pyfive: A pure-Python HDF5 reader

Bryan Lawrence [1], **Ezequiel Cimadevilla** [2], **Wout De Nolf** [3], **David Hassell** [1], **Jonathan Helmus**[1], **Brian Maranville** [4], **Kai Mühlbauer** [5], and **Valeriu Predoi** [1]

1 NCAS-CMS, Meteorology Department, University of Reading, Reading, United Kingdom. ROR **2** Instituto de Física de Cantabria (IFCA), CSIC-Universidad de Cantabria, Santander, Spain. **4** NIST Center for Neutron Research **5** Institute of Geosciences, Meteorology Section, University of Bonn, Germany. **3** European Synchrotron Radiation Facility (ESRF), Grenoble, France.
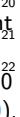
Summary

pyfive (https://pyfive.readthedocs.io/en/latest/) is an open-source thread-safe pure Python package for reading data stored in HDF5. While it is not a complete implementation of all the specifications and capabilities of HDF5, it includes all the core functionality necessary to read gridded datasets, whether stored contiguously or with chunks, and to carry out the necessary decompression for the standard options. All data access is fully lazy, the data is only read from storage when the numpy data arrays are manipulated. Originally developed some years ago, the package has recently been upgraded to support lazy access, and to add missing features necessary for handling all the environmental data known to the authors. It is now a realistic option for production data access in environmental science and more widely. The API is based on that of h5py (which is a Python shimmy over the HDF5 C-library, and hence is not thread-safe), with some API extensions to help optimise remote access. With these extensions, coupled with thread safety, many of the limitations precluding the efficient use of HDF5 (and netCDF4) on cloud storage have been removed.

Statement of need

HDF5[1](Folk et al., 2011) is probably the most important data format in physical science, used across the piste. It is particularly important in environmental science, particularly given the fact that netCDF4[2](Rew et al., 2006) is HDF5 under the hood. From satellite missions, to climate models and radar systems, the default binary format has been HDF5 for decades. While newer formats are starting to get mindshare, there are petabytes, if not exabytes, of existing HDF5, and there are still many good use-cases for creating new data in HDF5. However, despite the history, there are few libraries for reading HDF5 file data that do not depend on the official HDF5 library maintained by the HDF Group, and in particular, apart from pyfive, in Python there are none that cover the needs of environmental science. While the HDF5 c-library is reliable and performant, and battle-tested over decades, there are some caveats to depending upon it: Firstly, it is not thread-safe, secondly, the code is large and complex, and should anything happen to the financial stability of the HDF5 Group, it is not obvious the C-code could be maintained. Finally, the code complexity also meant that it is not suitable for developing bespoke code for data recovery in the case of partially corrupt data. From a long-term curation perspective both of these last two constraints are a concern.

The original implementation of pyfive (by JH), which included all the low-level functionality to deal with the internals of an HDF5 file was developed with POSIX access in mind. The

---

[1]https://www.hdfgroup.org/solutions/hdf5/
[2]https://www.unidata.ucar.edu/software/netcdf

⁴¹ recent upgrades were developed with the use-case of performant remote access to curated
⁴² data as the primary motivation, but with additional motivations of having a lightweight HDF5
⁴³ reader capable of deploying in resource or operating-system constrained environments (such
⁴⁴ as mobile), and one that could be maintained long-term as a reference reader for curation
⁴⁵ purposes. The lightweight deployment consequences of a pure-Python HDF5 reader needs no
⁴⁶ further introduction, but as additional motivation we now expand on the issues around remote
⁴⁷ access and curation.

