multitables Documentation

Release 2.0.0

G. H. Collin

May 01, 2021

Contents

multitables is a python library designed for high speed access to HDF5 files. Access to HDF5 is provided by the PyTables library (`tables`). Multiple processes are launched to read a HDF5 in parallel, allowing concurrent decompression. Data is streamed back to the invoker by use of shared memory space, removing the usual multiprocessing communication overhead.

The data is organised by rows of an array (elements of the outer-most dimension), and groups of these rows form blocks. By default, there is **no guarantee** on the ordering of the rows and/or blocks returned to the user, due to the concurrent nature of the library. They are returned as they become available. On-disk ordering can be forced using the `ordered` option, which may result in a performance penalty.

Performance gains of at least 2x can be achieved when reading from an SSD.

CHAPTER 1

Contents

1.1 Quick Start

1.1.1 Installation

```
pip install multitables
```

Alternatively, to install from HEAD, run

```
pip install git+https://github.com/ghcollin/multitables.git
```

You can also download or clone the repository and run

```
python setup.py install
```

`multitables` depends on `tables` (the pytables package), `numpy`, `msgpack`, and `wrapt`. The package is compatible with the latest versions of Python 3, as pytables no longer supports Python 2.

1.1.2 Quick start: Streaming

```
import multitables
stream = multitables.Streamer(filename='/path/to/h5/file')
for row in stream.get_generator(path='/internal/h5/path'):
    do_something(row)
```

1.1.3 Quick start: Random access

```
import multitables
reader = multitables.Reader(filename='/path/to/h5/file')

dataset = reader.get_dataset(path='/internal/h5/path')
stage = dataset.create_stage(10) # Size of the shared
                                 # memory stage in rows

req = dataset['col_A'][30:35] # Create a request as you
                              # would index normally.

future = reader.request(req, stage) # Schedule the request
with future.get_unsafe() as data:
    do_something(data)
data = None # Always set data to None after get_unsafe to
            # prevent a dangling reference

# ... or use a safer proxy method

req = dataset.col('col_A')[30:35,...,:100]

future = reader.request(req, stage)
with future.get_proxy() as data:
    do_something(data)

# ... or provide a function to run on the data

req = dataset.read_sorted('col_C', checkCSI=True, start=200, stop=300)

future = reader.request(req, stage)
future.get_direct(do_something)

# ... or get a copy of the data

req = dataset['col_A'][30:35,np.arange(500) > 45]

future = reader.request(req, stage)
do_something(future.get())

# once done, close the reader
reader.close(wait=True)
```

1.1.4 Examples

See the *How-To* for more in-depth documentation, and the unit tests for complete examples.

1.2 How To

All uses of the library start with creating a `Streamer` object, or a `Reader` object.

1.3 Streamer

The `Streamer` is designed for reading data from the dataset in approximately (or optionally forced) sequential order.

```
import multitables
stream = multitables.Streamer(filename="/path/to/h5/file", **kw_args)
```

Additional flags to pytables' `open_file` function can be passed through the optional keyword arguments.

1.3.1 Direct access

multitables allows low level access to the internal queue buffer. This access is synchronised with a guard object. When the guard object is created, an element of the buffer is reserved until the guard is released.

```
queue = stream.get_queue(
    path='/h5/path/', # Path to dataset within the H5file.
    n_procs=4,        # Number of processes to launch for parallel reads. Defaults to␣
↪4.
    read_ahead=5,     # Size of internal buffer in no. of blocks. Defaults to 2*n_
↪proc+1.
    cyclic=False,     # A cyclic reader wraps at the end of the dataset. Defaults to␣
↪False.
    block_size=32,    # Size (along the outer dimension) of the blocks that will be␣
↪read.
                      # Defaults to a multiple of the dataset chunk size, or a 128KB␣
↪block.
                      # Should be left to the default or carefully chosen for chunked␣
↪arrays,
                      # else performance degradation can occur.
    ordered=False     # Force the stream to return blocks in on-disk order. Useful if␣
↪two
                      # datasets need to be read synchronously. This option may have a
                      # performance penalty.
)

while True:
    guard = queue.get() # Get the guard object, will block until data is ready.
    if guard is multitables.QueueClosed:
        break # Terminate the loop once the dataset is finished.
    with guard as block: # The guard returns the next block of data in the buffer.
        do_something(block) # Perform actions on the data
```

Note that `block` here is a numpy reference to the internal buffer. Once the guard is released, `block` is no longer guaranteed to point to valid data. If the data need to be saved for later use, make a copy of it with `block.copy()`.

