SPIDAL Summer REU 2020: Implementing an HDF5-based File Format Coordinate Reader into MDAnalysis

Edis Jakupovic

Department of Physics, College of Liberal Arts and Sciences
Arizona State University, Tempe, Arizona 85281

August 28, 2020

Abstract

Molecular dynamics (MD) trajectory files are now commonly Terabytes in size, which comes with the negative consequence that reading the file and extracting useful data has become a bottleneck in the analysis of MD trajectories. Results in recent works have shown that a message passing interface implementation which reads an HDF5 file in parallel is a promising approach to speeding up trajectory analysis, so the goal of this project was to implement a trajectory reader class, *H5MDReader*, into MDAnalysis that is capable of reading an H5MD file in parallel through the parallel I/O features of an MPI library. This was done using the h5py Python package, which can handle HDF5 file objects with MPI drivers for parallel reading and writing. The parallel features of *H5MDReader* have not been thoroughly tested, however the serial capabilities were benchmarked by calculating the RMSD of sample trajectories versus various file formats. The results show that, in serial, *H5MDReader* takes a substantially longer time to read files than other formats, which requires analysis into where *H5MDReader* is spending time during I/O. ADD PROFILER RESULTS

Background

Molecular dynamics (MD) simulations provide a powerful and effective way to model the molecular happenings of a biological system. From the folding and unfolding of a protein molecule, to understanding the interaction between proteins and drug molecules, and even problems in materials science, MD simulations are a vital tool to understanding the function of biomolecules. These simulations are constructed by calculating the forces on each particle that are derived from a classical potential energy function and Newton's equations of motion, and are stored as "trajectory" files, which contain the position (but also can contain velocity, force, etc.) of every atom in the system for every frame of time in the length of the simulation[3]. As computing power continues to increase, and simulation data becomes more robust, the size of these trajectory files is now commonly Terabytes in size, which comes with the negative consequence that reading the file (input) and extracting useful data (output) has become a bottleneck in the analysis of MD trajectories[2]. Previous works[2,3] have shown that performing analysis tasks on trajectory data in parallel, rather than in series, can greatly boost the performance of certain types of analysis.

One of the issues in the field of computational biophysics is that trajectory files can be written in many different types of file formats, to the point where poor documentation and the barrier of learning how to access the data of one specific file format can be a hindrance to scientific productivity. MDAnalysis is a Python package that fixes this issue by handling the reading of over 25 different file formats internally, and providing an intuitive interface for accessing the useful data in a trajectory file[1]. Currently, trajectories can be analyzed in parallel using Parallel MDAnalysis (PMDA), which uses the Dask library that follows a split-apply-combine approach for trajectory analysis[3]. A message passing interface (MPI) implementation reading an HDF5 file in parallel is also a promising approach to speeding up trajectory analysis[2]. However, MDAnalysis did not have an HDF5-based format trajectory reader, so the goal of this project was to implement an HDF5-based trajectory reader into MDAnalysis that is capable of reading a file in parallel through the parallel I/O features of an MPI library.

H5MD, or "HDF5 for molecular data", is a file format that is used to store data from MD simulations such as particle coordinates, simulation box dimensions, and thermodynamic observables[4]. It is built upon the HDF5 file format, which is a structured, binary file format that organizes data into 2 objects: groups and datasets[4]. HDF5 files follow a hierarchical, tree-like structure, where groups represent nodes, or "splitting" points in the branches of the tree, and datasets represent the end of a branch, or the "leaves" of the tree. Both groups and datasets can contain metadata in the form of attributes, which can be especially useful in MD simulations with something like a "unit" attribute that is attached to a "positions" dataset. H5MD provides specifications with how these groups and datasets are organized and named for MD simulation data which is important for consistency in future scientific works that use HDF5-based file formats for MD simulations.

A Python package, pyh5md, already exists that can read and write H5MD files using the h5py libraries to handle HDF5 file objects with Python. Pyh5md was originally going to be used to handle the HDF5 file objects in MDAnalysis because it already adheres to H5MD's specifications, however it is no longer being maintained by its developers. For this reason, we decided to use h5py directly to implement an *H5MDReader* class that obeys the H5MD format specifications that will add to the array of MDAnalysis coordinate readers. Some simple benchmarks were done to compare the serial performance of an root-mean-square deviation

(RMSD) calculation of the alpha carbon positions of various file format readers in MDAnalysis.

Methods

Implementation

We added a module, *H5MD.py*, to MDAnalysis.coordinates that includes *class Timestep* and *class H5MDReader*. A host of unit tests were also written for *H5MDReader* that provided 93.95% Codecov coverage.

