MPIEvaluator: Run on multi-node HPC systems using mpi4py

Ewout ter Hoeven | 2023-11-07

Introduction

In exploratory modelling and simulation, dealing with uncertainty often demands large-scale computational experiments. While High Performance Computing (HPC) facilities offer the computational resources for these tasks, they can be complex to operate. The Exploratory Modeling Workbench, TU Delft's open-source Python library, facilitates this computational experimentation on standalone computers. However, its inability to natively run on multi-node systems, like HPCs, narrows its utility to models that are relatively simpler and faster.

This report discusses an enhancement to the EMAworkbench – the integration of the MPIEvaluator. With this addition, the workbench not only becomes compatible with multi-node systems, such as TU Delft's supercomputer DelftBlue, but it also ensures seamless transitioning between sequential and parallel experiments irrespective of the computational setup. The underlying motive is to empower researchers to employ larger models and to facilitate a higher number of iterations, which can, in turn, allows for scaling up both models and experiments to sizes not possible before.

This project was executed in from April 2023 to November 2023 under the supervision of Prof.dr.ir. J.H. Kwakkel. It was a 7 EC Capita Selecta as part as the EPA master at the TPM faculty of the TU Delft.

The development was done transparently in the open domain, and can be viewed back in these four items:

- Discussion #266: Multi-node (HPC) evaluator discussion
  Includes most of the requirements and design discussions
- Draft PR #292: Prototype of MPIEvaluator for multi-node workloads
  Includes the prototyping and iteration of the first draft implementation
- PR #299: Introducing MPIEvaluator: Run on multi-node HPC systems using mpi4py
  Includes the final implementation delivered, including extensive documentation
- PR #308: Docs: Add MPIEvaluator tutorial for multi-node HPC systems, including DelftBlue
  Includes a tutorial for using the MPIEvaluator, including an example for running on the DelftBlue SLURM-based system.

This report includes a technical summary of the new MPIEvaluator and the integration into the EMAworkbench.

Contents

Technical summary

In this technical summary the requirements of the solution will first be noted, after which the design choices are extensively discussed. A conceptual model will show the final implementation, after which the performance is presented. Finally, the current limitations are discussed.

Requirements

The main requirements to integrate multi-node system support into the EMAworkbench are:

1. Compatibility: The solution must integrate seamlessly with the existing framework of the EMAworkbench, requiring minimal changes to the current user workflows.
2. Scalability: The enhancement should enable the EMAworkbench to efficiently distribute and execute tasks across multiple nodes in an HPC environment, with the ability to scale as needed for large models and data sets.
3. Portability: The solution should not be overly specialized to any single HPC configuration, ensuring broad applicability across various HPC environments, including those not managed by SLURM.
4. Transparency and debuggability: The system must maintain a level of transparency in operations, providing sufficient logging and debugging capabilities to trace and correct errors effectively.

Some nice to have features are:

5. FileModel support: Support models other than Pytho-based ones, such as NetLogo or Vensim models.
6. SLURM support: For SLURM systems, the solution should provide mechanisms for user authentication, file transfer, job submission, and environment setup to facilitate the use of remote SLURM HPC clusters.
7. Efficient data transfer: The system must facilitate the efficient transfer of models and data between nodes to minimize latency and maximize performance.

Design

Abstraction level

The main big design decision was at which abstraction level this problem should be solved. From discussion with the supervisor[1], two distinct approaches where considered:

A. *Solving at the SLURM level*
   - The proposed SLURMEvaluator class focuses on addressing the problem at the level of SLURM, the job scheduler used on HPC clusters.
   - It involves creating a new evaluator class that can connect to remote SLURM HPC clusters, transfer files, submit batch jobs, and manage the environment.
   - The design allows for user authentication, environment setup, and job submission, making it possible to run experiments on remote SLURM clusters.

[1] Discussion #266: *Multi-node (HPC) evaluator discussion*

- This approach shifts the responsibility of handling SLURM-specific operations to the evaluator class, potentially simplifying the user experience.

This approach would have been the most user-friendly for a specific group of potential users. Since login, file transfer and submitting batch numbers would have been handled, it could potentially make running on a HPC system a one-liner from any existing Python script or notebook.

However, this approach would have had many disadvantages. Likely, we would make a working implementation for DelftBlue, the TU Delft supercomputer, but not for users of other SLURM systems, let alone users that have HPC clusters or servers which are not managed with SLURM at all.

Additionally, it would also lack transparency about what's happening in the process, and make debugging therefor very complex. It would also lack modularity, it would be build for a specific system and not be easily scalable to.

So this high abstract-level approach was abandoned in favor of a lower level solution.

B. Solving at the MPI level

- The proposed MPIEvaluator class tackles the problem at the level of MPI (Message Passing Interface), which operates on systems with already allocated nodes.
- This approach involves defining worker functions for executing individual experiments and initializing worker processes with the necessary environment and settings.
- It aims to distribute tasks over available nodes, allowing for efficient utilization of HPC resources without specifying the exact number of nodes in advance.
- This approach emphasizes a separation of data generation on the HPC and subsequent analysis locally.

By solving this at the MPI level, any system that can handle MPI can use the new multi-node evaluator, making it a very scalable and portable solution. Users of SLURM systems have to handle file transfers, login and scheduling jobs themselves on SLURM clusters. To aid those users, an extensive tutorial has been written[2].