Iterator

A convenience iterator is supplied to make loop termination easier.

```
for guard in queue.iter():
    with guard as block:
        do_something(block)
```

Remainder elements

In all the previous cases, if the supplied `read_size` does not evenly divide the dataset then the *remainder* elements will not be read. If needed, these remainder elements can be accessed using the following method

```
last_block = stream.get_remainder(path, queue.block_size)
```

Cyclic access

When the cyclic mode is enabled, the readers will wrap around the end of the dataset. The check for the end of the queue is no longer needed in this case.

```
while True:
    with queue.get() as block:
        do_something(block)
```

In cyclic access mode, the remainder elements are returned as part of a wrapped block that includes elements from the end and beginning of the dataset.

Once finished, the background processes can be stopped with `queue.close()`.

1.3.2 Generator

The generator provides higher level access to the streamed data. Elements from the dataset are returned one row at a time. These rows belong to a copied array, so they can be safely stored for later use. The remainder elements are also included in this mode.

```
gen = stream.get_generator(path, n_procs, read_ahead, cyclic, block_size)

for row in gen:
    do_something_else(row)
```

This is supposed to be in analogy to

```
dataset = h5_file.get_node(path)

for row in dataset:
    do_something_else(row)
```

When cyclic mode is enabled, the generator has no end and will continue until the loop is manually broken.

1.3.3 Concurrent access

Python iterators and generators are not thread safe. The low level direct access interface is thread safe.

1.4 Reader

The `Reader` is designed for random access, using an interface that is as close as possible to *numpy* indexing operations.

```
import multitables
reader = multitables.Reader(filename="/path/to/h5/file", **kw_args)
```

Additional flags to pytables' `open_file` function can be passed through the optional keyword arguments.

1.4.1 Dataset and stage

The basic workflow is to first open the desired dataset using the internal HDF5 path.

```
dataset = reader.get_dataset(path='/internal/h5/path')
```

Then, a stage must be created to host random access requests. This stage is an area of shared memory that is allocated and shared with the background reader processes. The result of all requests made with this stage must fit inside the allocated memory of the stage.

```
stage = dataset.create_stage(shape=10)
```

The provided `shape` parameter may be the full shape of the stage using the datatype of the dataset. Or, the shape may be left incomplete and the missing shape dimensions will be filled with the dataset shape. In this example, only the first dimension is specified, as so this stage has room for 10 rows of the dataset.

1.4.2 Requests

Requests happen through three operations. First, the description of a request is made through an indexing operation on the dataset.

```
req = dataset['col_A'][30:35]
```

Next, a future is made and a background task scheduled to fetch the requested data and load it into the provided stage.

```
future = reader.request(req, stage)
```

Finally, the future is waited upon using a get operation. Four types of get operations are provided. The first and simplest blocks on the task and returns a copy of the data.

```
data = request.get()
```

In the next type, a copy is avoided by providing a function that will be run with the data as the only argument. This get operation also blocks until the data is available and the provided function finishes.

```
def do_something(data):
    pass
data = request.get_direct(do_something)
```

The remaining two get operations use context managers to control access to the shared memory resource without creating a copy. The first is unsafe, in that if the resulting reference is not properly disposed of, memory errors may result.

```
with future.get_unsafe() as data:
    do_something(data)
data = None # Properly dispose of the reference
```

The final uses a wrapper object on the returned data, so that if the reference is not properly disposed of, an exception will be safely called.

```
with future.get_unsafe() as data:
    do_something(data)
data = None # Properly dispose of the reference
```

1.4.3 Cleaning up

Once finished, call the `close` method on the reader object.

```
reader.close(wait=True)
```

If the provided `wait` parameter is `True`, the `close` call will block until all background threads and processes have cleanly shut down.

