ndsharray Documentation

Release 1.1.1

Rune Monzel

Aug 14, 2023

CONTENTS:

CHAPTER

ONE

OVERVIEW

ndarray + sharing = ndsharray

1.1 Introduction

This python module let you share a numpy ndarray within different processes (either via python's multiprocessing or sharing between different python instances). The library behind this package is the lib mmap from official python - no extra library, except numpy, is needed. The mmap approach is using the shared memory, which can be accessed by different CPUs/python instances. Using shared memory is much faster than the pickle approach - you can even do a video streaming on a Raspberry Pi / Jetson Nano over multiple python processes.This library is easy-to-use, just initialize the shared array with a unique tag and write/read! You can even change the numpy array dimension/shape/dtype during runtime - the mmap will be silently rebuild if there is a change in the numpy array size/shape/dtype.

Small Example Code:

```
import numpy as np
from ndsharray import NdShArray

shared_array = NdShArray("my_unique_tag", r_w='w')  # r_w must be specified

my_array = np.random.random((400, 800))
shared_array.write(my_array)

print(my_array)
```

That's all for writing into shared memory. How to read? Open a second python instance:

```
import numpy as np
from ndsharray import NdShArray

shared_array = NdShArray("my_unique_tag", r_w='r')  # r_w must be specified

status, my_array = shared_array.read()

print(my_array)
```

1.2 Documentation

Can be found here: https://ndsharray.readthedocs.io/en/latest/

1.3 Requirements

- Python 3.6
- numpy

Tested with example codes on

- Windows 10, amd64, Python 3.6 / 3.8
- Ubuntu 20, amd64, Python 3.6 /3.8
- NVIDIA Jetson Nano with Ubuntu 18.04, ARM64-bit (aarch64), Python 3.6

1.4 Some technical notes

This library shall be an easy-to-use library and also shall be faster than pickling numpy arrays to another process. Please note that the python's provided shared_memory does the same as ndsharray, but is using byte array instead of numpy array! However, shared_memory is available since python 3.8 and not supported for python 3.6.

1.5 Installation from Github

Make sure to have git, python and pip in your environment path or activate your python environment.To install enter in cmd/shell:

```
git clone https://github.com/monzelr/ndsharray.git

cd ndsharray

pip install .
```

Alternative with python:

```
python setup.py install
```

CHAPTER

TWO

INSTALLATION

2.1 Via pip

This module can be found on pipy: https://pypi.org/project/ndsharray/

For installation, first activate your environment, and then install ndsharray:

```
pip install ndsharray
```

To uninstall the python package, type in this command:

```
pip uninstall ndsharray
```

2.2 From sources

The sources for ndsharray can be found in the github github repository.

Clone the repository:

```
git clone git://gitlab.com/monzelr/ndsharray
```

Best practice is just to link the source to to your python environment via the develop command:

```
cd ndsharray
python setup.py develop
```

Of course, you can also install the package with python normally:

```
$ python setup.py install
```

To uninstall the python package, type in this command:

```
pip uninstall ndsharray
```

2.3 Distribution

If you want to distribute the package, please build a python wheel which can be distributed:

```
python setup.py bdist_wheel
```

The wheel can be installed with the pip command.

CHAPTER

THREE

USAGE

3.1 Basics

NdShArray shall be used in different python processes. You can open another python process with python's subprocess module or with python's multiprocessing module.

NdShArray uses shared memory, thus different CPUs can access the same numpy array. Note that NdShArray does a memory view - exactly it uses the numpy function `frombuffer`, while using the `read` function. However, the `write` function does a copy into the shared memory.

To understand the libray open two different python console, and copy line for line the following code.

3.1.1 first console

In the first python console we create a random numpy array and create a unique identifier tag, which is important for the reading ndsharray.

```
import numpy as np
import ndsharray

my_array = np.random.random((4, 2))  # array size can be changed during runtime

# array must not be specified at instantiation
shared_array = ndsharray.NdShArray("my_unique_tag123", array=my_array, r_w='w')

print("name of the ndsharray: %s" % shared_array.name)  # prints the unique identifier␣
↪(uuid4)
print("access of the ndsharray: %s" % shared_array.access)  # prints access: 'r' or 'w'
print("current ndarray name: %s" % shared_array.ndarray_mmap_name)  # prints the current␣
↪mmap name of the ndarray
print(my_array)
```

3.1.2 second console

Open the second python console; and copy the following code:

```
import numpy as np
import ndsharray

_shared_array = ndsharray.NdShArray("my_unique_tag123", r_w='r')

status, my_array = _shared_array.read()

print("status: %s" % status)  # check if the read is valid with the status!
# check the read time: the elapsed time between write- and read-function
print("elapsed write-read time: %s ms" % _shared_array.read_time_ms)
print(my_array)
```

3.1.3 first console again

Ok, we just saw a successful transfer of the numpy array into the second process. But what if the numpy array size change in dimension, dtype or shape? The class NdShArray carries about this and creates silently a new shared memory, this can be seen by the property `ndarray_mmap_name`.