The Timestep class derives from base.Timestep, and overrides the method *_init_unitcell()* because H5MD stores unitcell information as a matrix, with the edge vectors of the triclinlic box being the rows of the matrix. The *dimensions()* property and setter are then overridden, where the *core.triclinic_box()* method is used to convert the box vectors given by the H5MD file into [lx, ly, lz, alpha, beta, gamma] format for MDAnalysis to display.

H5MDReader derives from *base.ReaderBase*, which derives from *base.ProtoReader*. To open an h5py file object with parallel MPI drivers, a *driver* and *comm* arguments must be passed to the *h5py.File()* method. We found that the easiest way for *H5MDReader* to accomplish this is by checking for the arguments, *driver* and *comm*, at initialization and storing them as class variables, and then passing them to *h5py.File()* in the method *open_trajectory()*, however this implementation is not thoroughly tested yet. Since there are no standard units defined in the H5MD format specifications, *H5MDReader* needed a way to check if units are provided in the input file, and whether MDAnalysis convert the given units to its native units. We found the best way to do this was to create a "translation dictionary" that includes all available MDAnalysis units and translates them from H5MD notation to MDA notation. The method, *_translate_h5md_units()*, checks if the given H5MD file contains any "unit" attributes and uses this dictionary to set the units in MDAnalysis.

There are two crucial methods that were overridden that *H5MDReader* uses to construct the Timestep object when a specific frame is called with MDAnalysis: *_read_frame()* and *_read_next_timestep()*, where *_read_next_timestep()* simply calls *_read_frame()* shifted by +1 frame. To avoid repetitive code and to make the flow more readable, we implemented several helper functions within the method *_read_frame()*. *_copy_data()* scans the "observables" group in the H5MD files and stores any data into the *ts.data* dictionary of an MDAnalysis universe. *_get_frame_dataset()* checks to see if the groups position, velocity, or force exist in the H5MD file and store the array for that specific frame into *ts.positions*, *ts.velocities*, and *ts.forces*, respectively. *_convert_units()* converts the values in the position, velocity, and force array to MDAnalysis standard units.

Benchmarking

In order the test the serial capabilities of *H5MDReader*, we ran some simple benchmarks that used the *time.time()* method to calculate the time it took for MDAnalysis to perform an RMSD calculation on the alpha carbon positions of the atoms of a sample trajectory in the MDAnalysis testsuite datafiles, and compared these results with *pyh5md* computing the same thing on the same array of positions.

To set up the files for benchmarking, we used the *cobrotoxin.tpr* topology file from MDAnalysisTests.datafiles, and used 4 identical trajectory files: *cobrotoxin.dcd*, *cobrotoxin.xtc*, *cobrotoxin.trr*, and *cobrotoxin.h5md*. It should be noted that *cobrotoxin.xtc* and *cobrotoxin.dcd* do not contain velocity and force data. The trajectories were extended by 1000x their original length by writing new files with the MDAnalysis *ChainReader* class to have a size of 3000 frames of 19385 atoms. A Python script was written where the file was opened, and iterated over each time frame to calculate the RMSD. Three time values were stored: the total I/O time for MDAnalysis to open the timestep, and the total time for the RMSD calculation to complete, and the total time for the loop to complete which included both the I/O time and RMSD calculation time. These total times were divided by the number of frames in the trajectory to give the time/frame for loop, I/O, and RMSD calculation. The benchmark was run 5 times, and the mean and standard deviation of the 5 runs was used to plot bar graphs illustrating how long each section of the benchmark took. The same benchmark was done using pyh5md to open *cobrotoxin.h5md* instead of MDAnalysis and added to the bar graphs.

Following the results of the benchmarks, a Python line profiler was used in a Jupyter Notebook performing the MDAnalysis and pyh5md benchmark on *cobrotoxin.h5md* with 300 frames to see where the discrepancy in time came from. All scripts used for benchmarking can be found at
https://github.com/Becksteinlab/h5md_examples/tree/master/h5md_benchmarking

