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 evaluate 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 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 bottle neck 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 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 that allows structural and temporal analysis of molecular dynamics (MD) trajectories generated by all major MD simulation packages [GLB+16], [MADWB11]. The size of these trajectories is growing as the simulation times is being extended from micro-seconds to milli-seconds and larger systems with increasing numbers of atoms are simulated. Thus, the amount of data to be analyzed is growing rapidly (into the terabyte range) and analysis is increasingly becoming a bottleneck [CR15]. Therefore, there is a need for high performance computing (HPC) approaches to increase the throughput. MDAnalysis does not yet provide a standard interface for parallel analysis; instead, various existing parallel libraries are currently used to parallelize MDAnalysis-based code. Here we evaluate performance for parallel map-reduce type analysis with the Dask parallel computing library [Roc15] for task-graph based distributed computing on HPC and local computing resources. As the computational task we perform an optimal structural superposition of the atoms of a protein to a reference structure by minimizing the RMSD of the C_α atoms. A range of commonly used MD file formats (CHARMM/NAMD DCD [BBIM+09], Gromacs XTC [AMS+15], Amber NetCDF [CCD+05]) and different trajectory sizes are benchmarked on different HPC resources including national supercomputers (XSEDE TACC Stampede and SDSC Comet), university supercomputers (ASU Research computing center (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. Such a heterogeneous environment creates a challenging problem for developing high performance programs without the effort required to use low-level, architecture specific parallel programming models for our domain-specific problem. Different storage systems such as solid state drives (SSDs), hard disk drives (HDDs), and the parallel Lustre file system (implemented on top of HDD) are also tested to examine effect of I/O on the performance. The benchmarks are performed both on a single node and across multiple nodes using the multiprocessing and distributed schedulers in Dask library. A protein system of $N = 3341$ atoms per frame but with different number of frames per trajectory was analyzed. We used different trajectory sizes of 50 GB, 150 GB, and 300 GB for Dask multiprocessing and 100 GB, 300 GB, 600 GB for Dask distributed; however, here we only present data for the 300 GB and 600 GB trajectory sizes, which represent typical medium and large size results. For an analysis of the full data set see the Technical Report [KB17]. All results for Dask distributed are obtained across three nodes on different clusters. Results are compared across all file formats, trajectory sizes, and machines. Our results show strong dependency on the storage system because a key problem is competition for access to the same file from multiple processes. However, the exact data access pattern depends on the trajectory file format and a strong dependence on the actual data format arises. Some trajectory formats are more robust against storage system specifics than others. In particular, analysis with the Gromacs XTC format can show strong ideal scaling over multiple nodes because this highly compressed format effectively reduces (global) I/O at the expense of increasing (local) per-core work for decompression. Our results show that there can be other challenges aside from the I/O bottleneck for achieving good speed-up. For instance, with numbers of processes matched to the available

‡ Arizona State University
§ RADICAL, ECE, Rutgers University, Piscataway, NJ 08854, USA
∗ Corresponding author: obeckste@asu.edu

cores, contention on the network may slow down individual tasks and lead to poor load balancing and poor overall performance. In order to identify the performance bottlenecks for our Map-Reduce Job, we have tested and examined several other factors including striping, oversubscribing, and the Dask Scheduler. We also compared a subset of systems with an MPI-based implementation (using mpi4py) in order to better understand the effect of using a high-level approach such as the Dask parallel library compared to a lower level one; in particular, we tried to identify possible underlying factors that may lead to low performance.

Methods

The data files consist of a topology file (in CHARMM PSF format) and a trajectory (DCD format); they are available from dropbox (adk4AKE.psf and 1ake_007-nowater-core-dt240ps.dcd). Files in XTC and NetCDF formats are generated from the DCD on the fly. To avoid operating system caching, files are copied and only used once for each benchmark. 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]. As computational load we implement the calculation of the root mean square distance of the C_α atoms of the protein adenylate kinase when it fitted to a reference structure using an optimal rigid body superposition, using the qcprot implementation [LAT10] in MDAnalysis [GLB+16]. The code for benchmarking is available from https://github.com/Becksteinlab/Parallel-analysis-in-the-MDAnalysis-Library .