1.4.4 Concurrent access pattern

The following is an example of launching and reading requests in separate threads. This uses the `create_stage_pool` method, that creates `N_stages` separate stages and places them in a rotating pool.

The `RequestPool` object is then used to create a queue of pending futures that returns futures in the same order that they are inserted.

```
N_stages = 10

stage_pool = dataset.create_stage_pool(1, N_stages)

reqs = multitables.RequestPool()

table_len = dataset.shape[0]
def loader():
    for idx in range(table_len):
        reqs.add(reader.request(dataset[idx:idx+1], stage_pool))

loader_thread = threading.Thread(target=loader)
loader_thread.start()

for idx in range(table_len):
    do_something(reqs.next().get())

reader.close(wait=True)
```

1.5 Benchmarking

These benchmarks have been performed with multitables_benchmark.py. Two compression methods are benchmarked, along with three different storage devices.

Note that the data used are random numbers from a normal distribution, which is not compressible. Thus, the numbers here reflect **only** the decompression overhead, and not the performance benefit that compression can give. They are intended to give a rough idea on the number of default processes to use, as well as the possible performance benefit of using this library.

Your mileage may vary, with factors including the workload, compression ratio, specific storage configuration, system memory, and dataset size. If your dataset fits wholly within the filesystem cache, your reads speeds will be significantly higher.

The following benchmarks use a 4GB file, reading two cycles, on a Haswell-E linux machine.

1.5.1 Using blosc

NVMe SSD

Direct queue

Table 1: pytables complevel (down) vs. number of parallel processes (across)

	1	2	3	4	5	6	7	8	9	10	11	12
0	954 MB/s	1694 MB/s	2061 MB/s	2380 MB/s	2399 MB/s	2400 MB/s	2252 MB/s	2346 MB/s	2221 MB/s	2377 MB/s	2252 MB/s	2260 MB/s
3	952 MB/s	1729 MB/s	2087 MB/s	2368 MB/s	2379 MB/s	2372 MB/s	2285 MB/s	2143 MB/s	2368 MB/s	2369 MB/s	2286 MB/s	2237 MB/s
9	394 MB/s	777 MB/s	1143 MB/s	1509 MB/s	1748 MB/s	2107 MB/s	2088 MB/s	2217 MB/s	2221 MB/s	2323 MB/s	2339 MB/s	2314 MB/s

Generator

Table 2: pytables complevel (down) vs. number of parallel processes (across)

	1	2	3	4	5	6	7	8	9	10	11	12
0	841 MB/s	1466 MB/s	1691 MB/s	1765 MB/s	1762 MB/s	1765 MB/s	1741 MB/s	1755 MB/s	1698 MB/s	1886 MB/s	1654 MB/s	1705 MB/s
3	834 MB/s	1430 MB/s	1668 MB/s	1693 MB/s	1712 MB/s	1659 MB/s	1669 MB/s	1618 MB/s	1647 MB/s	1644 MB/s	1572 MB/s	1559 MB/s
9	369 MB/s	737 MB/s	1073 MB/s	1393 MB/s	1573 MB/s	1578 MB/s	1669 MB/s	1587 MB/s	1634 MB/s	1524 MB/s	1478 MB/s	1588 MB/s

2x SATA III SSD in raid1

Direct queue

Table 3: pytables complevel (down) vs. number of parallel processes (across) :header-rows: 1

	1	2	3	4	5	6	7	8	9	10	11	12
0	358 MB/s	706 MB/s	821 MB/s	914 MB/s	960 MB/s	984 MB/s	996 MB/s	1009 MB/s	1012 MB/s	1027 MB/s	1026 MB/s	1013 MB/s
3	355 MB/s	686 MB/s	810 MB/s	915 MB/s	953 MB/s	985 MB/s	996 MB/s	1006 MB/s	1011 MB/s	1023 MB/s	1023 MB/s	1032 MB/s
9	237 MB/s	477 MB/s	687 MB/s	847 MB/s	907 MB/s	957 MB/s	988 MB/s	1012 MB/s	1033 MB/s	1048 MB/s	1056 MB/s	1062 MB/s