Results and Discussion

The goal of this project was to implement an HDF5-based format trajectory reader into MDAnalysis that is capable of reading a file in parallel through the parallel I/O features of an MPI library. This was accomplished, and *H5MDReader* will be available in MDAnalysis 2.0.0. While the parallel features of the implemented *H5MDReader* have not been thoroughly tested, the serial capabilities of the reader have been. A suite of tests with 93.95% coverage were written in MDAnalysisTests.coordinates.test_h5md.py that verified the validity of the data read into the reader, as well as making sure all of the appropriate errors are raised when needed, such as the H5MD file not containing units, yet *convert_units* is set to *True* upon initialization. *H5MDReader* unpacks an H5MD file's "particles" group and stores the box dimensions, positions, velocities, and forces into appropriate MDAnalysis Timestep attributes, along with any units associated with each dataset. It does not currently read charge or mass data from the "particles" group, however this is typically handled in the topology file in MDAnalysis. It also has the capability to store any datasets found in the "observables" group into the Timestep.data dictionary. As far as the remaining "connectivity" and "parameters" groups defined in the H5MD specifications, the implemented *H5MDReader* does not currently read any data from these groups, however it can easily be implemented given the accessibility allowed by h5py.

An important aspect to consider is that the implemented *H5MDReader* is serializable. A recent update in MDAnalysis allows *mda.Universe* objects to be pickled, which is crucial for utilizing the parallel capabilities of an MPI library, such as mpi4py. An H5MD universe can be sliced into many pieces to be handled in parallel with an MPI implementation, and the serializability allows the pieces to be converted into byte streams that make it possible for separate MPI processes to communicate with each other.

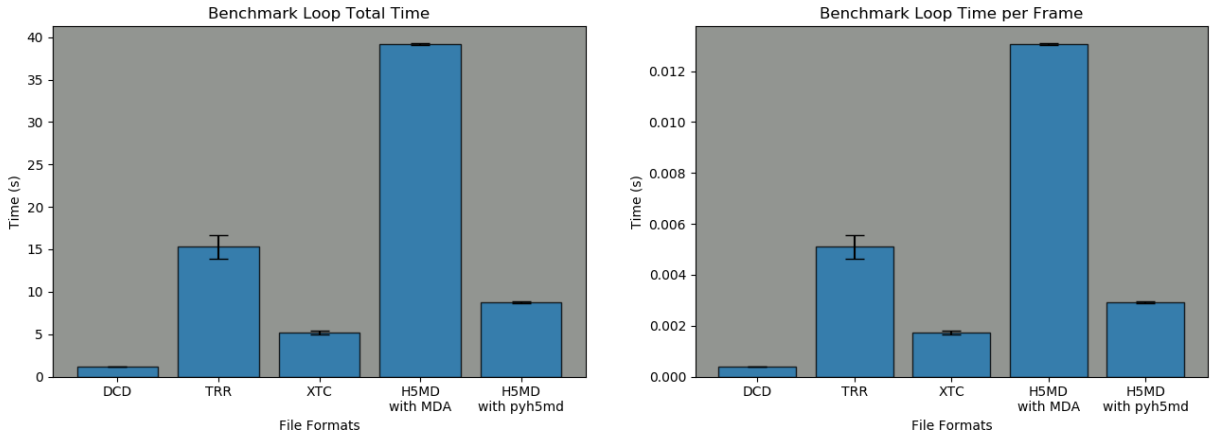
The benchmarks were performed on a remote workstation on a solid state drive, and the files were

prepared for benchmarking as given in the Methods section. Interestingly, the H5MD file with 3 frames was much larger than the TRR file which contained identical data. Table 1 gives the size of each file in its initial trajectory size, and with 1000x the frames. When the trajectories are expanded to a larger number of frames, however, this difference disappears, which seems to indicate the HDF5 data structures take up more space than the TRR file, but as the size of the arrays increases they begin to dominate the file size. Figure 1

Size	DCD	TRR	XTC	H5MD
1x	683 KB	2 MB	194 KB	22 MB
1000x	666 MB	2 GB	189 MB	2 GB

Table 1: File sizes for different trajectory formats for 1x frames and 1000x frames