Distributing tasks across HPC systems

With the EMAworkbench creating a large set of experiments, these should be executed distributed across multiple nodes within a High-Performance Computing (HPC) environment using Python. This requires a method that can robustly manage task allocation while minimizing the overhead associated with data and model transfer between nodes.

High-level Python runtimes often lack native support for efficient parallel execution across multiple HPC nodes. The essence of the problem lies in two core areas: the robust distribution of tasks (without excessive overhead) and the transfer of models and data between nodes. HPC systems, due to their distributed nature, present a challenge in synchronizing the computational workload without incurring significant communication costs that can offset the advantages of parallelism.

This is not a new problem, a solution used by many HPC systems is the Message Passing Interface. MPI provides a standard for efficient communication between processes in a distributed computing environment. Python's integration with MPI, through the mpi4py library, extends this capability to Python programs, allowing them to utilize multiple nodes in an HPC system. Despite the integration,

[2] PR #308: *Docs: Add MPIEvaluator tutorial for multi-node HPC systems, including DelftBlue*

a direct application of MPI in Python can be complex due to its low-level nature, which does not align with Python's high-level syntax and dynamic features.

The MPIPoolExecutor from the mpi4py.futures[3] module presents a solution by offering a high-level interface for asynchronous execution over a pool of MPI processes. It follows the design of concurrent.futures[4] from Python's standard library, and fullfilles the requirements for distributing experiments generated by the EMAworkbench.

- Task distribution: MPIPoolExecutor manages a pool of worker processes, enabling the delegation of tasks to different nodes. This pool abstracts the complexity of MPI's inter-process communication, thus facilitating ease of use for Python runtime tasks.
- Data transfer efficiency: The executor efficiently handles the transmission of tasks and associated data between nodes. Its design minimizes overhead by reducing the frequency and volume of inter-node communication required to dispatch and execute tasks.
- SLURM integration: The design of MPIPoolExecutor complements SLURM's scheduling capabilities. It does not necessitate pre-determination of the number of nodes or processors, allowing SLURM to manage resource allocation dynamically based on availability and workload demand. The ability to integrate with SLURM scheduling simplifies the execution of distributed tasks, conforming to established HPC job management practices.

Initializer

A critical component in the design of MPIEvaluator is the initializer, which is responsible for setting up the environment for each worker process. In earlier iterations, the use of a global initializer function was intended to streamline the process of configuring each MPI worker with necessary settings. However, this design encountered issues with re-initialization, particularly evident when the MPIEvaluator pool was invoked consecutively.

To resolve this, the decision was made to eliminate the common initializer function, thereby addressing the 'BrokenExecutor' error and enhancing the robustness of the system. This refinement in design meant that each MPI worker would independently configure its environment, increasing the reliability of consecutive invocations.

The removal of a centralized initializer aligns with the principles of MPI, which favours autonomy and decentralization of process management. By allowing each worker to handle its initialization, the design mitigates potential points of failure that can arise from a shared initial setup. This approach also simplifies the overall structure of the MPIEvaluator, adhering to the Pythonic principle of "simple is better than complex."

Limitations

The current implementation packages the model within each task packet sent to worker nodes. While this simplifies the distribution process, there is a potential for performance enhancement by segregating the model transfer to a singular operation, reducing data duplication. Additionally, memory-intensive models could challenge the single callback design, suggesting the need for a disk-based streaming callback to handle large data sets effectively.

SLURM scheduling

The main objective of this project was to allow the EMAworkbench to run on multi-node systems. However, a significant portion of HPC systems use the SLURM scheduler, including the TU Delft's

[3] https://mpi4py.readthedocs.io/en/stable/mpi4py.futures.html
[4] https://docs.python.org/3/library/concurrent.futures.html

supercomputer DelftBlue. Since the main problem would be solved at the MPI level (as discussed in the Abstraction level section), a tutorial was written to support users running on SLURM systems. It includes efficient job scheduling, manages file transfers, and maintains dependency management within the constraints of an HPC environment.

Instead of configuring a one-size-fits-all script, a template and tutorial approach was adopted. Users are guided to tailor the SLURM script to their specific needs, ensuring a balance between resource allocation and availability.

The tutorial includes environment configuration and dependency management through the use of modules and user-level package installations. This ensures that users can set up their computational environment independently, without the need for root access, which is in line with the security policies of shared HPC resources.

By creating a tutorial, the aim was to inform users on the end-to-end process, from job script creation to file transfer, and job submission to output retrieval. This allows users to adapt their workflows to the SLURM system and troubleshoot common issues.

Logging and debuggability

In high-performance computing, having a way to trace and correct errors across multiple computers is a challenge. This section details the rationale behind the design choices for implementing debugging and logging capabilities in the MPIEvaluator component of the Exploratory Modeling Workbench.

Effective debugging and logging in a distributed system like a high-performance computing cluster requires a solution that captures and organizes the actions of each node. The goal is to maintain performance while also ensuring that logs are comprehensive and coherent.

Three approaches were considered:[5]

1. Individual log files**:** Each computing node would record its own log. While complete, this could leave a scattered set of data if the process is interrupted.
2. Centralized log file**:** All log data would be sent to and recorded by a single node. This keeps logs in one place but could slow down the system if logging interferes with computation.
3. Asynchronous logging**:** Logs are sent to a separate logging process that handles them without interrupting the computing nodes. This aims to keep the system running smoothly while still capturing log data, with the risk that some logs might be lost in case of a crash.

The third approach, asynchronous logging, was selected. It's the most user-friendly solution which didn't reduced performance too much.

