PMDA - Parallel Molecular Dynamics Analysis

Shujie Fan[¶†], Max Linke[‖†], Ioannis Paraskevakos[**], Richard J. Gowers[‡§], Michael Gecht[‖], Oliver Beckstein[¶*]

Abstract—MDAnalysis is an object-oriented Python library to analyze trajectories from molecular dynamics (MD) simulations in many popular formats. With the development of highly optimized MD software packages on high performance computing (HPC) resources, the size of simulation trajectories is growing up to many terabytes in size. However efficient usage of multicore architecture is a challenge for MDAnalysis, which does not yet provide a standard interface for parallel analysis. To address the challenge, we developed PMDA, a Python library that builds upon MDAnalysis to provide parallel analysis algorithms. PMDA parallelizes common analysis algorithms in MDAnalysis through a task-based approach with the Dask library. We implement a simple split-apply-combine scheme for parallel trajectory analysis. The trajectory is split into blocks, analysis is performed separately and in parallel on each block ("apply"), then results from each block are gathered and combined. PMDA allows one to perform parallel trajectory analysis with pre-defined analysis tasks. In addition, it provides a common interface that makes it easy to create user-defined parallel analysis modules. PMDA supports all schedulers in Dask, and one can run analysis in a distributed fashion on HPC machines, ad-hoc clusters, a single multi-core workstation or a laptop. We tested the performance of PMDA on single node and multiple nodes on a national supercomputer. The results show that parallelization improves the performance of trajectory analysis and, depending on the analysis task, can cut down time to solution from hours to minutes. Although still in alpha stage, it is already used on resources ranging from multi-core laptops to XSEDE supercomputers to speed up analysis of molecular dynamics trajectories. PMDA is available as open source under the GNU General Public License, version 2 and can be easily installed via the `pip` and `conda` package managers.

Index Terms—Molecular Dynamics Simulations, High Performance Computing, Python, Dask, MDAnalysis

Introduction

Classical molecular dynamics (MD) simulations have become an invaluable tool to understand the function of biomolecules [KM02], [DDG+12], [SB14], [Oro14], [BLL18], [HBD+19] (often with a view towards drug discovery [BS12]) and diverse problems in materials science [Rot09], [LS15], [VMMC+15], [LJYH18], [KAHC18], [FPM18]. Systems are modeled as particles (for example, atoms) whose interactions are approximated with a classical potential energy function [FS02], [BGM+18]. Forces on the particles are derived from the potential and Newton's equations of motion for the particles are solved with an integrator algorithm, typically using highly optimized MD codes that run on high performance computing (HPC) resources or workstations (often equipped with GPU accelerators). The resulting trajectories, the time series of particle positions $\mathbf{r}(t)$ (and possibly velocities), are analyzed with statistical mechanics approaches [Tuc10], [BGM+18] to obtain predictions or to compare to experimentally measured quantities. Currently simulated systems may contain millions of atoms and the trajectories can consist of hundreds of thousands to millions of individual time frames, thus resulting in file sizes ranging from tens of gigabytes to tens of terabytes. Processing and analyzing these trajectories is increasingly becoming a rate limiting step in computational workflows [CR15], [BFJ18]. Modern MD packages are highly optimized to perform well on current HPC clusters with hundreds of cores such as the XSEDE supercomputers [TCD+14] but current general purpose trajectory analysis packages [Gio19] were not designed with HPC in mind.

In order to scale up trajectory analysis from workstations to HPC clusters with the MDAnalysis Python library [MADWB11], [GLB+16] we leveraged Dask [Roc15], [Das16], a task-graph parallel framework, together with Dask's various schedulers (in particular *distributed*), and created the *Parallel MDAnalysis* (PMDA) library. By default, PMDA follows a simple split-apply-combine [Wic11] approach for trajectory analysis, whereby each task analyzes a single trajectory segment and reports back the individual results that are then combined into the final result [KPJB17]. Our previous work established that Dask worked well with MDAnalysis [KPJB17] and that this approach was competitive with other task-parallel approaches [PLK+18]. However, we did not provide a general purpose framework to write parallel analysis tools with MDAnalysis. Here we show how the split-apply-combine approach lends itself to a generalizable Python implementation that makes it straightforward for users to implement their own parallel analysis tools. At the heart of PMDA is the idea that the user only needs to provide a function that analyzes a single trajectory frame. PMDA provides the remaining framework via the `ParallelAnalysisBase` class to split the trajectory, apply the user's function to trajectory frames, run the analysis in parallel via Dask/*distributed*, and combines the data. It also contains a growing library of ready-to-use analysis classes, thus enabling users to immediately accelerate analysis that they previously performed in serial with the standard MDAnalysis analysis classes [GLB+16].

Methods

At the core of PMDA is the idea that a common interface makes it easy to create code that can be easily parallelized, especially

† *These authors contributed equally.*
¶ *Arizona State University*
‖ *Max Planck Institute of Biophysics*
** *Rutgers University*
‡ *University of New Hampshire*
§ *present address: NextMove Software Ltd.*
* *Corresponding author: obeckste@asu.edu*

Copyright © 2019 Shujie Fan et al. This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

if the analysis can be split into independent work over multiple trajectory slices and a final step, in which all data from the trajectory slices are combined. We first describe typical steps in analyzing MD trajectories and then outline the approach taken in PMDA.