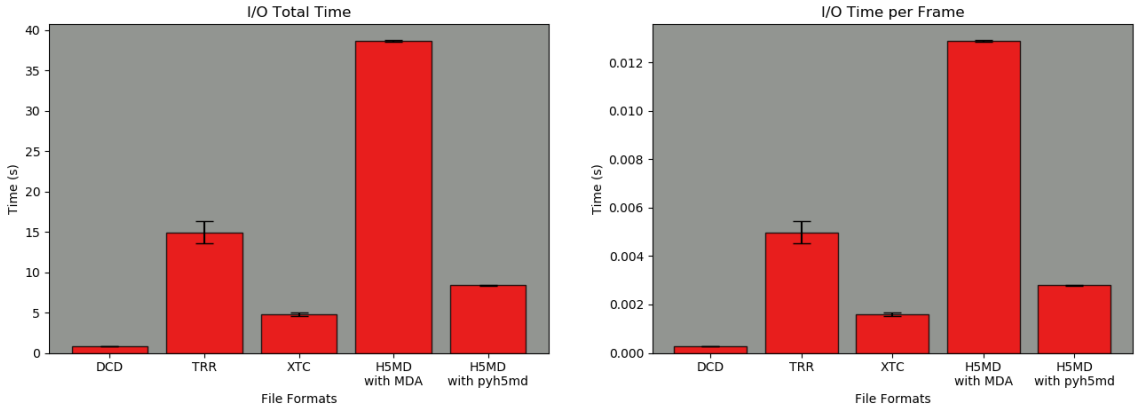
below gives the bar graphs for the benchmark loop total time and time per frame for each file format using MDAnalysis, as well as the same benchmark using the pyh5md package. From the figure, it seems that *H5MDReader* takes a substantially longer time to perform the benchmark than the other file formats. DCD and XTC do not contain velocity and force data, so it makes sense that their benchmarks would be faster, however the TRR file contains the same trajectory data, yet takes less than half the time has the H5MD file.



(a) (b)

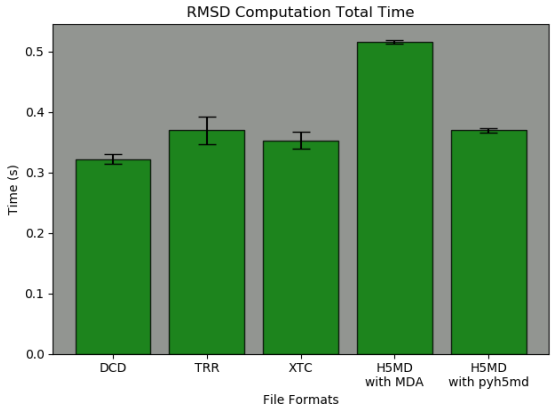
Figure 1: Benchmark loop times with standard deviation error bars. (a) gives the total time spent during the loop and (b) gives the time spent per frame.

To see at which point during the benchmark this discrepancy comes from, we can look at the I/O time and the RMSD calculation time of the benchmark, given in figure 2 below. From sub-figures (c) and (d) in Figure 2, we see that all of the file formats are relatively equal in their RMSD computation time, however H5MD with MDAnalysis is still slightly slower than the rest. Sub-figures (a) and (b), however, show the same large difference between the TRR and H5MD benchmark times. These results indicate that the I/O portion of MDAnalysis opening and reading the timestep is responsible for large discrepancy between the H5MD benchmark time and the other file formats.

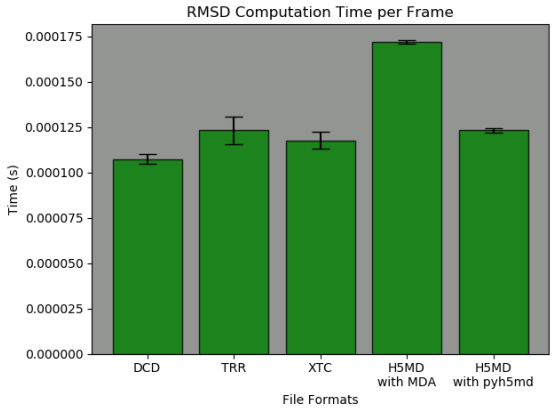


(a)

(b)



(c)



(d)

Figure 2: I/O and RMSD calculation times with standard deviation error bars. (a) and (c) give the total time spent during the benchmark. (b) and (d) give the time spent per frame.

Following this peculiar result, a line profiler ran in Jupyter Notebook, given in Figure 3, revealed that the most time during the benchmark was spent performing the *__getitem__()* method in h5py's dataset.py module, which imitates NumPy's array slicing notation to select subsets of a dataset.

```
         20752090 function calls (20585088 primitive calls) in 45.239 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    13553   13.139    0.001   16.400    0.001 dataset.py:476(__getitem__)
     1502   10.154    0.007   10.160    0.007 {method 'read' of 'MDAnalysis.lib.formats.libmdaxdr.TRRFile' objects}
      302    4.699    0.016    4.699    0.016 poll.py:102(select)
     1502    1.615    0.001    1.618    0.001 {method 'read' of 'MDAnalysis.lib.formats.libmdaxdr.XTCFile' objects}
    46903    1.237    0.000    2.114    0.000 group.py:255(__getitem__)
       50    0.855    0.017    0.858    0.017 arraysetops.py:484(in1d)
   969250    0.646    0.000    1.008    0.000 ntpath.py:44(normcase)
     1505    0.567    0.000    1.739    0.001 selections.py:343(__getitem__)
        9    0.556    0.062    3.090    0.343 utils.py:287(do_mtop)
   105358    0.541    0.000    1.209    0.000 selections.py:444(_handle_simple)
    22687    0.526    0.000    0.634    0.000 dataset.py:395(__init__)
    64787    0.431    0.000    0.440    0.000 dataset.py:282(shape)
```