This method incorporates a dedicated logging process for each MPI process with includes displaying the rank of the process, which helps in tracing logs back to their origin. A key aspect of this design is maintaining a uniform level of logging verbosity across all nodes. To achieve this, a specific flag, pass_root_logger_level, was introduced. This ensures that the logging level is consistent across the various components of the application, thereby preventing discrepancies in the level of detail reported in the logs.

[5] https://github.com/quaquel/EMAworkbench/discussions/266#discussioncomment-7132496

In this asynchronous model, log messages are sent to a separate process designated for logging tasks, which handles these messages without disrupting the main computational processes. This design choice is intended to prevent the potential for input/output blocking that could occur if the main processes were waiting for logging operations to complete. By sidestepping such blocks, the design aims to keep performance degradation to a minimum.

Implementation

This section will show a brief overview of the implemented code, based on the design choices discussed in the previous section.

MPIEvaluator Class

The MPIEvaluator class is at the main component of this implementation. Its primary role is to initiate a pool of workers across multiple nodes, evaluate experiments in parallel, and finalize resources when done.

Initialization:

- It imports mpi4py only when instantiated, preventing unnecessary dependencies for users who do not use the MPIEvaluator.
- The number of processes (nodes) is optionally accepted during initialization.
- The MPI pool of workers is started, with a warning given if the number of workers is low (indicating that the evaluator might be slower than its sequential or multiprocessing counterparts).

Evaluation:

- Experiments are first packed with the necessary information for processing across nodes, including the model name and the experiment details.
  - Note that currently the model is included in this package. This simplifies the implementation substantially, but with larger models there might be potential for performance gains if the model isn't send with each experiment, but just once to each worker.
- Experiments are then dispatched to worker nodes for parallel processing using MPIPoolExecutor.map().
- Once all experiments are done, outcomes are passed to a callback for post-processing.
  - Note: Models using a lot of memory could run out of memory before the (single) Callback. A new streaming-to-disk Callback class could help allow for models that gather data that exceeds the memory size.

Finalization:

- The MPI pool of workers is shut down.

run_experiment_mpi Function

This helper function is designed to unpack experiment data, set up the necessary logging configurations, run the experiment on the designated MPI rank (node), and return the results. This is the worker function that runs on each of the MPI ranks.

Logging:

- Logging configurations are set up based on the level passed during experiment packing. This ensures uniformity in logging verbosity across nodes.

- Messages include MPI rank details for easier debugging.

Logging Enhancements

A dedicated logger for the MPIEvaluator was introduced to provide clarity during debugging and performance tracking. Several measures were taken to ensure uniform logging verbosity across nodes and improve log readability:

- The MPI process name and rank is displayed alongside the log level.
- An optional flag to adjust root logger levels was introduced, ensuring uniformity across different modules.
  - pass_root_logger_level argument has be added to ema_logging.log_to_stderr. This ensures that the root logger level is passed to all modules, so that they will log identical levels. Example:

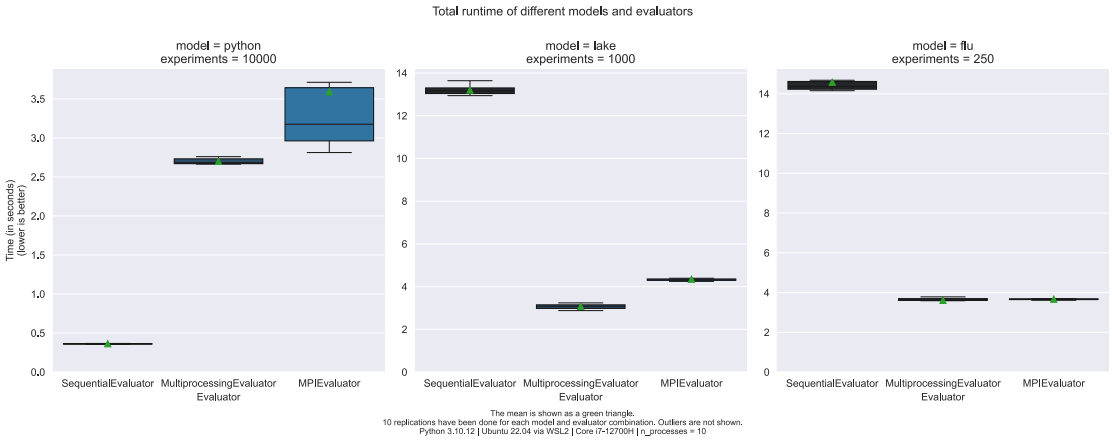
    ema_logging.log_to_stderr(level=20, pass_root_logger_level=True)

Performance

After implementation, the performance of the MPIEvaluator was measured. Two aspects were focussed on: The overhead of the MPIEvaluator compared to other evaluators, and the performance scaling on large systems.

Overhead

The overhead was measured by comparing the three evaluators (Sequential, Multiprocessing and MPI) on 3 different models for a fixed number of experiments. The faster these experiments would be completed, the higher the performance and the lower the overhead. The results are shown in the graph below.



Notable is that for a very small model like the simple python model, the SequentialEvaluator is the fastest, even as the other evaluators have access to 10 cores. For larger models like the lake and flu models, a significant speedup can be seem by using the Multiprocessing or MPIEvaluator. On the lake model the MPIEvaluator is a little slower than the Multiprocessing evaluator, but on the flu model they perform identical. This indicates that the MPIEvaluator has insignificant overhead on larger models.