Trajectory analysis

A trajectory with T saved time steps consists of a sequence of coordinates $\{(\mathbf{r}_1(t), \mathbf{r}_2(t), \ldots \mathbf{r}_N(t))\}_{1 \le t \le T}$ where $\mathbf{r}_i(t)$ are the Cartesian coordinates of particle i at time step t with N particles in the simulated system, i.e., $T \times N \times 3$ floating point numbers in total. To simplify notation, we consider t as an integer that indexes the trajectory frames; each frame index corresponds to a physical time in the trajectory that we could obtain if needed. In general, the coordinates are passed to a function $\mathscr{A}(\{\mathbf{r}_i(t)\})$ to compute a time-dependent quantity

$$A(t) = \mathscr{A}(\{\mathbf{r}_i(t)\}). \tag{1}$$

This quantity does not have to be a simple scalar; it may be a vector or a function of another parameter. In many cases, the *time series* $A(t)$ is the desired result. It is, however, also common to perform some form of *reduction* on the data, which can be as simple as a time average to compute a thermodynamic average $\langle A \rangle \equiv \bar{A} = T^{-1} \sum_{t=1}^{T} A(t)$. Such an average can be easily calculated in a post-analysis step after the time series has been obtained. An example of a more complicated reduction is the calculation of a histogram such as a radial distribution function (RDF) [FS02], [Tuc10] between two types of particles with numbers N_a and N_b,

$$g(r) = \left\langle \frac{1}{N_a N_b} \sum_{i=1}^{N_a} \sum_{j=1}^{N_b} \delta(|\mathbf{r}_{a,i} - \mathbf{r}_{b,j}| - r) \right\rangle \tag{2}$$

where the Dirac delta function counts the occurrences of particles i and j at distance r. To compute a RDF, we could generate a time series of histograms along the spatial coordinate r, i.e., $A(t;r)$ for each frame, and then perform the average in post-analysis. However, storage of such histograms becomes problematic, especially if instead of 1-dimensional RDFs, densities on 3-dimensional grids are being calculated. It is therefore better to reformulate the algorithm to perform a partial average (or reduction) during the analysis on a per-frame basis. For histograms, this could mean building a partial histogram and updating counts in the bins after every frame. PMDA supports the simple time series data collection and the per-frame reduction.

Split-apply-combine

The *split-apply-combine* strategy can be thought of as a simplified map-reduce [Wic11] that provides a conceptually simple approach to operate on data in parallel. It is based on the fundamental assumption that the data can be partitioned into blocks that can be analyzed independently. The trajectory is split along the time axis into M blocks of approximately equal size, $\tau = T/M$. One trajectory block can be viewed as a slice of a trajectory, e.g., for block k, $\{(\mathbf{r}_1(t), \mathbf{r}_2(t), \ldots \mathbf{r}_N(t))\}_{t_k \le t < t_k + \tau_k}$ with τ_k frames in the block. Each block k is analyzed in parallel by applying the function $\mathscr{A}$ to the frames in each block. Finally, the results from all blocks are gathered and combined.

The advantage of this approach is its simplicity. Many typical analysis tasks are based on calculations of time series from single trajectory frames as in Eq. 1 and it is this calculation that varies from task to task while the book-keeping and trajectory slicing

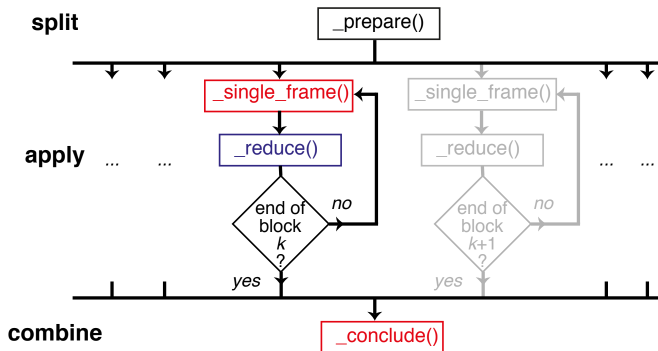

Fig. 1: High-level view of the split-apply-combine algorithm in PMDA. Steps are labeled with the methods in `pmda.parallel.ParallelAnalysisBase` that perform the corresponding function. Methods in red (`_single_frame()` and `_conclude()`) must be implemented for every analysis function because they are not general. The blue method `_reduce()` must be implemented unless a simple time series is being calculated. The `_prepare()` method is optional and provides a hook to initialize custom data structures.

is the same. Given a function $\mathscr{A}$ that performs the *single frame calculation*, PMDA provides code to perform the other necessary steps (Fig. 1).

As explained in more detail later, a class derived from `pmda.parallel.ParallelAnalysisBase` encapsulates one trajectory analysis calculation. Individual methods correspond to different steps and in the following (and in Fig. 1) we will mention the names of the relevant methods to make clear how PMDA abstracts parallel analysis. The calculation with M parallel workers is *prepared* by setting up data structures to hold the final result (method `_prepare()`). The indices for the M trajectory slices are created in such a way that the number of frames τ_k are balanced and do not differ by more than 1. For each slice or block k, the *single frame* analysis function $\mathscr{A}$ (`_single_frame()`) is sequentially applied to all frames in the slice. The result, $A(t)$, is *reduced*, i.e., added to the results for this block. For time series, $A(t)$ is simply appended to a list to form a partial time series for the block. More complicated reductions (method `_reduce()`) can be implemented, for example, the data may be histogrammed and added to a partial histogram for the block (as necessary for the implementation of the parallel RDF Eq. 2).

Implementation

PMDA is written in Python and, through MDAnalysis [GLB+16], reads trajectory data from the file system into NumPy arrays [Oli07], [VDWCV11]. Dask's `delayed()` function is used to build a task graph that is then executed using any of the schedulers available to Dask [Das16].

MDAnalysis combines a trajectory file (frames of coordinates that change with time) and a topology file (list of particles, their names, charges, bonds — all information that does not change with time) into a `Universe(topology, trajectory)` object. Arbitrary selections of particles (often atoms) are made available as an `AtomGroup` and the common approach in MDAnalysis is to work with these objects [GLB+16]; for instance, all coordinates of an `AtomGroup` with N atoms named `protein` are accessed as the $N \times 3$ NumPy array `protein.positions`.

pmda.parallel.ParallelAnalysisBase is the base class for defining a split-apply-combine parallel multi frame analysis in PMDA. It requires a Universe to operate on and any AtomGroup instances that will be used. A parallel analysis class must be derived from ParallelAnalysisBase and at a minimum, must implement the _single_frame(ts, agroups) and _conclude() methods. The arguments of _single_frame(ts, agroups) are a MDAnalysis Timestep instance and a tuple of AtomGroup instances so that the following code could be run (the code is a simplified version of the current implementation):

```
1 @delayed
2 def analyze_block(blockslice):
3     result = []
4     for ts in u.trajectory[blockslice]:
5         A = self._single_frame(ts, agroups)
6         result.append(A)
7     return result
```

The task graph is constructed by wrapping the above code into delayed() and appending a delayed instance for each trajectory slice to a (delayed) list:

```
7 blocks = delayed([analyze_block(blockslice)
8                   for blockslice in slices])
9 results = blocks.compute(**scheduler_kwargs)
```

Calling the compute() method of the delayed list object hands the task graph over to the scheduler, which then executes the graph on the available Dask workers. For example, the *multiprocessing* scheduler can be used to parallelize task graph execution on a single multiprocessor machine while the *distributed* scheduler is used to run on multiple nodes of a HPC cluster. After all workers have finished, the variable results contains a list of results from the individual blocks. PMDA actually stores these raw results as ParallelAnalysisBase._results and leaves it to the _conclude() method to process the results; this can be as simple as numpy.hstack(self._results) to generate a time series by concatenating the individual time series from each block.

