Applications and Performance of HPC Resources for Parallel Analysis of Molecular Dynamics Trajectories with Python

April 26, 2018

Abstract

Typical size of the molecular dynamics (MD) trajectories from data-intensive bio-molecular simulations ranges from gigabytes to terabytes. However, the computational biophysics community misses effective use of high performance computing (HPC) resources for efficiently analyzing these trajectories and more importantly achieving linear scaling still remains a big challenge. Present work aims to provide insights, guidelines and strategies to the community on how to take advantage of the available HPC resources to gain the best possible performance. We investigated a single program multiple data (SPMD) execution model where each process executes the same program, to parallelize the Map-Reduce root mean square distance (RMSD) and Dihedral Featurization algorithms for analysis of MD trajectories in the MDAnalysis library. We employ the Python language because it is widely used in the bio-molecular simulation field and focus on an MPI-based implementation. We notice that straggler tasks negatively impact the performance and act as scalability bottlenecks. Straggler tasks are a very common problem in heterogeneous environments and are significantly slower than the mean execution time of all tasks, impeding job completion time. Our initial analysis shows that accessing a single file on the distributed file system leads to stragglers, and as a result, prevents any scaling beyond one node. We introduce an important performance parameter $t_{Compute}/t_{IO}$ which determines whether we observe any stragglers. In addition, we show that there are two factors that lead to stragglers including I/O and communication. Taking advantage of Global Arrays (GA) toolkit we have been able to obtain significant improvement in communication cost and performance. In addition, we show two different approaches to overcome the I/O bottleneck and compare their performance. First approach is splitting the trajectory into as many trajectory segments as number of processes. The second approach is through MPI-based approach using Parallel HDF5 where we examine the performance differences between both collective and independent I/O. Applying these strategies, we obtained near ideal scaling and performance.

1 Introduction

The increase in computational power coupled with sophisticated algorithms has lead rapid increase in the amount of data produced by MD simulations. Typical trajectory sizes from MD simulations range from gigabytes to terabytes. Therefore, analyzing these trajectories has become a very tedious process in many workflows and as a result people are trying to look for state of the art HPC tools (MPI and OpenMP) or Big Data ecosystem to tackle this problem. The need for parallel programming and running these programs on parallel architectures is obvious, however, efficiently programming for a parallel environment can be a very daunting task.

MDAnalysis[??] is a widely used open-source Python library to analyze molecular dynamics (MD) simulations. *MDAnalysis* allows analysis of different file formats for trajectories generated by various packages for molecular dynamic simulations.

In our previous study, we used a parallel map-reduce approach to study the performance of RMSD task[?] . We previously looked at the *Dask* library[?] , which splits a computation in tasks and generates directed acyclic graphs of these tasks that are executed on a range of schedulers. We also implemented the parallel analysis scheme with MPI, using the *mpi4py* package[??] . For both Dask and MPI we found that our benchmark task, the calculation of the minimum C_α RMSD for a subset of the residues in the enzyme adenylate kinase from a long MD simulation, only showed good strong scaling within a single node (up to 24 cores on *SDSC Comet*). However, with a single compute node we are limited by the resources for executing a given problem. Distributed computing, allows parallelizing the SPMD paradigm for larger problem sizes and lead to performance gains. But, as soon as we extend the computation beyond a single node, performance drops due to *stragglers* tasks, a subset of Dask worker processes or MPI ranks that are significantly slower than the mean execution time of all tasks, increasing the total time to solution. Stragglers significantly impede job completion time and are a big challenge toward achieving improved performance.

MPI should have, in principle, close to ideal scaling for a pleasingly parallel task such as the analysis of trajectory blocks, and does not require additional considerations of, e.g., scheduler performance as for Dask. Therefore, in the present study, we analyze the MPI case in more detail to better understand the origin of the stragglers.

We want to provide simple and robust parallelization approaches to analysing molecular dynamics (MD) trajectories, in order to remove a narrowing bottle neck in the bio-molecular simulation field. We have selected two of the most common map-reduce algorithms in *MDAnalysis* one of which is I/O bound and the other is compute bound. We use SPMD paradigm to parallelize theses two algorithms on HPC resources. With SPMD, each process executes essentially the same code but on a different part of the data. We employ the Python language because it is widely used in this field and in the past has been shown to also perform well in HPC environments. We describe our experiences using

Python, machine-independent, byte-code interpreted, object-oriented programming (OOP) language, which is well-established in parallel environments [GA thesis]. Based on our initial analysis there is an important performance parameter, $t_{Compute}/t_{IO}$, that determines whether we observe stragglers. This depends on the order of complexity of the algorithm we are trying to parallelize. We show this behavior using RMSD and dihedral featurization algorithms. If $t_{Compute}/t_{IO}$ is large enough the algorithm scales very well, otherwise it does not scale beyond one node. For the algorithm with small $t_{Compute}/t_{IO}$ we need to come up with strategies on how to improve scaling and overcome straggler problems. Looking at the timing distribution across all ranks we noticed that communication and I/O are the two main scalability bottlenecks. Taking advantage of Global Array toolkit, each rank places its data in shared memory which allows direct access using the global address space rather than through a message-passing protocol. This reduces communication cost noticeably. Although Global Array toolkit is very helpful for improving the performance it is still not enough because I/O still remains a bottleneck. Our data shows that I/O time does not scale beyond one node. Due to the large file size and memory limit, processes are not able to load the whole trajectory into memory at once and as a result each process is only allowed to load one frame into memory at a time. Therefore, with large number of frames, there will be a lot of file access requests and when the compute time is small with respect to I/O, then I/O can be a major issue. Therefore, we needed to find ways to improve I/O scaling beyond a single compute node. In order to improve I/O we came up with two approaches: MPI-based approach using Parallel HDF5, and splitting our trajectory to as many trajectory segments as the number of processes. We provide the detail on these approaches on the following sections. But, both approaches significantly improved the performance and we were able to achieve near ideal scaling.

2 Molecular Dynamics Analysis Applications

2.1 MDAnalysis

Simulation data exist in trajectories in the form of three dimensional time series (atom positions and velocities), and these come in a plethora of different and idiosyncratic file formats. MDAnalysis is a widely used open source Python library to analyze these trajectory files with an object oriented interface. The package is written in Python (compatible with version 2.7 and 3.4+), with time critical code in C/Cython.

2.1.1 Root Mean Square Distance (RMSD)

The calculation of the root mean square distance (**RMSD**) for C_α atoms after optimal superposition with the QCPROT algorithm[? ?] is commonly required in the analysis of molecular dynamics simulations (Algorithm 1). The task used for the purpose of our benchmark is the MDAnalysis.analysis.rms.rmsd()

function from the MDAnalysis.analysis.rms module (implemented in Cython[?]). This function computes the coordinate root mean square distance between two sets of coordinates using the fast QCPROT algorithm to optimally superimpose two structures and then calculates the RMSD. RMSD values show how rigid the domains in a protein structure, are during the transition.

To this, the protein structure (selected C_α atoms) in the initial frame will be considered as the reference and as the mobile group at other time steps. RMSD calculation as a function of time are then performed by spatially aligning the mobile group to reference by doing a RMSD fit on selected atoms. The superposition is done in the following way: First, the mobile group is translated so that its center of mass coincides with the one of reference. Second, a rotation matrix is computed that minimizes the RMSD between the coordinates of selected atoms of the mobile group and reference structure. Finally, all atoms in Universe that contains mobile group are shifted and rotated. For each frame, a non-negative floating point number is calculated and the final result is a time-series of the RMSD. The order of complexity for RMSD algorithm 1 is $T \times N^2$ where T is the number of frames in the trajectory and N the number of particles in a frame.

