ose-code-templates

OpenSourceEconomics

Aug 13, 2020

CONTENTS

CHAPTER

ONE

EMBARISSINGLY PARALLEL LOOP

1.1 Core functions

Core functions for template

`core_functions.`**`distribute_tasks`**(*func_task*, *tasks*, *num_proc=1*, *is_distributed=False*)

Distribute workload.

This function distributes the workload using the `multiprocessing` or `mpi4py` library. It simply creates a pool of processes that allow to work on the tasks using shared or distributed memory.

Notes

We need to ensure that the number of processes is never larger as the number of tasks as otherwise the MPI implementation does not terminate properly.

- MP Pool, see here for details
- MPI Pool, see here for details

1.2 Test integration

Integration tests.

This module contains the integration tests that all the individual units are combined and tested together.

`test_integration.`**`get_random_request`**()

Random test case.

This function sets up a random test case that differs depending on whether MPI capabilities are available or not.

`test_integration.`**`test_1`**()

Test a random request.

This test simply evaluates a random request. It automatically checks whether a distributed evaluation is an option.

`test_integration.`**`test_2`**()

Varying the number of processes.

This test evaluates the same request with different number of processes and ensures that the amount resources do not matter for the results.

`test_integration.test_3()`

Alternating between shared and distributed memory.

This test evaluates the same request using the `multiprocessing` and `mpi4py` library and ensures that both yield the same result.

We show how to parallelize a loop using the `multiprocessing` and `mpi4py`. The setup allows to seamlessly switch between shared and distributed memory computing.

CHAPTER

TWO

NUMBA PARALLEL

[1]:
```
import numpy as np
from numba import prange, njit, guvectorize
```

Lets first get some test resources. The names and the structure from the examples are taken from the calculation of the expected value function in respy. The original function can be found here.

[2]:
```
wages = np.ones((100, 4))
nonpecs = np.ones((100, 4))
continuation_values = np.ones((100, 4))
period_draws_emax_risk = np.ones((50, 4))
delta = 0.95
```

2.1 Parallelization of `@jit` functions

`numba` offers automatic parallelization of jit functions. This can either happen implicit on array operations or explicit with the keyword statement `parallel=True` and e.g. parralel loops with `prange`. The resources for this can be found here.

[3]:
```
@njit(parallel=True)
def parralel_loop(wages, nonpecs, continuation_values, draws, delta):
    num_states, n_ch = wages.shape
    n_draws, n_choices = draws.shape
    out = 0
    for k in prange(num_states):
        for i in prange(n_draws):
            for j in prange(n_choices):
                out += (
                    wages[k, j] * draws[i, j]
                    + nonpecs[k, j]
                    + delta * continuation_values[k, j]
                )

    return out
```

2.2 Diagnostics

When calling an explicit parallelized function, `numba` tries to create separate calculations to run multiple kernels or threads. The optimization behavior can be inspected by using `func.parallel_diagnostics(level=4)`.

The levels can vary from one to four. The resources to this can be found here.