The default _reduce() method appends the results and is equivalent to line 6. In general, line 6 reads

```
6         result = self._reduce(result, A)
```

where variable result should have been properly initialized in _prepare(). In order to be parallelizable, the _reduce() method must be a static method that does not access any class variables but returns its modified first argument. For example, the default "append" reduction is

```
@staticmethod
def _reduce(res, result_single_frame):
    res.append(result_single_frame)
    return res
```

In general, the ParallelAnalysisBase controls access to instance attributes via a context manager ParallelAnalysisBase.readonly_attributes(). It sets them to "read-only" for all parallel parts to prevent the common mistake to set an instance attribute in a parallel task, which breaks under parallelization as the value of an attribute in an instance in a parallel process is never communicated back to the calling process.

Using PMDA

PMDA allows one to perform parallel trajectory analysis with pre-defined analysis tasks. In addition, it provides a common interface that makes it easy to create user-defined parallel analysis modules. Here, we will introduce some basic usages of PMDA.

Pre-defined Analysis

PMDA contains a growing number of pre-defined analysis classes that are modeled after functionality in MDAnalysis.analysis and that can be used right away. Current examples are pmda.rms for RMSD analysis, pmda.contacts for native contacts analysis, pmda.rdf for radial distribution functions, and pmda.leaflet for the LeafletFinder analysis tool [MADWB11], [PLK+18] for the topological analysis of lipid membranes. While the first three modules are based on pmda.parallel.ParallelAnalysisBase as described above and follow the strict split-apply-combine approach, pmda.leaflet is an example of a more complicated task-based algorithm that can also easily be implemented with MDAnalysis and Dask [PLK+18]. All PMDA classes can be used in a similar manner to classes in MDAnalysis.analysis, which makes it easy for users of MDAnalysis to switch to parallelized versions of the algorithms. One example is the calculation of the root mean square distance (RMSD) of C_α atoms of the protein with pmda.rms.RMSD. An analysis class object is instantiated with the necessary input data such as the AtomGroup containing the C_α atoms and a reference structure. To perform the analysis, the run() method is called.

```
import MDAnalysis as mda
from pmda import rms
# Create a Universe based on simulation topology
# and trajectory
u = mda.Universe(top, trj)

# Select all the C alpha atoms
ca = u.select_atoms('name CA')

# Take the initial frame as the reference
u.trajectory[0]
ref = u.select_atoms('name CA')

# Build the parallel rms object, and run
# the analysis with 4 workers and 4 blocks.
rmsd = rms.RMSD(ca, ref)
rmsd.run(n_jobs=4, n_blocks=4)

# The results can be accessed in rmsd.rmsd.
print(rmsd.rmsd)
```

Here the only difference between using the serial version and the parallel version is that the run() method takes additional arguments n_jobs and n_blocks, which determine the level of parallelization. When using the *multiprocessing* scheduler (the default), n_jobs is the number of processes to start and typically the number of blocks n_blocks is set to the number of available CPU cores. When the *distributed* scheduler is used, Dask will automatically learn the number of available Dask worker processes and n_jobs is meaningless; instead it makes more sense to set the number of trajectory blocks that are then spread across all available workers.

User-defined Analysis

PMDA makes it easy to create analysis classes such as the ones discussed above. If the per-frame analysis can be expressed as a simple function, then an analysis class can be created with a factory function. Otherwise, a class has to be derived from pmda.parallel.ParallelAnalysisBase. Both approaches are described below.

pmda.custom.AnalysisFromFunction(): PMDA provides helper functions in pmda.custom to rapidly build a parallel class for users who already have a *single frame* function that 1. takes one or more AtomGroup instances as input, 2. analyzes one frame in a trajectory and returns the result for this frame. For example, if we already have a function to calculate the radius of gyration [MM14] of a protein given in AtomGroup ag, namely ag.radius_of_gyration() (as available in MDAnalysis), then we can write a simple function rgyr() that returns for each trajectory frame a tuple containing the time at the current time step and the value of the radius of gyration:

```
import MDAnalysis as mda
u = mda.Universe(top, traj)
protein = u.select_atoms('protein')

def rgyr(ag):
    return (ag.universe.trajectory.time,
            ag.radius_of_gyration())
```

We can wrap rgyr() in the pmda.custom.AnalysisFromFunction() class instance factory function to build a parallel version of rgyr():

```
import pmda.custom
parallel_rgyr = pmda.custom.AnalysisFromFunction(
                    rgyr, u, protein)
```

This new parallel analysis class can be run just as the existing ones:

```
parallel_rgyr.run(n_jobs=4, n_blocks=4)
print(parallel_rgyr.results)
```

The time series of the results is stored in the attribute parallel_rgyr.results; for our example where each per-frame result is a tuple (time, Rgyr), the time series is stored as a $T \times 2$ array that can be plotted with

```
import matplotlib.pyplot as plt
data = parallel_rgyr.results
plt.plot(data[:, 0] , data[:, 1])
```

pmda.parallel.ParallelAnalysisBase: For more general cases, one can write the parallel class with the help of pmda.parallel.ParallelAnalysisBase, following the schema in Fig. 1. To build a new analysis class, one should derive a class from pmda.parallel.ParallelAnalysisBase that implements

1) the single frame analysis method _single_frame() (*required*),
2) the final results conclusion method _conclude() (*required*),
3) the additional preparation method _prepare() (*optional*),
4) the reduce method for frames within the same block _reduce() (*optional* for time series, *required* for anything else).

