PHIMECA
Solutions for robust engineering

Creating a distributed python wrapper with otwrapy

HPC and Uncertainty Treatment

PRACE Advanced Training Center, May 27-29 2019
EDF – Phimeca – Airbus Group – IMACS – CEA

Gaëtan Blondet, Phimeca Engineering

PRACE

MAISON DE LA SIMULATION

afaq ISO 9001 Qualité AFNOR CERTIFICATION

Outline

Introduction

- What is a wrapper?
  - a piece of python code that creates an interface with your external code.
  - able to take advantage of multi-core computers and HPC clusters in order to distribute multiple evaluations of your code.
- Presentation goal: Show you how to create a distributed wrapper to efficiently carry on uncertainty studies.
- Based on the module otwrapy available at GitHub. Initially developed at Phimeca engineering.
- A good working example can be found on the otwrapy repository example.

What makes a good wrapper ?

- It is distributed and avoids conflict between runs.
- You can use it as a script (argsparse module):

```
>> python wrapper.py -X 170 3 0.05
```

- It is able to run on different environments:
  - Workstation
  - Office made heterogeneous clusters –> e.g. IPyparallel or dispy
  - HPC through submission scripts –> e.g. TGCC or Poincare
  - Cloud solutions –> e.g. Simulagora or DominoUp
- It catches and logs errors for easy debugging.
- It can either run or simply prepare runs –> useful when running on clusters.

All of this might seem complex, but wrappers are repetitive and otwrapy is here for you !

Basic skeleton of a wrapper

- Assumption: you want to wrap an external code not written in Python.
- An OpenTURNS wrapper is a subclass of `ot.OpenTURNSPythonFunction` for which at least the method `_exec(X)` should be overloaded. Additionally, you can overload `_exec_sample(X)`, but with otwrapy.Parallelizer() there is no need to.
- If your code handles the gradient and the hessian, you can respectively overload `_gradient(X)` and `_hessian(X)`.

```
class Wrapper(ot.OpenTURNSPythonFunction):
    """Wrapper of my external code.
    """
    def __init__(self):
        """Initialize the wrapper with 4 and 1 as input and output dimension.
        """
        super(Wrapper, self).__init__(4, 1)
        # Do other stuff if necessary
    def _exec(self, X):
        """Run the model in the shell for the input vector X
        """
        pass
```

Overloading the exec function

- _exec is the default OpenTURNS method that executes the function on a given point.
- Semantically speaking, the function is divided on three parts :
  1. Prepare the input parameters, e.g., create an input file.
  2. Run the external code, e.g., on the shell.
  3. Get the output parameters by parsing the output given by the code.
- The three steps are executed on a temporary working directory using otwrapy.TempWorkDir

```
def _exec(self, X):
    """Run the model in the shell for the input vector X
    """

    # Move to temp work dir. Cleanup at the end
    with otw.TempWorkDir(cleanup=True):
        # Prepare the input
        self._prepare_input(X)
        # Run the external code
        self._run_code(X)
        # Parse the output parameters
        Y = self._parse_output()

    return Y
```

Temporary working directory

- Efficiently and safely work on temporary directories with otwrapy.TempWorkDir
  - Avoid conflict between simulations running in parallel.
  - If an exception is raised during execution, the Python interpreter should come back to the preceding current directories.
  - Cleanup the temporary working directories upon exit. Or don't if you want a full backup of the simulations.
  - Transfer necessary files needed by the external code
- Example:

```
import otwrapy as otw
# I'm on a given dir, e.g. ~/beam-wrapper
with otw.TempWorkDir(base_temp_work_dir='/tmp', prefix='run-', cleanup=True, transfer=None):
    """
    ...
    Do stuff safely on an exclusive temporary directory and erase it afterwards
    ...
    """
    # The current working directory is something like /tmp/run-pZYpzQ

# Back on ~/beam-wrapper and /tmp/run-pZYpzQ does not exist anymore
```

Prepare the input parameters

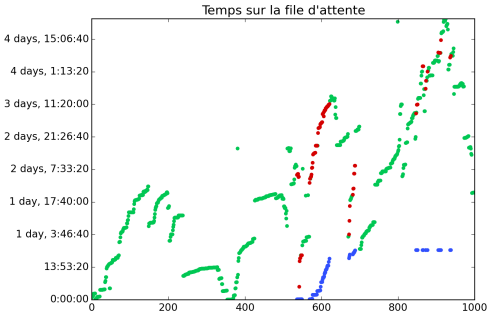
- For each simulation, your wrapper must communicate the input parameters to the external code.
- Most scientific codes use input files that describe, among other thing, the parameters of your model/simulation.
- Using OpenTURNS coupling tools, the values of the vector X are placed on an input template file that have tokens/placeholders for where the expected parameters should be.

