MPI-parallel Molecular Dynamics Trajectory Analysis with the H5MD Format in the MDAnalysis Python Package

Edis Jakupovic[‡], Oliver Beckstein[‡*]

Abstract—Molecular dynamics (MD) computer simulations help elucidate details of the molecular processes in complex biological systems, from protein dynamics to drug discovery. One major issue is that these MD simulation files are now commonly terabytes in size, which means analyzing the data from these files becomes a painstakingly expensive task. In the age of national supercomputers, methods of parallel analysis are becoming a necessity for the efficient use of time and high performance computing (HPC) resources but for any approach to parallel analysis, simply reading the file from disk becomes the performance bottleneck that limits overall analysis speed. One promising way around this file I/O hurdle is to use a parallel message passing interface (MPI) implementation with the HDF5 (Hierarchical Data Format 5) file format to access a single file simultaneously with numerous processes on a parallel file system. Our previous feasibility study suggested that this combination can lead to favorable parallel scaling with hundreds of CPU cores, so we implemented a fast and user-friendly HDF5 reader (the `H5MDReader` class) that adheres to H5MD (HDF5 for Molecular Dynamics) specifications. We made `H5MDReader` (together with a H5MD output class `H5MDWriter`) available in the MDAnalysis library, a Python package that simplifies the process of reading and writing various popular MD file formats by providing a streamlined user-interface that is independent of any specific file format. We benchmarked H5MDReader's parallel file reading capabilities on three HPC clusters: ASU Agave, SDSC Comet, and PSC Bridges. The benchmark consisted of a simple split-apply-combine scheme of an I/O bound task that split a 90k frame (113 GiB) coordinate trajectory into N chunks for N processes, where each process performed the commonly used RMSD (root mean square distance after optimal structural superposition) calculation on their chunk of data, and then gathered the results back to the root process. For baseline performance, we found maximum I/O speedups at 2 full nodes, with Agave showing 20x, and a maximum computation speedup on Comet of 373x on 384 cores (all three HPCs scaled well in their computation task). We went on to test a series of optimizations attempting to speed up I/O performance, including adjusting file system stripe count, implementing a masked array feature that only loads relevant data for the computation task, front loading all I/O by loading the entire trajectory into memory, and manually adjusting the HDF5 dataset chunk shapes. We found the largest improvement in I/O performance by optimizing the chunk shape of the HDF5 datasets to match the iterative access pattern of our analysis benchmark. With respect to baseline serial performance, our best result was a 98x speedup at 112 cores on ASU Agave. In terms of absolute time saved, the analysis went from 4623 seconds in the baseline serial run to 47 seconds in the parallel, properly chunked run. Our results emphasize the fact that file I/O is not just dependent on the access pattern of the file, but more so the synergy between access pattern and the layout of the file on disk.

Index Terms—Molecular Dynamics Simulations, High Performance Computing, Python, MDAnalysis, HDF5, H5MD, MPI I/O

‡ *Arizona State University*
∗ *Corresponding author: obeckste@asu.edu*

Introduction

The molecular dynamics (MD) simulation approach [HBD+19] is widely used across the biomolecular and materials sciences, accounting for more than one quarter of the total computing time [FQC+19] in the Extreme Science and Engineering Discovery Environment (XSEDE) network of national supercomputers in the US [TCD+14]. MD simulations, especially in the realm of studying protein dynamics, serve an important purpose in characterizing the dynamics, and ultimately the function of a protein [Oro14]. For example, recent award-winning work [CDG+21] involving the SARS-CoV-2 spike protein was able to use all-atom MD simulations to elucidate the dynamics of the virus-to-human cell interaction that was inaccessible to experiment. While the parameters involved in fine tuning the physics driving these simulations continue to improve, the computational demand of longer, more accurate simulations increases [DDG+12]. As high performance computing (HPC) resources continue to improve in performance, the size of MD simulation files are now commonly terabytes in size, making serial analysis of these trajectory files impractical [CR15]. Parallel analysis is a necessity for the efficient use of both HPC resources and a scientist's time [BFJ18], [FQC+19]. MD trajectory analysis can be parallelized using task-based or MPI-based (message passing interface) approaches, each with their own advantages and disadvantages [PLK+18]. Here we investigate parallel trajectory analysis with the MDAnalysis Python library [MADWB11], [GLB+16]. MDAnalysis is a widely used package in the molecular simulation community that can read and write over 25 popular MD trajectory file formats while providing a common object-oriented interface that makes data available as numpy arrays [HMvdW+20]. MDAnalysis aims to bridge the entrenched user communities of different MD packages, allowing scientists to more easily (and productively) move between these entrenched communities. Previous work that focused on developing a task-based approach to parallel analysis found that an I/O bound task only scaled to 12 cores due to a file I/O bottleneck [SFMLIP+19]. Our recent feasibility study suggested that parallel reading via MPI-IO and the HDF5 file format could

lead to good scaling although only a reduced size custom HDF5 trajectory was investigated and no usable implementation of a true MD trajectory reader was provided [KPF+20].

H5MD, or "HDF5 for molecular data", is an HDF5-based file format that is used to store MD simulation data, such as particle coordinates, box dimensions, and thermodynamic observables [dBCH14]. A Python reference implementation for H5MD exists (pyh5md [dBCH14]) but the library is not maintained anymore, and with advice from the original author of pyh5md, we implemented native support for H5MD I/O in the MDAnalysis package. HDF5 is a structured, binary file format that organizes data into two objects: groups and datasets. It implements a hierarchical, tree-like structure, where groups represent nodes of the tree, and datasets represent the leaves [Col14]. An HDF5 file's datasets can be stored either contiguously on disk, or scattered across the disk in different locations in *chunks*. These chunks must be defined on initialization of the dataset, and for any element to be read from a chunk, the entire chunk must be read. The HDF5 library can be built on top of a message passing interface (MPI) implementation so that a file can be accessed in parallel on a parallel file system such as Lustre or BeeGFS. We implemented a parallel MPI-IO capable HDF5-based file format trajectory reader into MDAnalysis, H5MDReader, that adheres to the H5MD specifications. H5MDReader interfaces with h5py, a high level Python package that provides a Pythonic interface to the HDF5 format [Col14]. In `h5py`, accessing a file in parallel is accomplished by passing a keyword argument into `h5py.File`, which then manages parallel disk access.

The BeeGFS and Lustre parallel file systems are well suited for multi-node MPI parallelization. One key feature of a Lustre parallel file systems is file striping, which is the ability to store data from a file across multiple physical locations, known as object storage targets (OSTs), where "stripe count" refers to the number of OSTs to which a single file is striped across. Thinking carefully about the synchronization of chunk shape and stripe settings can be crucial to establishing optimal I/O performance [How10]. We tested various algorithmic optimizations for our benchmark, including using various stripe counts (1, 48, 96), loading only necessary coordinate information with numpy masked arrays [HMvdW+20], and front loading all I/O by loading the entire trajectory chunk into memory prior to the RMSD calculation.