Since the Multiprocessing and MPI evaluator had access to 10 cores, it's useful to look at the performance per core, to determine the overhead.

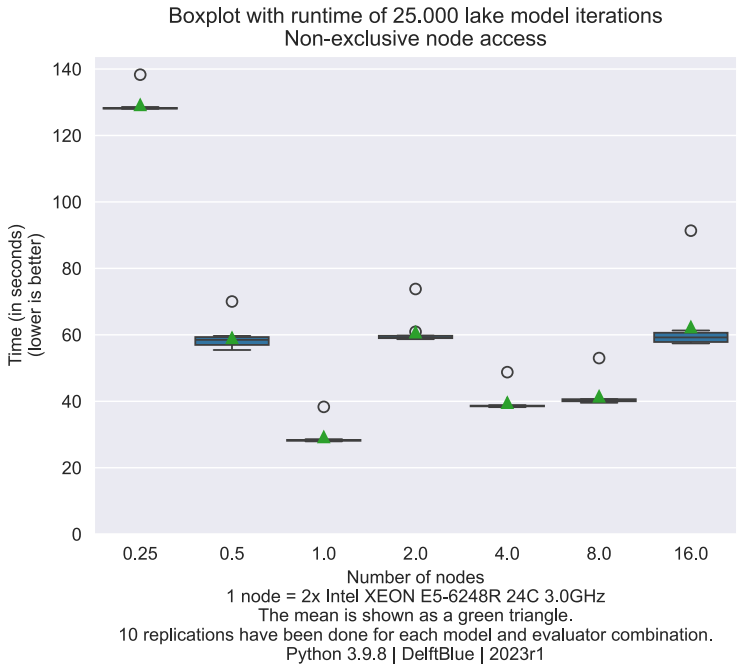
Model	SequentialEvaluator	MultiprocessingEvaluator	MPIEvaluator
python	1	0.0135	0.0101
lake	1	0.4296	0.3037
flu	1	0.4041	0.3976

For the small python model, the performance per core of the MPIEvaluator was only 0.01x the performance of the Sequential evaluator, indicating around 100x overhead. For the larger models that decreases significantly, to only around 2.5x overhead on the flu model.

Scaling

Looking at performance scaling to multiple nodes, three experiments were executed to measure how much the performance increases when scaling up to many nodes.

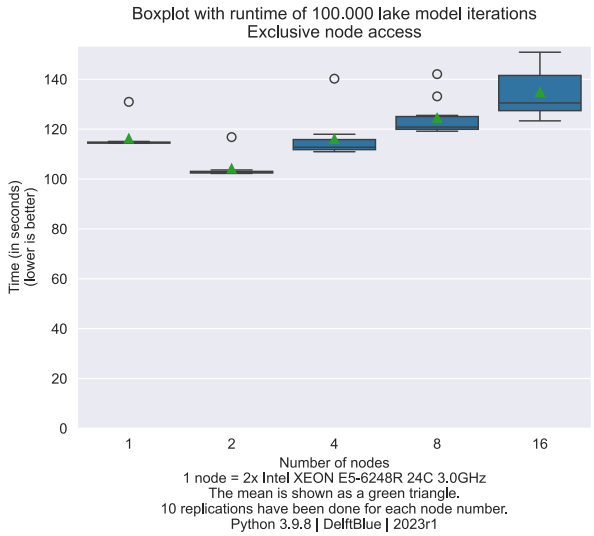
The first experiment run 25.000 iterations of the lake model on 0.25 to 16 nodes, each with 10 replications. The nodes were granted with non-exclusive access, meaning that basically there was a number of cores reserved that was 48 times the number of cores. However, these workers could be spread out over more nodes that listed in the graph.

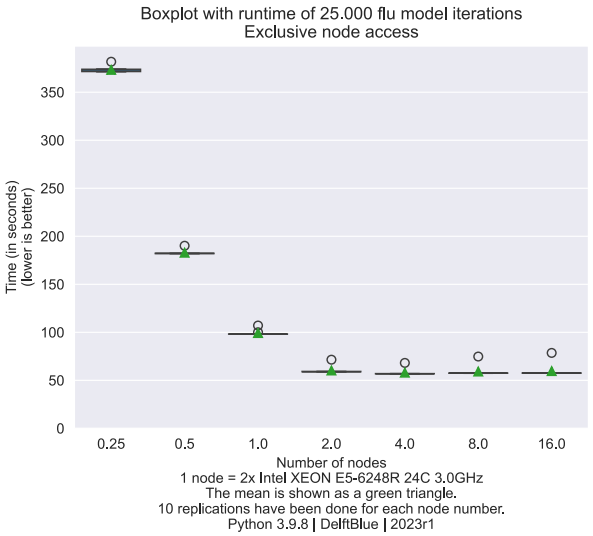


Notable is the almost linear performance scaling from 0.25 to 1 node, but the inconsistent scaling after that, without any system being faster than the 1 node one.

Therefore, two experiments with exclusive node access where performed. From a HPC perspective this isn't ideal, since it requires full nodes to free up with takes longer and is more difficult to schedule. The number of iterations was also increased to get more consistent results

The first experiment runs the lake model 100.000 times and the second one runs the flu model 25.000 times, both from 1 to 16 nodes exclusive nodes.





The lake model shows an performance improvement when running on two nodes, but nowhere near linear scaling. The flue model shows near-linear performance scaling from 0.25 to 2 nodes, but no significant scaling after that.