[4]:
```
# An example of the two things above:
parralel_loop(
    wages, nonpecs, continuation_values, period_draws_emax_risk, delta
)
parralel_loop.parallel_diagnostics(level=4)
```

```
================================================================================
 Parallel Accelerator Optimizing:  Function parralel_loop, <ipython-
input-3-cf75dd448160> (1)
================================================================================


Parallel loop listing for  Function parralel_loop, <ipython-input-3-cf75dd448160> (1)
-----------------------------------------------------------------------|loop #ID
@njit(parallel=True)                                                   |
def parralel_loop(wages, nonpecs, continuation_values, draws, delta):  |
    num_states, n_ch = wages.shape                                     |
    n_draws, n_choices = draws.shape                                   |
    out = 0                                                            |
    for l in prange(num_states):---------------------------------------| #2
        for i in prange(n_draws):--------------------------------------| #1
            for j in prange(n_choices):--------------------------------| #0
                out += (                                               |
                    wages[l, j] * draws[i, j]                          |
                    + nonpecs[l, j]                                    |
                    + delta * continuation_values[l, j]                |
                )                                                      |
                                                                       |
    return out                                                         |
--------------------------------- Fusing loops ---------------------------------
Attempting fusion of parallel loops (combines loops with similar properties)...
------------------------------ Before Optimisation ------------------------------
Parallel region 0:
+--2 (parallel)
   +--1 (parallel)
      +--0 (parallel)



--------------------------------------------------------------------------------
------------------------------ After Optimisation ------------------------------
Parallel region 0:
+--2 (parallel)
   +--1 (serial)
      +--0 (serial)



Parallel region 0 (loop #2) had 0 loop(s) fused and 2 loop(s) serialized as part
 of the larger parallel loop (#2).
--------------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
--------------------------------------------------------------------------------

---------------------------Loop invariant code motion---------------------------
Allocation hoisting:
No allocation hoisting found

Instruction hoisting:
loop #2:
  Failed to hoist the following:
    dependency: out_4 = out.3
--------------------------------------------------------------------------------
```

2.3 Parallelization of `@guvectorize` functions

When using `@guvectorize`, you can define functions on multiple arrays, which then can be parallelized across the entries of the arrays with `target="parallel"`. Details to `@guvectorize` can be found here.

[5]:
```python
@guvectorize(
    ["f8[:], f8[:], f8[:], f8[:, :], f8, f8[:]"],
    "(n_choices), (n_choices), (n_choices), (n_draws, n_choices), () -> ()",
    nopython=True,
    target="parallel",
)
def calculate_expected_value_functions(
    wages, nonpecs, continuation_values, draws, delta, expected_value_functions
):
    n_draws, n_choices = draws.shape

    expected_value_functions[0] = 0

    for i in range(n_draws):

        max_value_functions = 0

        for j in range(n_choices):
            value_function = (
                wages[j] * draws[i, j]
                + nonpecs[j]
                + delta * continuation_values[j]
            )

            if value_function > max_value_functions:
                max_value_functions = value_function

        expected_value_functions[0] += max_value_functions

    expected_value_functions[0] /= n_draws
```

The statement `target="parallel"` does not explicitly state that the code inside the `@guvectorize` function is parallelized itself. However, one can rule out this possibility, if the function diagnosed with the tools described above does not offer any parallelization. Thus, to my knowledge, there is no explicit possibility to fix a parallelization structure. One can only design the code, such that the intended parallelization happens when the `@guvectorized` function is called.

We collect resources and demonstrate parallelization with `numba`. Our focus lies on the analysis of nested parallelism and the working example is inspired by `respy`.

CHAPTER

THREE

MPI MAIN-CHILD APPLICATION

We illustrate the concept of a main-child application using our research code `respy`. As a use case, we are interested in capturing the uncertainties in the model's predictions about average final schooling. For that purpose we start a main process that distributes sampled parameter values from the imposed distribution of the discount factor and the return to schooling.

We can start the script using the terminal.

```
mpiexec -n 1 -usize 5 python main.py
```

This starts the main process and allows to create up to five additional child processes.

```
import shutil
import glob
import sys
import os

if "PMI_SIZE" not in os.environ.keys():
    raise AssertionError("requires MPI access")
from mpi4py import MPI

import chaospy as cp
import numpy as np

from auxiliary import aggregate_results
from auxiliary import TAGS


if __name__ == "__main__":

    # We specify the number of draws and number of children.
    num_samples, num_children = 5, 2

    # We draw a sample from the joint distribution of the parameters that is␣
↪subsequently
    # distributed to the child processes.
    distribution = cp.J(cp.Uniform(0.92, 0.98), cp.Uniform(0.03, 0.08))
    samples = distribution.sample(num_samples, rule="random").T

    info = MPI.Info.Create()
    info.update({"wdir": os.getcwd()})

    # We start all child processes and make sure they can work in their own␣
↪respective directory.
    [shutil.rmtree(dirname) for dirname in glob.glob("subdir_child_*")]
```