As an example, we show how one can build a class to calculate the radius of gyration of a protein given in AtomGroup protein; of course, in this case the simple approach with pmda.custom.AnalysisFromFunction() would be easier.

```
import numpy as np
from pmda.parallel import ParallelAnalysisBase

class RGYR(ParallelAnalysisBase):
    def __init__(self, protein):
        universe = protein.universe
        super(RGYR, self).__init__(universe,
                                   (protein,))
    def _prepare(self):
        self.rgyr = None
    def _conclude(self):
        self.rgyr = np.vstack(self._results)
```

The _conclude() method reshapes the attribute self._results, which always holds the results from all blocks, into a time series. The call signature for method _single_frame() is fixed and ts must contain the current MDAnalysis Timestep and agroups must be a tuple of AtomGroup instances. The current frame number, time and radius of gyration are returned as the single frame results:

```
def _single_frame(self, ts, atomgroups):
    protein = atomgroups[0]
    return (ts.frame, ts.time,
            protein.radius_of_gyration())
```

Because we want to return a time series, it is not necessary to define a _reduce() method. This class can be used in the same way as the class that we defined with pmda.custom.AnalysisFromFunction:

```
parallel_rgyr = RGYR(protein)
parallel_rgyr.run(n_jobs=4, n_blocks=4)
print(parallel_rgyr.results)
```

Performance Evaluation

In order to characterize the performance of PMDA on a typical HPC machine we performed computational experiments for two different analysis tasks, the RMSD calculation after optimum superposition (*RMSD*) and the water oxygen radial distribution function (*RDF*).

For the *RMSD* task we computed the time series of root mean square distance after optimum superposition (RMSD) of all 564 C_α atoms of a protein with the initial coordinates at the first frame as reference, as implemented in class pmda.rms.RMSD. The RMSD calculation with optimum superposition was performed with the fast QCPROT algorithm [The05] as implemented in MDAnalysis [MADWB11].

As a second test case we computed the water oxygen-oxygen radial distribution function (*RDF*, Eq. 2) in 75 bins up to a cut-off of 5 Å for all 24,239 oxygen atoms in the water molecules in our test system, using the class pmda.rdf.InterRDF. The RDF calculation is compute-intensive due to the necessity to calculate and histogram a large number ($\mathcal{O}(N)$ because of the use of a cut-off) of distances for each time step; it additionally exemplifies a non-trivial reduction.

These two common computational tasks differ in their computational cost and represent two different requirements for data reduction and thus allow us to investigate two distinct use cases. We investigated a long (9000 frames) and a short trajectory (900 frames) to assess to which degree parallelization remained practical. The computational experiments were performed in different scenarios to assess the influence of different Dask schedulers (*multiprocessing* and *distributed*) and the role of the file storage system (shared Lustre parallel file system and local SSD), as described below and summarized in Table 1.

Test system, benchmarking environment, and data files

We tested PMDA 0.2.1, MDAnalysis 0.20.0 (development version), Dask 1.2.0, and NumPy 1.15.4 under Python 3.6. All

configuration label	file storage	scheduler	max nodes	max processes
Lustre-distributed-3nodes	Lustre	*distributed*	3	72
Lustre-distributed-6nodes	Lustre	*distributed*	6	72
Lustre-multiprocessing	Lustre	*multiprocessing*	1	24
SSD-distributed	SSD	*distributed*	1	24
SSD-multiprocessing	SSD	*multiprocessing*	1	24

TABLE 1: *Testing configurations on SDSC Comet.* ***max nodes*** *is the maximum number of nodes that were tested; the multiprocessing scheduler is limited to a single node.* ***max processes*** *is the maximum number of processes or Dask workers that were employed.*

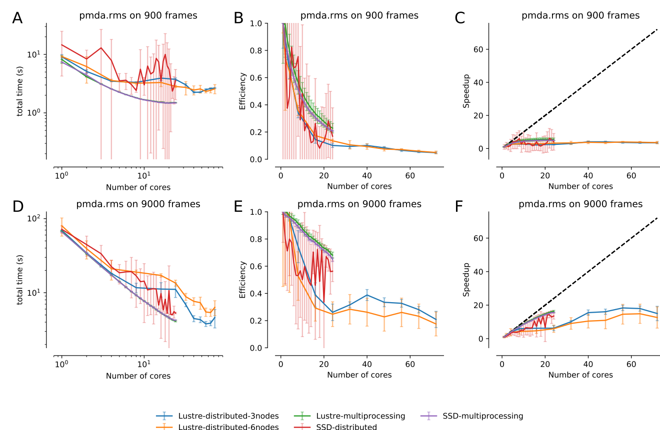

Fig. 2: *Strong scaling performance of the RMSD analysis task with short (900 frames) and long (9000) frames trajectories on* SDSC Comet*, where a single node contains 24 cores. The total time to completion t^{total} was measured for different testing configurations (Table 1).* ***A*** *and* ***D****: t^{total} as a function of processes or Dask workers, i.e., the number of CPU cores that were actually used. The number of trajectory blocks was the same as the number of CPU cores.* ***B*** *and* ***E****: efficiency E. The ideal case is $E = 1$.* ***C*** *and* ***F****: speed-up S. The dashed line represents ideal strong scaling $S(M) = M$. Points represent the mean over five repeats with the standard deviation shown as error bars.*

packages except PMDA and MDAnalysis were installed with the conda package manager from the conda-forge channel. PMDA and MDAnalysis development versions were installed from source in a conda environment with `pip install`.

Benchmarks were run on the CPU nodes of XSEDE's [TCD+14] *SDSC Comet* supercomputer, a 2 PFlop/s cluster with 1,944 Intel Haswell Standard Compute Nodes in total. Each node contains two Intel Xeon CPUs (E5-2680v3, 12 cores, 2.5 GHz) with 24 CPU cores per node, 128 GB DDR4 DRAM main memory, and a non-blocking fat-tree InfiniBand FDR 56 Gbps node interconnect. All nodes share a Lustre parallel file system and have access to node-local 320 GB SSD scratch space. Jobs are run through the SLURM batch queuing system. Our SLURM submission shell scripts and Python benchmark scripts for *SDSC Comet* are available in the repository https://github.com/Becksteinlab/scipy2019-pmda-data and are archived under DOI 10.5281/zenodo.3228422.