Results and Discussion

Effect of I/O Environment

In MDAnalysis library, trajectories from MD simulations are a frame by frame description of the motion of particles as a function of time. To allow the analysis of large trajectories, MDAnalysis only loads a single frame into memory at any time [GLB+16], [MADWB11]. Some file systems are designed to run on a single CPU while others like Network File System (NFS) which is among distributed file systems are designed to let different processes on multiple computers access a common set of files. These file systems guarantees sequential consistency which means that it prevents any process from reading a file while another process is reading the file. Distributed parallel file systems (Lustre) allow simultaneous access to the file by different processes; however it is very important to have a parallel I/O library; otherwise the file system will process the I/O requests it gets serially, yielding no real benefit from doing parallel I/O. Figure *fig:pattern-formats* shows the I/O pattern compared between different file formats.

XTC file format takes advantage of in-built compression and as a result has smaller file size as compared to the other formats. In addition, MDAnalysis implements a fast frame scanning algorithm for XTC files. This algorithm computes frame offsets and saves the offsets to disk as a hidden file once the trajectory is read the first time. When a trajectory is loaded again then instead of reading the whole trajectory the offset is used to seek individual frames. As a result, the time it takes a process to load a frame into memory is short (Figure 2 B and 3 B). In addition, each frame I/O will be followed by decompressing of that frame as soon as it is loaded into memory (see Figure 1 A). Thus, as soon as the frame is loaded into memory by one process, the file system will let the next process to load another frame into memory. This happens while the first process is decompressing the loaded frame.

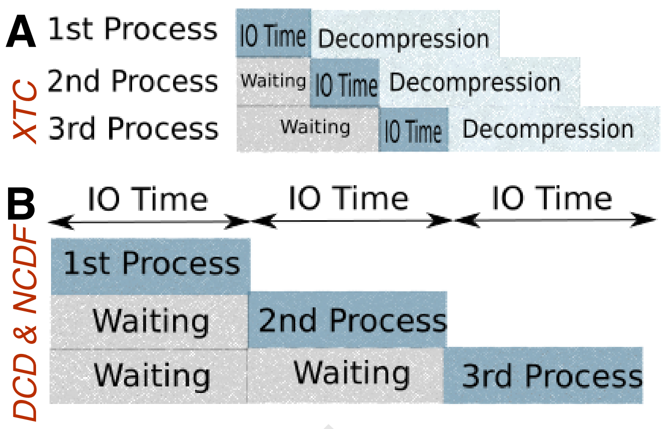

Fig. 1: I/O pattern for reading frames in parallel from commonly used MD trajectory formats. ***A*** *Gromacs XTC file format.* ***B*** *CHARMM/NAMD DCD file format and Amber netCDF format.*

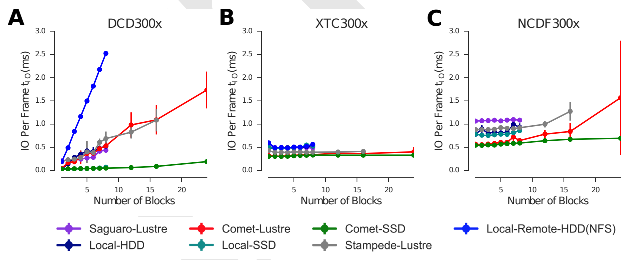

Fig. 2: Comparison of IO time between 300x trajectory sizes using dask multiprocessing on a single node. *The trajectory was split into N blocks and computations were performed using $N_{cores} = N$ CPU cores. The runs were performed on different resources (ASU RC* Saguaro, *SDSC* Comet, *TACC* Stampede, local *workstations with different storage systems (locally attached* HDD, remote HDD *(via network file system), locally attached* SSD, Lustre *parallel file system with a single stripe).*