Both models are relatively simple with large communication overhead. Testing the performance on a more compute intensive model would be interesting for future research.

Limitations & future enhancements

There are two main limitations currently:

- The MPIEvaluator is not tested with file-based models, such as NetLogo and Vensim. It might still work, but it's not tested. Originally this was in scope for this project, but due to difficulties in creating the proper environment, this was cut out of the scope of this effort.
- The model object is currently passed to the worker for each experiment. For large models with a relatively short runtime, this introduces significant performance overhead. Therefore, and optimization could be made to send the model only once to a worker on initialization.
  - Building on this, submitting experiment parameter sets in batches could also help increate performance, instead of sending them to the workers one-by-one.

Some other future improvements could be:

- The decision was made to solve the problem on the MPI level. However, since that's a lower abstraction level than SLURM, both wrappers for generals SLURM systems or the DelftBlue system specifically could still be added. Those would use the MPIEvaluator under the hood, but could create the SBATCH scripts for scheduling, authentication and file transfers.
  - This would allow a one-line solution, running from Python scripts or Jupyter notebooks.
- A new Callback class could be implemented that streams to the disk instead of keeping all results in memory. This would allow for handling very large model that

gather lots of data, probably at the costs of some performance. See issue #304 for more details.

- Further performance profiling could be done on the current design, to see any components can be sped up, like the distribution of experiments and models to workers, the logging or the.
  - It would be interesting to see how larger models perform on many-node systems, and if the scaling is better than with the small *lake* and *flu* models.

Appendixes

Two appendixes are included, with the final code diff to the EMAworkbench and the tutorial.