The test data files consist of a topology file `YiiP_system.pdb` (with $N = 111{,}815$ atoms) and two trajectory files `YiiP_system_9ns_center.xtc` (Gromacs XTC format, $T = 900$ frames) and `YiiP_system_90ns_center.xtc` (Gromacs XTC format, $T = 9000$ frames) of the membrane protein YiiP in a lipid bilayer together with water and ions. The test trajectories are made available on figshare at DOI 10.6084/m9.figshare.8202149.

We tested different combinations of Dask schedulers (*distributed*, *multiprocessing*) with different means to read the trajectory data (either from the shared Lustre parallel file system or from local SSD) as shown in Table 1. Using either the *multiprocessing* scheduler or the SSD restrict runs to a single node (maximum 24 CPU cores). With *distributed* (and Lustre) we tested fully utilizing all cores on a node and also only occupying half the available cores, while doubling the total number of nodes. In all cases the trajectory were split in as many blocks as there were available processes or Dask workers. We performed five independent repeat runs for all scenarios in Table 1 and plotted the mean of the reported timing quantity together with the standard deviation from the mean to indicate the variance of the runs.

Measured timing quantities

The `ParallelAnalysisBase` class collects detailed timing information for all blocks and all frames and makes these data available in the attribute `ParallelAnalysisBase.timing`: We measured the time t^{prepare} for `_prepare()`, the time t_k^{wait} that each task k waits until it is executed by the scheduler, the time t_k^{Universe} to create a new `Universe` for each Dask task (which includes opening the shared trajectory and topology files and loading the topology into memory), the time $t_{k,t}^{\text{I/O}}$ to read each frame t in each block k from disk into memory, the time $t_{k,t}^{\text{compute}}$ to perform the computation in `_single_frame()` and reduction in `_reduce()`, the time t_k^{conclude} to perform the final processing of all data in `_conclude()`, and the total wall time to solution t^{total}.

We analyzed the total time to completion as a function of the number of CPU cores, which was equal to the number of trajectory blocks, so that each block could be processed in parallel. We quantified the strong scaling behavior by calculating the *speed-up* for running on M CPU cores with M parallel Dask tasks as $S(M) = t^{\text{total}}(1)/t^{\text{total}}(M)$, where $t^{\text{total}}(1)$ is the performance of the PMDA code using the serial scheduler. The *efficiency* was calculated as $E(M) = S(M)/M$. The errors of these quantities were derived by the standard error propagation.

To gain better insight into the performance-limiting steps in our algorithm (Fig. 1) we plotted the *maximum* times over all ranks because the overall time to completion cannot be faster than the slowest parallel process. For example, for the read I/O time we calculated the total read I/O time for each rank k as $t_k^{\text{I/O}} = \sum_{t=t_k}^{t_k+\tau_k} t_{k,t}^{\text{I/O}}$ and then reported $\max_k t_k^{\text{I/O}}$.

RMSD analysis task

The parallelized RMSD analysis in `pmda.rms.RMSD` scaled well only to about half a node (12 cores), as shown in Fig. 2 A, D, regardless of the length of the trajectory. The efficiency dropped below 0.8 (Fig. 2 B, E) and the maximum achievable speed-up remained below 10 for the short trajectory (Fig. 2 C) and below 20 for the long one (Fig. 2 F). Overall, using the *multiprocessing* scheduler and either Lustre or SSD gave the best performance and shortest time to solution. The *distributed* scheduler with SSD gave widely variable results as seen by large standard deviations over

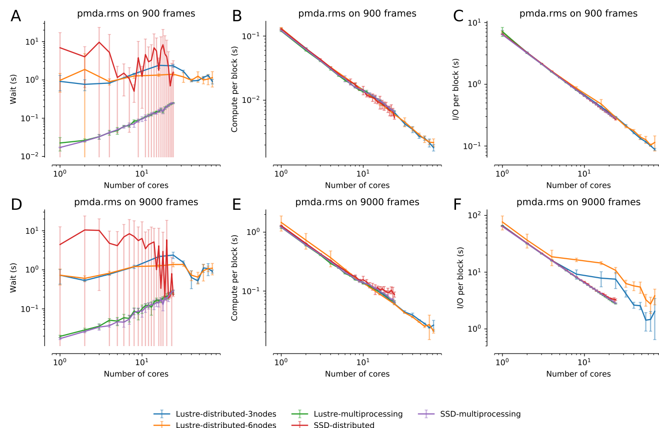

Fig. 3: *Detailed per-task timing analysis for parallel components of RMSD analysis task. Individual times per task were measured for different testing configurations (Table 1).* ***A*** *and* ***D****: Maximum waiting time for the task to be executed by the Dask scheduler.* ***B*** *and* ***E****: Maximum total compute time per task.* ***C*** *and* ***F****: Maximum total read I/O time per task. Points represent the mean over five repeats with the standard deviation shown as error bars.*

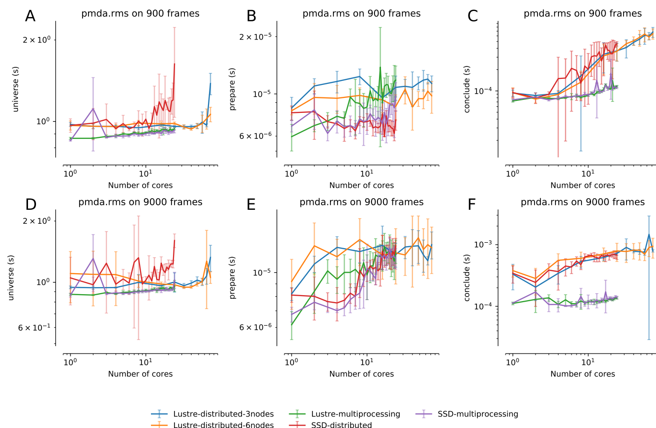

Fig. 4: *Detailed timing analysis for other components of the RMSD analysis task. Individual times per task were measured for different testing configurations (Table 1).* ***A*** *and* ***D****: Maximum time for a task to load the* `Universe`*.* ***B*** *and* ***E****: Time $t^{prepare}$ to execute* `_prepare()`*.* ***C*** *and* ***F****: Time $t_k^{conclude}$ to execute* `_conclude()`*. Points represent the mean over five repeats with the standard deviation shown as error bars.*

multiple repeats. It still performed better than when the Lustre file system was used but overall, for a single node, the *multiprocessing* scheduler always gave better performance with less variation in run time. These results were consistent with findings in our earlier pilot study where we had looked at the RMSD task with Dask and had found that *multiprocessing* with both SSD and Lustre had given good single node performance but, using *distributed*, had not scaled well beyond a single *SDSC Comet* node [KPJB17].