As a result, the overlapping of the data requests for the same calculation will be less frequent. However, there is no in-built compression for DCD and netCDF file formats and as a result file sizes are larger. This will result in higher I/O time and therefore overlapping of per frame trajectory data access (Figure 1 B). The I/O time is larger for netCDF file format as compared to DCD file format due to larger file size (Figure 2 A, C). This is since netCDF has a more complicated file format. Reading an existing netCDF dataset involves opening the dataset; inquiring about dimensions, variables, and attributes; reading variable data; and closing the dataset [ref]. In fact, netCDF has a very sophisticated format, while DCD has a very simple file format. This is why DCD is showing a weak scaling by increasing parallelism whereas netCDF file format is being scaled reasonably well by increasing parallelism across many cores. Figures *fig:IO-multiprocessing* and *fig:IO-distributed* compare the difference in I/O time for different file formats for 300X and 600X trajectories for multiprocessing and distributed scheduler respectively. According to figure *fig:IO-multiprocessing*, SSD can be very helpful (especially for dcd file format) and can improve the performance due to speed up in access time. Also we anticipate that, for heavier analyses that have higher compute time per frame, per frame trajectory data access happens less often and accession times gradually become staggered across CPUs which can be considered for future studies.

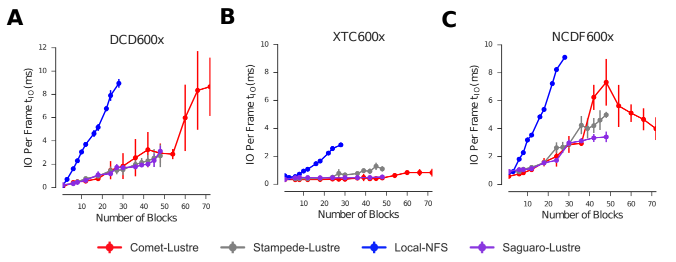

Fig. 3: Comparison of IO time between 600x trajectory sizes using dask distributed on one to three nodes. The trajectory was split into N blocks and computations were performed using $N_{cores} = N$ CPU cores. The runs were performed on different resources (ASU RC Saguaro, *SDSC* Comet, *TACC* Stampede, *all using Lustre with a single stripe as the parallel file system and* local *workstations with NFS).*

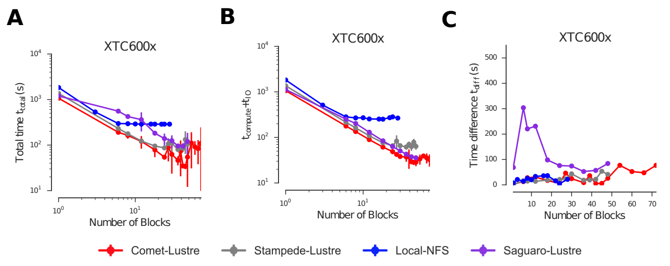

Fig. 4: Timings for various parts of the computation for the 600x XTC trajectory on HPC resources using dask distributed. The runs were performed on different resources (ASU RC Saguaro, *SDSC* Comet, *TACC* Stampede, *all using Lustre with a single stripe as the parallel file system and* local *workstations with NFS). **A** Total time to solution (wall clock), t_N for N trajectory blocks using $N_{cores} = N$ CPU cores. **B** Sum of the measured I/O time $t_{I/O}$ and the (constant) time for the RMSD computation t_{comp} (data not shown). **C** Difference $t_N - (t_{I/O} + t_{comp})$, accounting for other load that is not directly measured.*

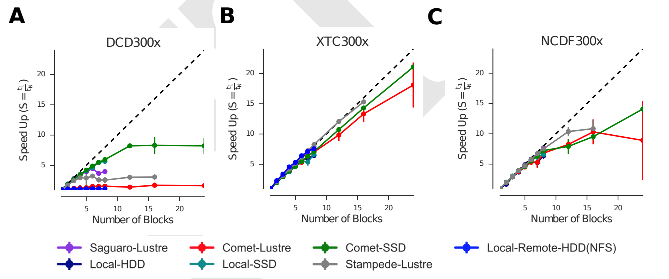

Fig. 5: Speed-up for the analysis of the 300x trajectory on HPC resources using dask multiprocessing. The dashed line shows the ideal limit of strong scaling. The runs were performed on different resources (ASU RC Saguaro, *SDSC* Comet, *TACC* Stampede, *all using Lustre with a single stripe as the parallel file system and* local *workstations with NFS). **A** CHARMM/NAMD DCD. **B** Gromacs XTC. **C** Amber netCDF.*

