Parallel Analysis in MDAnalysis using the Dask Parallel Computing Library

Mahzad Khoshlessan[§], Ioannis Paraskevakos[‡], Shantenu Jha[‡], Oliver Beckstein[§*]

Abstract—The analysis of biomolecular computer simulations has become a challenge because the amount of output data is now routinely in the terabyte range. We evaluated if this challenge can be met by a parallel map-reduce approach with the Dask parallel computing library for task-graph based computing coupled with our MDAnalysis Python library for the analysis of molecular dynamics (MD) simulations. We performed a representative performance evaluation, taking into account the highly heterogeneous computing environment that researchers typically work in together with the diversity of existing file formats for MD trajectory data. We found that the underlying storage system (solid state drives, parallel file systems, or simple spinning platter disks) can be a deciding performance factor that leads to data ingestion becoming the primary bottleneck in the analysis work flow. However, the choice of the data file format can mitigate the effect of the storage system; in particular, the commonly used Gromacs XTC trajectory format, which is highly compressed, can exhibit strong scaling close to ideal due to trading a decrease in global storage access load against an increase in local per-core cpu-intensive decompression. Scaling was tested on a single node and multiple nodes on national and local supercomputing resources as well as typical workstations. In summary, we show that, due to the focus on high interoperability in the scientific Python eco system, it is straightforward to implement map-reduce with Dask in MDAnalysis and provide an in-depth analysis of the considerations to obtain good parallel performance on HPC resources.

Index Terms—MDAnalysis, High Performance Computing, Dask, Map-Reduce, MPI for Python

Introduction

MDAnalysis is a Python library that provides users with access to raw simulation data and enables structural and temporal analysis of molecular dynamics (MD) trajectories generated by all major MD simulation packages [GLB+16], [MADWB11]. MD trajectories are time series of positions (and sometimes also velocities) of the simulated atoms or particles; using statistical mechanics one can calculate experimental observables from these time series [FS02], [MM14]. The size of these trajectories is growing as the simulation times are being extended beyond micro-seconds and larger systems with increasing numbers of atoms are simulated. The amount of data to be analyzed is growing rapidly into the terabyte range and analysis is increasingly becoming a bottleneck in MD workflows [CR15]. Therefore, there is a need for high performance computing (HPC) approaches for the analysis of MD trajectory data [TRB+08], [RCI13].

MDAnalysis does not yet provide a standard interface for parallel analysis; instead, various existing parallel libraries such as Python multiprocessing, joblib, and mpi4py [DPS05], [DPKC11] are currently used to parallelize MDAnalysis-based code on a case-by-case basis. Here we evaluated performance for parallel map-reduce [DG08] type analysis with the Dask parallel computing library [Roc15] for task-graph based distributed computing on HPC and local computing resources. Although Dask is able to implement much more complex computations than map-reduce, we chose Dask for this task because of its ease of use and because we envisage using this approach for more complicated analysis applications whose parallelization cannot be easily expressed as a simple map-reduce algorithm.

As the computational task we performed a common task in the analysis of the structural dynamics of proteins: we computed the time series of the root mean squared distance (RMSD) of the positions of all C_α atoms to their initial coordinates at time 0; for each time step ("frame") in the trajectory, rigid body degrees of freedom (translations and rotations) have to be removed through an optimal structural superposition that minimizes the RMSD [MM14] (Figure 1). A range of commonly used MD file formats (CHARMM/NAMD DCD [BBIM+09], Gromacs XTC [AMS+15], Amber NCDF [CCD+05]) and different trajectory sizes were benchmarked.

We looked at different HPC resources including national supercomputers (XSEDE TACC *Stampede* and SDSC *Comet*), university supercomputers (Arizona State University Research

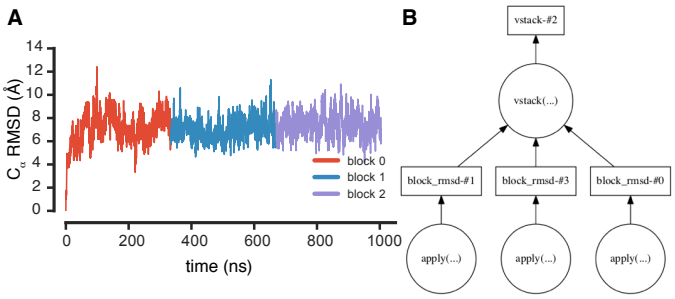

Fig. 1: *Calculation of the root mean square distance (RMSD) of a protein structure from the starting conformation via map-reduce with Dask.* ***A*** *RMSD as a function of time, with partial time series colored by trajectory block.* ***B*** *Dask task graph for splitting the RMSD calculation into three trajectory blocks.*

§ Arizona State University, Department of Physics, Tempe, AZ 85287, USA
‡ RADICAL, ECE, Rutgers University, Piscataway, NJ 08854, USA
∗ Corresponding author: obeckste@asu.edu

Computing *Saguaro*), and local resources (Gigabit networked multi-core workstations). The tested resources are parallel and heterogeneous with different CPUs, file systems, high speed networks and are suitable for high-performance distributed computing at various levels of parallelization. Different storage systems such as solid state drives (SSDs), hard disk drives (HDDs), network file system (NFS), and the parallel Lustre file system (using HDDs) were tested to examine the effect of I/O on the performance. The benchmarks were performed both on a single node and across multiple nodes using the multiprocessing and distributed schedulers in the Dask library.

We previously showed that the overall computational cost scales directly with the length of the trajectory, i.e., the weak scaling is close to ideal and is fairly independent from other factors [KB17]. Here we focus on the strong scaling behavior, i.e., the dependence of overall run time on the number of CPU cores used. Competition for access to the same file from multiple processes appears to be a bootleneck and thefore the storage system is an important determinant of performance. But because the trajectory file format dictates the data access pattern, overall performance also depends on the actual data format, with some formats being more robust against storage system specifics than others. Overall, good strong scaling performance could be obtained for a single node but robust across-node performance remained challenging. In order to identify performance bottlenecks we examined several other factors including the effect of striping in the parallel Lustre file system, oversubscribing (using more tasks than Dask workers), the performance of the Dask scheduler itself, and we also benchmarked an MPI-based implementation in contrast to the Dask approach. From these tests we tentatively conclude that poor across-nodes performance is rooted in contention on the shared network that may slow down individual tasks and lead to poor load balancing. Nevertheless, Dask with MDAnalysis appears to be a promising approach for high-level parallelization for analysis of MD trajectories, especially at moderate CPU core numbers.