⁴⁸ Thread safety has become a concern given the wide use of Dask[3] in Python based analysis
⁴⁹ workflows, and this, coupled with a lack of user knowledge about how to efficiently use HDF5,
⁵⁰ has led to a community perception that HDF5 is not fit for remote access (especially on cloud
⁵¹ storage). Issues with thread safety arise from the underlying HDF5 c-library, and cannot be
⁵² resolved in any solution depending on that library, hence the desire for a pure Python solution.
⁵³ Remote access has been bedevilled by the widespread need to access remotely data which has
⁵⁴ been chunked and compressed, combined with the use of HDF5 data which was left in the
⁵⁵ state it was when the data was produced - often with default unsuitable chunking (Rew, 2013)
⁵⁶ and with interleaved chunk indexes and data. Solutions have mainly consisted of reformatting
⁵⁷ the data (and rechunking it at the same time) or utilising kerchunk mediated direct access
⁵⁸ to chunked HDF5 data[4]. However, in practice using kerchunk requires the data provider to
⁵⁹ generate kerchunk indices to support remote users, and it leads to issues of synchronicity
⁶⁰ between indices and changing datasets.

⁶¹ This version of pyfive was developed with these use-cases in mind. There is now full support
⁶² for lazy loading of chunked data, and methods are provided to give users all the benefits of
⁶³ using kerchunk, but without the need for a priori generation. Because pyfive can access and
⁶⁴ cache (in the client) the b-tree (index) on a variable-by-variable basis, most of the benefits of
⁶⁵ kerchunk are gained without any of the constraints. However, the kerchunk index is always a
⁶⁶ contiguous object accessible with one get transaction, this is not necessarily the case with the
⁶⁷ b-tree, unless the source data has been repacked to ensure contiguous metadata using a tool
⁶⁸ like h5repack. Much of the community is unaware of the possibility of repacking the index
⁶⁹ metadata, and this together with relatively opaque information about the internal structure
⁷⁰ of files (and hence the necessity or other wise of such repacking), means that repacking is
⁷¹ rarely done. To help with this process, pyfive also includes extensions to expose information
⁷² about how data and indexes are distributed in the files. With these tools, index extraction
⁷³ with pyfive can be comparable in performance to obtaining a kerchunk index, and completely
⁷⁴ opaque to the user.

⁷⁵ With the use of pyfive, suitably repacked and rechunked HDF5 data can now be considered
⁷⁶ "cloud-optimised", insofar as with lazy loading, improved index handling, and thread-safety,
⁷⁷ there are no "format-induced" constraints on performance during remote access. To aid in
⁷⁸ discovering whether or not a given HDF5 dataset is cloud-optimised, pyfive also now provides
⁷⁹ simple methods to expose information about file layout - both in API extensions, and in a
⁸⁰ new p5dump utility packaged with the pyfive library, which provides (in the default view)
⁸¹ functionality similar to ncdump, and when used with p5dump -s, information about storage
⁸² characteristics.

⁸³ The issues of the dependency on a complex code maintained by one private company in the
⁸⁴ context of maintaining data access (over decades, and potentially centuries), can only be
⁸⁵ mitigated by ensuring that the data format is well documented, that data writers use only the
⁸⁶ documented features, and that public code exists which can be relatively easily maintained.
⁸⁷ The HDF5group have provided good documentation for the core features of HDF5 which
⁸⁸ include all those of interest to the weather and climate community who motivated this reboot
⁸⁹ of pyfive, and while there is a community of developers beyond the HDF5 group (including
⁹⁰ some at the publicly funded Unidata institution), recent events suggest that given most of

---

[3]https://www.dask.org/
[4]https://fsspec.github.io/kerchunk/

those developers and their existing funding are US based, some spreading of risk would be desirable. To that end, a pure Python code, which is relatively small and maintained by an international constituency, alongside the existing C-code, provides some assurance that the community can maintain HDF5 access for the foreseeable future. A pure Python code also makes it easier to develop scripts which can work around data and metadata damage should they occur.

Examples

We now introduce three aspects of the new functionality that `pyfive` now provides: remote access, configurable lazy loading, and determining whether files are cloud optimised.