Effect of File Format

Figure 5 and 6 shows speedups for 300x and 600x trajectories for the multiprocessing and distributed scheduler as an example of using HPC resources for a big trajectory. The DCD file format does not scale at all by increasing parallelism across different cores (Figure 6 A). This is due to the overlapping of the data access requests from different processes. 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). XTC file format express 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. 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, the 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 4). This difference increases with increase in trajectory size for all file formats for all machines (not shown here). 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 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. Table 1 summarizes the average and max total compute and I/O time measured through our code, max total compute and I/O time measured using the web interface and job execution time for each of the cases tested. 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. As seen from the tests performed on our local machines, there is a very small difference between maximum total compute and I/O time and job execution time. This difference is mostly due to communications performed in the reduction process. In addition, maximum total compute and I/O time measured using the web interface and our code are very close. As seen in Table and Figure 7, for SDSC Comet ($N_{\text{cores}} = 54$), there is a very small difference between maximum total compute and I/O time measured using the web interface and job execution time. However, there is a considerable difference between maximum total compute and I/O time measured using the web interface and our code. There is one process which is much slower as compared to others. As can be seen from the results, some tasks (so-called Stragglers) are considerably slower than the others, delaying the completion of the job and as a result affect the overal performance.

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, the time the jobs take to run are 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 filesystems 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

Resource	N_{cores}	Average total compute and I/O time(s)	Max total compute and I/O time(s)	Max total compute and I/O time measured using web interface(s)	Job executio time(s)
Local	24	93.83	110.58	110.43	111.83
Local	28	86.54	111.54	111.24	112.81
SDSC Comet	30	37.79	41.11	41.12	42.23
SDSC Comet	54	36.15	43.58	104.25	105.1

TABLE 1: *Summary of the measured times for different calculations, tested on different machines for 600X trajectory and XTC file format. N_{cores} is the number of cores used in each test, average total compute and I/O time is the I/O plus compute time for all frames per process averaged across all processes, max total compute and I/O time is the I/O plus compute time for all frames for the slowest process measured through the code, max total compute and I/O time measured using web interface is the I/O plus compute time for all frames for the slowest process measured through web interface.*

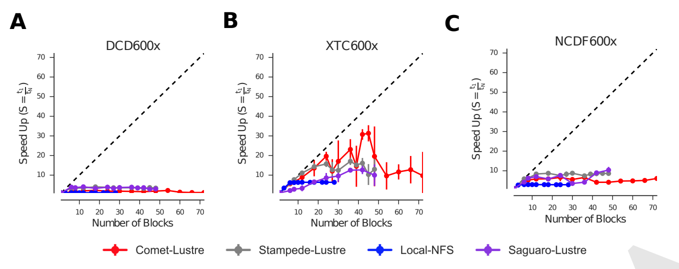

Fig. 6: *Speed-up for the analysis of the 600x trajectory on HPC resources using dask distributed. The dashed line shows the ideal limit of strong scaling. The runs were performed on different resources (ASU RC* Saguaro*, SDSC* Comet*, TACC* Stampede*, all using Lustre with a single stripe as the parallel file system and* local *workstations with NFS).* ***A*** *CHARMM/NAMD DCD.* ***B*** *Gromacs XTC.* ***C*** *Amber netCDF.*

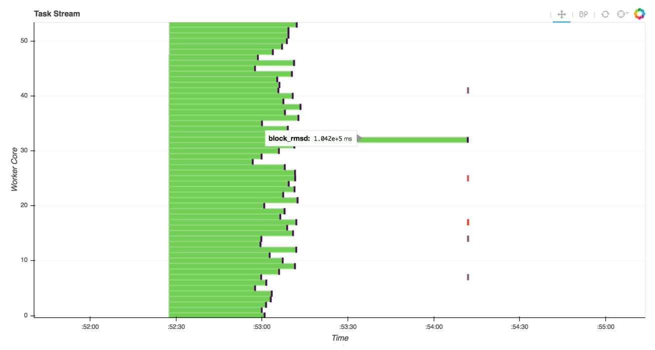

Fig. 7: *Task stream plots showing the fraction of time spent on different computations by each worker obtained using dask web interface (tested on SDSC Comet using $N_{cores} = 54$ for 600X trajectory and XTC file format-Green bars represent time spent on RMSD calculations)*