Note: The silently re-creation of a shared memory takes its time, so if you have 3 different-shaped numpy arrays which just updates its values over time, it is the best way to create 3 NdShArrays!

Go to the first console and enter the following code snippet:

```
my_int_array = (255 * np.random.random((6, 3))).astype(np.uint8)
shared_array.write(my_int_array)

# the name of the ndarray mmap have been changed now - it is using a different uuid4␣
↪identifier:
print("current ndarray name: %s" % shared_array.ndarray_mmap_name)
print(my_int_array)
```

3.1.4 second console again

Let's go to the second console and check the numpy array:

```
status, my_array_2 = _shared_array.read()

print("status: %s" % status)  # check if the read is valid with the status!
# check the read time: the elapsed time between write- and read-function
print("elapsed write-read time: %s ms" % _shared_array.read_time_ms)
print("current ndarray name: %s" % _shared_array.ndarray_mmap_name)
print(my_array_2)
```

3.1.5 Supported numpy types

To check the supported numpy types just take a look into `ndsharray.supported_types`:

```
import ndsharray

print("supported numpy types:")
for _dtype in ndsharray.supported_types:
    print(_dtype)
```

As you can see, currently not all numpy dtypes are supported (e. g. `bytes`, `str` and `object` data types are missing).

3.1.6 More Example Code

More examples can be found on the github project in the example folder: https://github.com/monzelr/ndsharray/tree/main/examples

CHAPTER

FOUR

API

Top-level package

ndsharray.**supported_types**

alias of [<class 'numpy.int8'>, <class 'numpy.int16'>, <class 'numpy.int32'>, <class 'numpy.int64'>, <class 'numpy.uint8'>, <class 'numpy.uint16'>, <class 'numpy.uint32'>, <class 'numpy.uint64'>, <class 'numpy.float16'>, <class 'numpy.float32'>, <class 'numpy.float64'>, <class 'numpy.float128'>, <class 'numpy.complex64'>, <class 'numpy.complex128'>, <class 'numpy.complex256'>, <class 'bool'>]

class ndsharray.**NdShArray**(*name: str*, *array: numpy.ndarray = array([], dtype=uint8)*, *r_w: Optional[str] = None*)

sharing numpy array between different processes

__init__(*name: str*, *array: numpy.ndarray = array([], dtype=uint8)*, *r_w: Optional[str] = None*)

Parameters

- **name** –
- **array** –
- **r_w** – 'r' or 'w' for 'read' or 'write' functionality, must be specified

property access: str

access of the ndsharray; either 'w' for only writeable or 'r' for only readable

Return access

property name: str

unique name of the mmap memory, serves as identifier for other processes the name must be declared at class instantiation and is read only after instantiation

Return name

property ndarray_mmap_name: str

returns the name of the mmap which holds the current ndarray

ndarray_mmap_name consists of the name and an uuid which is generated for each new ndarray size (changes in dtype, shape or dimension does a change in size)

Return ndarray_mmap_name

read() → Tuple[bool, numpy.ndarray]

reading the shared memory with mmap and numpy's frombuffer, which returns a view of the buffer and not a copy.

Citing the documentation from numpy.frombuffer: "This function creates a view into the original object. This should be safe in general, but it may make sense to copy the result when the original object is mutable or untrusted." Source: https://numpy.org/doc/stable/reference/generated/numpy.frombuffer.html

Return validity boolean displaying if the numpy array is ok or if it is either old or corrupt or not (e.g. mixed numpy ndarray from previous writing). Note: validity is checked by checking if buffer[0] and buffer[-1] have the same time stamp!

Return array numpy.ndarray

property read_time_ms: float

returns the write-read time of the two processes in milliseconds

Returns

write(*array: numpy.ndarray*) → None

write a numpy array into the mmap file, it might be from any type, shape or dimension

Important Note: a mmap will be silently re-created if type, dimension or shape will be changed. the other process will read the first line of the mmap and will also re-create its mmap. Re-creating the mmap needs more time than a normal read.

Parameters **array** – a numpy.ndarray which shall be saved in mmap

Return None

CHAPTER

FIVE

HISTORY

5.1 Version 1.0.0

- initial release for PyPi

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

A

M

N

R

S

W