Generator

Table 4: pytables complevel (down) vs. number of parallel processes (across) :header-rows: 1

Com-plevel/n_proc	1	2	3	4	5	6	7	8	9	10	11	12
0	338 MB/s	661 MB/s	797 MB/s	906 MB/s	941 MB/s	974 MB/s	980 MB/s	970 MB/s	999 MB/s	998 MB/s	1001 MB/s	1003 MB/s
3	338 MB/s	657 MB/s	796 MB/s	889 MB/s	938 MB/s	952 MB/s	977 MB/s	962 MB/s	981 MB/s	989 MB/s	982 MB/s	976 MB/s
9	239 MB/s	473 MB/s	677 MB/s	822 MB/s	898 MB/s	942 MB/s	968 MB/s	985 MB/s	994 MB/s	995 MB/s	1004 MB/s	1002 MB/s

SATA III 7200 RPM HDD

Direct queue

Table 5: pytables complevel (down) vs. number of parallel processes (across)

	1	2	3	4	5	6
0	97 MB/s	97 MB/s	69 MB/s	67 MB/s	68 MB/s	62 MB/s
3	103 MB/s	94 MB/s	69 MB/s	66 MB/s	68 MB/s	62 MB/s
9	101 MB/s	92 MB/s	65 MB/s	66 MB/s	70 MB/s	63 MB/s

Generator

Table 6: pytables complevel (down) vs. number of parallel processes (across)

	1	2	3	4	5	6
0	119 MB/s	94 MB/s	72 MB/s	69 MB/s	68 MB/s	62 MB/s
3	121 MB/s	94 MB/s	69 MB/s	70 MB/s	67 MB/s	62 MB/s
9	119 MB/s	92 MB/s	60 MB/s	66 MB/s	70 MB/s	63 MB/s

1.5.2 Using zlib

NVMe SSD

Direct queue

Table 7: pytables complevel (down) vs. number of parallel processes (across) :header-rows: 1

	1	2	3	4	5	6	7	8	9	10	11	12
0	958 MB/s	1773 MB/s	2132 MB/s	2202 MB/s	2393 MB/s	2249 MB/s	2345 MB/s	2243 MB/s	2373 MB/s	2262 MB/s	2290 MB/s	2171 MB/s
3	301 MB/s	597 MB/s	902 MB/s	1208 MB/s	1497 MB/s	1766 MB/s	1869 MB/s	2073 MB/s	2116 MB/s	2302 MB/s	2149 MB/s	2390 MB/s
9	269 MB/s	524 MB/s	787 MB/s	1047 MB/s	1234 MB/s	1499 MB/s	1621 MB/s	1647 MB/s	1684 MB/s	1934 MB/s	2021 MB/s	1934 MB/s

Generator

Table 8: pytables complevel (down) vs. number of parallel processes (across) :header-rows: 1

	1	2	3	4	5	6	7	8	9	10	11	12
0	830 MB/s	1444 MB/s	1629 MB/s	1706 MB/s	1599 MB/s	1721 MB/s	1746 MB/s	1761 MB/s	1740 MB/s	1773 MB/s	1872 MB/s	1689 MB/s
3	297 MB/s	581 MB/s	869 MB/s	1153 MB/s	1412 MB/s	1590 MB/s	1575 MB/s	1653 MB/s	1623 MB/s	1655 MB/s	1644 MB/s	1546 MB/s
9	258 MB/s	504 MB/s	766 MB/s	1004 MB/s	1192 MB/s	1402 MB/s	1486 MB/s	1478 MB/s	1517 MB/s	1601 MB/s	1542 MB/s	1554 MB/s

1.5.3 Conclusion

Parallel reads **hurt** performance on HDDs. This is expected, as seek time is a major limiter in this case.

Parallel reads can give at least a 2x performance increase when using SSDs. Diminishing returns kick in above 4 processes.

While high levels of compression can have a serious processing overhead on single processor reads, parallel reads can achieve parity with an uncompressed dataset. Thus, the compression ratio of the data will translate directly to increased read performance.