Methods

We implemented a simple map-reduce scheme to parallelize processing of trajectories over contiguous blocks. We tested libraries in the following versions: MDAnalysis 0.15.0, Dask 0.12.0 (also 0.13.0), distributed 1.14.3 (also 1.15.1), and NumPy 1.11.2 (also 1.12.0) [VCV11].

```
import numpy as np
import MDAnalysis as mda
from MDAnalysis.analysis.rms import rmsd
```

The trajectory is split into `n_blocks` blocks with inital frame `start` and final frame `stop` set for each block. The calculation on each block (function `block_rmsd()`, corresponding to the *map* step) is *delayed* with the `delayed()` function in Dask:

```
from dask.delayed import delayed

def analyze_rmsd(ag, n_blocks):
    """RMSD of AtomGroup ag, parallelized n_blocks"""
    ref0 = ag.positions.copy()
    bsize = int(np.ceil(
                ag.universe.trajectory.n_frames \
                / float(n_blocks)))
    blocks = []
    for iblock in range(n_blocks):
        start, stop = iblock*bsize, (iblock+1)*bsize
        out = delayed(block_rmsd, pure=True)(
                ag.indices, ag.universe.filename,
                ag.universe.trajectory.filename,
                ref0, start, stop)
        blocks.append(out)
    return delayed(np.vstack)(blocks)
```

In the *reduce* step, the partial time series from each block are concatenated in the correct order (`np.vstack`, see Figure 1 A); because results from delayed objects are used, this step also has to be delayed.

As computational load we implement the calculation of the root mean square distance (RMSD) of the C_α atoms of the protein adenylate kinase [SB14] when fitted to a reference structure using an optimal rigid body superposition [MM14], using the qcprot implementation [LAT10] in MDAnalysis [GLB+16]. The RMSD is calculated for each trajectory frame in each block by iterating over `u.trajectory[start:stop]`:

```
def block_rmsd(index, topology, trajectory, ref0,
               start, stop):
    u = mda.Universe(topology, trajectory)
    ag = u.atoms[index]
    out = np.zeros([stop-start, 2])
    for i, ts in enumerate(
            u.trajectory[start:stop]):
        out[i, :] = ts.time, rmsd(ag.positions, ref0,
                        center=True, superposition=True)
    return out
```

Dask produces a task graph (Figure 1 B) and the computation of the graph is executed in parallel through a Dask scheduler such as `dask.multiprocessing` (or `dask.distributed`):

```
from dask.multiprocessing import get

u = mda.Universe(PSF, DCD)
ag = u.select_atoms("protein and name CA")
result = analyze_rmsd(ag, n_blocks)
timeseries = result.compute(get=get)
```

The complete code for benchmarking is available from https://github.com/Becksteinlab/Parallel-analysis-in-the-MDAnalysis-Library under the MIT License.

The data files consist of a topology file `adk4AKE.psf` (in CHARMM PSF format; $N = 3341$ atoms) and a trajectory `1ake_007-nowater-core-dt240ps.dcd` (DCD format) of length 1.004 µs with 4187 frames; both are freely available from figshare at DOI 10.6084/m9.figshare.5108170 [SB17]. Files in XTC and NCDF formats are generated from the DCD on the fly using MDAnalysis. To avoid operating system caching, files were copied and only used once for each benchmark. All results for Dask distributed were obtained across three nodes on different clusters.

Trajectories with different number of frames per trajectory were analyzed to assess the effect of trajectory file size. These trajectories were generated by concatenating the base trajectory 50, 100, 300, and 600 times and are referred to as, e.g., "DCD300x" or "XTC600x". Run time was analyzed on single nodes (1–24 CPU cores) and up to three nodes (1–72 cores) as function of the number of cores (strong scaling behavior) and trajectory sizes (weak scaling). However, here we only present strong scaling data for the 300x and 600x trajectory sizes, which represent typical medium size results. For an analysis of the full data including weak scaling results set see the Technical Report [KB17].

The DCD file format is a binary representation for 32-bit floating point numbers (accuracy of positions about 10^{-6} Å) and the DCD300x trajectory has a file size of 47 GB (DCD600x is twice as much); XTC is a lossy compressed format that effectively rounds floats to the second decimal (accuracy about 10^{-2} Å, which is sufficient for typical analysis) and XTC300x is only 15 GB.

Amber NCDF is implemented with netCDF classic format version 3.6.0 (same accuracy as DCD) and trajectories are about the same size as DCD. DCD and NCDF natively allow fast random access to frames or blocks of frames, which is critical to implement the map-reduce algorithm. XTC does not natively support frame seeking but MDAnalysis implements a fast frame scanning algorithm for XTC files that caches all frame offsets and so enables random access for the XTC format, too [GLB+16].

Performance was quantified by measuring the average time per trajectory frame to load data from storage into memory (I/O time per frame, $t_{I/O}$), the average time to complete the RMSD calculation (compute time per frame, t_{comp}), and the total wall time for job execution t_N when using N CPU cores. Strong scaling was assessed by calculating the speed up $S(N) = t_1/t_N$ and the efficiency $E(N) = S(N)/N$.

Results and Discussion

Trajectories from MD simulations record snapshots of the positions of all particles are regular time intervals. A snapshot at a specified time point is called a frame. MDAnalysis only loads a single frame into memory at any time [GLB+16], [MADWB11] to allow the analysis of large trajectories that may contain, for example, $n_{frames} = 10^7$ frames in total. In a map-reduce approach, N processes will iterate in parallel over N chunks of the trajectory, each containing n_{frames}/N frames. Because frames are loaded serially, the run time scales directly with n_{frames} and the weak scaling behavior (as a function of trajectory length) is trivially close to ideal as seen from the data in [KB17]. Weak scaling with the system size also appears to be fairly linear, according to preliminary data (not shown). Therefore, in the following we focus exclusively on the harder problem of strong scaling, i.e., reducing the run time by employing parallelism.

Effect of File Format on I/O

We first sought to quantify the effect of the trajectory format on the analysis performance. The overall run time depends strongly on the trajectory file format as well as the underlying storage system as shown for the 300x trajectories in Figure 2; results for other trajectory sizes are similar (see [KB17]) except for the smallest 50x trajectories where possibly caching effects tend to improve overall performance. Using DCD files with SSDs on a single node (Figure 2 A) is about one order of magnitude faster than the other formats (Figure 2 B, C) and scales near linearly for small CPU core counts ($N \leq 12$). However, DCD does not scale at all with other storage systems such as HDD of NFS and run time only improves up to $N = 4$ on the Lustre file system. On the other hand, the run time with NCDF and especially with XTC trajectories improves linearly with increasing N, with XTC on Lustre and $N = 24$ cores almost obtaining the best DCD run time of about 30 s (SSD, $N = 12$); at the highest single node core count $N = 24$, XTC on SSD performs even better (run time about 25 s). For larger N on multiple nodes, only a shared file system (Lustre or NFS) based on HDD was available. All three file formats only show small improvements in run time at higher core counts ($N > 24$) on the Lustre file system on supercomputers with fast interconnects and no improvements on NFS over Gigabit (Figure 2 D–F).