We benchmarked H5MDReader's parallel reading capabilities with MDAnalysis on three HPC clusters: ASU Agave at Arizona State University, and SDSC Comet and PSC Bridges, which are part of XSEDE [TCD+14]. The benchmark consisted of a simple split-apply-combine scheme [Wic11] of an I/O-bound task that split a 90k frame (113 GiB) trajectory into N chunks for N processes, where each process performed a computation on their chunk of data, and the results were finally gathered back to the root process. For the computational task, we computed the time series of the root mean squared distance (RMSD) of the positions of the C_α (alpha carbon) atoms in the protein to their initial coordinates at the first frame of the trajectory. At each frame (time step) in the trajectory, the protein was optimally superimposed on the reference frame to remove translations and rotations. The RMSD calculation is a very common task performed to analyze the dynamics of the structure of a protein [MM14]. Because it is a fast computation that is bounded by how quickly data can be read from the file it is a suitable task to test the I/O capabilities of H5MDReader.

We tested the effects of HDF5 file chunking and file compression on I/O performance. In general we found that altering the stripe count and loading only necessary coordinates via masked arrays provided little improvement in benchmark times. Loading the entire trajectory into memory in one pass instead of iterating through, frame by frame, showed the greatest improvement in performance. This was compounded by our results with HDF5 chunking. Our baseline test file was auto-chunked with the auto-chunking algorithm in `h5py`. When we recast the file into a contiguous form and a custom, optimized chunk layout, we saw improvements in serial I/O on the order of 10x. Additionally, our results from applying gzip compression to the file showed no loss in performance at higher processor counts, indicating H5MD files can be compressed without losing performance in parallel analysis tasks.

Methods

HPC environments

We tested the parallel MPI I/O capabilities of our H5MD implementation on three supercomputing environments: ASU Agave, PSC Bridges, and SDSC Comet. The Agave supercomputer offers 498 compute nodes. We utilized the Parallel Compute Nodes that offer 2 Intel Xeon E5-2680 v4 CPUs (2.40GHz, 14 cores/CPU, 28 cores/node, 128GB RAM/node) with a 1.2PB scratch BeeGFS file system that uses an Intel OmniPath interconnect system. The Bridges supercomputer offers over 850 compute nodes that supply 1.3018 Pf/s and 274 TiB RAM. We utilized the Regular Shared Memory Nodes that offer 2 Intel Haswell E5-2695 v3 CPUs (2.3-3.3GHz, 14 cores/CPU, 28 cores/node, 128GB RAM/node) with a 10PB scratch Lustre parallel file system that uses an InfiniBand interconnect system. The Comet supercomputer offers 2 Pf/s with 1944 standard compute nodes. We utilized the Intel Haswell Standard Compute Nodes that offer 2 Intel Xeon E5-2680 v3 CPUs (2.5GHz, 12 cores/CPU, 24 cores/node, 128GB RAM/node) with a 13PB scratch Lustre parallel file system that also uses an InfiniBand interconnect system.

Our software library stacks were built with conda environments. Table 1 gives the versions of each library involved in the stack. We used GNU C compilers on Agave and Bridges and the Intel C-compiler on Comet for MPI parallel jobs as recommended by the Comet user guide. We used OpenMPI as the MPI implementation on all HPC resources as this was generally the recommended environment and in the past we found it also the easiest to build against [KPF+20]. The mpi4py [DPKC11] package was used to make MPI available in Python code, as required by `h5py`. In general, our software stacks were built in the following manner:

- module load anaconda3
- create new conda environment
- module load parallel hdf5 build
- module load OpenMPI implementation
- install mpi4py with `env MPICC=/path/to/mpicc pip install mpi4py`
- install h5py with `CC="mpicc" HDF5_MPI="ON" HDF5_DIR=/path/to/parallel-hdf5 pip install --no-binary=h5py h5py`
- install development MDAnalysis as outlined in the MDAnalysis User Guide

System	ASU Agave	PSC Bridges	SDSC Comet
Python	3.8.5	3.8.5	3.6.9
C compiler	gcc 4.8.5	gcc 4.8.5	icc 18.0.1
HDF5	1.10.1	1.10.2	1.10.3
OpenMPI	3.0.0	3.0.0	3.1.4
h5py	2.9.0	3.1.0	3.1.0
mpi4py	3.0.3	3.0.3	3.0.3
MDAnalysis	2.0.0-dev0	2.0.0-dev0	2.0.0-dev0

TABLE 1: *Library versions installed for each HPC environment.*

name	format	file size (GiB)
H5MD-default	H5MD	113
H5MD-chunked	H5MD	113
H5MD-contiguous	H5MD	113
H5MD-gzipx1	H5MD	77
H5MD-gzipx9	H5MD	75
DCD	DCD	113
XTC	XTC	35
TRR	TRR	113

TABLE 2: *Data files benchmarked on all three HPCS.* ***name*** *is the name that is used to identify the file in this paper.* ***format*** *is the format of the file, and* ***file size*** *gives the size of the file in gibibytes.* ***H5MD-default*** *original data file written with pyh5md which uses the auto-chunking algorithm in h5py.* ***H5MD-chunked*** *is the same file but written with chunk size (1, n atoms, 3) and* ***H5MD-contiguous*** *is the same file but written with no HDF5 chunking.* ***H5MD-gzipx1*** *and* ***H5MD-gzipx9*** *have the same chunk arrangement as* ***H5MD-chunked*** *but are written with gzip compression where 1 is the lowest level of compression and 9 is the highest level.* ***DCD****,* ***XTC****, and* ***TRR*** *are copies* ***H5MD-contiguous*** *written with MDAnalysis.*

Benchmark Data Files

The test data files used in our benchmark consist of a topology file `YiiP_system.pdb` with 111,815 atoms and a trajectory file `YiiP_system_9ns_center100x.h5md` with 90100 frames. The initial trajectory data file (H5MD-default in Table 2) was generated with pyh5md [dBCH14] using the XTC file `YiiP_system_9ns_center.xtc` [SFMLIP+19], [LRFK+21], using the "ChainReader" facility in MDAnalysis with the list `100 * ["YiiP_system_9ns_center.xtc"]` as input. The rest of the test files were copies of H5MD-default and were written with MDAnalysis using different HDF5 chunking arrangements and compression settings. Table 2 gives all of the files benchmarked with how they are identified in this paper as well as their corresponding file size.

Parallel Algorithm Benchmark

We implemented a simple split-apply-combine parallelization algorithm [Wic11], [SFMLIP+19], [KPF+20] that divides the number of frames in the trajectory evenly among all available processes. Each process receives a unique `start` and `stop` for which to iterate through their section of the trajectory. As the computational task, the root mean square distance (RMSD) of the protein C_α atoms after optimal structural superposition [MM14] is computed at each frame with the QCProt algorithm [The05], as described in our previous work [SFMLIP+19], [KPF+20].