2.1.2 Dihedral Featurization

As a real-world compute-bound task we investigated **Dihedral featurization**[?] (Algorithm 2) whereby a time series of feature vectors consisting of the two backbone dihedral angles per residue (ϕ_i and ψ_i) is calculated for all 212 non-terminal residues. For each frame, an array of dihedral angles is calculated where for later convenience, an angle θ_i is actually represented as $(\cos\theta_i, \sin\theta_i)$. The order of complexity for Dihedral featurization algorithm 2 is $T \times N$.

3 Background

3.1 MPI for Python

MPI for Python is a Python wrapper written over Message Passing Interface (MPI) standard and allows any Python program to employ multiple processors[? ?] . Python has several advantages that makes it a very attractive language including rapid development of small scripts and code prototypes as well as large applications and highly portable and reusable modules and libraries. MPI for Python's interface is designed to be as conforming as possible with standard MPI for C++, saving users from learning a new interface specification. In addition, Python's interactive nature, and other factors like lines of codes (LOC), number of function invocation, and development time adds to its attractiveness and clarifies why it is a good investment to extend Python use to message-passing parallel programming applications. Based on the efficiency tests, the performance degradation due to using MPI for Python is not prohibitive and the overhead introduced by MPI for Python is far smaller than the overhead associated to the use of interpreted versus compiled languages [].

Algorithm 1 RMSD Algorithm

Input: *mobile*: the desired atom groups to perform RMSD on them
bsize: Total number of frames assigned to each rank
xref0: mobile group in the initial frame which will be considered as rerference
start & stop: that tell which block of trajectory (frames) is assigned to each rank
topology & trajectory: files to read the data structure from
Output: Calculated RMSD arrays

```
1: procedure Block_RMSD(index, topology, trajectory, xref0, start=None, stop=None, step=None)
2:     start00 = time.time()
3:     u = Universe(topology, trajectory)
4:     g = u.atoms[index]
5:
6:     start1 = time.time()
7:     start0 = start1
8:     for ∀iframe, ts in enumerate(u.trajectory[start : stop : step]) do
9:         start2 = time.time()
10:        results[iframe, :]  = ts.time, MDAnalysis.analysis.rms.rmsd(g.positions, xref0, center=True, superposition=True)
11:        t_comp[iframe] = time.time()-start2                    ▷ Compute per frame
12:        t_IO[iframe] = start2-start1                           ▷ I/O per frame
13:        start1 = time.time()
14:    end for
15:    start3 = time.time()
16:    return results
17: end procedure
18:
19: start4 = time.time()
20: mobile = Select Cα atoms
21: index = indices of mobile atom group
22: xref0 = mobile.positions-mobile.center_of_mass()
23: out = Block_RMSD(index, topology, trajectory, xref0, start=start, stop=stop, step=1)
24:
25: start5 = time.time()
26: if rank == 0 then
27:    data1 = np.zeros([size*bsize,2], dtype=float)
28: else
29:    data1 = None
30:    comm.Gather(out[0], data1, root=0)
31: end if
32:
33: start6 = time.time()
```

In addition, there are works on improving the communication performance in MPI4py and it shows minimal overheads compared to C code if efficient raw memory buffers are used for communication[?] .

3.2 Applications of Global Array

Global array (GA) toolkit offers the best features of both shared and distributed programming models and allows manipulating physically distributed dense multi-dimensional arrays without the need for explicit cooperation by other processes. The primary mechanisms provided by GA for accessing data are block copy operations that transfer data between layers of memory hierarchy, i.e. global memory (distributed array) and local memory. GA compliments message passing programming model and is compatible with MPI which allows the users to take advantage of existing MPI software/libraries.

The classic message-passing paradigm of parallel programming not only transfers data but also synchronizes the sender and receiver. Asynchronous

(nonblocking) send/receive operations can be used to avoid synchronization, but cooperation between sender and receiver is still required. The synchronization effect is beneficial in certain classes of algorithms, such as parallel linear algebra algorithms. However, for other algorithms, this synchronization can be unnecessary and undesirable, and a source of performance degradation and programming complexity. In the shared-memory programming model, data is located either in ***private*** memory which is accessible only by a specific process or in ***global*** memory which is accessible to all processes.

In shared-memory systems, regardless of the implementation, the shared-memory paradigm eliminates the synchronization that is required when message-passing is used to access shared data. A disadvantage of many shared-memory models is that they do not expose the none-uniform memory access (***NUMA***) hierarchy of the underlying distributed-memory hardware. The shared memory model based on Global Arrays combines the advantages of a distributed memory model with the ease of use of shared memory. This is achieved by function calls that provide information on which portion of the distributed data is held locally and the use of explicit calls to functions that transfer data between a shared address space and local storage. The combination of these functions allows users to make use of the fact that remote data is slower to access than local data and to optimize data reuse and minimize communication in their algorithms.

The basic components of the Global Arrays toolkit are function calls to create global arrays (*ga_create*), copy data to (*ga_put*), from (*ga_get*), and between global arrays, and identify and access the portions of the global array data that are held locally (*ga_access*). In addition, there are also functions to destroy arrays (*ga_destroy*) and free up the memory originally allocated to them.

Another study by [Thesis] extends GAiN to a pure Python extension module. Even though a compiled extension module generally performs faster than its pure Python counterpart having a Python extension modules have several benefits: First and foremost it would require very little to be installed by the end user. Second, it leverages Python?s strengths such as its readability, maintainability, and the elimination of the need to compile machine code [Thesis].

NumPy is inherently serial and although NumPy's computational capabilities are efficient, it is still not enough for larger problem sizes. GA library attempts to leverage the substantial work that is Global Arrays in support of large parallel array computation within the NumPy framework.

3.3 MPI and Parallel HDF5

MPI-based applications work by launching multiple parallel instances of the Python interpreter which communicate with each other via the MPI library. HDF5 itself handles nearly all the details involved with coordinating file access through MPI library. This is advantageous to avoid multiple processes to compete over accessing the same file on disk. In fact in python, MPI-IO opens shared files with a mpio driver. In addition, MPI communicator should be supplied as well and the users also need to follow some constraints for data consistency [Python- HDF5 Book].

3.3.1 Collective Versus Independent Operations

MPI has two flavors of operation: collective, which means that all processes have to participate in the same order, and independent, which means each process can perform the operation or not and the order also does not matter [Python- HDF5 Book]. With HDF5, modifications to file metadata must be done collectively. In contrast, Opening or closing a file, Creating or deleting new datasets, groups, attributes, or named types, Changing a datasets shape, and Moving or copying objects in the file or any type of data operations can be performed independently by processes. It should be noted that collective does not mean synchronized. Although in collective I/O all processes perform the same task, but they do not wait until the others catch up [Python- HDF5 Book].

4 Method

The test system contained the protein adenylate kinase with 214 amino acid residues and 3341 atoms in total**?** . The trajectory was in Gromacs XTC format trajectory ("600x" in **?**) with a size of about 30 GB and 2,512,200 time frames (corresponding to 602.4 µs simulated time) which represents a typical medium per-frame size but is very long for current standards.