In order to explain the differences in performance and scaling of the file formats, we analyzed the time to load the coordinates of a single frame from storage into memory ($t_{I/O}$) and the time to perform the computation on a single frame using the in-memory

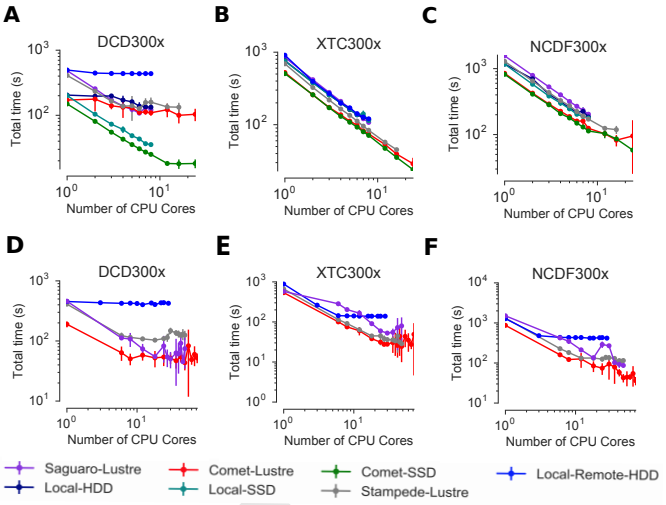

Fig. 2: *Comparison of total job execution time t_N for different file formats (300x trajectory size) using Dask multiprocessing on a* single node *(1–24 CPU cores, **A** – **C**) and Dask distributed on* up to three nodes *(1–72 CPU cores, **D** – **F**). The trajectory was split into M blocks and computations were performed using $N = M$ CPU cores. The runs were performed on different resources (ASU RC* Saguaro*, SDSC* Comet*, TACC* Stampede*,* local *workstations with different storage systems (locally attached* HDD*,* remote HDD *(via network file system, NFS), locally attached* SSD*,* Lustre *parallel file system with a single stripe). **A**, **D** CHARMM/NAMD DCD. **B**, **E** Gromacs XTC. **C**, **F** Amber NetCDF.*

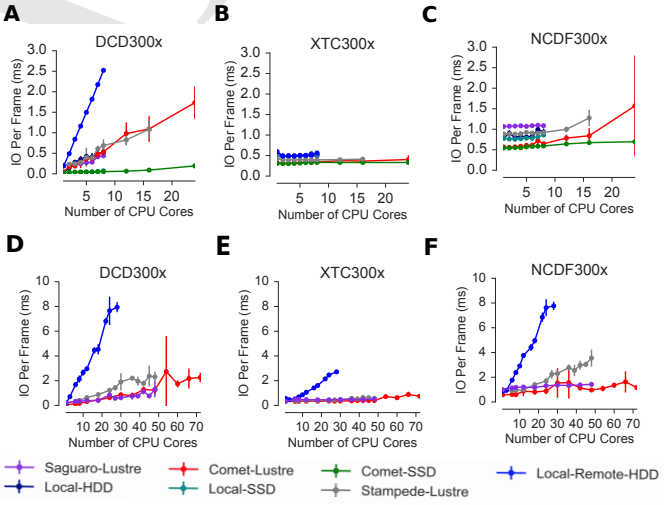

Fig. 3: *Comparison of I/O time $t_{I/O}$ per frame between different file formats (300x trajectory size) using Dask multiprocessing on a* single node *(**A** – **C**) and Dask distributed on* multiple nodes *(**D** – **F**). **A**, **D** CHARMM/NAMD DCD. **B**, **E** Gromacs XTC. **C**, **F** Amber NetCDF. All parameters as in Fig. 2.*

data (t_{comp}). As expected, t_{comp} is independent from the file format, n_{frames}, and N and only depends on the CPU type itself (mean and standard deviation on SDSC *Comet* 0.098 ± 0.004 ms, TACC *Stampede* 0.133 ± 0.000 ms, ASU RC *Saguaro* 0.174 ± 0.000 ms, local workstations 0.225 ± 0.022 ms, see [KB17]). Figure 3, however shows how $t_{I/O}$ (for the 300x trajectories) varies widely and in most cases, is at least an order of magnitude larger than t_{comp}. The exception is $t_{I/O}$ for the DCD file format using SSDs, which remains small (0.06 ± 0.04 ms on SDSC *Comet*) and almost

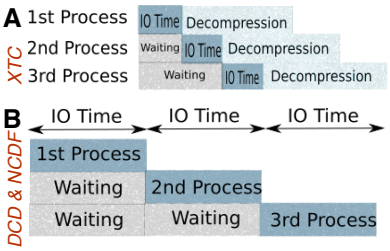

Fig. 4: *I/O pattern for reading frames in parallel from commonly used MD trajectory formats.* ***A*** *Gromacs XTC file format.* ***B*** *CHARMM/NAMD DCD file format and Amber NCDF format.*

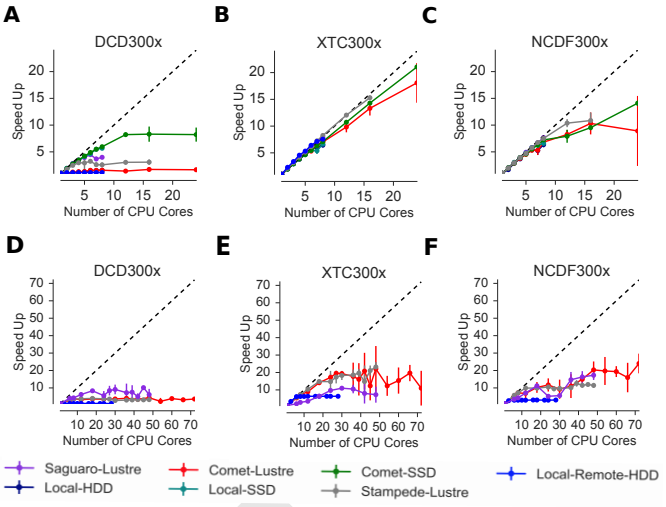

Fig. 5: *Speed-up S for the analysis of the 300x trajectory on HPC resources using Dask multiprocessing (single node,* ***A*** *–* ***C****) and distributed (up to three nodes,* ***D*** *–* ***F****). The dashed line shows the ideal limit of strong scaling. All other parameters as in Fig. 3.*

constant with $N \leq 12$ (Figure 3 A) and as a result, the DCD file format shows good scaling and the best performance on a single node. For HDD-based storage, the time to read data from a DCD frame increases with the number of processes that are simultaneously trying to access the DCD file. XTC and NCDF show flat $t_{\mathrm{I/O}}$ with N on a single node (Figure 3 B, C) and even for multiple nodes, the time to ingest a frame of a XTC trajectory is almost constant, except for NFS, which broadly shows poor performance (Figure 3 E, F).