In order to obtain detailed timing information we instrumented code as follows below. Table 3 outlines the specific lines in the code that were timed in the benchmark.

```
1 import MDAnalysis as mda
2 from MDAnalysis.analysis.rms import rmsd
3 from mpi4py import MPI
```

line number	id	description
11	$t^{\text{init\_top}}$	load topology file
12	$t^{\text{init\_traj}}$	load trajectory file
38	$t^{\text{I/O}}$	read data from time step into memory
39	t^{compute}	perform rmsd computation
42	t^{wait}	wait for processes to synchronize
47	$t^{\text{comm\_gather}}$	combine results back into root process

TABLE 3: *All timings collected from the example benchmark code.* ***id*** *gives the reference name used in this paper to reference the corresponding line number and timing collected.* ***description*** *gives a short description of what that specific line of code is doing in the benchmark.*

```
4 import numpy as np
5
6 comm = MPI.COMM_WORLD
7 size = comm.Get_size()
8 rank = comm.Get_rank()
9
10 def benchmark(topology, trajectory):
11     u = mda.Universe(topology)
12     u.load_new(trajectory,
13                driver="mpio",
14                comm=comm)
15     CA = u.select_atoms("protein and name CA")
16     x_ref = CA.positions.copy()
17
18     # make_balanced_slices divides n_frames into
19     # equally sized blocks and returns start:stop
20     # indices for each block
21     slices = make_balanced_slices(n_frames,
22                                   size,
23                                   start=0,
24                                   stop=n_frames,
25                                   step=1)
26      start = slices[rank].start
27      stop = slices[rank].stop
28      bsize = stop - start
29
30      # sendcounts is used for Gatherv() to know how
31      # many elements are sent from each rank
32      sendcounts = np.array([
33          slices[i].stop - slices[i].start
34          for i in range(size)])
35
36      rmsd_array = np.empty(bsize, dtype=float)
37      for i, frame in enumerate(range(start, stop)):
38         ts = u.trajectory[frame]
39         rmsd_array[i] = rmsd(CA.positions,
40                              x_ref,
41                              superposition=True)
42      comm.Barrier()
43      rmsd_buffer = None
44      if rank == 0:
45         rmsd_buffer = np.empty(n_frames,
46                                dtype=float)
47      comm.Gatherv(sendbuf=rmsd_array,
48                   recvbuf=(rmsd_buffer, sendcounts), root=0)
```

The HDF5 file is opened with the `mpio` driver and the `MPI.COMM_WORLD` communicator to ensure the file is accessed in parallel via MPI I/O. The topology and trajectory initialization times must be analyzed separately because the topology file is not opened in parallel and represents a fixed cost each process must pay to open the file. MDAnalysis reads data from MD trajectory files one frame, or "snapshot" at a time. Each time the `u.trajectory[frame]` is iterated through, MDAnalysis reads the file and fills in numpy arrays [HMvdW+20] corresponding to that time step. Each MPI process runs an identical copy of the script, but receives a unique `start` and `stop` variable

such that the entire file is read in parallel. Gathering the results is done collectively by MPI, which means all processes must finish their iteration blocks before the results can be returned. Therefore, it is important to measure t^{wait} as it represents the existence of "straggling" processes. If one process takes substantially longer than the others to finish its iteration block, all processes are slowed down. These 6 timings are returned and saved as an array for each benchmark run.

We applied this benchmark scheme to H5MD test files on Agave, Bridges, and Comet. Each benchmark run received a unique, freshly copied test file that was only used once so as to avoid any caching effects of the operating system or file system. We also tested three algorithmic optimizations: Lustre file striping, loading the entire trajectory into memory, and using masked arrays in numpy to only load the C_α coordinates required for the RMSD calculation. For striping, we ran the benchmark on Bridges and Comet with a file stripe count of 48 and 96. For the into memory optimization, we used `MDAnalysis.Universe.transfer_to_memory()` to read the entire file in one go and pass all file I/O to the HDF5 library. For the masked array optimization, we allowed `u.load_new()` to take a list or array of atom indices as an argument, `sub`, so that the `MDAnalysis.Universe.trajectory.ts` arrays are instead initialized as `numpy.ma.masked_array` instances and only the indices corresponding to `sub` are read from the file.

Performance was quantified by measuring the I/O timing returned from the benchmarks, and strong scaling was assessed by calculating the speedup $S(N) = t_1/t_N$ and the efficiency $E(N) = S(N)/N$.

Data Sharing

All of our SLURM submission shell scripts and Python benchmark scripts for all three HPC environments are available in the repository https://github.com/Becksteinlab/scipy2021-mpiH5MD-data and are archived under DOI 10.5281/zenodo.5083858.

Results and Discussion

Baseline Benchmarks

We first ran benchmarks with the simplest parallelization scheme of splitting the frames of the trajectory evenly among all participating processes. The H5MD file involved in the benchmarks was written with the pyh5md library, the original Python reference implementation for the H5MD format [dBCH14]. The datasets in the data file were chunked automatically by the auto-chunking algorithm in `h5py`. File I/O remains the largest contributor to the total benchmark time, as shown by Figure 1 (A). Figure 1 (B, D-F) also show that the initialization, computation, and MPI communication times are negligible with regards to the overall analysis time. t^{wait}, however, becomes increasingly relevant as the number of processes increases (Figure 1 C), indicating a growing variance in the iteration block time across all processes. In effect, t^{wait} is measuring the occurrence of "straggling" processes, which has been previously observed to be an issue on busy, multi-user HPC environments [KPF+20]. We found that the total benchmark time continues to decrease as the number of processes increases to over 100 (from 4648 $\pm$ 319 seconds at $N = 1$ to 315.6 $\pm$ 59.8 seconds at $N = 112$ on Agave) (Fig. 2 A). While the absolute time of each benchmark is important in terms of measuring the actual amount of time saved with our parallelization scheme, results are often highly variable in a crowded HPC environment [How10] and therefore we focus our analysis on the speedup and efficiency of each benchmark run. The maximum total I/O speedup observed is only 15x and efficiencies at around 0.2 (Fig. 2 B, C). The RMSD computation scaling, on the other hand, remains high, with nearly ideal scaling on Bridges and Comet, with Agave trailing behind at 71x speedup at 122 cores. Therefore, for a computationally bound analysis task, our parallel H5MD implementation will likely scale well.