The experiments were executed on the XSEDE Supercomputers: *SDSC Comet*. SDSC Comet is a 2.7 PFlop/s cluster with 6,400 nodes in total. The standard compute nodes consist of Intel Xeon E5-2680v3 processors, 128 GB DDR4 DRAM (64 GB per socket), and 320 GB of SSD local scratch memory. The large memory nodes contain 1.5 TB of DRAM and four Haswell processors each and the network topology is 56 Gbps FDR Infini-Band. All the experiments in the present study are performed using compute nodes.

Timing observables

We model MPI performance based on the RMSD algorithm (1) and Dihedral Featurization algorithm (2). The notation for our models is summarized in Table 2. We directly measured inside our code (in the function `block_rmsd()`) the "compute" time per trajectory frame to perform the computation $t_{\text{compute\_final}}$ (`t_comp` in the code) and the "I/O" time for ingesting the data from the file system into memory, $t_{\text{IO\_final}}$ (`t_IO`). t_{comp_final} is the summation of "compute" time per frame and t_{IO_final} is the summation of "I/O" time per frame for all the frames assigned to each rank. t_{end_loop} is the time delay between the end of the last iteration and exiting the for loop. $t_{opening\_trajectory}$ is the time for where data structures are initialized and topology and trajectory files are opened.

Outside the `block_rmsd()` function, relevant probs were taken and stored as variables `start4`, `start5`, and `start6`, which we will abbreviate in the following as t_4 (after setting up the problem and before entering `block_rmsd()` to *ingest* and *compute*), t_5, (before communicating back results with `MPI.COMM_WORLD.Gather()`), and t_6 (after all work has been done but before timing statistics are computed). The total time (for all frames) spent in

Item	Definition
t_{end_loop}	$t_3 - t_1$
$t_{opening\_trajectory}$	$t_0 - t_{00}$
t_{comp_final}	$np.sum(t_{comp})$
t_{IO_final}	$np.sum(t_{IO})$
t_{all_frame}	$t_3 - t_0$
t_{RMSD}	$t_5 - t_4$
$t_{Communication_{MPI}}$	$t_6 - t_5$
$t_{Communication_{GA}}$	$(t_6 - t_5)+(t_7 - t_6)$
$t_{Overhead1}$	$t_{all\_frame} - t_{IO\_final} - t_{comp\_final} - t_{end\_loop}$
$t_{Overhead2}$	$t_{RMSD} - t_{all\_frame} - t_{opening\_trajectory}$
t_N	$t_{RMSD} + t_{Communication}$
t_{total}	$\max t_N$

Table 1: Notation of our performance modeling (relevant probes in the codes are stored as variables `start*`, which we will abbreviate in here as t_*)

`block_rmsd()` is $t_{\mathrm{RMSD}} = t_5 - t_4$. The "Shuffle" time to gather ("reduce") all data from all processor ranks is $t_{\mathrm{comm}} = t_6 - t_5$.

There are parts of the code in `block_rmsd()` that are not covered by the detailed timing information of $t_{\mathrm{compute\_final}}$ and $t_{\mathrm{IO\_final}}$. To measure the unaccounted time we define the "overheads". $t_{Overhead1}$ and $t_{Overhead2}$ are the overhead of the calculations and they should be ideally very small. The total time to completion for a single process on N cores is t_N, which is mathematically equivalent to $t_N \equiv t_{RMSD} + t_{\mathrm{comm}}$.

We also recorded the total time to solution $t_{\mathrm{total}}(N)$ with N MPI processes on N cores (which is effectively $t_{\mathrm{total}}(N) \approx \max t_N$). Strong scaling was quantified by calculating the speed-up relative to performance on a single core (using MPI).

$$S = \frac{t_{\mathrm{total}}(N)}{t_{\mathrm{total}}(1)} \tag{1}$$

5 Performance Study

We did not discern any specific patterns that could be traced to the underlying hardware. Stragglers were observed on *SDSC Comet*, *TACC Stampede* and *TACC Data Analytics System Wrangler* (data not shown). There was also no clear pattern in which certain MPI ranks would always be a straggler and we could also not trace stragglers to specific cores or nodes (or at least our data did not reveal an obvious pattern). Therefore, the phenomenon of stragglers in the RMSD test appears to be independent from the underlying hardware.

5.0.1 RMSD Benchmarks

We showed previously that the RMSD task only scaled well up to 24 cores, on a single compute node on *Comet* (and similarly also only on a single node on other machines), using either dask or MPI[?] . Although it is not clear that the root cause is the same for dask and MPI, here we focus on the MPI implementation (via *mpi4py*[? ?]) for its greater simplicity than dask, in order to better understand the cause for the poor scaling across nodes.

For both dask and MPI we observed stragglers, individual workers or MPI ranks, that much longer to complete than mean execution time of all other workers or ranks. Waiting for these stragglers destroys strong scaling performance, as shown in Figure 1a, 1b for MPI.

In the example run in Figure 1c, ten ranks out of 72 take almost 65 s whereas the remaining ranks only take about 40 s. The detailed breakdown of the time spent on each rank (Figure 1c) shows that time for the actual computation, $t_{\text{compute\_final}}$, is fairly constant across ranks. The time spent on reading data from the shared trajectory file on the Lustre file system into memory, $t_{\text{IO\_final}}$, shows variability across different ranks. Stragglers, however, appear to be defined by occasional much larger *communication* times, t_{comm}, that are on the order of 30 s in this example. For other ranks, t_{comm} varies across different ranks and for a few ranks $t_{\text{comm}} < 10$ s or is barely measurable. t_{comm} is the time to perform `comm.Gather()` (lines 25-30 in 1).

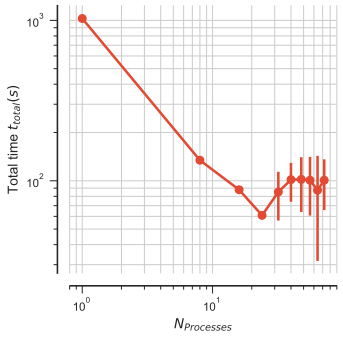
Identification of Scalability Bottleneck

This initial analysis (especially Figure 1c) indicates that communication is a major issue. Figure 2 shows the scaling of $t_{\text{compute\_final}}$ and $t_{\text{IO\_final}}$ individulally. As shown, $t_{\text{compute\_final}}$ scales very well; however, $t_{\text{IO\_final}}$ does not show good scaling and that explains why we are seeing these variations in $t_{\text{IO\_final}}$ across different ranks (Figure 1c).

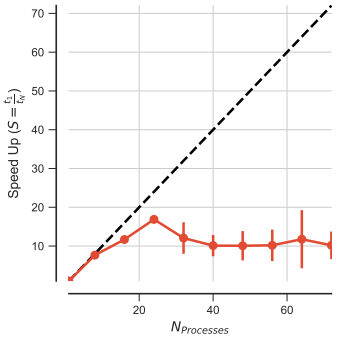
5.1 Dihedral Featurization Benchmarks

We briefly tested a much larger computational workload ($t_{\text{compute\_final}} = 40$ ms, $t_{\text{IO\_final}} = 0.4$ ms, thus $t_{\text{compute\_final}}/t_{\text{IO\_final}} \approx 100$), namely dihedral featurization on *Comet* with Infiniband and Lustre file system.