A detailed look at the maximum times (Fig. 3) that the Dask worker processes spent on waiting to be executed, performing the RMSD calculation with data in memory, and reading the trajectory frame data from the file into memory showed that the waiting time (Fig. 3 A, D) either increased from about 0.02 s to 0.1 s for *multiprocessing* or was roughly a constant 1 s for *distributed* (on Lustre). For reasons that were not clear, the *distributed* scheduler with SSD had on average the largest wait times, with large fluctuations, ranging from 0.1 s to 10 s (red lines in Fig. 3 A, D). The computation itself scaled very well (Fig. 3 B, E) with only small variations, indicating that split-apply-combine is a robust approach to parallelization, once the data are in memory. The reading time scaled fairly well but exhibited some variation beyond a single node (24 cores) and an unexplained decline in performance for the longer trajectory, as seen in Fig. 3 C, F. The read I/O results indicated that both Lustre and SSD can perform equally well. Beyond 12 cores, the waiting time started approaching the time for read I/O (compute was an order of magnitude less than I/O) and hence parallel speed-up was limited by the wait time.

The second major component that limited scaling performance was the time to create the `Universe` data structure (Fig. 4 A, D). The time to read the topology and open the trajectory file on the shared file system typically increased from 1 s to about 2 s and thus, for the given total trajectory lengths, also became comparable to the time for read I/O. The other components (prepare and conclude, see Fig. 4) remained negligible with times below 10^{-3} s.

The parallelizable fraction of the workload consisted of the

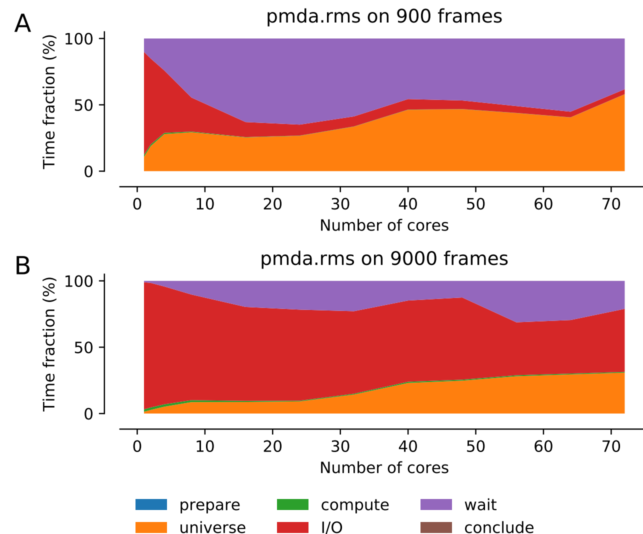

Fig. 5: *Fraction of the total run time taken by individual steps in the parallel* RMSD *calculation for* distributed *on up to three nodes (Lustre-distributed-3nodes). Compute (green) and read I/O (red) represent the parallelizable fraction of the program; all other components are effectively serial.* ***A*** *Trajectory with 900 frames.* ***B*** *Trajectory with 9000 frames.*

compute and read I/O steps. Because this fraction was relatively small and was dominated by the wait time from the Dask scheduler and the time to initialize the `Universe` data structure (Fig. 5), the overall performance gain by parallelization remained modest, as explained by Amdahl's law [Amd67]. Thus, for a highly optimized and fast computation such as the RMSD calculation, the best performance (speed-up on the order of 10 fold) could already be achieved on the equivalent of a modern workstation. The *multiprocessing* scheduler seemed to be the more consistent and better performing choice in this scenario; therefore PMDA defaults to *multiprocessing*. Performance would likely improve with longer trajectories because the "fixed" serial costs (waiting, `Universe`

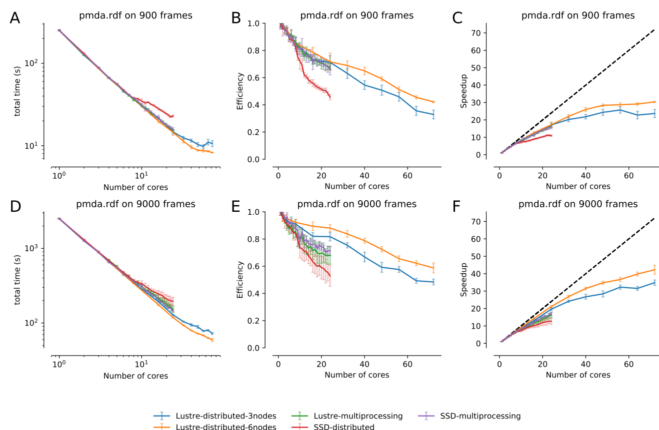

Fig. 6: Strong scaling performance of the RDF analysis task. The total time to completion t^{total} was measured for different testing configurations (Table 1). ***A*** *and* ***D****: t^{total} as a function of processes or Dask workers, i.e., the number of CPU cores that were actually used. The number of trajectory blocks was the same as the number of CPU cores.* ***B*** *and* ***E****: efficiency E. The ideal case is $E = 1$.* ***C*** *and* ***F****: speed-up S. The dashed line represents ideal strong scaling $S(M) = M$. Points represent the mean over five repeats with the standard deviation shown as error bars.*