Depending on the file format the loading time of frames into memory will be different, as illustrated in Figure 4. The XTC file format is compressed and has a smaller file size when compared to the other formats. When a compressed XTC frame is loaded into memory, it is immediately decompressed (see Figure 4 A). During decompression by one process, the file system allows the next process to load its requested frame into memory. As a result, competition for file access between processes and overall wait time is reduced and $t_{\mathrm{I/O}}$ remains almost constant, even for large number of parallel processes (Figure 3 B, E). Neither DCD nor NCDF files are compressed and multiple processes compete for access to the file (Figure 4 B) although NCDF files is a more complicated file format than DCD and has additional computational overhead. Therefore, for DCD the I/O time per frame is very small as compared to other formats when the number of processes is small (and the storage is fast), but even at low levels of parallelization, $t_{\mathrm{I/O}}$ increases due to the overlapping of per frame trajectory data access (Figure 3 A, D). Data access with NCDF is slower but due to the additional computational overhead, is amenable to some level of parallelization (Figure 3 C, F).

Performance Comparison between Different File Format

Figure 5 shows speed up comparison for 300x trajectories between multiprocessing and distributed schedulers. The DCD file format does not scale at all by increasing parallelism across different cores (Figure 5 A, D). This is due to the fact that IO time does not remain level by increasing the number of processes as discussed in the previous section. Our study showed that SSDs can be very helpful and can lead to better performance for all file formats especially DCD file format (Figure 5 A, D). XTC file format expresses reasonably well scaling with the increase in parallelism up to the limit of 24 (single node) for both multiprocessing and distributed scheduler. The NCDF file format scales very well up to 8 cores for all trajectory sizes. As the number of prcocesses increases the IO time also increases for NCDF file format and as a result the scaling is limited up to 8 CPU cores. For XTC file format, the I/O time is leveled up to 50 cores and compute time also remains level across parallelism up to 72 cores. Therefore, it is expected to achieve speed up, across parallelism up to 50 cores. However, based on Figure 5 E, XTC format only scales well up to 20 cores. Based on the present result, there is a difference between job execution time, and total compute and I/O time averaged over all processes (Figure 6 A). This difference increases with increase in trajectory size for all file formats for all machines (For details refer to the Technical Report [KB17]). This time difference is much smaller for Comet and Stampede as compared to other machines. The difference between job execution time and total compute and I/O time measured inside our code is very small for the results obtained using multiprocessing scheduler; however, it is considerable for the results obtained using distributed scheduler.

In order to obtain more insight on the underlying network behavior both at the worker level and communication level and in order to be able to see where this difference originates from we have used the web-interface of the Dask library. This web-interface is launched whenever Dask scheduler is launched. Figure 7 B, shows the comparison between timing measurements from instrumentation inside the Python code and Dask web-interface (average $n_{\mathrm{frames}}/N(t_{\mathrm{comp}}+t_{\mathrm{I/O}})$, $\max[n_{\mathrm{frames}}/N(t_{\mathrm{comp}}+t_{\mathrm{I/O}})]$, and t_N) for XTC600x on SDSC Comet for two different CPU cores ($N_{\mathrm{cores}} = 30$, $N_{\mathrm{cores}} = 54$). For $N_{\mathrm{cores}} = 54$, the measured $\max[n_{\mathrm{frames}}/N(t_{\mathrm{comp}}+t_{\mathrm{I/O}})]$ through our instrumentation inside the Python code and web-interface shows two different values. $\max[n_{\mathrm{frames}}/N(t_{\mathrm{comp}}+t_{\mathrm{I/O}})]$ measured using Dask web-interface is closer to the measured job execution time. The reason why $\max[n_{\mathrm{frames}}/N(t_{\mathrm{comp}}+t_{\mathrm{I/O}})]$ measured using Dask web-interface and our instrumentation are different is open to question. Based on task stream plot shown in Figure 7 A, the "straggler" task (#32) is much slower as compared to others and as a result slows down the whole process. But, the reason why the "straggler" task (#32) is delayed is not clear. The next sections in the present study aim to find the reason for which we are seeing these delayed tasks (so called "stragglers").

Challenges for Good HPC Performance

It should be noted that all the present results were obtained during normal, multi-user, production periods on all machines. In fact,

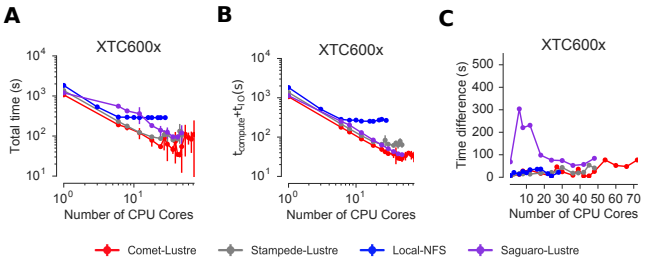

Fig. 6: *Detailed analysis of timings for the 600x XTC trajectory on HPC resources using Dask distributed. All other parameters as in Fig. 3.* ***A*** *Total time to solution (wall clock), t_N for N trajectory blocks using $N_{cores} = N$ CPU cores.* ***B*** *Sum of the I/O time per frame $t_{I/O}$ and the (constant) time for the RMSD computation t_{comp} (data not shown).* ***C*** *Difference $t_N - n_{frames}(t_{I/O} + t_{comp})$, accounting for the cost of communications and other overheads.*