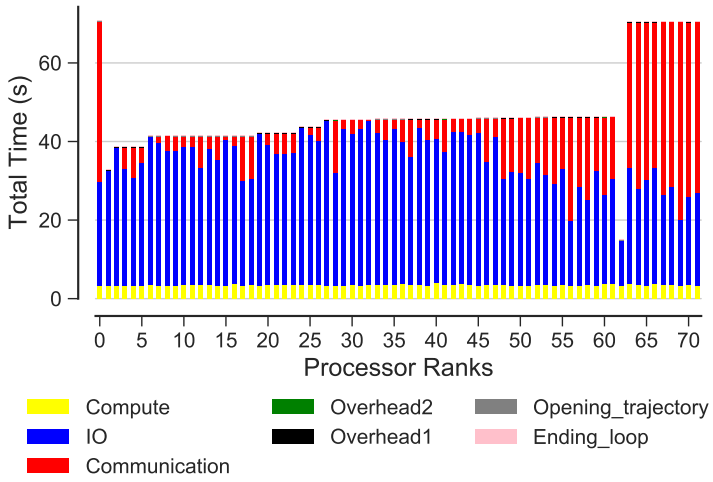
The system scales linearly and close to ideal (Figure 3). Although there is communication of large result arrays, which is costly for multiple ranks, the speed-up curve (Eq. 1) in Figure 3c demonstrates very good scaling with the number of cores (up to 72 cores on 3 nodes). The reason is that the communication cost (for $MPI - COMM\_WORLD - Gather()$) decreases with increasing the number of cores because the result array size that has to be communicated also decreases (Figure 4). Based on Figure 4, communication scales fairly well with the number of processes. This can be attributed to larger array sizes compared to the RMSD task and according to **?** the overhead of the *mpi4py* package decreases as the array size to be communicated increases. The dihedral



(a) Scaling total



(b) Speed-up



(c) Compute $t_{\text{compute\_final}}$, IO $t_{\text{IO\_final}}$, communication t_{comm}, ending the for loop t_{end_loop}, opening the trajectory $t_{opening\_trajectory}$, and overheads $t_{Overhead1}$, $t_{Overhead2}$ per MPI rank (as described in methods).

Figure 1: Performance of the RMSD task with MPI which is I/O-bound $t_{\text{compute\_final}}/t_{\text{IO\_final}} \approx 0.3$. Data are read from the file system (I/O included) and results are communicated back to rank 0 (communications included).

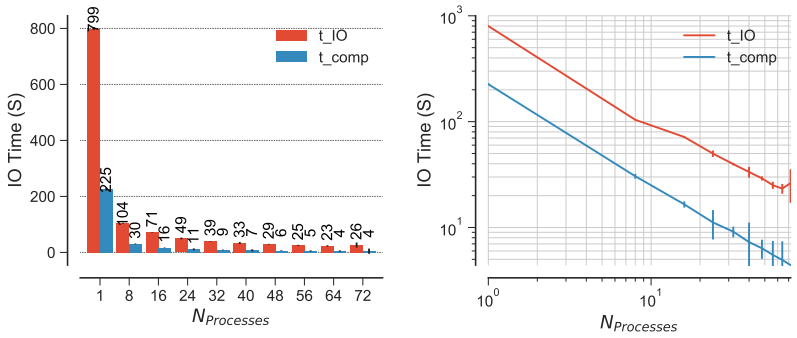


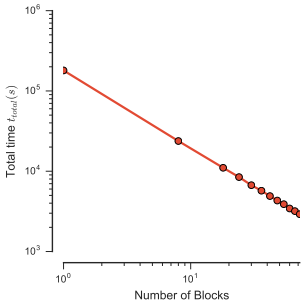
Figure 2: Scaling of the $t_{\text{compute\_final}}$ and $t_{\text{IO\_final}}$ of the RMSD task with MPI.

featurization workload has larger array size for all processor sizes (per task, a time series of feature vectors of size $N_b \times (213 \times 2 \times 2)$ when compared to the RMSD workload (per task a float array of length N_b) and therefore we are hypothesizing that the higher performance of *mpi4py* for larger array sizes has lead to better overall scaling.

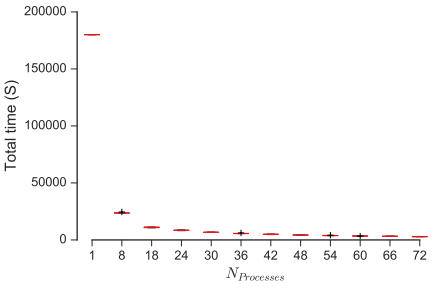
Overall, increasing the computational workload over I/O improves scaling. For large compute-bound workloads such as the dihedral featurization task, stragglers are eliminated and nearly ideal strong scaling is achieved. Even for moderately compute-bound workloads such as the 20× and 40× RMSD tasks, increasing the computational workload over I/O reduces the impact of stragglers even though they still contribute to large variations and thus to somewhat erratic scaling.

Achieving ideal scaling for large per frame workloads is also dependent on the size of the array to be communicated. Our results show that with dihedral featurization workload, nearly ideal scaling is possible. However, the performance of 40× and 20× RMSD workload, although greatly improved, is still affected by slow processes due to communication and overhead. The processes with large communication time can be attributed to the overhead of *mpi4py* package with respect to pure C code for smaller array size communication. Additionally, comparison of dihedral featurization workload, and 40× and 20× RMSD workload shows that overhead is less pronounced when the work becomes more compute bound.

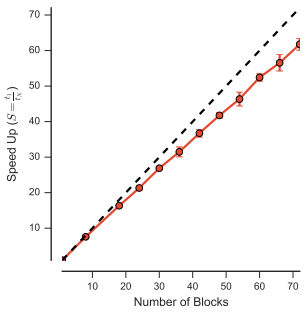
The fact that linear scaling is possible, even with expensive communications, makes parallelization a valuable strategy to reduce the total time to solution for large computational workloads. In a real-world application to one of our existing data sets on local workstations with Gigabit interconnect and Network File System (NFS) (using the Dask parallel library instead of MPI), analysis time was reduced from more than a week to a few hours (data not shown).



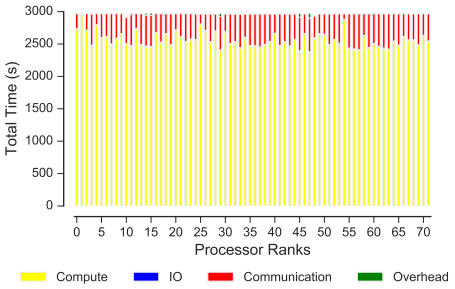
(a) Scaling total



(b) Total job execution time along with the mean and standard deviations across 5 repeats.

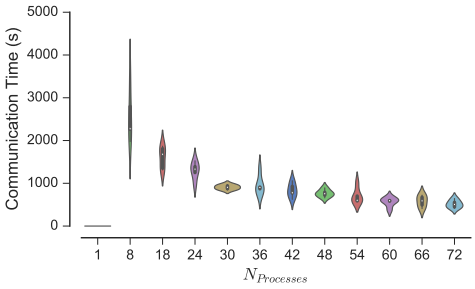


(c) Speed-up

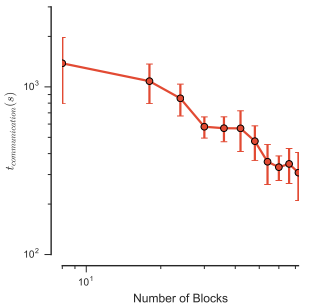


(d) time per MPI rank for a typical run (note, IO is almost too small to be visible)

Figure 3: Performance for the **dihedral featurization** workload, which is compute-bound $t_{\text{compute\_final}}/t_{\text{IO\_final}} \approx 100$, when *communications are included*.



(a) Communication time distribution for different number of processes (shown as violin plots [?] with kernel density estimates as implemented in *seaborn*); the white dot indicates the median.