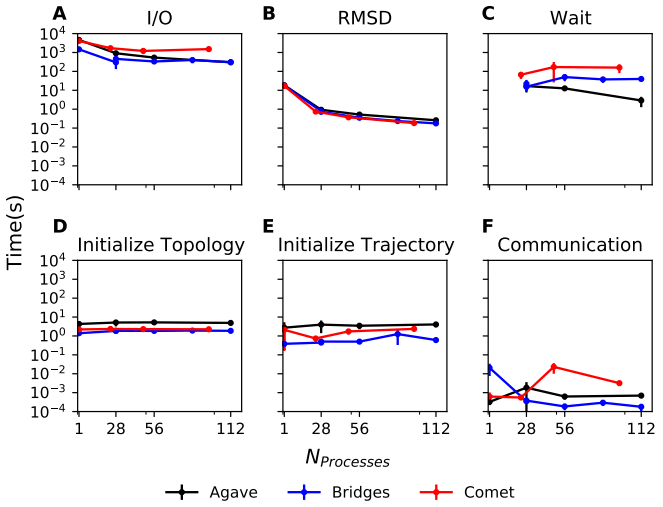

Fig. 1: *Benchmark timings breakdown for the ASU Agave, PSC Bridges, and SDSC Comet HPC clusters. The benchmark was run on up to 4 full nodes on each HPC, where $N_{processes}$ was 1, 28, 56, and 112 for Agave and Bridges, and 1, 24, 48, and 96 on Comet. The `H5MD-default` file was used in the benchmark, where the trajectory was split in N chunks for each corresponding N process benchmark. Points represent the mean over three repeats with the standard deviation shown as error bars.*

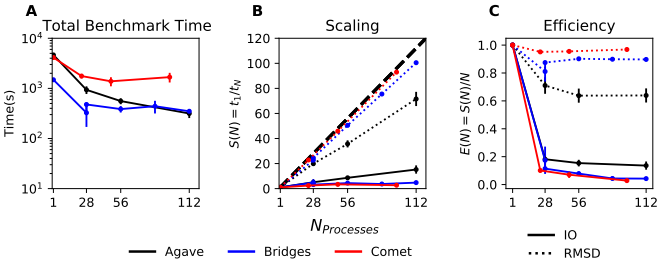

Fig. 2: *Strong scaling I/O and RMSD performance of the RMSD analysis task of the `H5MD-default` data file on Agave, Bridges, and Comet. $N_{processes}$ ranged from 1 core, to 4 full nodes on each HPC, and the number of trajectory blocks was equal to the number of processes involved. Points represent the mean over three repeats where the error bars are derived with the standard error propagation from the standard deviation of absolute times.*

Effects of Algorithmic Optimizations on File I/O

We tested three optimizations aimed at shortening file I/O time for the same data file. In an attempt to optimize I/O, we tried to minimize "wasted I/O". For example, in any analysis task, not all coordinates in the trajectory may be necessary for the computation. In our analysis test case, the RMSD was calculated for only the C_α atoms of the protein backbone, therefore the

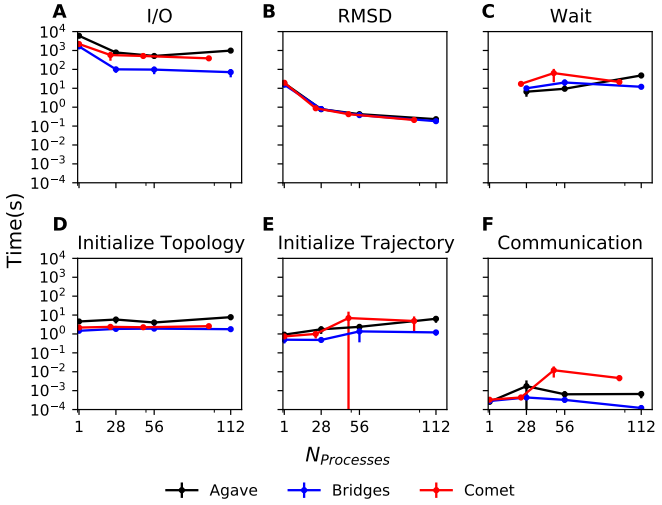

Fig. 3: Benchmark timings breakdown for the ASU Agave, PSC Bridges, and SDSC Comet HPC clusters for the `masked_array` optimization technique. The benchmark was run on up to 4 full nodes on each HPC, where N processes was 1, 28, 56, and 112 for Agave and Bridges, and 1, 24, 48, and 96 on Comet. The `H5MD-default` file was used in the benchmark, where the trajectory was split in N chunks for each corresponding N process benchmark. Points represent the mean over three repeats with the standard deviation shown as error bars.

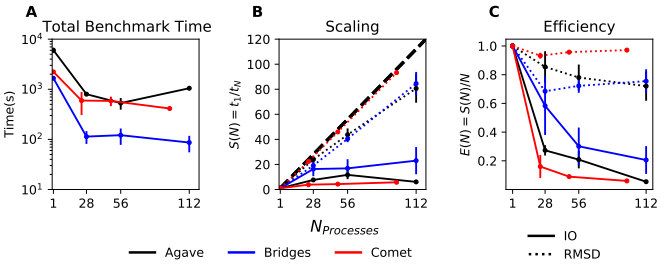

Fig. 4: Strong scaling performance of the RMSD analysis task with the `masked_array` optimization technique. The benchmark used the `H5MD-default` data file on Agave, Bridges, and Comet. $N_{processes}$ ranged from 1 core, to 4 full nodes on each HPC, and the number of trajectory blocks was equal to the number of processes involved. Points represent the mean over three repeats where the error bars are derived with the standard error propagation from the standard deviation of absolute times.

coordinates of all other atoms read from the file is essentially wasted I/O. To circumvent this issue, we implemented the use of NumPy `ma.masked_array` [HMvdW+20], where the arrays of coordinate data are instead initialized as masked arrays that only fill data from selected coordinate indices. We found that Bridges showed the best scaling with the masked array implementation, with a total scaling of 23x at 4 full nodes (1642 $\pm$ 115 seconds at $N = 1$ to 71 $\pm$ 33 seconds at $N = 112$ cores) as seen in Figure 4 (A, B). Agave showed a maximum scaling of 11x at 2 full nodes, while Comet showed 5x scaling at 4 full nodes (Figure 4 B). In some cases, the masked array implementation resulted in slower I/O times. For example, Agave went from 4623 seconds in the baseline 1 core run to 5991 seconds with masked arrays. This could be due to the HDF5 library not being optimized to work with masked arrays as with numpy arrays. On the other hand, for Bridges and Comet, we observed an approximate 5x speedup in I/O time (Fig. 4 B) for the masked array case when compared to the baseline benchmark. In terms of the RMSD computation scaling, we once again found all three systems scaled well, with Comet displaying ideal scaling all the way to 4 full nodes, while Agave and Bridges hovering around 85x at 112 cores.