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, all of our processes have the same amount of work to do and our Map-Reduce job is load balanced. Therefore, observing these stragglers discussed in the previous section 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 observation using 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 is more frequent. 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 more than one node, even XTC files are affected by this, as is also shown in the previous sections. 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. 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 for XTC file format (600X) when striping is activated. As can be seen, IO time is level across parallelism

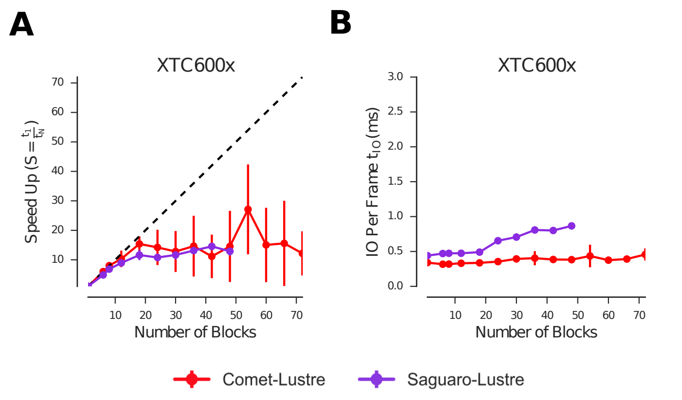

Fig. 8: ***A*** *Speed-up for the analysis of the 600x trajectory on HPC resources using dask distributed with strip count of three. The dashed line shows the ideal limit of strong scaling.* ***B*** *Comparison of IO time between 600x trajectory sizes using dask distributed on one to three nodes. The runs were performed on different resources (ASU RC* Saguaro*, SDSC* Comet*, all using Lustre with strip count of three as the parallel file system).*

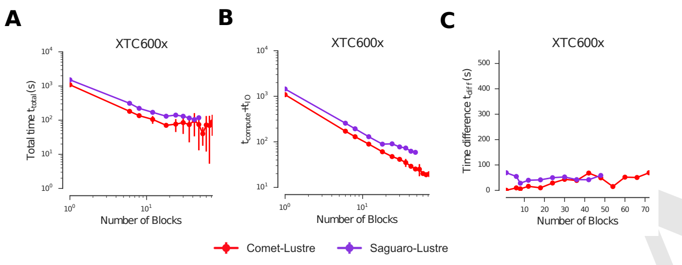

Fig. 9: Timings for various parts of the computation for the 600x XTC trajectory on HPC resources using dask distributed. The runs were performed on different resources (ASU RC Saguaro*, SDSC* Comet*, all using Lustre with stripe count of three as the parallel file system and* local *workstations with NFS).* ***A*** *Total time to solution (wall clock), t_N for N trajectory blocks using $N_{cores} = N$ CPU cores.* ***B*** *Sum of the measured I/O time $t_{I/O}$ and the (constant) time for the RMSD computation t_{comp} (data not shown).* ***C*** *Difference $t_N - (t_{I/O} + t_{comp})$, accounting for other load that is not directly measured.*

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 aresult the overal speed-up is not improved as compared to what we had in Figure 6.

Effect of Oversubscribing: One useful way to robust our code to uncertainty in computations is to submit many more tasks than the numer 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 of tasks to be three times the number of workers ($N_{\text{Blocks}} = 3 * N_{\text{cores}}$). 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 (600X). 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 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 aresult the overal speed-up is not

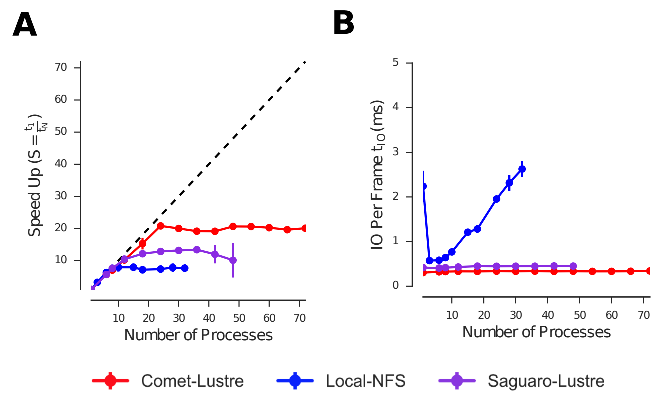

Fig. 10: ***A*** *Speed-up for the analysis of the 600x trajectory on HPC resources using dask distributed with strip count of three. The dashed line shows the ideal limit of strong scaling.* ***B*** *Comparison of IO time between 600x trajectory sizes using dask distributed on one to three nodes. The runs were performed on different resources (ASU RC* Saguaro*, SDSC* Comet*, and our local machines, all using Lustre with strip count of three as the parallel file system). N number of blocks is three times the number of processes $N = 3 * N_{cores}$*