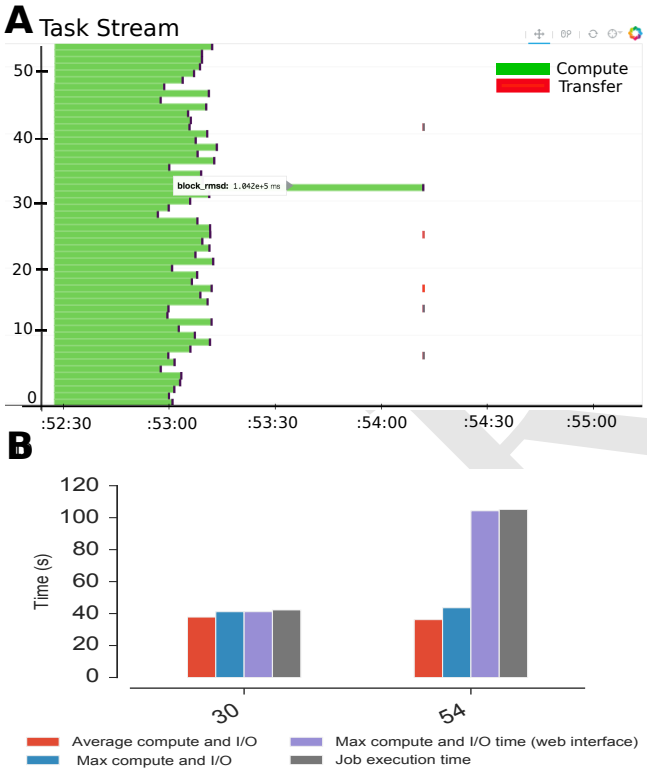

Fig. 7: *Evidence for uneven distribution of task execution times, shown for the XTC600x trajectory on SDSC* Comet *on the Lustre file system.* ***A*** *Task stream plot showing the fraction of time spent on different parts of the task by each worker, obtained using the Dask web-interface. (54 tasks for 54 workers that used $N = 54$ cores). Green bars ("Compute") represent time spent on RMSD calculations, including trajectory I/O, red bars show data transfer. A "straggler" task (#32) takes much longer than any other task and thus determines the total execution time.* ***B*** *Comparison between timing measurements from instrumentation inside the Python code (average compute and I/O time per task $n_{frames}/N\,(t_{comp} + t_{I/O})$, $\max[n_{frames}/N\,(t_{comp} + t_{I/O})]$, and t_N) and Dask web-interface for $N = 30$ and $N = 54$ cores.*

the time the jobs take to run is affected by the other jobs on the system. This is true even when the job is the only one using a particular node, which was the case in the present study. There are shared resources such as network file systems that all the nodes use. The high speed interconnect that enables parallel jobs to run is also a shared resource. The more jobs are running on the cluster, the more contention there is for these resources. As a result, the same job runs at different times will take a different amount of time to complete. In addition, remarkable fluctuations in task completion time across different processes is observed through monitoring network behavior using Dask web-interface. These fluctuations differ in each repeat and are dependent on the hardware and network. These factors further complicate any attempts at benchmarking. Therefore, this makes it really hard to optimize codes, since it is hard to determine whether any changes in the code are having a positive effect. This is because the margin of error introduced by the non-deterministic aspects of the cluster's environment is greater than the performance improvements the changes might produce. There is also variability in network latency, in addition to the variability in underlying hardware in each machine. This causes the results to vary significantly across different machines. Since our Map-reduce job is pleasantly parallel, each or a subset of computations can be executed independently on each process. Also, the claculations are load balanced which means that all of our processes have the same amount of work to do (One block per process). Therefore, observing these stragglers shown in Figure 7 A is unexpected and the following sections in the present study aim to identify the reason for which we are seeing these stragglers.

Performance Optimization

In the present section, we have tested different features of our computing environment to see if we can identify the reason for those stragglers and improve performance by avoiding the stragglers. Lustre striping, oversubscribing, scheduler throughput are tested to examine their effect on the performance. In addition, scheduler plugin is also used to validate our observations from Dask web-interface. In fact, we create a plugin that performs logging whenever a task changes state. Through the scheduler plugin we will be able to get lots of information about a task whenever it finishes computing.

Effect of Lustre Striping: As discussed before, the overlapping of data requests from different processes can lead to higher I/O time and as a result poor performance. This is strongly affecting our results since our compute per frame is not heavy and therefore the overlapping of data requests will be more frequent depending on the file format. The effect on the performance is strongly dependent on file format and some formats like XTC file formats which take advantage of in-built decompression are less affected by the contention from many data requests from many processes. However, when extending to multiple nodes, even XTC files are affected by this, as is also shown in Figure 3 B, E. In Lustre, a copy of the shared file can be in different physical storage devices (OSTs). Single shared files can have a stripe count equal to the number of nodes or processes which access the file. In the present study, we set the stripe count equal to three which is equal to the number of nodes used for our benchmark using distributed scheduler. This may be helpful to improve performance, since all the processes from each node will have a copy of the file and as a result the contention due to many data requests will decrease. Figure 8 show the speed up and I/O time per frame plots obtained

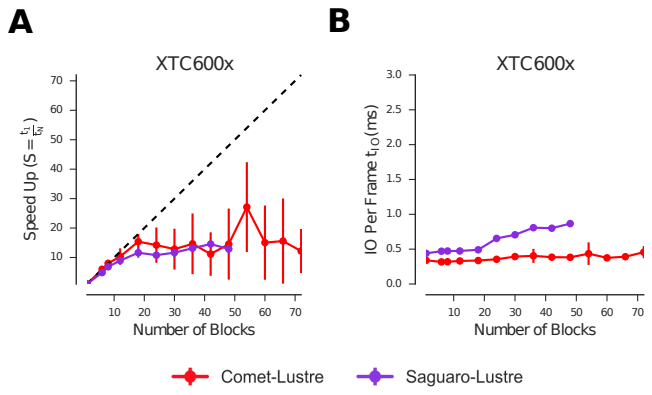

Fig. 8: *Effect of striping with the Lustre distributed file system. The XTC600x trajectory was analyzed on HPC resources (ASU RC* Saguaro*, SDSC* Comet*) with Dask distributed and a Lustre stripe count of three, i.e., data were replicated across three servers. One trajectory block was assigned to each worker, i.e., the number of tasks equaled the number of CPU cores.* ***A*** *Speed-up.* ***B*** *Average I/O time per frame, $t_{I/O}$.*

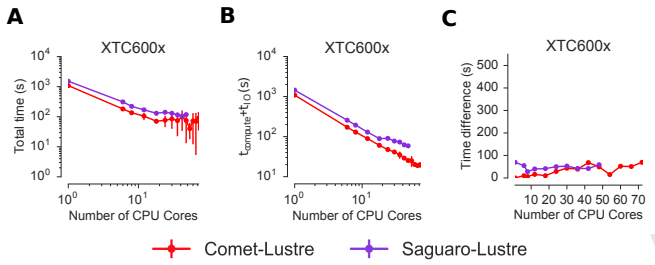

Fig. 9: *Detailed timings for three-fold Lustre striping (see Fig. 8 for other parameters).* ***A*** *Total time to solution (wall clock), t_N for M trajectory blocks using $N = M$ CPU cores.* ***B*** *$t_{comp} + t_{I/O}$, average sum of the I/O time ($t_{I/O}$, Fig. 8 B) and the (constant) time for the RMSD computation t_{comp} (data not shown).* ***C*** *Difference $t_N - n_{frames}(t_{I/O} + t_{comp})$, accounting for communications and overheads that are not directly measured.*