With an MPI implementation, processes participating in parallel I/O communicate with one another. It is commonly understood that repeated, small file reads performs worse than a large, contiguous read of data. With this in mind, we tested this concept in our benchmark by loading the entire trajectory into memory prior to the RMSD task. Modern super computers make this possible as they contain hundreds of GiB of memory per node. On Bridges, loading into memory strangely resulted in slower I/O times (1466s baseline to 2196s at $N = 1$ and 307s baseline to 523s at $N = 112$, Fig. 1 A and Fig. 5 A). Agave and Comet, on the other hand, showed surprisingly different results. They both performed substantially better for the $N = 1$ core case. Agave's serial I/O performance was boosted from 4623s to 891s (Fig. 5 A) by loading the data into memory in one slurp rather than iterating through the trajectory frame by frame. Similarly, Comet's serial I/O performance went from 4101s to 1740s, with multi-node performance continuing to show improvement versus the baseline numbers (excluding the peak at $N = 48$). Agave steady improvements in performance all the way to 4 full nodes, where the I/O time reached 73s (Fig. 5 A, Fig. 6 A). Figure 7 gives a direct comparison on Agave of the baseline benchmark performance with the two optimization methods outlined. With respect to the baseline serial performance, loading into memory gives a 91x speedup (4658s at 1 core to 73s at 112 cores) (Figure 7, A). This result was interesting in that the only difference between the two was the access pattern of the data - in one case, the file was read in small repeated bursts, while in the other the file was read from start to finish with HDF5. We hypothesized that this was due to layout of the file itself on disk.

Also, we found that the t^{wait} does not increase as the number of processes increases as in all of the other benchmark cases (Figure 5 C). In the other benchmarks, t^{wait} was typically on the order of 10-200 seconds, whereas t^{wait} on the order of 0.01 seconds for the memory benchmarks (Figure 7 C). This indicates that the cause of the iteration block time variance among processes stems from MPI rank coordination when many small read requests are made.

To investigate MPI rank competition, we increased the stripe count on Bridge's and Comet's Lustre file system up to 96, where found marginal I/O scaling improvements of 1.2x on up to 4 full nodes (not shown). While our data showed no improvement with altering the stripe count, this may have been a byproduct the poor chunk layout of the original file on disk. In the next section we discuss the effects of HDF5 chunking on I/O performance.

Effects of HDF5 Chunking on File I/O

To test the hypothesis that the increase in serial file I/O between the baseline performance in loading into memory performance was caused by the layout of the file on disk, we created *H5MDWriter*, an MDAnalysis file format writer class that gives one the ability to write H5MD files with the MDAnalysis user interface. These files can be written with user-decided custom chunk layouts, file compression settings, and can be opened with MPI parallel drivers that enable parallel writing. We ran some initial serial writing tests and found that writing from DCD, TRR, and XTC to H5MD

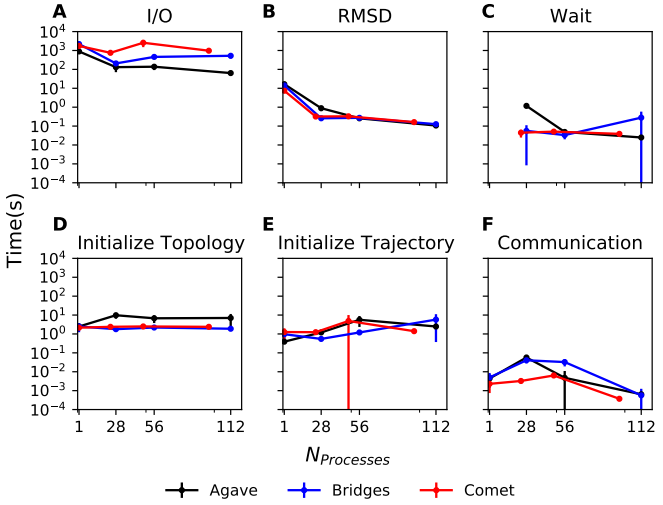

Fig. 5: *Benchmark timings breakdown for the ASU Agave, PSC Bridges, and SDSC Comet HPC clusters for the loading-into-memory optimization technique. The benchmark was run on up to 4 full nodes on each HPC, where N processes was 1, 28, 56, and 112 for Agave and Bridges, and 1, 24, 48, and 96 on Comet. The* `H5MD-default` *file was used in the benchmark, where the trajectory was split in N chunks for each corresponding N process benchmark. Points represent the mean over three repeats with the standard deviation shown as error bars.*

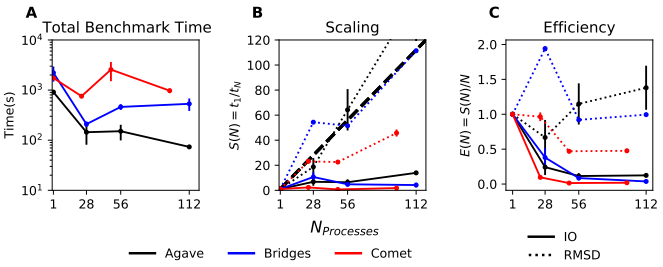

Fig. 6: *Strong scaling I/O performance of the RMSD analysis task with the loading-into-memory optimization technique. The benchmark used the* `H5MD-default` *data file on Agave, Bridges, and Comet. $N_{processes}$ ranged from 1 core, to 4 full nodes on each HPC, and the number of trajectory blocks was equal to the number of processes involved. Points represent the mean over three repeats where the error bars are derived with the standard error propagation from the standard deviation of absolute times.*

typically took ~360 seconds on Agave. For the 113 GiB test file, this was a 0.31 GiB/s write bandwidth. We rewrote the H5MD-default test file and tested two cases: one in which the file is written with no chunking applied (H5MD-contiguous), and one in which we applied a custom chunk layout to match the access pattern on the file (H5MD-chunked). Our benchmark follows a common MD trajectory analysis scheme in that it iterates through the trajectory one frame at a time. Therefore, we applied a chunk shape of `(1, n atoms, 3)` which matched exactly the shape of data to be read at each iteration step. An important feature of HDF5 chunking to note is that, for any element in a chunk to be read, the **entire** chunk must be read. When we investigated the chunk shape of the H5MD-default that was auto-chunked with h5py's chunking algorithm, we found that each chunk contained data elements from multiple different time steps. This means,