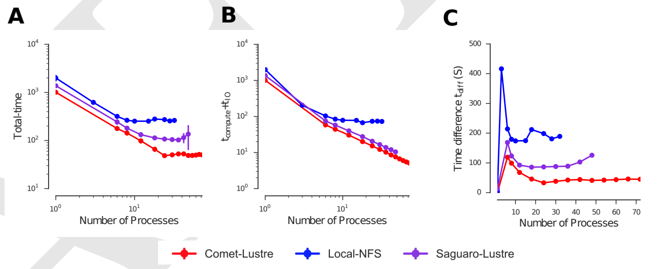

Fig. 11: Timings for various parts of the computation for the 600x XTC trajectory on HPC resources using dask distributed. The runs were performed on different resources (ASU RC Saguaro*, SDSC* Comet*, and our local machines, all using Lustre with stripe count of three as the parallel file system and* local *workstations with NFS).* ***A*** *Total time to solution (wall clock), t_N for N trajectory blocks using $N = 3 * N_{cores}$ CPU cores.* ***B*** *Sum of the measured I/O time $t_{I/O}$ and the (constant) time for the RMSD computation t_{comp} (data not shown).* ***C*** *Difference $t_N - (t_{I/O} + t_{comp})$, accounting for other load that is not directly measured.*

improved as compared to what we had in Figure 6. In order to see if the calculation is load balanced and the same amount of load is assigned to each worker by the scheduler, scheduler pluging is used to get detailed information about each task and to also validate our observationis from Dask web-interface. The results from scheduler pluging is described in the following section.

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 12 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 for each available core. Also, the Distributed with just one worker process and a number of threads equal to the number of

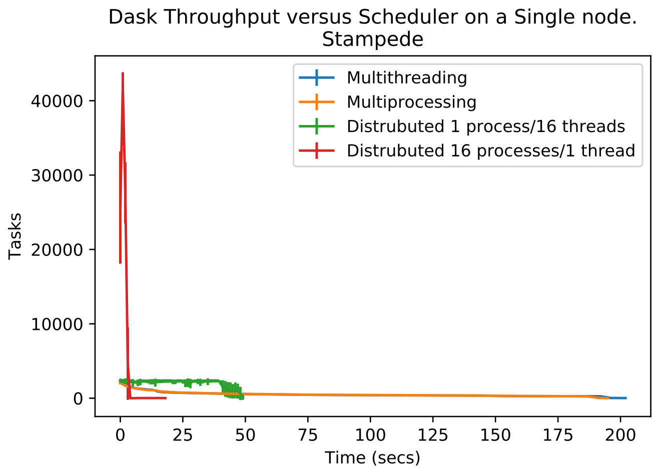

Fig. 12: *Dask Throughput on a single node vs Scheduler type. X axis is time and Y axis is the number of tasks that were executed in a second.*

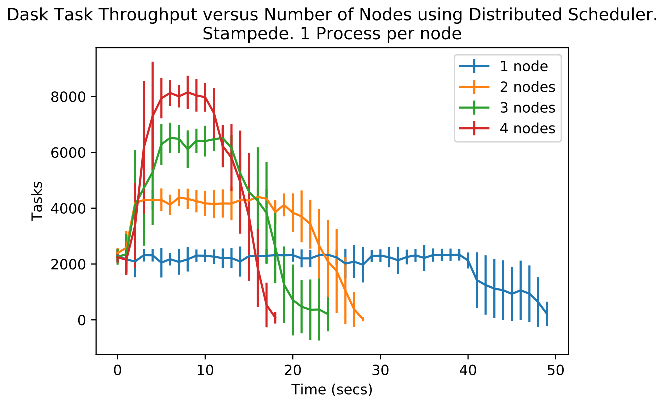

Fig. 13: *Dask Throughput vs Number of Nodes. X axis is time and Y axis is the number of tasks that were executed in a second.*

available cores is still able to schedule and execute these 100000 tasks. The Multiprocessing and Multithreading Schedulers have similar behavior again, but need significantly more time to finish compared to the Distributed.