Figure 3: Line profiler results on benchmarks with 300 frame *cobrotoxin* test file.

The next question the benchmark results seem to ask is, why is pyh5md so much faster than MDAnalysis? Figure 4 gives the profiler results for

```
         3935010 function calls (3909073 primitive calls) in 14.337 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    12008   10.035    0.001   11.184    0.001 dataset.py:476(__getitem__)
    34593    0.758    0.000    1.275    0.000 group.py:255(__getitem__)
    55562    0.331    0.000    0.339    0.000 dataset.py:282(shape)
    13542    0.292    0.000    0.350    0.000 dataset.py:395(__init__)
    16509    0.156    0.000    0.156    0.000 {method 'reduce' of 'numpy.ufunc' objects}
     4503    0.155    0.000   10.564    0.002 H5MD.py:654(_get_frame_dataset)
       10    0.148    0.015    0.149    0.015 arraysetops.py:484(in1d)
   193850    0.136    0.000    0.208    0.000 ntpath.py:44(normcase)
     1501    0.128    0.000    1.338    0.001 H5MD.py:638(_copy_to_data)
     1501    0.098    0.000   13.101    0.009 H5MD.py:593(_read_frame)
```

(a)

```
         9232345 function calls (9161439 primitive calls) in 5.210 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1505    0.505    0.000    1.563    0.001 selections.py:343(__getitem__)
    93350    0.434    0.000    0.947    0.000 selections.py:444(_handle_simple)
     1545    0.352    0.000    2.067    0.001 dataset.py:476(__getitem__)
        5    0.303    0.061    1.610    0.322 utils.py:287(do_mtop)
    12310    0.277    0.000    0.520    0.000 group.py:255(__getitem__)
  2809450    0.205    0.000    0.205    0.000 {method 'append' of 'list' objects}
    94855    0.170    0.000    0.281    0.000 selections.py:421(_expand_ellipsis)
     9145    0.169    0.000    0.201    0.000 dataset.py:395(__init__)
       10    0.159    0.016    0.160    0.016 arraysetops.py:484(in1d)
```

(b)

Figure 4

1 Conclusion

The aim of this project was to implement an HDF5-based file format coordinate reader into the MDAnalysis package. The implemented *H5MDReader* class successfully reads trajectory data from an H5MD file, however our benchmarks show I/O speed of handling an H5MD file with MDAnalysis is much slower than a TRR file with the same data. ADD PROFILER RESULTS. Further testing is also required on the parallel MPI I/O capabilities of *H5MDReader*.

2 Acknowledgements

Funding was provided by the National Science Foundation for a REU supplement to award ACI1443054.

References

[1] R. J. Gowers, M. Linke, J. Barnoud, T. J. E. Reddy, M. N. Melo, S. L. Seyler, D. L. Dotson, J. Domanski, S. Buchoux, I. M. Kenney, and O. Beckstein. MDAnalysis: A Python package for the rapid analysis of molecular dynamics simulations. In S. Benthall and S. Rostrup, editors, Proceedings of the 15th Python in Science Conference, pages 102-109, Austin, TX, 2016. SciPy

[2] M. Khoshlessan, I. Paraskevakos, G. C. Fox, S. Jha, and O. Beckstein. Parallel performance of molecular dynamics trajectory analysis. Concurrency and Computation: Practice and Experience, e-pub:e5789, 2020. doi: 10.1002/cpe.5789

[3] Shujie Fan, Max Linke, Ioannis Paraskevakos, Richard Gowers, Michael Gecht, and Oliver Beckstein. "PMDA - Parallel Molecular Dynamics Analysis". In: Python in Science Conference. Austin, Texas, 2019, pp. 134–142.

[4] Pierre Buyl, Peter Colberg, Felix Höfling (2013). H5MD: A structured, efficient, and portable file format for molecular data. Computer Physics Communications. 185. 10.1016/j.cpc.2014.01.018.