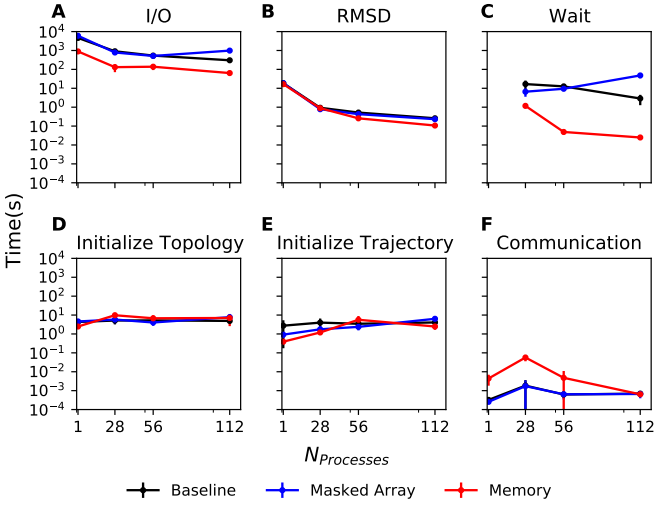

Fig. 7: *Benchmark timings on ASU Agave comparing the baseline benchmark with the masked array and loading into memory optimizations. Each benchmark was run on up to 4 full nodes where N processes was 1, 28, 56, and 112. The* `H5MD-default` *test file was used in all benchmarks. Points represent the mean over three repeats with the standard deviation shown as error bars.*

for every time step of data read, an exorbitant amount of excess data was being read and discarded at each iteration step. Before approaching the parallel tests, we tested how the chunk layout affects baseline serial I/O performance for the file. We found I/O performance strongly depends on the chunk layout of the file on disk. The auto-chunked H5MD-default file I/O time was 4101s, while our custom chunk layout resulted in an I/O time of 460s (Figure 8). So, we effectively saw a 10x speedup just from optimizing the chunk layout alone, where even the file with no chunking applied showed similar improvements in performance. In our previous serial I/O tests, we found that H5MD performed worse than other file formats, so we repeated those tests with our custom chunked file, H5MD-chunked. We found for our test file of 111,815 atoms and 90100 frames, H5MD outperformed XTC and TRR, while performing equally well to the DCD file, an encouraging result (Fig. 9).

Next, we investigated what effect the chunk layout had on parallel I/O performance. We repeated our benchmarks on Agave (at this point, Bridges had been decommissioned and our Comet allocation had expired) but with the H5MD-chunked and H5MD-contiguous data files. For the serial one process case, we found a similar result in that the I/O time was dramatically increased with an approximate 10x speedup for both the contiguous and chunked file, with respect to the baseline benchmark (Figure 10 A). The rest of the timings remained unaffected (Figure 10 B-F). Although the absolute total benchmark time is much improved (Figure 11 A), the scaling remains challenging, with a maximum observed speedup of 12x for the contiguous file (Figure 11 B). The $N = 112$ H5MD-contiguous run's I/O time was 47s (Fig. 10 A). When compared to the 4623s baseline serial time, this is a 98x speedup. Similarly, the H5MD-chunked 4 node run resulted in an I/O time of 83s, which is a 56x speedup when compared to baseline serial performance. Therefore, the boost in performance seen by loading the H5MD-default trajectory into memory rather than iterating frame by frame is indeed most likely due to the original file's chunk layout. This emphasizes the point that one

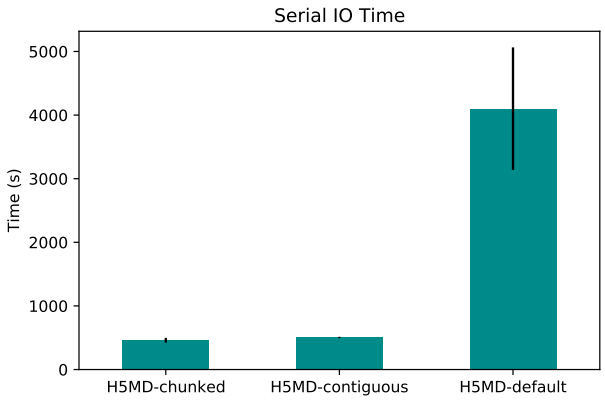

Fig. 8: *Serial I/O time for H5MD-default, H5MD-contiguous, and H5MD-chunked data files. Each file contained the same data (113 GiB, 90100 frames) but was written with a different HDF5 chunk arrangement, as outlined in Table 2. Each bar represents the mean of 5 repeat benchmark runs, with the standard deviation shown as error bars.*

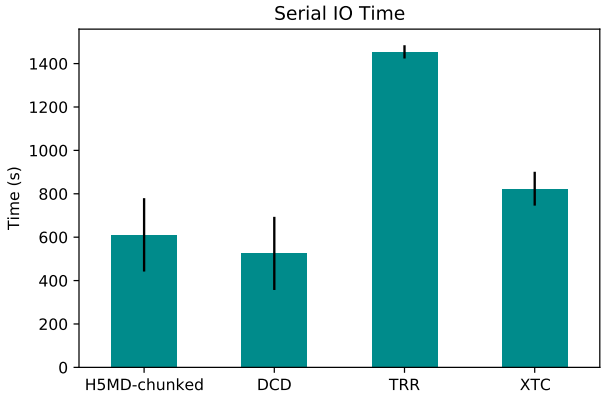

Fig. 9: *Comparison of serial I/O time for various popular MD file formats. All files contain the same amount of data (90100 frames). Each bar represents the mean of 10 repeat benchmark runs, with the standard deviation shown as error bars.*

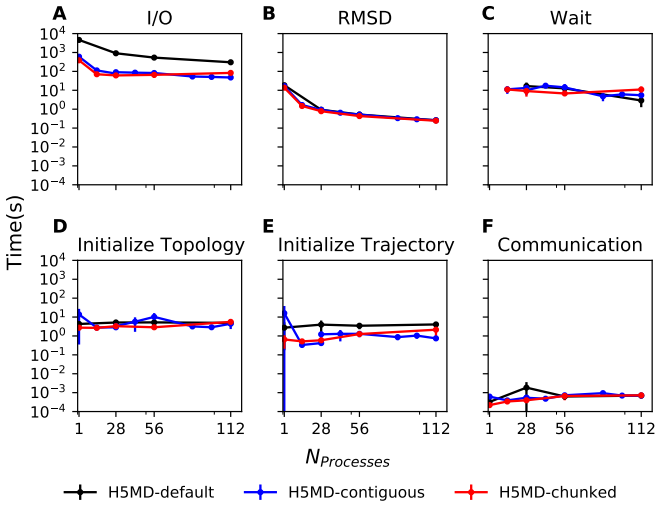

Fig. 10: *Benchmark timings breakdown on ASU Agave for the three chunk arrangements tested. The benchmark was run on up to 4 full nodes on each HPC, where N processes was 1, 28, 56, and 112.* ***H5MD-default*** *was auto-chunked by* `h5py`. ***H5MD-contiguous*** *was written with no chunking applied, and* ***H5MD-chunked*** *was written with a chunk shape of* `(1, n atoms, 3)`. *The trajectory was split in N chunks for each corresponding N process benchmark. Points represent the mean over three repeats with the standard deviation shown as error bars.*