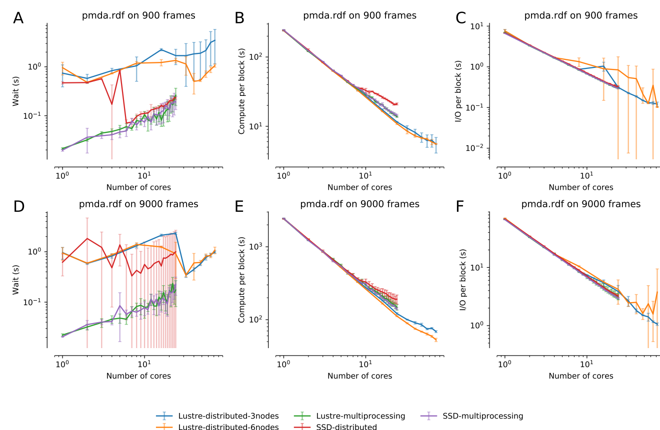

Fig. 7: Detailed per-task timing analysis for parallel components of the RDF analysis task. Individual times per task were measured for different testing configurations (Table 1). ***A*** *and* ***D****: Maximum waiting time for the task to be executed by the Dask scheduler.* ***B*** *and* ***E****: Maximum total compute time per task.* ***C*** *and* ***F****: Maximum total read I/O time per task. Points represent the mean over five repeats with the standard deviation shown as error bars.*

creation) would decrease in relevance to the time spent on computation and data ingestion (see Fig. 5 B), which benefit from parallelization [Gus88]. However, all things considered, a single node seemed sufficient to accelerate RMSD analysis.

RDF analysis task

Unlike the RMSD analysis task, the parallelized RDF analysis in `pmda.rdf.InterRDF` showed decreasing total time to solution up to the highest number of CPU cores tested (see Fig. 6 A, D). The efficiency on a single node remained above 0.6 for almost all cases (Fig. 6 B, E) and remained above 0.6 for the best case (*distributed* on Lustre and half-filling of nodes for the long trajectory), up to 3 nodes (72 cores, Fig. 6 E). Even when filling complete nodes, the efficiency for the long trajectory remained above 0.5 (Fig. 6 E). Consequently, a sizable speed-up could be maintained that approached 40 fold in the best case (Fig. 6 F), which cut down the time to solution from about 40 min to under 1 min. On a single node, all approaches performed similarly well, with the *distributed* scheduler now having a slight edge over *multiprocessing* (Fig. 6), with the exception of the combination of *distributed* with the SSD, which for unknown reasons performed much worse than everything else (similar to the situation observed for the *RMSD* case).

The detailed analysis of the individual components in Fig. 7 clearly showed that the RDF analysis task required much more computational effort than the RMSD task and that it was dominated by the compute component (Fig. 8), which scaled very well to the highest core numbers (Fig. 7 B, E). However, *multiprocessing* and especially *distributed* with *SSD* took longer for the computational part at ≥ 8 cores (one third of a single node), indicating that in these cases some sort of competition reduced performance. For comparison, serial computation required about 250 s while read I/O required less than 10 s, and this ratio was approximately maintained as the read I/O also scaled reasonably well (Fig. 7 C, F) Although the variance increased markedly when multiple nodes were included such as when using six half-filled nodes, this effect did not strongly impact overall performance because $t_{k,t}^{\text{compute}} \gg t_{k,t}^{\text{I/O}}$. The differences between using all cores on a node compared to only using half the cores on each node were small but only using half a node was consistently better, especially in the compute time, and hence the overall performance of the latter approach was better. For the shorter trajectory, the wait time was a factor in reducing performance at higher core numbers (Fig. 7 A). The other components ($t_k^{\text{Universe}} < 2$ s, $t^{\text{prepare}} < 3 \times 10^{-5}$ s , $t_k^{\text{conclude}} < 4 \times 10^{-4}$ s) were similar or better (i.e., shorter) than the ones shown for the RMSD task in Fig. 4 and are not shown; only the time to set up the `Universe` played a role in reducing the scaling performance in the *Lustre-distributed-3nodes* scenario at 60 or more CPU cores.

In summary, the performance increase for a compute-intensive task such as RDF was sizable and, although not extremely efficient, was large enough (about 30-40) to justify the use of about 100 cores on a HPC supercomputer. Because scaling seemed mostly limited by constant costs such as the scheduling wait time (see Fig. 8), processing longer trajectories, for which more work has to be done in the parallelizable compute and read I/O steps, should improve the scaling behavior [Gus88].

Conclusions

The PMDA Python package provides a framework to parallelize analysis of MD trajectories with a simple *split-apply-combine* approach by combining Dask with MDAnalysis. Although still in early development, it provides useful functionality for users to speed up analysis, ranging from a growing library of included tools to different approaches for users to write their own parallel analysis. In simple cases, just wrapping a user supplied function is enough to immediately use PMDA but the package also provides a documented API to derive from the `pmda.parallel.ParallelAnalysisBase` class. We showed that performance depends on the type of analysis that is being performed. Compute-intensive tasks such as the RDF calculation can show good strong scaling up to about a hundred cores on a typical supercomputer and speeding up the time to

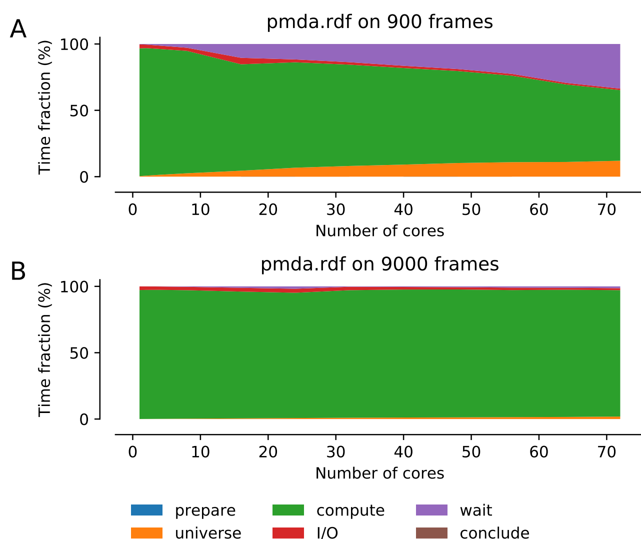

Fig. 8: *Fraction of the total run time taken by individual steps in the parallel* RDF *calculation for* distributed *on up to three nodes (Lustre-distributed-3nodes). Compute (green) and read I/O (red) represent the parallelizable fraction of the program; all other components are effectively serial.* ***A*** *Trajectory with 900 frames.* ***B*** *Trajectory with 9000 frames.*

solution from hours in serial to minutes in parallel should make this an attractive solution for many users. For other analysis tasks such as the RMSD calculation and other similar ones (e.g., simple distance calculations), a single multi-core workstation seems sufficient to achieve speed-ups on the order of 10 and HPC resources would not be useful. But thanks to the design of Dask, running a PMDA analysis on a laptop, workstation, or supercomputer requires absolutely no changes in the code and users are free to immediately choose the computational resource that best fits their purpose.