Figure 13 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 14 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, 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's web interface, we implemented a Dask Scheduler Plugin. This plugin captures task execution events from the scheduler and their respective timestamps. These captured

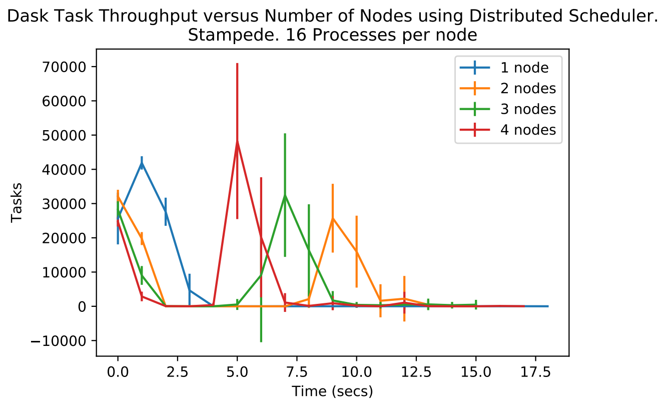

Fig. 14: *Dask Throughput vs Number of Nodes. X axis is time and Y axis is the number of tasks that were executed in a second*

profiles were later use to analyze the execution of and XTC 300x on Stampede. Figure 15 shows characteristic executions. The delay in execution (Figure 15 A) is because of the specific cores and not something that is part of the framework. This is eliminated when cores are oversubscribed by a factor of three (i.e., three times as many blocks as cores (Figure 15 B).

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 network, shared resources or scheduler throughput. Lustre striping improves I/O time; however, the job computation is still delayed and as a result lead to poor speed-up when extended to multiple nodes. 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 MPI-based implementation. This will let us figure out whether the stragglers observed in the present benchmark using Dask parallel libray are as a result of scheduler overhead or the environment itself.

Performance of our Map-reduce job with a different task than RMSD calculation

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).

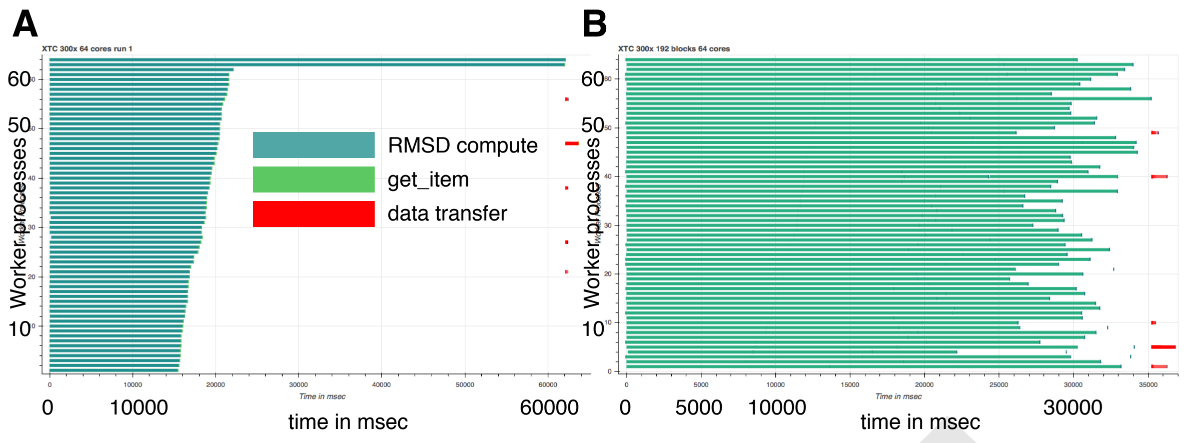

Fig. 15: Task Stream of RMSD with MDAnalysis and Dask with XTC 300x over 64 cores on Stampede with 64 blocks (right) and 192 blocks (left). The X axis is time in milliseconds and the Y axis Worker process ID. Dark Green is the computation of RMSD for each data chunk, Light Green are the Get Item tasks and Red is data transfer.

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.

[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.

[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.

[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.