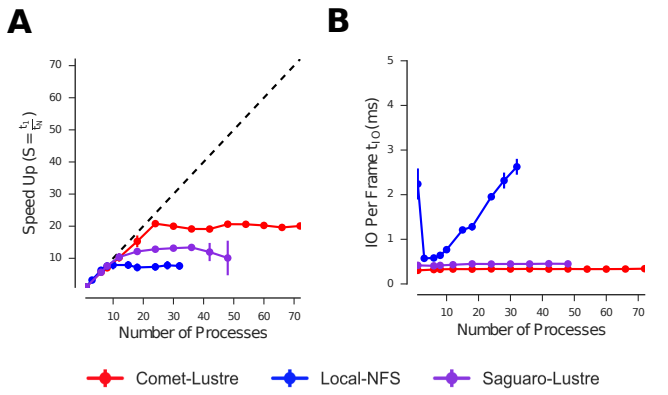

Fig. 10: *Effect of three-fold oversubscribing distributed workers. The XTC600x trajectory was analyzed on HPC resources (Lustre stripe count of three) and local NFS using Dask distributed where M number of trajectory blocks (tasks) is three times the number of worker processes, $M = 3N$, and there is one worker per CPU core.* ***A*** *Speed-up S.* ***B*** *I/O time $t_{I/O}$ per frame.*

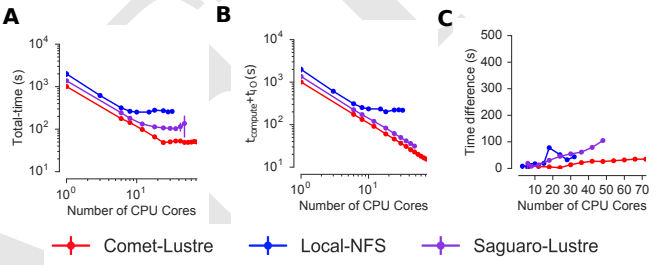

Fig. 11: *Detailed timings for three-fold oversubscribing distributed workers.* ***A*** *Total time to solution (wall clock), t_N.* ***B*** *$t_{comp} + t_{I/O}$, average sum of $t_{I/O}$ (Fig. 10 B) and the (constant) computation time t_{comp} (data not shown) per frame.* ***C*** *Difference $t_N - n_{frames}(t_{I/O} + t_{comp})$, accounting for communications and overheads that are not directly measured. Other parameters as in Fig. 10.*

for XTC file format (600X) when striping is activated. As can be seen, IO time is level across parallelism up to 72 cores which means that striping is helpful for leveling IO time per frame across all cores. However, based on the timing plots shown in Figure 9, there is a time difference between average total compute and I/O time and job execution time which is due to the stragglers and as a result the overall speed-up is not improved.

Effect of Oversubscribing: One useful way to robust our code to uncertainty in computations is to submit much more tasks than the number of cores. This may allow Dask to load balance appropriately, and as a result cover the extra time when there are some stragglers. In order for this, we set the number M of tasks to be three times the number of workers, $M = 3N$, where the number of workers $N = N_{\text{cores}}$ equals the number of CPU cores. Lustre-striping is also activated and is set to three which is also equal to number of nodes. Figures 10 show the speed up, and I/O time per frame plots obtained for XTC file format (XTC600x). As can be seen, IO time is level across parallelism up to 72 cores because of striping. However, based on the timing plots shown in Figure 11, there is a time difference between average total compute and I/O time and job execution time which reveals that oversubscribing does not help to remove the stragglers and as a result the overall speed-up is not improved. Figure 12 shows time comparison on different parts of the calculations. Bars are subdivided into the contribution of overhead in the calculations, communication time and RMSD calculation across parallelism from 1 to 72. RMSD calculation is the time spent on RMSD tasks, and communication time is the time spent for gathering RMSD arrays calculated by each processor rank. As can be seen in Figure 12, the overhead in the calculations is small up to 24 cores (Single node). The largest fraction of the calculations is spent on the calculation of RMSD arrays (computation time) which decreases pretty well as the number of cores increases from 1 to 72. However, when extending to multiple nodes the time due to overhead and communication increases which affects the overall performance.

Examining Scheduler Throughput

An experiment were executed with Dask schedulers (multi-threaded, multiprocessing and distributed) on Stampede. In each run a total of 100000 zero workload tasks were executed. Figure 13 A shows the Throughput of each scheduler over time on a single Stampede node - Dask scheduler and worker are on the same node. Each value is the mean throughput value of several runs for each scheduler.

Our understanding is that the most efficient scheduler is the distributed scheduler, especially when there is one worker process

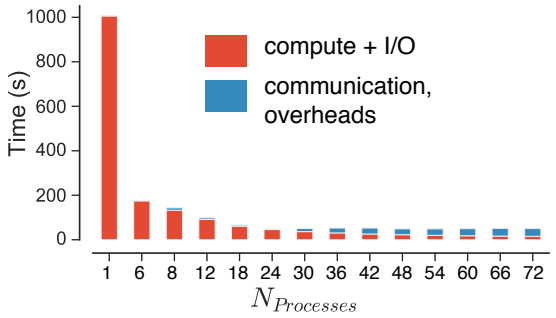

Fig. 12: *Time comparison for three-fold oversubscribing distributed workers (XTC600x on SDSC* Comet *on Lustre with stripe count three). Bars indicate the mean total execution time t_N (averaged over repeats) as a function of available worker processes, with one wo per CPU core. Time for compute + I/O (red, see Fig. 11 B) domin for smaller core counts (up to one node, 24) but is swamped communication and overheads (blue, see see Fig. 11 C) beyon single node.*

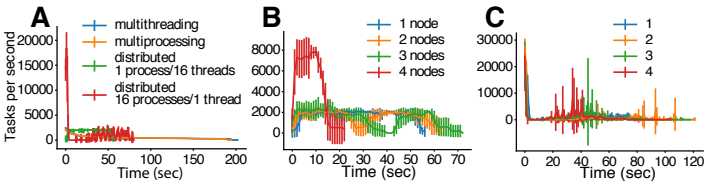

Fig. 13: *Benchmark of Dask scheduler throughput on TACC* Stampede. *Performance is measured by the number of empty `pass` tasks that were executed in a second. The scheduler had to lauch 100,000 tasks and the run ended when all tasks had been run. **A** single node with different schedulers; multithreading and multiprocessing are almost indistinguishable from each other. **B** multiple nodes with the distributed scheduler and 1 worker process per node. **C** multiple nodes with the distributed scheduler and 16 worker processes per node.*

for each available core. Also, the distributed with just one worker process and a number of threads equal to the number of available cores are still able to schedule and execute these 100,000 tasks. The multiprocessing and multithreading schedulers have similar behavior again, but need significantly more time to finish compared to the distributed.