Code availability and development process

PMDA is available in source form under the GNU General Public License v2 from the GitHub repository MDAnalysis/pmda, and as a PyPi package and conda package (via the conda-forge channel). Python 2.7 and Python $\geq$ 3.5 are fully supported and tested. The package uses semantic versioning to make it easy for users to judge the impact of upgrading. The development process uses continuous integration (Travis CI): extensive tests are run on all commits and pull requests via pytest, resulting in a current code coverage of 97% and documentation is automatically generated by Sphinx and published as GitHub pages. Users are supported through the community mailing list (Google group) and the GitHub issue tracker.

Acknowledgments

We would like to thank reviewer Cyrus Harrison for the idea to plot the fractional time spent on different stages of the program (Figs. 5 and 8). This work was supported by the National Science Foundation under grant numbers ACI-1443054 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 was used under allocation TG-MCB130177. Max Linke was supported by NumFOCUS under a small development grant.

References

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi:10.1145/1465482.1465560.

[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*, November 2018. URL: https://www.exascale.org/bdec/sites/www.exascale.org.bdec/files/whitepapers/Beckstein-Fox-Jha_BDEC2_WP_0.pdf.

[BGM+18] Efrem Braun, Justin Gilmer, Heather B. Mayes, David L. Mobley, Jacob I. Monroe, Samarjeet Prasad, and Daniel M. Zuckerman. Best practices for foundations in molecular simulations [article v1.0]. *Living Journal of Computational Molecular Science*, 1(1):5957–, 11 2018. doi:10.33011/livecoms.1.1.5957.

[BLL18] Sandro Bottaro and Kresten Lindorff-Larsen. Biophysical experiments and biomolecular simulations: A perfect match? *Science*, 361(6400):355–360, 2018. doi:10.1126/science.aat4010.

[BS12] David W Borhani and David E Shaw. The future of molecular dynamics simulations in drug discovery. *J Comput Aided Mol Des*, 26(1):15–26, Jan 2012. doi:10.1007/s10822-011-9517-y.

[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.

[Das16] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016. URL: https://dask.org.

[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.

[FPM18] Pim W J M Frederix, Ilias Patmanidis, and Siewert J Marrink. Molecular simulations of self-assembling bio-inspired supramolecular systems and their connection to experiments. *Chem Soc Rev*, 47(10):3470–3489, May 2018. doi:10.1039/c8cs00040a.

[FS02] Daan Frenkel and Berend Smit. *Understanding Molecular Simulations*. Academic Press, San Diego, 2 edition, 2002.

[Gio19] Toni Giorgino. Analysis libraries for molecular trajectories: a cross-language synopsis. In M. Bonomi and C. Camilloni, editors, *Biomolecular Simulations: Methods and Protocols*. Springer, 2019.

[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. URL: https://www.mdanalysis.org, doi:10.25080/Majora-629e541a-00e.

[Gus88] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988. doi:10.1145/42411.42415.

[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.

[KAHC18] Grit Kupgan, Lauren J. Abbott, Kyle E. Hart, and Coray M. Colina. Modeling amorphous microporous polymers for CO_2 capture and separations. *Chemical Reviews*, 118(11):5488–5538, 2018. PMID: 29812911. arXiv:https://doi.org/10.1021/acs.chemrev.7b00691, doi:10.1021/acs.chemrev.7b00691.

[KM02] Martin Karplus and J Andrew McCammon. Molecular dynamics simulations of biomolecules. *Nat Struct Biol*, 9(9):646–52, Sep 2002. doi:10.1038/nsb0902-646.

[KPJB17] Mahzad Khoshlessan, Ioannis Paraskevakos, Shantenu Jha, and Oliver Beckstein. Parallel analysis in MDAnalysis using the Dask parallel computing library. In Katy Huff, David Lippa, Dillon Niederhut, and M Pacer, editors, *Proceedings of the 16th Python in Science Conference*, pages 64–72, Austin, TX, 2017. SciPy. doi:10.25080/shinma-7f4c6e7-00a.

[LJYH18] Denvid Lau, Wei Jian, Zechuan Yu, and David Hui. Nano-engineering of construction materials using molecular dynamics simulations: Prospects and challenges. *Composites Part B: Engineering*, 143:282 – 291, 2018. doi:10.1016/j.compositesb.2018.01.014.

[LS15] Chunyu Li and Alejandro Strachan. Molecular scale simulations on thermoset polymers: A review. *Journal of Polymer Science Part B: Polymer Physics*, 53(2):103–122, 2015. doi:10.1002/polb.23489.

[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. arXiv:http://dx.doi.org/10.1080/08927022.2014.935372, doi:10.1080/08927022.2014.935372.

[Oli07] Travis E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, 2007. doi:10.1109/mcse.2007.58.

[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.

[Roc15] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*, number 130–136, 2015. URL: https://github.com/dask/dask.

[Rot09] Jörg Rottler. Fracture in glassy polymers: a molecular modeling perspective. *Journal of Physics: Condensed Matter*, 21(46):463101, oct 2009. doi:10.1088/0953-8984/21/46/463101.

[SB14] Sean L Seyler and Oliver Beckstein. Sampling of large conformational transitions: Adenylate kinase as a testing ground. *Molec. Simul.*, 40(10–11):855–877, 2014. doi:10.1080/08927022.2014.919497.

[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.

[Tuc10] Mark E. Tuckerman. *Statistical Mechanics: Theory and Molecular Simulation*. Oxford University Press, Oxford, UK, 2010.

[VDWCV11] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011. doi:10.1109/MCSE.2011.37.

[VMMC+15] L. M. Varela, T. Méndez-Morales, J. Carrete, V. Gómez-González, B. Docampo-Álvarez, L. J. Gallego, O. Cabeza, and O. Russina. Solvation of molecular cosolvents and inorganic salts in ionic liquids: A review of molecular dynamics simulations. *Journal of Molecular Liquids*, 210:178–188, 2015. doi:10.1016/j.molliq.2015.06.036.

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