There is no appreciable difference between the direct, low level access and the generator access method and low read speeds. The limiting factor in that regime is the read speed. At high read speeds, a significant difference is observed; therefore, one should use the direct, low-level access method when high speed NVMe storage is available.

1.5.4 Running the benchmark

Running the benchmark requires HDF5 to be built with the `--enable-direct-vfd` configure option (and then a recompile of pytables), to enable bypassing of the filesystem cache. If the direct driver is not available on your system, the driver may be turned off. However, in this case alternative measures must be taken to avoid the filesystem cache (such as using an appropriately large benchmarking file).

Additionally the benchmark requires the `tqdm` python package.

The most accurate results for your use case can only be obtained by testing the library directly in your application.

1.6 Reference

class multitables.**Streamer**(*filename*, ***kw_args*)

Provides methods for streaming data out of HDF5 files.

class Queue(*request_pool*, *stop*, *block_size*)

Abstract queue that is backed by the internal circular buffer.

close()

Signals to the background processes to stop, and closes the queue.

get()

Get the next element from the queue of data. This method returns a guard object that synchronises access to the underlying buffer. The guard, when placed in a with statement, returns a reference to the next available element in the buffer. This method blocks until data is available.

Returns A guard object that returns a reference to the element.

iter()

Convenience method for easy iteration over elements in the queue. Each iteration of the iterator will block until an element is available to be read.

Returns An iterator for the queue.

get_generator(*path*, *n_procs=None*, *read_ahead=None*, *cyclic=False*, *block_size=None*, *ordered=False*, *field=None*, *remainder=True*)

Get a generator that allows convenient access to the streamed data. Elements from the dataset are returned from the generator one row at a time. Unlike the direct access queue, this generator also returns the remainder elements. Additional arguments are forwarded to get_queue. See the get_queue method for documentation of these parameters.

Parameters **path** –

Returns A generator that iterates over the rows in the dataset.

get_queue(*path*, *n_procs=None*, *read_ahead=None*, *cyclic=False*, *block_size=None*, *ordered=False*, *field=None*, *remainder=False*)

Get a queue that allows direct access to the internal buffer. If the dataset to be read is chunked, the block_size should be a multiple of the chunk size to maximise performance. In this case it is best to leave it to the default. When cyclic=False, and block_size does not divide the dataset evenly, the remainder elements will not be returned by the queue. When cyclic=True, the remainder elements will be part of a block that wraps around the end and includes element from the beginning of the dataset. By default, blocks are returned in the order in which they become available. The ordered option will force blocks to be returned in on-disk order.

Parameters

- **path** – The HDF5 path to the dataset that should be read.
- **n_procs** – The number of background processes used to read the datset in parallel.
- **read_ahead** – The number of blocks to allocate in the internal buffer.
- **cyclic** – True if the queue should wrap at the end of the dataset.
- **block_size** – The size along the outer dimension of the blocks to be read. Defaults to a multiple of the chunk size, or to a 128KB sized block if the dataset is not chunked.
- **ordered** – Force the reader return data in on-disk order. May result in performance penalty.
- **field** – The field or column name to read. If omitted, all fields/columns are read.

- **remainder** – Also return the remainder elements, these will be returned as array smaller than the block size.

Returns A queue object that allows access to the internal buffer.

get_remainder(*path*, *block_size*)
Get the remainder elements. These elements will not be read in the direct queue access cyclic=False mode.

Parameters

- **path** – The HDF5 path to the dataset to be read.
- **block_size** – The block size is used to calculate which elements will remain.

Returns A copy of the remainder elements as a numpy array.

class multitables.**Reader**(*filename*, *n_procs=4*, *notify=None*, ***kw_args*)
Provides methods for random access of HDF5 datasets.

close(*wait=False*)
Close the reader. After this point, no more requests can be made. Pending requests will still be fulfilled. Any attempt to made additional requests will raise an exception. Once all requests have been fulfilled, the background processes and threads will be shut down.