Appendix A: Code diff

This appendix provides the final code diff applied to the EMAworkbench through pull request #299.

```
diff --git a/.github/workflows/ci.yml b/.github/workflows/ci.yml
index 6c0972ae8..cc3747c4f 100644
--- a/.github/workflows/ci.yml
+++ b/.github/workflows/ci.yml
@@ -23,6 +23,7 @@ jobs:
         include:
           - os: ubuntu-latest
             python-version: "3.10"
+            test-mpi: true
           - os: ubuntu-latest
             python-version: "3.9"
           - os: ubuntu-latest
@@ -42,6 +43,12 @@ jobs:
       run: |
         pip install --upgrade pip
         pip install .[dev,cov] ${{ matrix.pip-pre }}
+    - name: Install MPI and mpi4py
+      if: matrix.test-mpi == true
+      run: |
+        sudo apt-get update
+        sudo apt-get install -y libopenmpi-dev
+        pip install mpi4py
     - name: Test with Pytest
       timeout-minutes: 15
       run:
diff --git a/ema_workbench/__init__.py b/ema_workbench/__init__.py
index bce53c12c..ba928e9ef 100644
--- a/ema_workbench/__init__.py
+++ b/ema_workbench/__init__.py
@@ -15,6 +15,7 @@
     Constant,
     Scenario,
     Policy,
+    MPIEvaluator,
     MultiprocessingEvaluator,
     IpyparallelEvaluator,
     SequentialEvaluator,
diff --git a/ema_workbench/em_framework/__init__.py
b/ema_workbench/em_framework/__init__.py
index e3b6f3394..357565b65 100644
--- a/ema_workbench/em_framework/__init__.py
+++ b/ema_workbench/em_framework/__init__.py
@@ -30,6 +30,7 @@
     "perform_experiments",
     "optimize",
     "IpyparallelEvaluator",
+    "MPIEvaluator",
     "MultiprocessingEvaluator",
     "SequentialEvaluator",
     "ReplicatorModel",
@@ -76,6 +77,7 @@
 from .evaluators import (
     perform_experiments,
     optimize,
```

```
+    MPIEvaluator,
     MultiprocessingEvaluator,
     SequentialEvaluator,
     Samplers,
diff --git a/ema_workbench/em_framework/evaluators.py
b/ema_workbench/em_framework/evaluators.py
index d5f316cd5..bcfa1f91b 100644
--- a/ema_workbench/em_framework/evaluators.py
+++ b/ema_workbench/em_framework/evaluators.py
@@ -13,6 +13,7 @@
 import sys
 import threading
 import warnings
+import logging

 from ema_workbench.em_framework.samplers import AbstractSampler
 from .callbacks import DefaultCallback
@@ -415,6 +416,71 @@ def evaluate_experiments(self, scenarios, policies,
callback, combine="factorial
         add_tasks(self.n_processes, self._pool, ex_gen, callback)


+class MPIEvaluator(BaseEvaluator):
+    """Evaluator for experiments using MPI Pool Executor from mpi4py"""
+
+    def __init__(self, msis, n_processes=None, **kwargs):
+        super().__init__(msis, **kwargs)
+        self._pool = None
+        self.n_processes = n_processes
+
+    def initialize(self):
+        # Only import mpi4py if the MPIEvaluator is used, to avoid
unnecessary dependencies.
+        from mpi4py.futures import MPIPoolExecutor
+
+        self._pool = MPIPoolExecutor(max_workers=self.n_processes)  #
Removed initializer arguments
+        _logger.info(f"MPI pool started with {self._pool._max_workers}
workers")
+        if self._pool._max_workers <= 10:
+            _logger.warning(
+                f"With only a few workers ({self._pool._max_workers}), the
MPIEvaluator may be slower than the Sequential- or
MultiprocessingEvaluator"
+            )
+        return self
+
+    def finalize(self):
+        self._pool.shutdown()
+        _logger.info("MPI pool has been shut down")
+
+    def evaluate_experiments(self, scenarios, policies, callback,
combine="factorial"):
+        ex_gen = experiment_generator(scenarios, self._msis, policies,
combine=combine)
+        experiments = list(ex_gen)
+        log_level = _logger.getEffectiveLevel()
+
+        packed = [
+            (experiment, experiment.model_name, self._msis, log_level) for
experiment in experiments
```

```
+        ]
+
+        _logger.info(
+            f"MPIEvaluator: Starting {len(packed)} experiments using MPI
pool with {self._pool._max_workers} workers"
+        )
+        results = self._pool.map(run_experiment_mpi, packed)
+
+        _logger.info(f"MPIEvaluator: Completed all {len(packed)}
experiments")
+        for experiment, outcomes in results:
+            callback(experiment, outcomes)
+        _logger.info(f"MPIEvaluator: Callback completed for all
{len(packed)} experiments")
+
+
+def run_experiment_mpi(packed_data):
+    from mpi4py.MPI import COMM_WORLD
+
+    rank = COMM_WORLD.Get_rank()
+
+    experiment, model_name, msis, level = packed_data
+
+    logging.basicConfig(level=level,
format="[%(processName)s/%(levelname)s] %(message)s")
+    _logger.debug(f"MPI Rank {rank}: starting {repr(experiment)}")
+
+    models = NamedObjectMap(AbstractModel)
+    models.extend(msis)
+    experiment_runner = ExperimentRunner(models)
+
+    outcomes = experiment_runner.run_experiment(experiment)
+
+    _logger.debug(f"MPI Rank {rank}: completed {experiment}")
+
+    return experiment, outcomes
+
+
 class IpyparallelEvaluator(BaseEvaluator):
     """evaluator for using an ipypparallel pool"""

diff --git a/ema_workbench/util/ema_logging.py
b/ema_workbench/util/ema_logging.py
index 47538eb9e..ca2f484c7 100644
--- a/ema_workbench/util/ema_logging.py
+++ b/ema_workbench/util/ema_logging.py
@@ -178,7 +178,7 @@ def get_rootlogger():
     return _rootlogger


-def log_to_stderr(level=None):
+def log_to_stderr(level=None, pass_root_logger_level=False):
     """
     Turn on logging and add a handler which prints to stderr

@@ -186,6 +186,10 @@ def log_to_stderr(level=None):
     ----------
     level : int
             minimum level of the messages that will be logged
+    pas_root_logger_level: bool, optional. Default False
+            if true, all module loggers will be set to the
```

```
+            same logging level as the root logger.
+            Recommended True when using the MPIEvaluator.

     """

@@ -206,4 +210,8 @@ def log_to_stderr(level=None):
     logger.addHandler(handler)
     logger.propagate = False

+    if pass_root_logger_level:
+        for _, mod_logger in _module_loggers.items():
+            mod_logger.setLevel(level)
+
     return logger
diff --git a/test/test_em_framework/test_evaluators.py
b/test/test_em_framework/test_evaluators.py
index bf0b0bfc1..fcf4753bb 100644
--- a/test/test_em_framework/test_evaluators.py
+++ b/test/test_em_framework/test_evaluators.py
@@ -4,6 +4,7 @@
 """
 import unittest.mock as mock
 import unittest
+import platform

 import ema_workbench
 from ema_workbench.em_framework import evaluators
@@ -74,6 +75,51 @@ def test_ipyparallel_evaluator(
             evaluator.evaluate_experiments(10, 10, mocked_callback)
             lb_view.map.called_once()

+    # Check if mpi4py is installed and if we're on a Linux environment
+    try:
+        import mpi4py
+
+        MPI_AVAILABLE = True
+    except ImportError:
+        MPI_AVAILABLE = False
+    IS_LINUX = platform.system() == "Linux"
+
+    @unittest.skipUnless(
+        MPI_AVAILABLE and IS_LINUX, "Test requires mpi4py installed and a
Linux environment"
+    )
+    @mock.patch("mpi4py.futures.MPIPoolExecutor")
+    @mock.patch("ema_workbench.em_framework.evaluators.DefaultCallback")
+
@mock.patch("ema_workbench.em_framework.evaluators.experiment_generator")
+    def test_mpi_evaluator(self, mocked_generator, mocked_callback,
mocked_MPIPoolExecutor):
+        try:
+            import mpi4py
+        except ImportError:
+            self.fail(
+                "mpi4py is not installed. It's required for this test.
Install with: pip install mpi4py"
+            )
+
+        model = mock.Mock(spec=ema_workbench.Model)
+        model.name = "test"
+
```

```
+        # Create a mock experiment with the required attribute
+        mock_experiment = mock.Mock()
+        mock_experiment.model_name = "test"
+        mocked_generator.return_value = [mock_experiment]
+
+        pool_mock = mock.Mock()
+        pool_mock.map.return_value = [(1, ({}, {}))]
+        pool_mock._max_workers = 5  # Arbitrary number
+        mocked_MPIPoolExecutor.return_value = pool_mock
+
+        with evaluators.MPIEvaluator(model) as evaluator:
+            evaluator.evaluate_experiments(10, 10, mocked_callback)
+
+            mocked_MPIPoolExecutor.assert_called_once()
+            pool_mock.map.assert_called_once()
+
+        # Check that pool shutdown was called
+        pool_mock.shutdown.assert_called_once()
+
    def test_perform_experiments(self):
        pass
```

Appendix B: Tutorial

Appendix B provides the full tutorial for using the MPIEvaluator on HPC systems from PR #308.

