*)

Decorator that denotes a function as an "agent" thread that is launched when a Thinker process is started

Parameters

- **func** – Do not directly pass this variable. It is used as an argument to the decorator
- **startup** – Whether this agent exiting normally should trigger other agents to stop. All agents will still stop if it exits with an exception

colmena.thinker.**event_responder**(*func: Callable | None = None*, *event_name: str | None = None*, *reallocate_resources: bool = False*, *gather_from: str | None = None*, *gather_to: str | None = None*, *disperse_to: str | None = None*, *max_slots: int | str | None = None*, *slot_step: int = 1*)

Decorator that builds agents which respond to an event being set.

The Thinker associated with this agent must have a class attribute that is an Event with the same name as event_name. The agent will run once the event is set and will reset the event once the function completes (i.e., event.clear). If more than one agent is started by an event, the event will be reset only after all agents finish.

The event responder can launch a thread to acquire resource temporarily. The thread is created if you set reallocate_resources=True in the decorator and transfers resources to a specific pool until the decorated function completes or a user-defined resource cap is set. You must configure from where these resources are acquired, in which resource pool they are placed, and where they are re-allocated after the thread completes. The thread will allocate up to the maximum number of slots defined and then reallocate *all slots available to that pool* to the designated resource.

Parameters

- **func** – Do not directly pass this variable. It is used as an argument to the decorator
- **event_name** – Name of the event to watch
- **reallocate_resources** – Whether to re-allocate resources while function is running
- **gather_from** – Name of a resource pool from which to acquire resources
- **gather_to** – Name of the resource pool to place re-allocated resources
- **disperse_to** – Name of the resource pool to move resources to after function completes
- **max_slots** – Maximum number of resource slots to acquire. Can be an integer, the name of a class attribute of the thinker, or 'none' it no maximum is needed
- **slot_step** – Number of slots to acquire per request

colmena.thinker.**result_processor**(*func: Callable | None = None*, *topic: str = 'default'*)

Decorator that builds agents which respond to results becoming available in a queue

Decorated functions must take a single argument: a result object

Parameters

- **func** – Do not directly pass this variable. It is used as an argument to the decorator
- **topic** – Topic of the queue to pull from

colmena.thinker.**task_submitter**(*func: Callable | None = None*, *task_type: str | None = None*, *n_slots: int | str = 1*)

Decorator that builds agents which respond to computing resources becoming available

Decorated functions should assume that resources are available and reserved when the function is called

Parameters

- **func** – Do not directly pass this variable. It is used as an argument to the decorator
- **task_type** – Name of task pool from which to request resources
- **n_slots** – Number of resources to request. Must be either an integer or the name of a class attribute

colmena.thinker.resources

Utilities for tracking resources

class colmena.thinker.resources.**ReallocatorThread**(*resource_counter: ResourceCounter*, *gather_from: str | None*, *gather_to: str | None*, *disperse_to: str | None*, *max_slots: int | None = None*, *stop_event: Event | None = None*, *slot_step: int = 1*, *logger_name: str | None = None*)

Bases: Thread

Thread that reallocates resources until an event is set.

Create a thread by defining the procedure the thread should follow for reallocation (e.g., from where to gather resources, where to store them, where to put them when done).

The resource allocation thread is stopped by calling `obj.stop_event.set()`. Note that you can provide an Event object to the initializer to use instead of the `stop_event` attribute.

Runs as a daemon thread.

Parameters

- **resource_counter** – Resource counter used to track resources
- **stop_event** – Event which controls when the thread should give resources back. If unset, a new Event is created.
- **logger_name** – Name of the logger, if desired
- **gather_from** – Name of a resource pool from which to acquire resources
- **gather_to** – Name of the resource pool to place re-allocated resources
- **disperse_to** – Name of the resource pool to move resources to after function completes
- **max_slots** – Maximum number of resource slots to acquire
- **slot_step** – Number of slots to acquire per request

run() → None

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

class colmena.thinker.resources.**ResourceCounter**(*total_slots: int*, *task_types: List[str] = ()*)

Bases: object

Utility class for keeping track of resources available for different tasks.

The class manages two pieces of state: the amount of resources allocated to a certain task, and the amount of resources that are currently available for that task. Users of this class can change either state using a series of thread-safe methods.

Tracking Allocations: The resource counter is initialized with a certain count of resources, which represent the total number of a certain computing device available (e.g., node, GPU). They all begin as "unallocated" for any task.

Users change the amount of resources dedicated to tasks by "reallocating" them from one task to another. The *reallocate()* method achieves this by requesting a certain number of units from one task and adding them to a second task's available resources once those units are marked as available.

Tracking Utilization: The amount of resources in use for a certain task is tracked by an internal counter. Users of this class request the use a certain number of resources by calling the *acquire()* method. The method blocks until either the request is completely fulfilled (i.e., the specified amount of resources are marked as available) or the operation times out.

Resources are marked as available again using the *release()* method. The release method marks those resources as available to be re-used for other tasks of the same type. Resources must be re-allocated using *reallocate()*.

Implementation: All of the operations described above are thread-safe. Resource utilization is tracked using a semaphore so that threads can acquire and release resources simultaneously. Resources are acquired as first-come-first-served by using a lock to control access to the "acquire" function of the resource utilization semaphore.