Figure 13 B shows the distributed scheduler's throughput over time when the number of Nodes increases. Each node has a single worker process and each worker launches a thread to execute a task (maximum 16 threads per worker). By increasing the number of nodes we can see that Dask's throughput increases by the same factor. Figure 13 C shows the same execution with the Dask cluster being setup to have one worker process per core. In this figure, the scheduler does not reach its steady throughput state, compared to 13 B, thus it is not clear what is the effect of the extra nodes. Another interesting aspect is that when a worker process is assigned to each core, Dask's Throughput is an order of magnitude larger allowing for even faster scheduling decisions and task execution.

Scheduler Plugin Results

In addition to Dask web-interface, we implemented a Dask scheduler plugin. This plugin captures task execution events from the scheduler and their respective timestamps. These captured profiles were later used to analyze the execution of XTC 300x on Stampede. In all the previous benchmarks in the present study,

RMSD Blocks	Run 1	Run 2	Run 3	Run 4	Run 5
1	0	0	1	0	0
2	8	5	7	7	2
3	48	54	56	50	60
4	8	5	0	7	2

TABLE 1: *Summary of the number of worker processes per submitted RMSD blocks. Each column shows the total number of Worker process that executed a number of RMSD blocks per run. Executed on TACC Stampede utilizing 64 cores*

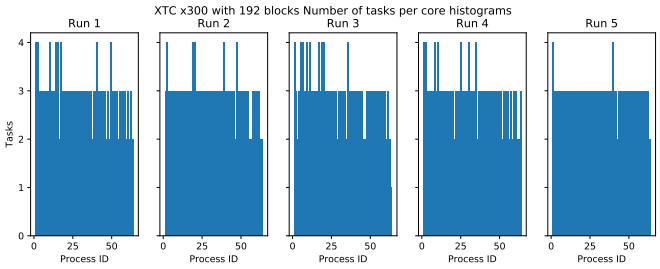

Fig. 14: *Task Histogram of RMSD with MDAnalysis and Dask with XTC 300x over 64 cores on Stampede with 192 blocks. Each histogram is a different run of the same execution. The X axis is worker process ID and the Y axis the number of tasks submitted to that process.*

number of blocks is equal to the number of processes ($N = N_{\text{cores}}$). However, when extended to multiple nodes the whole calculation is delayed due to the stragglers and as a result the overall performance was affected. In the present section, we repeated the benchmark where the number of blocks is three times the number of processes ($N = 3 * N_{\text{cores}}$). We were able to measure how many tasks are submitted per worker process. This exexutions are performed to see why oversubscribing introduced in the previous section was not helpful. Table 1 summarizes the results and Figure 14 shows in detail how RMSD blocks were submitted per worker process in each run. As it is shown the execution is not balanced between worker processes. Although, most workers are calculating three RMSD blocks, as it is expected by oversubscribing, there are a few workers that are receiving a smaller number of blocks and workers that receive more than three. Therefore, we can conclude that over-subscription does not necessarily lead to a balanced execution, adding additional execution time.

Comparison of Performance of Map-Reduce Job Between MPI for Python and Dask Frameworks

Based on the results presented in previous sections, it turned out that the stragglers are not because of the scheduler throughput. Lustre striping improves I/O time; however, the job computation is still delayed due to stragglers and as a result performance is not improved. In order to make sure if the stragglers are created because of scheduler overhead in Dask framework we have tried to measure the performance of our Map-Reduce job using an MPI-based implementation, which makes use of mpi4py [DPS05], [DPKC11]. This will let us figure out whether the stragglers observed in the present benchmark using Dask parallel library are as a result of scheduler overhead or any other factor than scheduler. The comparison is performed on XTC 600x using SDSC Comet. Figure 15 A shows time comparison on different parts of the

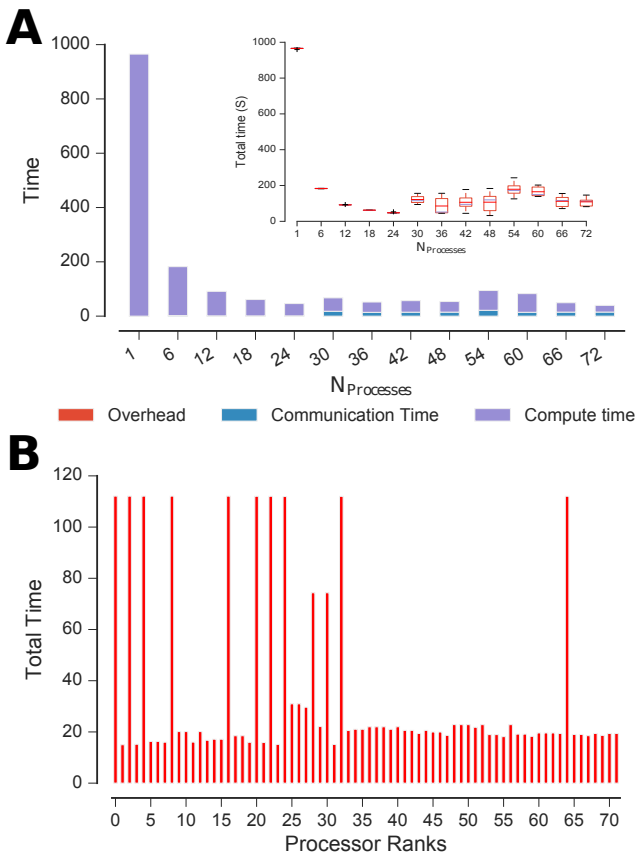

Fig. 15: ***A*** *Time comparison on different parts of the calculations obtained using MPI for python. In this aggregate view, the time spent on different parts of the calculation are combined for different number of processes tested. The bars are subdivided into the contributions of each time spent on different parts. Reported values are the mean values across 5 repeats.* ***A inset*** *Total job execution time along with the mean and standard deviations across 5 repeats across parallelism from 1 to 72 obtained using MPI for python. The calculations are performed on XTC 600x using SDSC Comet.* ***B*** *Comparison of job execution time across processor ranks for 72 CPU cores obtained using MPI for python. There are several stragglers which slow down the whole process.*