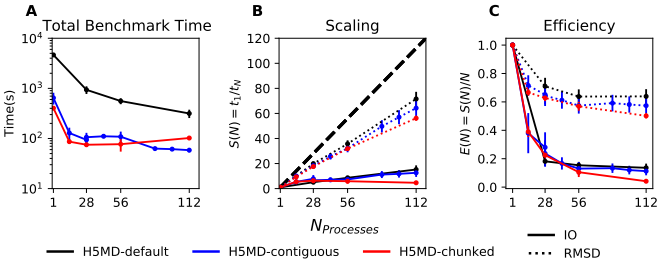

Fig. 11: *Strong scaling I/O performance of the RMSD analysis task with various chunk layouts tested on ASU Agave.* $N_{processes}$ *ranged from 1 core, to 4 full nodes, and the number of trajectory blocks was equal to the number of processes involved. Points represent the mean over three repeats where the error bars are derived with the standard error propagation from the standard deviation of absolute times.*

may garner substantial I/O improvements if one thinks carefully not only about how their algorithm accesses the file, but also how the file is actually stored on disk. The relationship between layout on disk and disk access pattern is crucial for optimized I/O. Furthermore, as the auto-chunked layout of the `H5MD-default` file scattered data from a single time step across multiple chunks, it is very likely that these chunks themselves were also scattered across stripes. In this case, multiple processes are still attempting to read from the same chunk which would nullify any beneficial effect striping has on file contention. We would have liked to further test the effects of striping with a proper chunk layout, but our XSEDE allocation expired.

Effects of HDF5 GZIP Compression on File I/O

HDF5 files also offer the ability to compress the files. With our writer, users are easily able to apply any of the compression settings allowed by HDF5. To see how compression affected parallel I/O, we tested HDF5's gzip compression with a minimum setting of 1 and a maximum setting of 9. In the serial 1 process case, we found that I/O performance is slightly hampered, with I/O times approximately 4x longer with compression applied (Figure 13 A) This is expected as you are giving up disk space for the time it takes to decompress the file, as is seen in the highly compressed XTC format (Fig. 9). However, at increasing number of processes ($N > 28$), we found that this difference disappears (Figure 13 A and Figure 12 A). This shows a clear benefit of applying gzip compression to a chunked HDF5 file for parallel analysis tasks, as the compressed file is ~2/3 the size of the original. Additionally we found speedups of up to 36x on 2 full nodes for the compressed data file benchmarks (Figure 13 B), although we recognize this number is slightly inflated due to the slower serial I/O time. From this data we can safely assume that H5MD files can be compressed without fear of losing parallel I/O performance, which is a nice boon in the age of terabyte sized trajectory files.

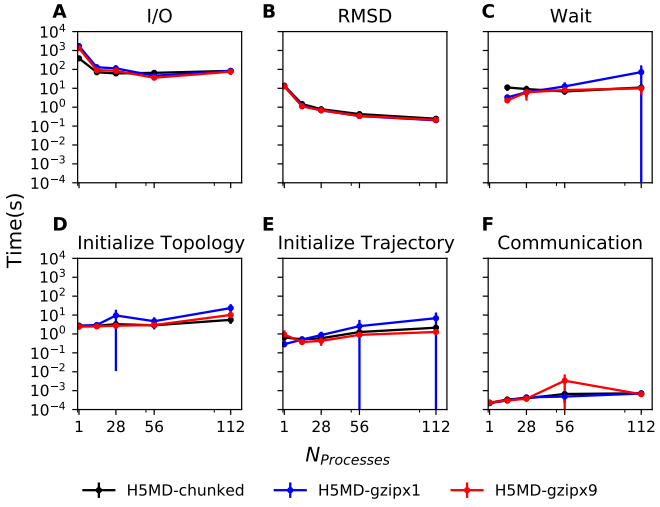

Fig. 12: *Benchmark timings breakdown on ASU Agave for the minimum gzip compression 1 and maximum gzip compression 9. The benchmark was run on up to 4 full nodes on each HPC, where N processes was 1, 28, 56, and 112. The trajectory was split in N chunks for each corresponding N process benchmark. Points represent the mean over three repeats with the standard deviation shown as error bars.*

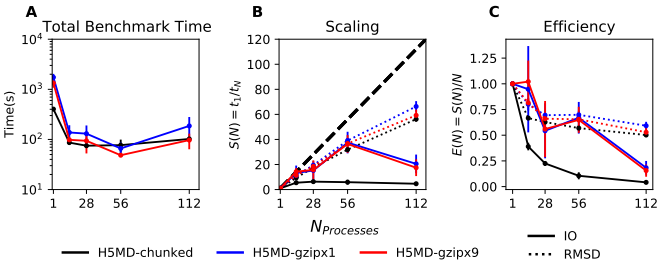

Fig. 13: *Strong scaling I/O performance of the RMSD analysis task with minimum and maximum gzip compression applied. $N_{processes}$ ranged from 1 core, to 4 full nodes, and the number of trajectory blocks was equal to the number of processes involved. Points represent the mean over three repeats where the error bars are derived with the standard error propagation from the standard deviation of absolute times.*

Conclusions

The growing size of trajectory files demands parallelization of trajectory analysis. However, file I/O has become a bottleneck in the workflow of analyzing simulation trajectories. Our implementation of an HDF5-based file format trajectory reader in MDAnalysis can perform parallel MPI I/O, and our benchmarks on various national HPC environments show that speed-ups on the order of 20x for 48 cores are attainable. Scaling up to achieve higher parallel data ingestion rates remains challenging, so we developed several algorithmic optimizations in our analysis workflows that lead to improvements in I/O times. The results from these optimization attempts led us to find that the our original data file that was auto-chunked by h5py's chunking algorithm had an incredibly inefficient chunk layout of the original file. With a custom, optimized chunk layout and gzip compression, we found maximum scaling of 36x on 2 full nodes on Agave. In terms of speedup with respect to the file chunked automatically, our properly chunked file led to I/O time speedups of 98x at 112 cores on Agave, which means carefully thinking not only about how your file is accessed, but also how the file is stored on disk can result in a reduction of analysis time from 4623 to 47 seconds. To garner further improvements in parallel I/O performance, a more sophisticated I/O pattern may be required, such as two-phase MPI I/O or carefully synchronizing chunk sizes with Lustre stripes. The addition of the HDF5 reader provides a foundation for the development of parallel trajectory analysis with MPI and the MDAnalysis package.

Acknowledgments

The authors thank Dr. Pierre de Buyl for advice on the implementation of the h5md format reading code and acknowledge Gil Speyer and Jason Yalim from the Research Computing Core Facilities at Arizona State University for support with the Agave cluster and BeeGFS. This work was supported by the National Science Foundation through a REU supplement to award ACI1443054 and used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. The SDSC Comet computer at the San Diego Supercomputer Center and PSC Bridges computer at the Pittsburgh Supercomputing Center were used under allocation TG-MCB130177. The authors acknowledge Research Computing at Arizona State University for providing HPC and storage resources that contributed to the research results reported within this paper.

References

[BFJ18] Oliver Beckstein, Geoffrey Fox, and Shantenu Jha. Convergence of data generation and analysis in the biomolecular simulation community. In *Online Resource for Big Data and Extreme-Scale Computing Workshop*, page 4, November 2018. URL: https://www.exascale.org/bdec/sites/www.exascale.org.bdec/files/whitepapers/Beckstein-Fox-Jha_BDEC2_WP_0.pdf.