MPIEvaluator: Run on multi-node HPC systems

The MPIEvaluator is a new addition to the EMAworkbench that allows experiment execution on multi-node systems, including high-performance computers (HPCs). This capability is particularly useful if you want to conduct large-scale experiments that require distributed processing. Under the hood, the evaluator leverages the MPIPoolExecutor from mpi4py.futures.

Limiations

- Currently, the MPIEvaluator is only tested on Linux, while it might work on other operating systems.
- Currently, the MPIEvaluator is only tested with Python-based models, while it might work with other file-based model types (like NetLogo or Vensim).
- The MPIEvaluator is most useful for large-scale experiments, where the time spent on distributing the experiments over the cluster is negligible compared to the time spent on running the experiments. For smaller experiments, the overhead of distributing the experiments over the cluster might be significant, and it might be more efficient to run the experiments locally.

This tutorial will first show how to set up the environment, and then how to use the MPIEvaluator to run a model on a cluster. Finally, we'll use the DelftBlue supercomputer as an example, to show how to run on a system which uses a SLURM scheduler.

1. Setting up the environment

To use the MPIEvaluator, MPI and mpi4py must be installed.

Installing MPI on Linux typically involves the installation of a popular MPI implementation such as OpenMPI or MPICH. Below are the instructions for installing OpenMPI:

1a. Installing OpenMPI

You can install OpenMPI using you package manager. First, update your package repositories, and then install OpenMPI:

For **Debian/Ubuntu**:

```
sudo apt update
sudo apt install openmpi-bin libopenmpi-dev
```

For **Fedora**:

```
sudo dnf check-update
sudo dnf install openmpi openmpi-devel
```

For **CentOS/RHEL**:

```
sudo yum update
sudo yum install openmpi openmpi-devel
```

Many times, the necessary environment variables are automatically set up. You can check if this is the case by running the following command:

```
mpiexec --version
```

If not, add OpenMPI's bin directory to your PATH:

```
export PATH=$PATH:/usr/lib/openmpi/bin
```

1b. Installing mpi4py

The python package mpi4py needs to installed as well. This is most easily done from PyPI, by running the following command:

pip install -U mpi4py

2. Creating a model

First, let's set up a minimal model to test with. This can be any Python-based model, we're using the example_python.py model from the EMA Workbench documentation as example.

We recommend crafting and testing your model in a separate Python file, and then importing it into your main script. This way, you can test your model without having to run it through the MPIEvaluator, and you can easily switch between running it locally and on a cluster.

2a. Define the model

First, we define a Python model function.

In [ ]:

```
def some_model(x1=None, x2=None, x3=None):
    return {"y": x1 * x2 + x3}
```

Now, create the EMAworkbench model object, and specify the uncertainties and outcomes:

In [ ]:

```
from ema_workbench import Model, RealParameter, ScalarOutcome, ema_logging,
perform_experiments

if __name__ == "__main__":
    # We recommend setting pass_root_logger_level=True when running on a
cluster, to ensure consistent log levels.
    ema_logging.log_to_stderr(level=ema_logging.INFO,
pass_root_logger_level=True)

    ema_model = Model("simpleModel", function=some_model)  # instantiate
the model

    # specify uncertainties
    ema_model.uncertainties = [
        RealParameter("x1", 0.1, 10),
        RealParameter("x2", -0.01, 0.01),
        RealParameter("x3", -0.01, 0.01),
    ]
    # specify outcomes
    ema_model.outcomes = [ScalarOutcome("y")]
```

2b. Test the model

Now, we can run the model locally to test it:

In [ ]:

```
from ema_workbench import SequentialEvaluator

with SequentialEvaluator(ema_model) as evaluator:
    results = perform_experiments(ema_model, 100, evaluator=evaluator)
```

In this stage, you can test your model and make sure it works as expected. You can also check if everything is included in the results and do initial validation on the model, before scaling up to a cluster.

3. Run the model on a MPI cluster

Now that we have a working model, we can run it on a cluster. To do this, we need to import the MPIEvaluator class from the ema_workbench package, and instantiate it with our model. Then, we can use the perform_experiments function as usual, and the MPIEvaluator will take care of distributing the experiments over the cluster. Finally, we can save the results to a pickle file, as usual.