(b) Scaling communcation time (mean and standard deviation)

Figure 4: Comparison of communication cost for different number of processes over 5 repeats for the **dihedral featurization** workload (with *communications included*).

5.2 Effect of $t_{\text{compute\_final}}/t_{\text{IO\_final}}$ on Performance

The RMSD task turned out to be I/O bound, i.e.,

$$\frac{t_{\text{compute\_final}}}{t_{\text{IO\_final}}} \ll 1.$$

We hypothesized that decreasing the relative I/O load with respect to compute load would also reduce the stragglers. We therefore increased the computational load so that the work became compute bound.

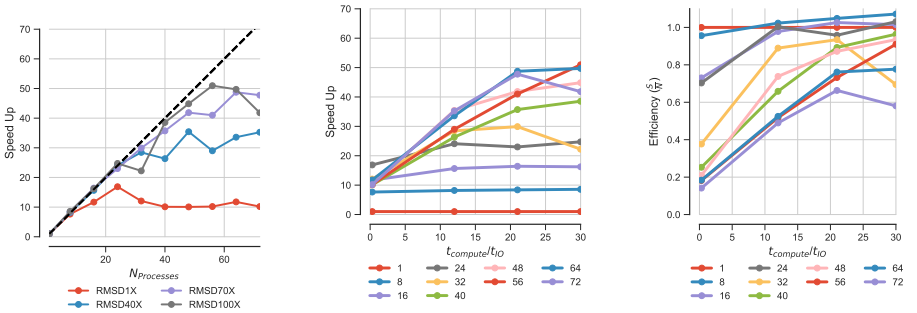
$$\frac{t_{\text{compute\_final}}}{t_{\text{IO\_final}}} \gg 1.$$

i.e., now processes are not constantly performing I/O and instead, I/O is interleaved with longer periods of computation. In order to artificially increase the computational load we also repeated the same RMSD calculation (line 10, algorithm 1) 40, 70 and 100 times in a loop respectively.

5.2.1 Increased workload (RMSD)

The RMSD workload was artificially increased forrty-fold, seventy-fold, and hundred-fold and we measured performance as before. Figure 5 shows the effect of $t_{\text{compute\_final}}/t_{\text{IO\_final}}$ ratio on performance. On average, each rank took $N_b \times t_{\text{IO\_final}}$ (where $N_b = N_{\text{frames}}/N$ is the number of frames per trajectory block, i.e., the number of frames processed by each MPI rank for N processes) for I/O, $X \times N_b \times t_{\text{compute\_final}}$ for the $X\times$ RMSD computation.

As the $t_{\text{compute\_final}}/t_{\text{IO\_final}}$ ratio increases the speed-up and performance improves and shows overall better scaling than the I/O-bound workload, i.e.



(a) Effect of $t_{\text{compute\_final}}/t_{\text{IO\_final}}$ on the Speed-Up

(b) Change in the Speed-Up with respect to $t_{\text{compute\_final}}/t_{\text{IO\_final}}$ for different processor counts

(c) Change in the efficiency with respect to $t_{\text{compute\_final}}/t_{\text{IO\_final}}$ for different processor counts

Figure 5: Performance change of the RMSD task with MPI with different $t_{\text{compute\_final}}/t_{\text{IO\_final}}$ ratios. We tested performance for $t_{\text{compute\_final}}/t_{\text{IO\_final}}$ ratios of 0.3, 12, 21, 30 which correspond to 1× RMSD, 40× RMSD, 70× RMSD, and 100× RMSD respectively (communication is included)

1× RMSD (Figure 5a). When $t_{\text{compute\_final}}/t_{\text{IO\_final}}$ ratio increases, the RMSD calculation consistently scales up to larger numbers of cores ($N = 56$ for 70× RMSD). Figures 5b and 5c shows the improvment in performance more clearly. In fact as the $t_{\text{compute\_final}}/t_{\text{IO\_final}}$ ratio increases the values of speed-up and efficiency get closer to their ideal value for each number of processor counts.

The measured average execution times of different communication strategies

.

Given the results for Dihedral featurization and RMSD algorithms (Algorithms 2, and 1) and $X\times$ RMSD (Figure 5) **We hypothesize that MPI competes with Lustre on the same network interface, which would explain why communication appears to be primarily a problem in the presence of I/O when $t_{\text{compute\_final}}/t_{\text{IO\_final}}$ is small.** In fact, decreasing the I/O load relative to the compute load should open up the network for communication.

5.3 Communication Cost and Application of Global Array

As discussed in the RMSD calculation, 1c, communication acted as the scalability bottleneck. In fact, when the processes communicate result arrays back to the master (rank 0) process, some processes take much longer as compared to other processes. Now we want to know what strategies can be used to avoid communication cost.

Algorithm 3 describes the global array algorithm. Given the speed up plots 6b and total time results 6a global array improves strong scaling performance. Although communication time has significantly decreased using global array (compare Figure 6c to Figure 1c), the existing variation in the dominant I/O

part of the calculation would still prevent ideal scaling (Figure 7). In fact, scaling did not improve over the runs when communications was performed using global arrays because there remained a few slow processes that take about 50 s, which is slower than the mean execution time of all ranks which is about 17 s.

5.4 I/O Cost

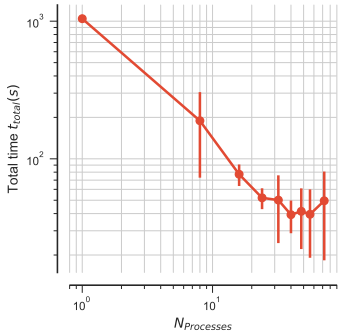
We showed previously that the I/O system can have a large effect on the parallel performance of the RMSD task[?] , especially because the average time to perform the computation $t_{\text{compute\_final}}$ (about 0.09 ms) is about four times smaller than the I/O time $t_{\text{IO\_final}}$ (about 0.3 ms) (Figures 2 and 1). In fact, poor I/O performance responsible for the stragglers, and the question is "are stragglers waiting for file access?" Due to the large file size and memory limit, processes are not able to load the whole trajectory into memory at once and as a result each process is only allowed to load one frame into memory at a time. Therefore, there are about $2,512,200$ frames and as a result file access requests and when the compute time is small with respect to I/O, then I/O can be a major issue as we also see in our results (igures 2 and 1). For example, file locking provided by the parallel file system might impose significant overheads such as to communicate acquiring and releasing locks and to compute assignments of new locks[?] . Read throughput might be limited by the available bandwidth on the Infini-band network interface that serves the Lustre file system and access for some files might be throttled. In fact, we need to test this hypothesis and come up with ways and strategies to avoid the competition over file access across different ranks. To this aim, we experimented two different ways for reducing I/O cost and examined their effect on the performance. These two ways include: Splitting the trajectories and MPI-based Parallel HDF5. We discuss these two approaches in detail in the following sections.

5.4.1 Splitting the Trajectories

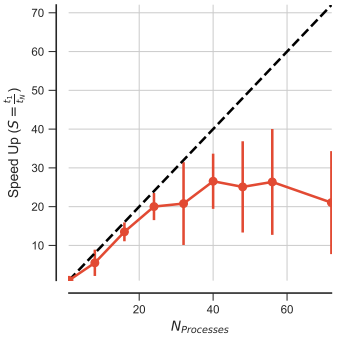
In all the previous benchmarks all processes were using a shared trajectory file. In order to test our hypothesis we splitted our trajectory file into as many trajectory segments as the number of processes. This means that if we have N processes the trajectory file is splitted into N segments and each segment will have N_b frames in it. Through this approach each process will have access to its own segment and there will be no competition over file access. For reference the necessary time for splitting the trajectory file is give in Appendix A.