[CDG+21] Lorenzo Casalino, Abigail C Dommer, Zied Gaieb, Emilia P Barros, Terra Sztain, Surl-Hee Ahn, Anda Trifan, Alexander Brace, Anthony T Bogetti, Austin Clyde, Heng Ma, Hyungro Lee, Matteo Turilli, Syma Khalid, Lillian T Chong, Carlos Simmerling, David J Hardy, Julio DC Maia, James C Phillips, Thorsten Kurth, Abraham C Stern, Lei Huang, John D McCalpin, Mahidhar Tatineni, Tom Gibbs, John E Stone, Shantenu Jha, Arvind Ramanathan, and Rommie E Amaro. AI-driven multiscale simulations illuminate mechanisms of SARS-CoV-2 spike dynamics. *The International Journal of High Performance Computing Applications*, page 10943420211006452, April 2021. Publisher: SAGE Publications Ltd STM. URL: https://doi.org/10.1177/10943420211006452, doi:10.1177/10943420211006452.

[Col14] Andrew Collette. Python and hdf5. In Meghan Blanchette and Rachel Roumeliotis, editors, *Python and HDF5*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., 2014.

[CR15] T. Cheatham and D. Roe. The impact of heterogeneous computing on workflows for biomolecular simulation and analysis. *Computing in Science Engineering*, 17(2):30–39, 2015. doi:10.1109/MCSE.2015.7.

[dBCH14] Pierre de Buyl, Peter H. Colberg, and Felix Höfling. H5MD: A structured, efficient, and portable file format for molecular data. *Computer Physics Communications*, 185(6):1546 – 1553, 2014. doi:10.1016/j.cpc.2014.01.018.

[DDG+12] Ron O Dror, Robert M Dirks, J P Grossman, Huafeng Xu, and David E Shaw. Biomolecular simulation: a computational microscope for molecular biology. *Annu Rev Biophys*, 41:429–52, 2012. doi:10.1146/annurev-biophys-042910-155245.

[DPKC11] Lisandro D. Dalcín, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124 – 1139, 2011. New Computational Methods and Software Tools. doi:10.1016/j.advwatres.2011.04.013.

[FQC+19] Geoffrey Fox, Judy Qiu, David Crandall, Gregor von Laszewski, Oliver Beckstein, John Paden, Ioannis Paraskevakos, Shantenu Jha, Fusheng Wang, Madhav Marathe, Anil Vullikanti, and III Cheatham, Thomas E. Contributions to high-performance big data computing. In L. Grandinetti, G. R. Joubert, K. Michielsen, S. L. Mirtaheri, M. Taufer, and R Yokota, editors, *Future Trends of HPC in a Disruptive Scenario*, volume 34 of *Advances in Parallel Computing*, pages 34–81. IOS Press, 2019. doi:10.3233/APC190005.

[GLB+16] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, David L Dotson, Jan Domański, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A Python package for the rapid analysis of molecular dynamics simulations. In Sebastian Benthall and Scott Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 98–105, Austin, TX, 2016. SciPy. doi:10.25080/Majora-629e541a-00e.

[HBD+19] David J. Huggins, Philip C. Biggin, Marc A. Dämgen, Jonathan W. Essex, Sarah A. Harris, Richard H. Henchman, Syma Khalid, Antonija Kuzmanic, Charles A. Laughton, Julien Michel, Adrian J. Mulholland, Edina Rosta, Mark S. P. Sansom, and Marc W. van der Kamp. Biomolecular simulations: From dynamics and mechanisms to computational assays of biological activity. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 9(3):e1393, 2019. doi:10.1002/wcms.1393.

[HMvdW+20] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández Del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E Oliphant. Array programming with numpy. *Nature*, 585(7825):357–362, 09 2020. doi:10.1038/s41586-020-2649-2.

[How10] Mark Howison. Tuning HDF5 for Lustre File Systems. September 2010. URL: https://escholarship.org/uc/item/46r9d86r.

[KPF+20] Mahzad Khoshlessan, Ioannis Paraskevakos, Geoffrey C. Fox, Shantenu Jha, and Oliver Beckstein. Parallel performance of molecular dynamics trajectory analysis. *Concurrency and Computation: Practice and Experience*, 32:e5789, 2020. doi:10.1002/cpe.5789.

[LRFK+21] Maria Lopez-Redondo, Shujie Fan, Akiko Koide, Shohei Koide, Oliver Beckstein, and David L. Stokes. Zinc binding alters the conformational dynamics and drives the transport cycle of the cation diffusion facilitator YiiP. *Journal of General Physiology*, 153(8), July 2021. URL: https://doi.org/10.1085/jgp.202112873, doi:10.1085/jgp.202112873.

[MADWB11] Naveen Michaud-Agrawal, Elizabeth Jane Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. *J Comp Chem*, 32:2319–2327, 2011. doi:10.1002/jcc.21787.

[MM14] Cameron Mura and Charles E. McAnany. An introduction to biomolecular simulations and docking. *Molecular Simulation*, 40(10-11):732–764, 2014. doi:10.1080/08927022.2014.935372.

[Oro14] Modesto Orozco. A theoretical view of protein dynamics. *Chem. Soc. Rev.*, 43:5051–5066, 2014. doi:10.1039/C3CS60474H.

[PLK+18] Ioannis Paraskevakos, Andre Luckow, Mahzad Khoshlessan, Goerge Chantzialexiou, Thomas E. Cheatham, Oliver Beckstein, Geoffrey Fox, and Shantenu Jha. Task-parallel analysis of molecular dynamics trajectories. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA*, page Article No. 49, New York, NY, USA, August 13–16 2018. Association for Computing Machinery, ACM. doi:10.1145/3225058.3225128.

[SFMLIP+19] Shujie Fan, Max Linke, Ioannis Paraskevakos, Richard J. Gowers, Michael Gecht, and Oliver Beckstein. PMDA - Parallel Molecular Dynamics Analysis. In Chris Calloway, David Lippa, Dillon Niederhut, and David Shupe, editors, *Proceedings of the 18th Python in Science Conference*, pages 134 – 142, Austin, TX, 2019. SciPy. doi:10.25080/Majora-7ddc1dd1-013.

[TCD+14] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkins-Diehr. XSEDE: Accelerating scientific discovery. *Computing in Science & Engineering*, 16(5):62–74, Sept.-Oct. 2014. doi:10.1109/MCSE.2014.80.

[The05] Douglas L Theobald. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. *Acta Crystallogr A*, 61(Pt 4):478–80, Jul 2005. doi:10.1107/S0108767305015266.

[Wic11] Hadley Wickham. The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1):1–29, 2011. doi:10.18637/jss.v040.i01.