(continues on next page)

(continued from previous page)

```
    file_ = os.path.dirname(os.path.realpath(__file__)) + "/child.py"
    comm = MPI.COMM_SELF.Spawn(
        sys.executable, args=[file_], maxprocs=num_children, info=info
    )

    # We send all problem-specific information once and for all.
    prob_info = dict()
    prob_info["num_params"] = samples.shape[1]
    comm.bcast(prob_info, root=MPI.ROOT)

    status = MPI.Status()
    for sample in samples:

        comm.recv(status=status)
        rank_sender = status.Get_source()

        comm.send(None, tag=TAGS.RUN, dest=rank_sender)

        sample = np.array(sample, dtype="float64")
        comm.Send([sample, MPI.DOUBLE], dest=rank_sender)

    # We are done and now terminate all child processes properly and finally the turn␣
↪off the
    # communicator. We need for all to acknowledge the receipt to make sure we do not␣
↪continue here
    # before all tasks are not only started but actually finished.
    [comm.send(None, tag=TAGS.EXIT, dest=rank) for rank in range(num_children)]
    [comm.recv() for rank in range(num_children)]
    comm.Disconnect()

    rslt = aggregate_results()
```

The behavior of the child processes is governed in the following script.

```
#!/usr/bin/env python
"""This script provides all capabilities for the child processes."""

import os

# In this script we only have explicit use of MPI as our level of parallelism. This␣
↪needs to be
# done right at the beginning of the script.
update = {
    "NUMBA_NUM_THREADS": "1",
    "OMP_NUM_THREADS": "1",
    "OPENBLAS_NUM_THREADS": "1",
    "NUMEXPR_NUM_THREADS": "1",
    "MKL_NUM_THREADS": "1",
}
os.environ.update(update)

from mpi4py import MPI
import pandas as pd
import numpy as np
import respy as rp

from auxiliary import TAGS
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    comm = MPI.Comm.Get_parent()
    num_slaves, rank = comm.Get_size(), comm.Get_rank()
    status = MPI.Status()

    # We need some additional task-specific information.
    prob_info = comm.bcast(None)

    subdir = f"subdir_child_{rank}"
    os.mkdir(subdir)
    os.chdir(subdir)

    # We now set up the simulation function of  `respy` and receive some task-
↪specific information.
    params, options, df = rp.get_example_model("kw_94_one")
    simulate = rp.get_simulate_func(params, options)

    rslt = list()
    while True:

        # Signal readiness
        comm.send(None, dest=0)

        # Receive instructions and act accordingly.
        comm.recv(status=status)
        tag = status.Get_tag()

        if tag == TAGS.EXIT:
            # We set up a container to store the results.
            df = pd.DataFrame(rslt, columns=["qoi", "delta", "exp_edu"])
            df.index.name = "sample"
            df.to_pickle(f"rslt_child_{rank}.pkl")

            # Now we are ready to acknowledge completion and disconnect.
            comm.send(None, dest=0)
            comm.Disconnect()
            break

        elif tag == TAGS.RUN:
            # We are called to sample the quantity of interest and need to update the␣
↪parameters
            # accordingly.
            sample = np.empty(prob_info["num_params"], dtype="float64")
            comm.Recv([sample, MPI.DOUBLE])

            (
                params.loc["delta", "value"],
                params.loc[("wage_a", "exp_edu"), "value"],
            ) = sample

            stat = simulate(params).groupby("Identifier")["Experience_Edu"].max().
↪mean()
            rslt.append([stat, *sample])

        else:
```

(continues on next page)

(continued from previous page)

```
            raise AssertionError
```

We show how to set up a main-child application. We use the example of uncertainty propagation using `respy` as the motivating use-case.

CHAPTER

FOUR

POWERED BY

PYTHON MODULE INDEX

INDEX