Performance without Global Array

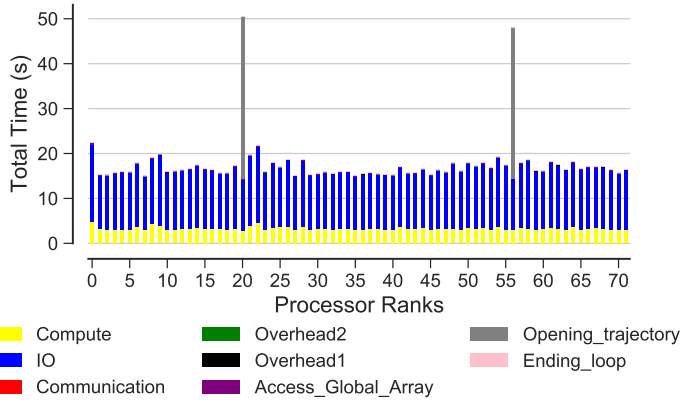
We ran a benchmark up to 8 nodes (192 cores) and, we observed rather better scaling behavior with efficiencies above 0.6 (Figure 8a and 8b) and the delay time for stragglers has also reduced (Figure 8c). However, the scaling is still far from ideal due to the communication. Although the delay due to communication is much smaller as compared to RMSD with a shared trajectory file (Figure 1c),



(a) Scaling total



(b) Speed-up



(c) Compute $t_{\text{compute\_final}}$, IO $t_{\text{IO\_final}}$, communication t_{comm}, ending the for loop t_{end_loop}, opening the trajectory $t_{opening\_trajectory}$, and overheads $t_{Overhead1}$, $t_{Overhead2}$ per MPI rank (as described in methods).

Figure 6: Performance of the RMSD task with MPI using global array ($t_{\text{compute\_final}}/t_{\text{IO\_final}} \approx 0.3$). Data are read from the file system (I/O included). All ranks update the global array and rank 0 accesses the whole RMSD array through the global memory address.

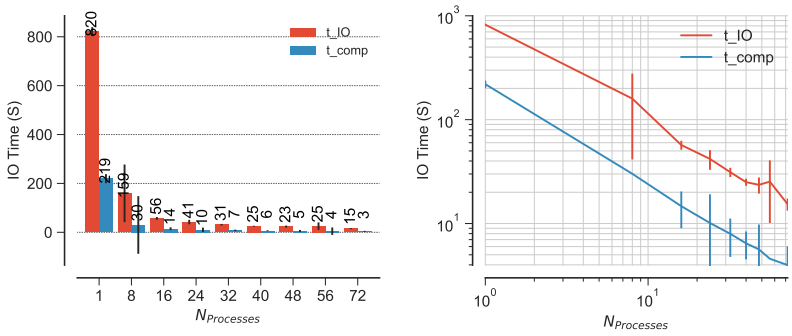


Figure 7: Scaling of the $t_{\text{compute\_final}}$ and $t_{\text{IO\_final}}$ of the RMSD task with Global Array.

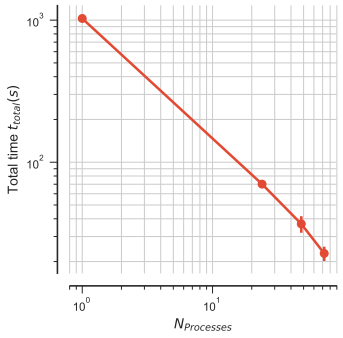
it is still delaying several processes and as a result leads to longer job completion time (Figure 8c). These delayed tasks impact performance as well and hence the speed-up is not still close to ideal scaling.

Performance using Global Array

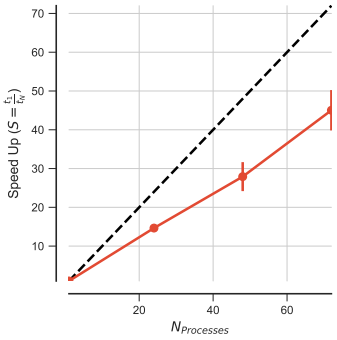
Previously, we showed that global array significantly reduces the communication cost. We want to see how the performance looks like if we split our trajectory file and take advantage of global array. Again, we ran our benchmark up to 8 nodes (192 cores) and, we observed near ideal scaling behavior with efficiencies above 0.9 (Figure 10a and 10b) with no straggler tasks (Figure 10c). The present results show that contention for a file impacts the performance. The initial results with splitting the trajectory file suggests that there is in fact an effect, which possibly also interfers with the communications when $t_{\text{compute\_final}}/t_{\text{IO\_final}} \ll 1$ (i.e. with a I/O bound workload).

5.4.2 MPI-based Parallel HDF5

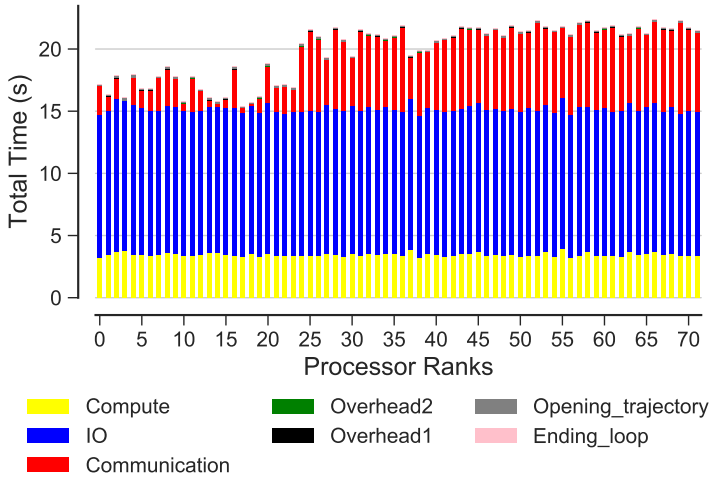
Another approach we examined to improve I/O scaling is MPI-based Parallel HDF5. We converted our XTC trajectory file into HDF5 format so that we can test the performance of parallel IO with HDF5 file format. The time it took to convert our XTC file with $2,512,200$ frame into HDF5 format was about 5400 s. Again, we ran our benchmark up to 8 nodes (192 cores) and, we observed near ideal scaling behavior with efficiencies above 0.8 (Figure 12a and 12b) with no straggler tasks (Figure 12c).



(a) Scaling total



(b) Speed-up



(c) Compute $t_{\text{compute\_final}}$, IO $t_{\text{IO\_final}}$, communication t_{comm}, ending the for loop t_{end_loop}, opening the trajectory $t_{opening\_trajectory}$, and overheads $t_{Overhead1}$, $t_{Overhead2}$ per MPI rank (as described in methods).

Figure 8: Performance of the RMSD task with MPI using global array ($t_{\text{compute\_final}}/t_{\text{IO\_final}} \approx 0.3$). Data are read from the file system (I/O included). All ranks update the global array and rank 0 accesses the whole RMSD array through the global memory address.