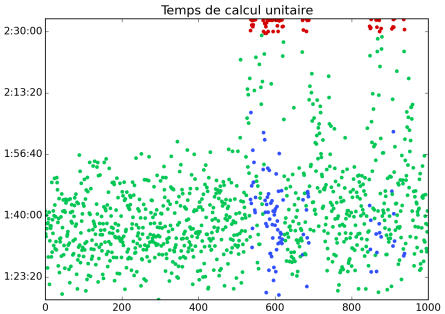
```
def _prepare_input(self, X):
    """Create the input file required by the code.
    """
    ot.coupling_tools.replace(
        infile='input_templatefile.xml',
        outfile='input.xml',
        tokens=['@X1','@X2','@X3','@X4'],
        values=X)
```

Run the external code

- Most of the time this is a fairly straightforward call to an executable with an input file.
- Sometimes, it is useful to time your runtime.

```
def _run_code(self):
    time_start = time.time()
    ot.coupling_tools.execute('/path/to/executable -x input.xml'))
    return time.time() - time_start
```





Parse output parameters 1/2

- Common practice among scientific code is to create output files with the results of the simulation.
- The output should then be parsed in order to get the output parameters of interest.
- If it is a `.csv` file
  - pandas.read_csv is the fastest option, but it introduces pandas as a dependency.
  - if speed is not an issue, try ot.coupling_tools.get_value,
  - or numpy.loadttxt.

Parse output parameters 2/2

- For `.xml` files minidom package from the python standard library does the trick.
- If by any chance the external code returns the output parameters of interest to STDOUT, set `get_stdout=True` when calling `ot.coupling_tools.execute(...)`. (or use subprocess.check_output)
- For standard binary formats, there are python interfaces to netcdf and HDF5.
- Otherwise, be creative and pythonic !

```
def _parse_output(self):
    # Retrieve output (see also )
    xmldoc = minidom.parse('outputs.xml')
    itemlist = xmldoc.getElementsByTagName('outputs')
    Y = float(itemlist[0].attributes['Y1'].value)

    return [Y]
```

Managing data backups

- Uncertainty studies tend to be expensive in computational time, it is then in your best interest to backup your simulation results !
- otwrapy has two useful functions to do so: otwrapy.dump_array and otwrapy.load_array
- They are faster than a simple pickle.dump and pickle.load (because they use protocol=2)
- They offer the possibility to compress data with the gzip library. If the extension is 'pklz', it compresses by default.
- Advice: Convert your ot.Sample to a np.array before dumping. An np.array is lighter !
- Example: dump and compress

```
import otwrapy as otw
otw.dump_array(np.array(X), 'InputSample.pklz', compress=True)
```

- ... and load

```
import otwrapy as otw
import openturns as ot
X = otw.load_array('InputSample.pklz')
X = ot.Sample(X)
```

Catch exceptions when your code fails

- In order to catch exceptions use the decorator otwrapy.Debug() !
- It encloses what happens inside a function into a try/catch structure and logs Exceptions when they are raised.
- Useful when your wrapper is not used on an interactive environment like IPython or a Jupyter notebook.

```
import otwrapy as otw
class Wrapper(ot.OpenTURNSPythonFunction):
    @otw.Debug('wrapper.log')
    def _exec(self, X):
        #Do stuff
        return Y
```

Creating a CLI for your wrapper

- A command line interface allows you to run your wrapper in detached mode, e.g., through submission scripts on HPC clusters.
- The argparse library might seem complicated, but they have a great cookbook and there are good chances that a simple copy/paste will be enough.
- Take a look at the beam wrapper for an example of a CLI interface

```
if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser(description="Python wrapper example.")
    parser.add_argument('-X', nargs=3, metavar=('X1', 'X2', 'X3'),
        help='Vector on which the model will be evaluated')
    args = parser.parse_args()

    model = Wrapper(3, 1)
    X = ot.NumericalPoint([float(x) for x in args.X])
    Y = model(X)
    dump_array(X, 'InputSample.pkl')
    dump_array(Y, 'OutputSample.pkl')
```

- You can then execute your code from the command line :

```
python wrapper.py -X 170 3 0.05
```

Parallelizing the wrapper

- Uncertainty studies fall into what we call *embarrassingly parallel* (or *pleasantly parallel*) patterns –> Repeat similar non communicating tasks over and over.
- Good news, this means that they are very simple to parallelize.
- But don't bother... just let the magic happen with otwrapy.Parallelizer() !!

```
import otwrapy as otw
from otwrapy.examples.beam import Wrapper
parallelized_beam_wrapper = otw.Parallelizer(Wrapper())
```

Distributing calls on clusters or the cloud

- But what if you want to distribute your wrapper calls on the cloud or on a cluster ?
- otw.Parallelizer is no longer the way to go, for the moment...
- You can manage to make an heterogeneous office cluster with IPyparallel or dispy
- For clusters and the cloud, rely on a good CLI interface of your wrapper and distribute your calls through submission scripts or cloud APIs (e.g., Simulagora or Domino)

Conclusion

- Take away message : Making a wrapper is all about preparing the input, executing the code and parsing the output on isolated working directories. Don't forget, in a multi-core era you don't have a choice, make your wrapper distributed !
- By creating a CLI of your wrapper, you can easily distribute your calls on a cluster or on cloud platforms.
- It is important to protect your wrapper with otw.Debug() so that you can have a traceback of raised Exceptions.
- ot.PythonFunction() is a simpler alternative to ot.OpenTURNSPythonFunction(), but you loose the ability to parameterize your wrapper when instantiating it.
- otwrapy is here for you ! Use it to avoid code boilerplate or as a simple cookbook.

Thank you for your attention

Gaëtan Blondet

blondet@phimeca.com

Github : otwrapy