Parameters **wait** – If True, block until all background threads/processes have shut down. False by default.

get_dataset(*path*)
Create a dataset proxy that can be used to create requests. :param path: The internal HDF5 path to the dataset within the HDF5 file. :return: A dataset proxy object.

request(*key*, *stage*)
Generate and queue a request. The details of the request should be provided in the key argument, through operations on one of the dataset proxy objects generated by get_dataset. The result of the request will be stored in the provided stage. A request object will be returned, which can be used to wait on the result and access the result when it is ready. :param key: Operations created by a dataset proxy. :param stage: A stage or stage pool in which the result will be stored. :return: A request object.

stop()
Stop the reader. All background processes and threads will immediately shut down. This will invalidate all pending requests. Attempts to access pending requests, or already waiting requests will raise an exception stating that the reader has stopped.

class multitables.**RequestPool**
A helper class for managing a pool of requests.

add(*req*)
Add a request to the pool. :param req: An object instance that should be place in the pool.

next()
Get the next object in the pool. Blocks until an object is available. :return: The next object in the pool.

exception multitables.**QueueClosedException**

exception multitables.**SubprocessException**
Base class for forwarding exceptions that happen inside a subprocess.

exception multitables.**SharedMemoryError**

class multitables.dataset.**TableDataset**(*reader*, *path*, *dtype*, *shape*)
Proxy for dataset operations on pytables Tables.

col(*name*)
Proxy a column retrieval operation. The interface for this method is equivalent to the pytables method of the same name.

read(*start=None*, *stop=None*, *step=None*, *field=None*)
Proxy a read operation. The interface for this method is equivalent to the pytables method of the same name.

read_coordinates(*coords*, *field=None*)
Proxy a coordinate read operation. The interface for this method is equivalent to the pytables method of the same name.

read_sorted(*sortby*, *checkCSI=False*, *field=None*, *start=None*, *stop=None*, *step=None*)
Proxy a sorted read operation. The interface and requirements for this method are equivalent to the pytables method of the same name.

where(*condition*, *condvars=None*, *start=None*, *stop=None*, *step=None*)
Proxy a conditional selection operations. The interface for this method are equivalent to the pytables method of the same name.

class multitables.dataset.**ArrayDataset**(*reader*, *path*, *dtype*, *shape*)

read(*start=None*, *stop=None*, *step=None*)
Proxy a read operation. The interface for this method is equivalent to the pytables method of the same name.

class multitables.dataset.**VLArrayDataset**(*reader*, *path*, *dtype*, *shape*)

read(*start=None*, *stop=None*, *step=None*)
Proxy a read operation. The interface for this method is equivalent to the pytables method of the same name.

class multitables.request.**Request**(*details*, *stage*)
Public interface for managing requests.

get()
A safe method for accessing the result of the request. This method makes a copy of the result and returns it. This copy can be used in any fashion, as it no longer has resource contraints. :return: A copy of the result of the request.

get_direct(*action*)
A safer method for directly accessing the shared memory. This method blocks until the request is fulfilled. Once ready, it called the provided action function with a direct reference to the shared memory as an argument. Care should be taken that this direct reference does not leave the scope of the function, or else the problems enumerated in the get_unsafe context manager may result.

Parameters **action** – A function that takes one argument, which will be supplied as a direct reference to the shared memory.

get_proxy()
A safe context manager for indirectly accessing the shared memory. This manager waits until the request is fulfilled. Once ready, it yields a proxy to the underlying shared memory. Once the context manager expires, the proxy will be released, and access to the shared memory is no longer possible. Any attempt to access the shared memory past this point raises an exception.

get_unsafe()
A context manager for accessing the result of the request directly. This manager waits until the request is fulfilled. Once ready, it yields a direct reference to the underlying shared memory. If an exception was raised when fielding this request, the exception is re-raised here. Use of this context manager can be

unsafe, as it yields a direct reference to the shared memory. If this reference is not properly managed, it can lead to a dangling pointer that causes an exception when the associated stage is closed. The contents of this dangling pointer will also change when the associated stage is re-used for another request. It is recommended to use a safer access method, or immediately delete or set to None the local variable bound to the yielded reference after use.

CHAPTER 2

Licence

This software is distributed under the MIT licence. See the LICENSE.txt file for details.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index