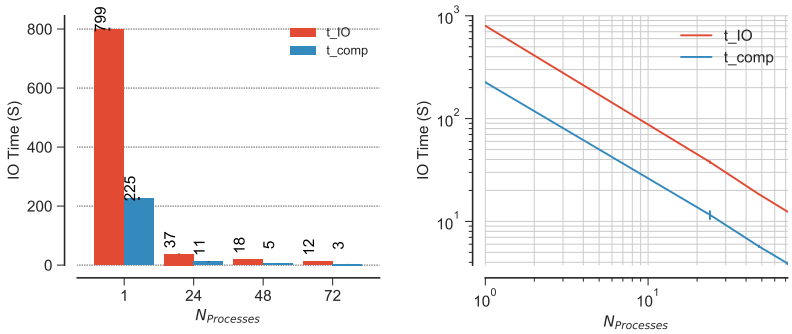


Figure 9: Scaling of the $t_{\text{compute\_final}}$ and $t_{\text{IO\_final}}$ of the RMSD task with Global Array.

6 Guidelines on the Parallel Analysis of Three Dimensional Time Series

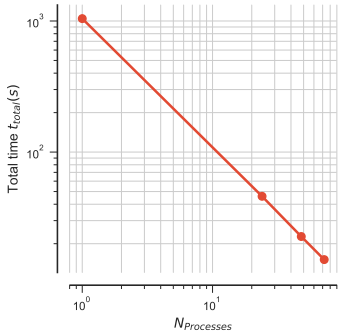
Parallel implementation of the three dimensional time series is performed in the following steps:

Step 1 Calculate the value of $t_{\text{compute\_final}}/t_{\text{IO\_final}}$ to see whether the task is compute bound ($\frac{t_{\text{compute\_final}}}{t_{\text{IO\_final}}} \gg 1$) or IO bound ($\frac{t_{\text{compute\_final}}}{t_{\text{IO\_final}}} \ll 1$). As discussed in Section 5.2 for I/O bound problems both communication and I/O will be a problem and the performance of the task will be affected by the straggler tasks which delay job completion time.

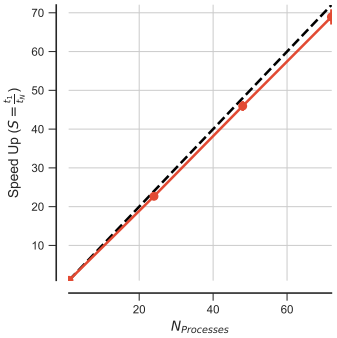
Step 2 For $\frac{t_{\text{compute\_final}}}{t_{\text{IO\_final}}} \geqslant 100$ the task is compute bound and the task will scale very well as we showed in Section 5.1. However, if the size of the data to be communicated to rank 0 using the blocking collective communication ($MPI.Gather()$) is large, one might consider using global arrays to achieve near ideal scaling behavior (Section 5.3). Application of Global array is useful in the sense that it replaces message-passing interface with a distributed shared array where its blocks will be updated by the associated rank in the communication domain (Algorithm 3).

Step 3 For $\frac{t_{\text{compute\_final}}}{t_{\text{IO\_final}}} \leqslant 100$ the task is I/O bound and then you need to take the following steps:

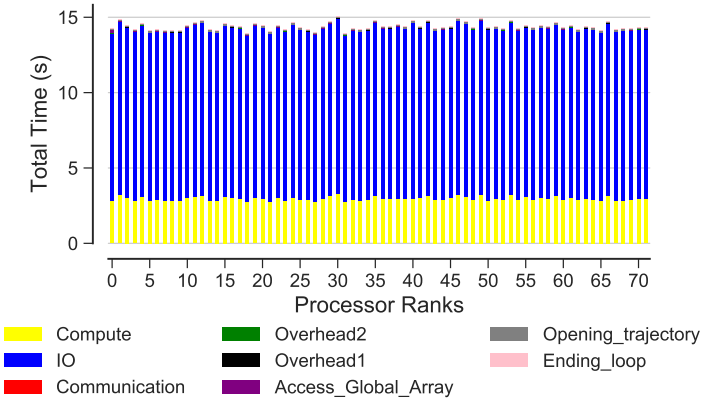
- If there is access to HDF5 format the recommended way will be to use MPI-based Parallel HDF5 (Section 5.4.2). Since converting the XTC file to HDF5 is expensive if the trajectory file formats are not in HDF5 form then one might prefer to split the trajectories.
- The trajectory file should be splitted into as many trajectory segments as the number of processes.
- The appropriate parallel implementation along with *Global Array* should be used on the trajectory segments (Section 5.4.1).



(a) Scaling total



(b) Speed-up



(c) Compute $t_{\text{compute\_final}}$, IO $t_{\text{IO\_final}}$, communication t_{comm}, ending the for loop t_{end_loop}, opening the trajectory $t_{opening\_trajectory}$, and overheads $t_{Overhead1}$, $t_{Overhead2}$ per MPI rank (as described in methods).

Figure 10: Performance of the RMSD task with MPI using global array ($t_{\text{compute\_final}}/t_{\text{IO\_final}} \approx 0.3$). Data are read from the file system (I/O included). All ranks update the global array and rank 0 accesses the whole RMSD array through the global memory address.

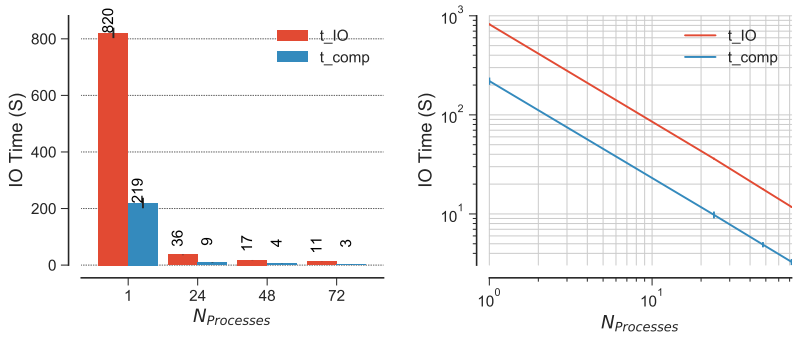


Figure 11: Scaling of the $t_{\text{compute\_final}}$ and $t_{\text{IO\_final}}$ of the RMSD task with Global Array.

7 Conclusion

We have been able to identify the root cause for stragglers in our MPI test case and our data suggest a **new specific hypothesis**. We tested our benchmark on RMSD (I/O bound) and Dihedral featurization (compute bound) algorithms in *MDAnalysis*. Both communication and I/O appeared to be the scalability bottleneck for our RMSD benchmark test case.

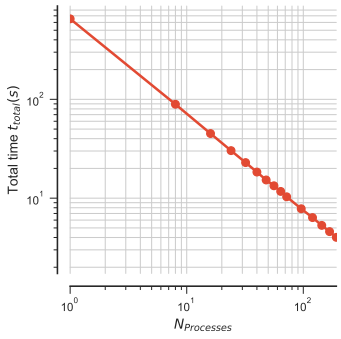
In fact, for I/O-bound workload, *stragglers are due to the competition between MPI and the Lustre file system on the shared Infini-band interconnect.* This hypothesis appears to be consistent with the observation that for larger number of MPI ranks, communication is primarily a problem in the presence of (nearly continuous) I/O as produced by I/O-bound workloads.

The ratio between compute load and I/O load appeared to be crucial. For sufficiently large per-frame workloads, close to ideal scaling was achievable (Figure 3). Additionally, the effect of communication was less pronounced when the work became more compute-bound. We could also achieve much better performance in our RMSD benchmark taking advantage of global array toolkit. Using global array for our communication, we did not observe any delayed task due to communication (Figure 6) and it we were able to reduce the communication cost.