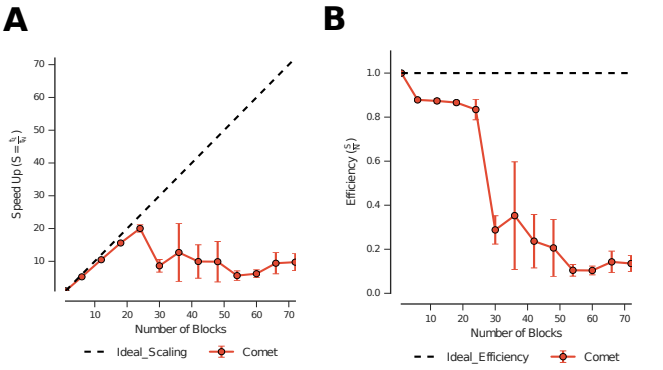

Fig. 16: ***A*** *Speed-up and* ***B*** *efficiency plots for benchmark performed on XTC 600x on SDSC Comet across parallelism from 1 to 72 using MPI for python. Five repeats are run for each block size to collect statistics and the reported values are the mean values across 5 repeats.*

calculations. Bars are subdivided into the contribution of overhead in the calculations, communication time and RMSD calculation across parallelism from 1 to 72. Computation time is the time spent on RMSD tasks, and communication time is the time spent for gathering RMSD arrays calculated by each processor rank. Total time is the summation of communication time, computation time and the overhead in the calculations. As can be seen in Figure 15 A, the overhead in the calculations is small up to 24 cores (Single node). Based on Figure 15, the communication time is very small up to a single node and increases as the calculations are extended to multiple nodes. Overall, only a small fraction of total time is spent on communications. Overhead in the calculations is also very small. The largest fraction of the calculations is spent on the calculation of RMSD arrays (computation time) which decreases pretty well as the number of cores increases for a sigle node. However, when extending to multiple nodes computation time also increases. We believe that this is caused due to stragglers which is also confirmed based on Figure 15 A.

Figure 15 B, shows comparison of job execution time across all ranks tested with 72 cores. As seen in Figure 15 B, there are several slow processes as compared to others which slow down the whole process and as a result affect the overall performance. These stragglers are observed in all cases when number of cores is more than 24 (extended to multiple cores). However, they are only shown for $N = 72$ CPU cores for the sake of brevity.

Overall speed-up along with the efficiency plots are shown in Figure 16. As seen the overall performance is affected when extended to multiple nodes (more than 24 CPU cores).

Based on the results from MPI for python the reason for stragglers is not the Dask scheduler overhead. In order to make sure that the reason for stragglers is not the qcprot RMSD calculation we tested the performance of our code using another metric MDAnalysis.lib.distances.distance_array. This metric calculates all distances between a reference set and another configuration. Even with the new metric the same behavior observed and hence we can conclude that qcprot RMSD calculation is not the reason why we are seeing the stragglers. Further studies are necessary to identify the underlying reason for the stragglers observed in the present benchmark.

Conclusions

In summary, Dask together with MDAnalysis makes it straightforward to implement parallel analysis of MD trajectories within a map-reduce scheme. We show that obtaining good parallel performance depends on multiple factors such as storage system and trajectory file format and provide guidelines for how to optimize trajectory analysis throughput within the constraints of a heterogeneous research computing environment. Nevertheless, implementing robust parallel trajectory analysis that scales over many nodes remains a challenge.

Acknowledgments

MK and IP were supported by grant ACI-1443054 from the National Science Foundation. SJ and OB were supported in part by grant ACI-1443054 from the National Science Foundation. Computational resources were in part provided by the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number

ACI-1053575 (allocation MCB130177 to OB and allocation TG-MCB090174 to SJ) and by Arizona State University Research Computing.

REFERENCES

[AMS+15] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1–2:19 – 25, 2015. URL: http://www.gromacs.org, doi:10.1016/j.softx.2015.06.001.

[BBIM+09] B R Brooks, C L Brooks III., A D Jr Mackerell, L Nilsson, R J Petrella, B Roux, Y Won, G Archontis, C Bartels, S Boresch, A Caflisch, L Caves, Q Cui, A R Dinner, M Feig, S Fischer, J Gao, M Hodoscek, W Im, K Kuczera, T Lazaridis, J Ma, V Ovchinnikov, E Paci, R W Pastor, C B Post, J Z Pu, M Schaefer, B Tidor, R M Venable, H L Woodcock, X Wu, W Yang, D M York, and M Karplus. CHARMM: the biomolecular simulation program. *J Comput Chem*, 30(10):1545–1614, Jul 2009. URL: https://www.charmm.org, doi:10.1002/jcc.21287.

[CCD+05] David A Case, Thomas E Cheatham, 3rd, Tom Darden, Holger Gohlke, Ray Luo, Kenneth M Merz, Jr, Alexey Onufriev, Carlos Simmerling, Bing Wang, and Robert J Woods. The amber biomolecular simulation programs. *J Comput Chem*, 26(16):1668–1688, 2005. URL: http://ambermd.org/, doi:10.1002/jcc.20290.

[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.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. doi:10.1145/1327452.1327492.

[DPKC11] Lisandro D. Dalcin, 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.

[DPS05] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. Mpi for python. *Journal of Parallel and Distributed Computing*, 65(9):1108 – 1115, 2005. doi:10.1016/j.jpdc.2005.03.010.

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

[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 102 – 109, Austin, TX, 2016. SciPy. URL: http://mdanalysis.org.

[KB17] Mahzad Khoshlessan and Oliver Beckstein. Parallel analysis in the MDAnalysis library: Benchmark of trajectory file formats. Technical report, Arizona State University, Tempe, AZ, 2017. doi:10.6084/m9.figshare.4695742.

[LAT10] Pu Liu, Dimitris K Agrafiotis, and Douglas L. Theobald. Fast Determination of the Optimal Rotational Matrix for Macromolecular Superpositions. *J Comput Chem*, 31(7):1561–1563, 2010. doi:10.1002/jcc.21439.

[MADWB11] Naveen Michaud-Agrawal, Elizabeth Jane Denning, Thomas B. Woolf, and Oliver Beckstein. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. *J Comput Chem*, 32:2319–2327, 2011. URL: http://mdanalysis.org, 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.

[RCI13] Daniel R. Roe and Thomas E. Cheatham III. PTRAJ and CPPTRAJ: Software for processing and analysis of molecular dynamics trajectory data. *J Chemical Theory Computation*, 9(7):3084–3095, 2013. URL: https://github.com/Amber-MD/cpptraj, doi:10.1021/ct400341p.

[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.

[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.

[SB17] Sean Seyler and Oliver Beckstein. Molecular dynamics trajectory for benchmarking MDAnalysis, 6 2017. URL: https://figshare.com/articles/Molecular_dynamics_trajectory_for_benchmarking_MDAnalysis/5108170, doi:10.6084/m9.figshare.5108170.

[TRB+08] T. Tu, C.A. Rendleman, D.W. Borhani, R.O. Dror, J. Gullingsrud, MO Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K.A. Stafford, and David E. Shaw. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *International Conference for High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008.*, pages 1–12, Austin, TX, 2008. IEEE. doi:10.1109/SC.2008.5214715.

[VCV11] Stefan Van Der Walt, S. Chris Colbert, and Gael Varoquaux. The NumPy array: A structure for efficient numerical computation. *Comput Sci Eng*, 13(2):22–30, 2011. URL: http://www.numpy.org/, arXiv:1102.1523, doi:10.1109/MCSE.2011.37.