In [ ]:

```
# ema_example_model.py
from ema_workbench import (
    Model,
    RealParameter,
    ScalarOutcome,
    ema_logging,
    perform_experiments,
    MPIEvaluator,
)
import pickle


def some_model(x1=None, x2=None, x3=None):
    return {"y": x1 * x2 + x3}


if __name__ == "__main__":
    ema_logging.log_to_stderr(level=ema_logging.INFO,
pass_root_logger_level=True)

    ema_model = Model("simpleModel", function=some_model)

    ema_model.uncertainties = [
        RealParameter("x1", 0.1, 10),
        RealParameter("x2", -0.01, 0.01),
        RealParameter("x3", -0.01, 0.01),
    ]
    ema_model.outcomes = [ScalarOutcome("y")]

    # Note that we switch to the MPIEvaluator here
    with MPIEvaluator(ema_model) as evaluator:
        results = evaluator.perform_experiments(scenarios=10000)

    # Save the results to a pickle file
    with open("ema_mpi_test.pickle", "wb") as handle:
        pickle.dump(results, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

To run this script on a cluster, we need to use the mpiexec command:

mpiexec python3 -m mpi4py.futures ema_example_model.py

This command will execute the ema_example_model.py Python script using MPI, leveraging the mpi4py.futures module for parallel processing. The number of processes and other MPI-specific configurations would be inferred from default settings or any configurations provided elsewhere, such as in an MPI configuration file or additional flags to mpiexec (not shown in the provided command).

The output of the script will be saved to the ema_mpi_test.pickle file, which can be loaded and analyzed as usual.

Example: Running on the DelftBlue supercomputer (with SLURM)

As an example, we'll show how to run the model on the DelftBlue supercomputer, which uses the SLURM scheduler. The DelftBlue supercomputer is a cluster of 218 nodes, each with 2 Intel Xeon Gold E5-6248R CPUs (48 cores total), 192 GB of RAM, and 480 GB of local SSD storage. The nodes are connected with a 100 Gbit/s Infiniband network.

These steps roughly follow the DelftBlue Crash-course for absolute beginners. If you get stuck, you can refer to that guide for more information.

1. Creating a SLURM script

First, you need to create a SLURM script. This is a bash script that will be executed on the cluster, and it will contain all the necessary commands to run your model. You can create a new file, for example slurm_script.sh, and add the following lines:

```
#!/bin/bash

#SBATCH --job-name="Python_test"
#SBATCH --time=00:02:00
#SBATCH --ntasks=25
#SBATCH --cpus-per-task=1
#SBATCH --partition=compute
#SBATCH --mem-per-cpu=1GB
#SBATCH --account=research-tpm-mas

module load 2023r1
module load openmpi
module load python
module load py-numpy
module load py-mpi4py
module load py-pip

pip install -U --user ema_workbench

mpiexec python3 -m mpi4py.futures ema_example_model.py
```

Modify the script to suit your needs:

- Set the --job-name to something descriptive.
- Update the maximum --time to the expected runtime of your model. The job will be terminated if it exceeds this time limit.
- Set the --ntasks to the number of cores you want to use. Each node has 48 cores, so for example --ntasks=96 might use two nodes, but can also be distributed over more nodes.
- Update the memory --mem-per-cpu to the amount of memory you need per core. Each node has 192 GB of memory, so you can use up to 4 GB per core.
- Add --exclusive if you want to claim a full node for your job. In that case, specify --nodes instead of --ntasks. This will reduce overhead, but it will also delay you scheduling time, because you need to wait for a full node to become available.
- Set the --account to your project account. You can find this on the Accounting and Shares page of the DelftBlue docs.

See Submit Jobs at the DelftBlue docs for more information on the SLURM script configuration.

Then, you need to load the necessary modules. You can find the available modules on the DHPC modules page of the DelftBlue docs. In this example, we're loading the 2023r1 toolchain, which includes Python 3.9, and then we're loading the necessary Python packages.

You might want to install additional Python packages. You can do this with pip install -U --user <package>. Note that you need to use the --user flag, because you don't have root access on the cluster. To install the EMA Workbench, you can use pip install -U --user ema_workbench. If you want to install a development branch, you can use pip install -e -U --user git+https://github.com/quaquel/EMAworkbench@<BRANCH>#egg=ema-workbench, where <BRANCH> is the name of the branch you want to install.

Finally, the script uses mpiexec to run Python script in a way that allows the MPIEvaluator to distribute the experiments over the cluster.

Note that the bash scripts (sh), including the slurm_script.sh we just created, need LF line endings. If you are using Windows, line endings are CRLF by default, and you need to convert them to LF. You can do this with most text editors, like Notepad++ or Atom for example.

1. Setting up the environment

First, you need to log in on DelftBlue. As an employee, you can login from the command line with:

ssh <netid>@login.delftblue.tudelft.nl
where <netid> is your TU Delft netid. You can also use an SSH client such as PuTTY.

As a student, you need to jump though an extra hoop:

ssh -J <netid>@student-linux.tudelft.nl <netid>@login.delftblue.tudelft.nl
Note: Below are the commands for students. If you are an employee, you need to remove the -J <netid>@student-linux.tudelft.nl from all commands below.

Once you're logged in, you want to jump to your scratch directory (note it's not but is not backed up).

cd ../../scratch/<netid>
Create a new directory for this tutorial, for example mkdir ema_mpi_test and then cd ema_mpi_test

Then, you want to send your Python file and SLURM script to DelftBlue. Open a **new** command line terminal, and then you can do this with scp:

scp -J <netid>@student-linux.tudelft.nl ema_example_model.py slurm_script.sh <netid>@login.delftblue.tudelft.nl:/scratch/<netid>/ema_mpi_test
Before scheduling the SLURM script, we first have to make it executable:

chmod +x slurm_script.sh
Then we can schedule it:

sbatch slurm_script.sh
Now it's scheduled!

You can check the status of your job with squeue:

squeue -u <netid>
You might want to inspect the log file, which is created by the SLURM script. You can do this with cat:

cat slurm-<jobid>.out
where <jobid> is the job ID of your job, which you can find with squeue.

When the job is finished, we can download the pickle file created. Open the command line again (can be the same one as before), and you can copy the results back to your local machine with scp:

```
scp -J <netid>@student-linux.tudelft.nl
<netid>@login.delftblue.tudelft.nl:/scratch/<netid>/ema_mpi_test/ema_mpi_test.pickle .
```
Finally, we can clean up the files on DelftBlue, to avoid cluttering the scratch directory:

```
cd ..
 rm -rf "ema_mpi_test"
```