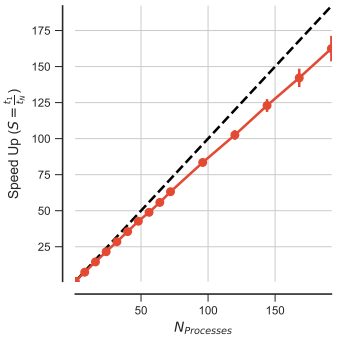
More importantly, we had observed that I/O appeared to be important as demonstrated by close to ideal scaling through splitting the trajectories and parallel I/O (Figures 8 and 12). The communication time t_{comm}, i.e., the `mpi4py.MPI.COMM_WORLD.Gather()` step, could take much longer for stragglers than for "normal" MPI ranks when I/O has to be performed (Figure 1c) through a shared trajectory file.

With splitting the trajectories, effect of communication is still apparent on the performance; however this problem can be tackled using global array toolkit (Compare Figure 8 to Figure 10).

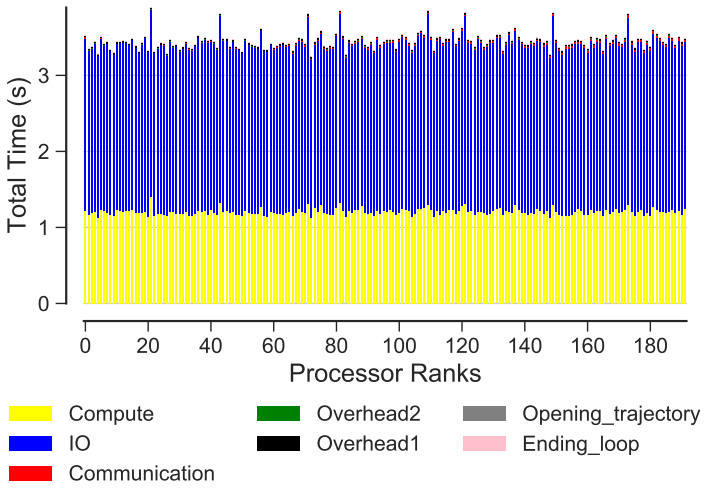
There are currently many freely available libraries for the analysis and processing of three-dimensional time series. However, dramatic increases in the



(a) Scaling total



(b) Speed-up



(c) Compute $t_{\text{compute\_final}}$, IO $t_{\text{IO\_final}}$, communication t_{comm}, ending the for loop t_{end_loop}, opening the trajectory $t_{opening\_trajectory}$, and overheads $t_{Overhead1}$, $t_{Overhead2}$ per MPI rank (as described in methods).

Figure 12: Performance of the RMSD task with MPI-based parallel HDF5 ($t_{\text{compute\_final}}/t_{\text{IO\_final}} \approx 0.3$). Data are read from the file system from a shared HDF5 file instead of XTC format (Independent I/O).

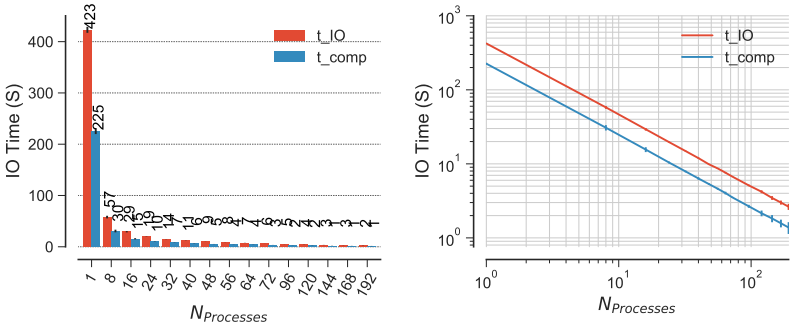


Figure 13: Scaling of the $t_{\text{compute\_final}}$ and $t_{\text{IO\_final}}$ of the RMSD task with MPI-based parallel HDF5 (Independent I/O).

size of trajectories combined with the serial nature of these libraries necessitates use of state of the art high performance computing tools for rapid analysis of these time series. All the above strategies, provides the bio-molecular simulation community the means to perform a wide variety of parallel analyses on data generated from computational simulations. The guidelines provided in the present study, help people on how to tackle their problem depending on the workload being I/O bound or compute bound. The analysis indicates that the splitting the trajectories or parallel I/O in combination with global array will make it feasible to run a I/O bound task on scalable computers up to 8 nodes and get near ideal scaling behavior.

Algorithm 2 Dihedral Featurization

Input: *mobile*: the desired atom groups to perform RMSD on them
start & stop: that tell which block of trajectory (frames) is assigned to each rank
topology & trajectory: files to read the data structure from
Output: Calculated Dihedral Angles

```
procedure angle2sincos(x)
    return [np.cos(x), np.sin(x)]
end procedure

procedure residues_to_dihedrals(residues)
    for ∀res in residues do
        list(ϕ(res), ψ(res))
    end for
end procedure

procedure featurize_dihedrals(dihedrals)
    angles = [dihedral.value() for dihedral in dihedrals]
    return angle2sincos(angles)
end procedure

procedure Block_Dihedral(index, topology, trajectory, xref0, start=None, stop=None, step=None)
    start00 = time.time()
    clone = mda.Universe(topology, trajectory)
    g = clone.atoms[index]

    start1 = time.time()
    start0 = start1
    for ∀iframe, ts in enumerate(clone.trajectory[start : stop : step]) do
        start2 = time.time()
        Append in results featurize_dihedrals(dihedrals)
        t_comp[iframe] = time.time()-start2                    ▷ Compute per frame
        t_IO[iframe] = start2-start1                           ▷ I/O per frame
        start1 = time.time()
    end for
    start3 = time.time()
end procedure

start4 = time.time()
out = Block_Dihedral(index, topology, trajectory, xref0, start=start, stop=stop, step=1)

start5 = time.time()
if rank == 0 then
    data1 = np.zeros([size*bsize,np.shape(out)[1]], dtype=float)
else
    data1 = None
    comm.Gather(out[0], data1, root=0)
end if

start6 = time.time()
```

Algorithm 3 Global Array instead of Message-Passing Paradigm

Input: *mobile*: the desired atom groups to perform RMSD on them
bsize: Total number of frames assigned to each rank N_b
g_a: Initialized global array of ($bsize * size, 2$)
buf: Initialized buffer to store final calculated RMSD results of size ($bsize * size, 2$)
xref0: mobile group in the initial frame which will be considered as rerference
start & stop: that tell which block of trajectory (frames) is assigned to each rank
topology & trajectory: files to read the data structure from
Output: Calculated RMSD arrays

```
1: bsize = int(np.ceil(mobile.universe.trajectory.n_frames / float(size)))
2: g_a = ga.create(ga.C_DBL, [bsize*size,2], "RMSD")
3: buf = np.zeros([bsize*size,2], dtype=float)
4:
5: start4 = time.time()
6: out = Block_RMSD(index, topology, trajectory, xref0, start=start, stop=stop,
   step=1)
7: start5 = time.time()
8:
9: ga.put(g_a, out[:,:], (start,0), (stop,2))
10:
11: start6 = time.time()
12: if rank == 0 then
13:     buf = ga.get(g_a, lo=None, hi=None)
14: end if
15:
16: start7 = time.time()
```

Number of trajectory segments	N_p used for writing the segments	time (s)
24	24	
48	48	
72	72	
96	96	
144	144	
196	196	

Table 2: The wall-clock time spent for writing N_p trajectory segments using N_p processes using MPI on SDSC Comet

A Appendix

Detailed timing for splitting the trajectories

Version of the libraries used for the present study