Remote Access

A notable feature of the recent `pyfive` upgrade is that it was carried out with thread-safety and remote access using fsspec (https://filesystem-spec.readthedocs.io) in mind. We provide two examples of using `pyfive` to access remote data, one in S3, and one behind a modern http web server:

For accessing the data on S3 storage, we will have to set up an s3fs virtual file system, then pass it to `pyfive`:

```python
import pyfive
import s3fs
# storage options for an anon S3 bucket
# there are also caching options for the s3 middleware, not shown here
storage_options = {
    "anon": True,
    "client_kwargs": {"endpoint_url": "https://s3server.ac.uk"}
}
fs = s3fs.S3FileSystem(**storage_options)
file_uri = "s3-bucket/myfile.nc"
with fs.open(file_uri, "rb") as s3_file:
    nc = pyfive.File(s3_file)
    dataset = nc["var"]
```

for an HTTPS data server, the usage is similar:

```python
import fsspec
import pyfive
# there are also caching options for the fsspec middleware, not shown here
fs = fsspec.filesystem("http")
with fs.open("https://site.com/myfile.nc", "rb") as http_file:
    nc = pyfive.File(http_file)
    dataset = nc[var]
```

This is of course exactly the same pattern as remote access using h5py, and that is by design - to make moving to `pyfive` easy for users!

Lazy Loading

A key tenet of efficient remote access is that variable inspection is quick and involves the minimimum of network traffic between storage and the client application.
However when this is coupled with the common pattern of using Dask, some flexibility in what is loaded when is beneficial.

By default when one inspects the contents of a file using `pyfive` nothing more is read from the file than the names of the variables ("datasets" in the language of HDF5): for example:

```
with pyfive.File("myfile.h5","r") as f:
    variables_in_file = [v for v in f]
```

117 involves nothing more than getting a set of variable names. When one wishes to inspect these
118 variables:

```
with pyfive.File("myfile.h5","r") as f:
    temp = f["temp"]
print(temp[1:10])
```

119 the default in pyfive is to get only the metadata associated with each variable - but crucially
120 at this point the b-tree index is also loaded and the variable can now be accessed outside the
121 context manager. Data loading is now completely lazy and the variable instance (temp) has all
122 the information needed to extract data as needed. This is done so that in Dask applications,
123 when one passes each Dask computational chunk a portion of the variable, each such Dask
124 chunk has already got the index, and when it does want to load data, it can be as efficient as
125 possible.

126 However, there are situations where loading the b-tree at variable instantiation is not wanted,
127 and all is wanted is to be able to view all the variable attributes. To support this option pyfive
128 also offers the get_lazy_view file method, so one can do:

```
with pyfive.File("myfile.h5","r") as f:
    temp = f.get_lazy_view("temp")
print(temp.attrs())
```

129 It is still possible to access the data and the b-tree index is loaded when data access is first
130 attempted.

131 This extra lazy view is new functionality. It would obviously have been possible to make the
132 default not load the b-tree, but in the opinion of the current pyfive maintainers loading the
133 b-tree at variable instantiation is likely more consistent with user expectations as it is more
134 equivalent to the behaviour of other packages like h5py.

Cloud Optimisation

136 To be fully cloud optimised - as defined by Stern et al. (2022) - an HDF5 file needs to have
137 a contiguous index for each variable, and the chunks for each variable need to be sensibly
138 chosen and broadly contiguous within the file. When these criteria are met, indexes can be
139 read efficiently, and middleware such as fsspec can make sensible use of readahead caching
140 strategies.

141 HDF5 data files direct from simulations and instruments are often not in this state as information
142 about the number of variables, the number of chunks per variable, and the compressed size of
143 those variables is not known as the data is being produced.
144 In such cases the data is also often not chunked along the dimensions being added to as the
145 file is written (since it would have to be buffered first).

146 Of course, once the file is produced, such information is available. Metadata can be repacked to
147 the front of the file and variables can be rechunked and made contiguous - which is effectively
148 the same process undertaken when HDF5 data is reformatted to other cloud optimised formats.

149 The HDF5 library provides a tool "h5repack" which can do this, provided it is driven with
150 suitable information about required chunk shape and the expected size of metadata fields.
151 pyfive supports both a method to query whether such repacking is necessary, and to extract
152 necessary parameters.