Parameters

- **total_slots** – Total number of nodes available to the resources
- **task_types** – Names of task types

acquire(*task: str | None*, *n_slots: int*, *timeout: float = -1.0*, *cancel_if: Event | None = None*) → bool

Request a certain number of nodes for a particular task

Draws only from the pool of nodes allocated to this task

Blocks until the request completes

Parameters

- **task** – Name of the task
- **n_slots** – Number of slots to request
- **timeout** – Maximum time to wait for the request to be filled
- **cancel_if** – Cancel the request if this event happens

Returns

Whether the request was fulfilled

allocated_slots(*task: str*) → int

Number of slots allocated to a certain task

Parameters
task – Name of the task

available_slots(*task: str | None*) → int

Get the number of nodes available for a certain task

Parameters
task – Name of the task

Returns
Number of slots available for that task

reallocate(*task_from: str | None*, *task_to: str | None*, *n_slots: int | str*, *block: bool = True*, *callback: Callable[[], Any] | None = None*, *timeout: float = -1*, *cancel_if: Event | None = None*) → bool

Transfer computer resources from one task to another

Blocks until complete, unless `block` is set to `False`

Parameters

- **task_from** – Which task to pull resources from (None to request un-allocated nodes)
- **task_to** – Which task to add resources to (None to de-allocate nodes)
- **n_slots** – Number of nodes to request. Set to "all" to reallocate all slots (all allocated slots, not just all available slots)
- **block** – Whether to block until the tasks completes
- **callback** – Callback function. Only used if the call is non-blocking
- **timeout** – Maximum time to wait for the request to be filled
- **cancel_if** – Cancel the request if this event happens

Returns
Whether request was fulfilled. Always `True` if `block==False`

release(*task: str | None = None*, *n_slots: int = 1*, *rerequest: bool = False*, *timeout: float = -1*) → bool | None

Register that nodes for a particular task are available and, optionally, re-request those nodes for the same task.

Blocks until the task request completes

Parameters

- **task** – Name of the task
- **n_slots** – Number of slots to mark as available
- **rerequest** – Whether to re-request nodes immediately after releasing them
- **timeout** – Maximum time to wait for the request to be filled

Returns
Whether the re-request was fulfilled

property unallocated_slots: int

Number of unallocated slots

1.10.6 colmena.exceptions

Exceptions type specific to the colmena

exception colmena.exceptions.**KillSignalException**

Bases: BaseException

Server has received a signal to stop

exception colmena.exceptions.**TimeoutException**

Bases: BaseException

Timeout on listening to a queue has occurred

1.10.7 colmena.proxy

Utilities for interacting with ProxyStore.

Utilities for interacting with ProxyStore

exception colmena.proxy.**ProxyJSONSerializationWarning**

Bases: Warning

colmena.proxy.**get_store**(*name: str*, *config: Dict[str, Any] | None = None*) → Store | None

Get a Store by name or create one if it does not already exist.

Parameters

- **name** (*str*) – name of the store.
- **config** – Store configuration that can be used to reinitialize the Store if provided and a store with *name* is not found.

Returns

The store registered as *name* or a newly intialized and registered store if *kind* is not None.

colmena.proxy.**proxy_json_encoder**(*proxy: Proxy*) → Any

Custom encoder function for proxies

Proxy objects are not JSON serializable so this function, when passed to *json.dumps()*, will attempt to JSON serialize the wrapped object. If the proxy is not resolved, a warning will be raised for the user and the proxy will be replaced with a placeholder string for the proxy. This 1) prevents JSON serialization from failing and 2) avoid unintended resolutions of proxies that may invoke expensive communication operations without the user being aware.

Usage:

```
>>> # With JSON dump/dumps
>>> json.dumps(json_obj_containing_proxy, default=proxy_json_encoder)
>>> # With Pydantic
>>> my_basemodel_instance.json(encoder=proxy_json_encoder)
```

Parameters

proxy (*Proxy*) – proxy to convert to JSON encodable object

Returns

The object wrapped by the proxy if the proxy has already been resolved otherwise a placeholder string.

Raises
TypeError – if *proxy* is not an instance of a Proxy.

colmena.proxy.**resolve_proxies_async**(*args: object | list | tuple | dict*) → List[Proxy]

Begin asynchronously resolving all proxies in input

Scan inputs for instances of *Proxy* and begin asynchronously resolving. This is useful if you have one or more proxies that will be needed soon so the underlying objects can be asynchronously resolved to reduce the cost of the first access to the proxy.

Parameters
args (*object*, *list*, *tuple*, *dict*) – possible object or iterable of objects that may be ObjectProxy instances

Returns
List of the proxies that are being resolved

colmena.proxy.**store_proxy_stats**(*proxy: Proxy*, *proxy_timing: dict*)

Store the timings associated with a proxy, if available

Parameters

- **proxy** – Proxy to evaluate
- **proxy_timing** – Dictionary in which to store timings to be updated

CHAPTER

TWO

WHY THE NAME "COLMENA?"

Colmena is Spanish for "beehive." Colmena applications, like their namesake, feature independent agents that perform a variety of tasks over their lifetimes without complicated communication between each other.

CHAPTER

THREE

CITING COLMENA

If you use Colmena in academic research cite our 2021 paper: link bibtex

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

Q

R

S

T

U

V

W