153 In the following example we compare and contrast the unpacked and repacked version of a
154 particularly pathological file, and in doing so showcase some of the pyfive API extensions
155 which help us understand why it is pathological, and how to address those issues for repacking.

```

If we extract just a piece of the output of p5dump -s on this file (which has surface wind velocity at three hour intervals for one hundred years):

```
float64 time(time) ;
        time:standard_name = "time" ;
        time:_n_chunks = 292192 ;
        time:_chunk_shape = (1,) ;
        time:_btree_range = (31808, 19854095942) ;
        time:_first_chunk = 9094 ;

float32 uas(time, lat, lon) ;
        uas:_Storage = "Chunked" ;
        uas:_n_chunks = 292192 ;
        uas:_chunk_shape = (1, 143, 144) ;
        uas:_btree_range = (28672, 19854809382) ;
        uas:_first_chunk = 36520 ;
```

we can immediately see that this will be a problematic file! The b-tree index is clearly interleaved with the data (compare the first chunk address with last index addresses of the two variables), and with a chunk dimension of (1,), any effort to use the time-dimension to locate data of interest will involve a ludicrous number of 1 number reads (all underlying libraries read the data one chunk at a time). It would feel like waiting for the heat death of the universe if one was to attempt to manipulate this data stored on an object store!

It is relatively easy (albeit slow) to use h5repack to fix this - e.g see Hassell & Cimadevilla Alvarez ([2025](#)) - after which we see:

```
float64 time(time) ;
        time:_Storage = "Chunked" ;
        time:_n_chunks = 1 ;
        time:_chunk_shape = (292192,) ;
        time:_btree_range = (11861, 11861) ;
        time:_first_chunk = 40989128 ;
        time:_compression = "gzip(4)" ;
float32 uas(time, lat, lon) ;
        uas:_Storage = "Chunked" ;
        uas:_n_chunks = 5844 ;
        uas:_chunk_shape = (50, 143, 144) ;
        uas:_btree_range = (18663, 347943) ;
        uas:_first_chunk = 41041196 ;
        uas:_compression = "gzip(4)" ;
```

Now data follows indexes, the time dimension is one chunk, and there is a more sensible number of actual data chunks. While this file would probably benefit from splitting, with a contiguous set of indexes, it is now possible to exploit this data via S3.

All the metadata shown in this dump output arises from pyfive extensions to the pyfive.h5t.DatasetID class. pyfive also provides a simple flag: consolidated_metadata for a File instance, which can take values of True or False for any given file, which simplifies at least the "is the index packed at the front of the file?" part of the optimisation question - though inspection of chunking is a key part of the workflow necessary to determine whether or not a file really is optimised for cloud usage.

Acknowledgements

References

Folk, M., Heber, G., Koziol, Q., Pourmal, E., & Robinson, D. (2011). An overview of the HDF5 technology suite and its applications. *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, 36–47. https://doi.org/10.1145/1966895.1966900

Hassell, D., & Cimadevilla Alvarez, E. (2025). *Cmip7repack: Repack CMIP7 netCDF-4 datasets*. Zenodo. https://doi.org/10.5281/zenodo.17550920

Rew, R. (2013). Chunking data: Choosing shapes. In *News @ Unidata*. https://www.unidata.ucar.edu/blog/developer/en//entry/chunking_data_choosing_shapes.

Rew, R., Hartnett, E., & Caron, J. (2006). NetCDF-4: Software implementing an enhanced data model for the geosciences. *22nd Inernational Conference on Interactive Information Processing Systems for Meteorology, Oceanography and Hydrology*.

Stern, C., Abernathey, R., Hamman, J., Wegener, R., Lepore, C., Harkins, S., & Merose, A. (2022). Pangeo forge: Crowdsourcing analysis-ready, cloud optimized data production. *Frontiers in Climate*, *3*. https://doi.org/10.3389/fclim.2021.782909