Guidelines for obtaining ideal Performance for Parallel Analysis of Molecular Dynamics Trajectories with Python on HPC Resources

Mahzad Khoshlessan[a], Ioannis Paraskevakos[c], Geoffrey C. Fox[d], Shantenu Jha[c], Oliver Beckstein[a,b,*]

[a] *Department of Physics, Arizona State University, Tempe, AZ 85281, USA*
[b] *Center for Biological Physics, Arizona State University, Tempe, AZ 85281, USA*
[c] *Department of Electrical & Computer Engineering, Rutgers University, Piscataway, NJ 08854, USA*
[d] *Digital Science Center, Indiana University, Bloomington, IN 47405*

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. ***oliver: We will have to be a bit more careful how we phrase this because VMD can analyze on thousands of cores and cpptraj is apparently also doing pretty well. There is also HiMach.Benchmarks, however, are scarce. ***mahzad: Need to make clear how this work distinguishes from their work, first 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. 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,

*Corresponding author
Email addresses: mkhoshle@asu.edu (Mahzad Khoshlessan), i.paraskev@rutgers.edu (Ioannis Paraskevakos), gcf@indiana.edu (Geoffrey C. Fox), shantenu.jha@rutgers.edu (Shantenu Jha), oliver.beckstein@asu.edu (Oliver Beckstein)

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 biomolecular simulation field and focus on an MPI-based implementations. 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 through independent I/O. Applying these strategies, we obtained near ideal scaling and performance.

Keywords: Python, MPI, HPC, MDAnalysis, Global Array, MPI I/O, Straggler, Molecular Dynamic

2010 MSC: 00-01, 99-00

---

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 [1]. 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. Therefore, executing analysis workflows in parallel becomes important; however, efficiently programming for a parallel environment can be a very daunting task.

MDAnalysis [2, 3] 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 [4]. We previously looked at the *Dask* library [5], which splits a computation in tasks and generates directed acyclic graphs (DAG) of these tasks that can be executed on a range of schedulers. We also implemented the parallel analysis scheme with MPI, using the *mpi4py* package [6, 7]. 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 our problems 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 ana-

lyzing molecular dynamics (MD) trajectories, in order to remove a narrowing bottle neck in the bio-molecular simulation field. We have selected two of the algorithms in *MDAnalysis* one of which is I/O bound (RMSD) and the other is compute bound (Dihedral Featurization). 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 use Python, a machine-independent, byte-code interpreted, object-oriented programming (OOP) language, which is well-established in HPC parallel environments [8]. Based on our initial analysis there is an important performance parameter, $t_{Compute}/t_{IO}$ which is the ratio of computational to I/O load, as measured by the time spent on the computation versus the time spent on reading data from the file system, that determines whether we observe stragglers. We show this behavior using RMSD and dihedral featurization algorithms. If $t_{Compute}/t_{IO} \gg 1$, the algorithm scales very well, otherwise it does not scale beyond one node. For the algorithms with small $t_{Compute}/t_{IO}$, we need to come up with strategies 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 we were able to reduce communication cost noticeably. In addition, our data shows that I/O time does not scale beyond one node. In order to improve I/O scaling, we came up with two approaches: MPI-based approach using Parallel HDF5 [9], 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. Background & Related Works

***oliver: The lit. review is required for a proper scholarly treatment of the subject. Background/Related Work should contain a literature review of

other approaches for analyzing MD trajectories on HPC: at a minimum: VMD, HiMach, cpptraj (and pytraj), possibly mdtraj ? read the papers/websites and check for benchmarks. Look for others (Giannis already mentioned some in his paper and/or the workshop draft). ***mahzad: Giannis, Overal we need more references on here. In addition you mention the features of the library, but we need to point out the strength and weakness of each library and make them bold. Maybe a Table like what you have added in ICPP paper on the comparison of frameworks that you mention what are their differences and how they differ from each other will be good to be added here as well. Could you please add that to this section?

CPPTraj [10] offers three levels of parallelization. The two are through MPI and the third through OpenMP. The MPI types of parallelization CPPTraj supports are show in Figure 1. In more detail, CCPTraj allows the parallel read between frames of the same trajectory or ensemble members of the same trajectory. When it is used to analyze a single trajectory, all frames of the trajectory are equally distributed over the number of MPI process that are used. Each process reads the frames that are assigned to it, executes and writes the results of all processes to the same output file. When ensemble mode is used, each ensemble member is assigned to an MPI process. As a consequence, there have to be as many MPI processes as ensemble members. The user has the ability to increase CPPTraj's throughput by assigning more than one MPI processes per ensemble member. Each ensemble member is divided further the same way as a single trajectory.

HiMach [11] was developed by D.E.Shaw Research group to provide a parallel analysis framework for molecular dynamics simulations. HiMach extends Google's MapReduce to provide a scalable API for MD trajectory analysis. HiMach API provides a series of Python classes that are used to define trajectories, do per frame data acquisition (Map class) and cross-frame analysis (Reduce class). After the user has defined all the above, HiMach's runtime is responsible to parallelize and distribute the Map and Reduce classes to the assigned cores. Data transfers between Map and Reduce phases are done through

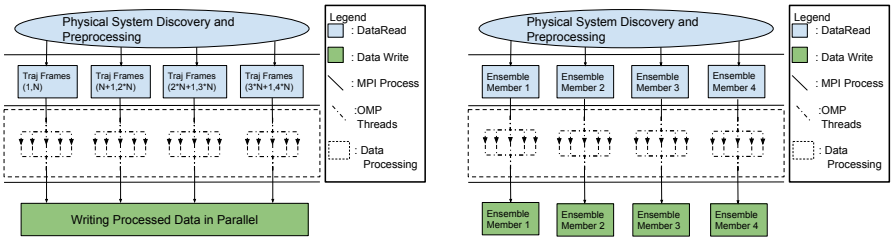


Figure 1: CPPTraj MPI modes of execution. The right figure shows the case where a single trajectory is given. The left figure shows the case where an ensemble of trajectories are given for analysis

a communication protocol created specifically for HiMach and it is transparent from the user.

3. Molecular Dynamics Analysis Applications

3.1. MDAnalysis

Simulation data exist in trajectories in the form of three dimensional time series (atoms 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.

3.1.1. 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 [12, 13] is commonly required in the analysis of molecular dynamics simulations (Algorithm 1). The task used for the purpose of our benchmark is the $RMSD = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\delta_i^2}$ implemented in MDAnalysis.analysis.rms module where δ_i is the distance between atom i and a reference structure (implemented in Cython [2]). This function computes the RMSD between two sets of coordinates using the fast QCPROT algorithm to optimally superimpose two structures and then calculates the RMSD.

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. 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 spatially aligns the mobile group to reference which minimizes the RMSD between the coordinates of the mobile group and reference structure. Finally, all atoms in 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. RMSD values show how rigid the domains in a protein structure are, during the transition. The order of complexity for RMSD algorithm 1 is $T \times N^2$ [14] where T is the number of frames in the trajectory and N the number of particles in a frame.

***mahzad: Is it fine we have included MPI commands in the pseudo code? Is this how people write it?

Algorithm 1 MPI-parallel Multi-frame RMSD Algorithm

Input: *size*: Total number of frames
ref: mobile group in the initial frame which will be considered as reference
start & stop: Starting and stopping frame index
topology & trajectory: files to read the data structure from
Output: Calculated RMSD arrays

```
procedure Block_RMSD(topology, trajectory, ref, start, stop)
    u ← Universe(topology, trajectory)        ▷ u hold all the information of the physical system
    g ← u.frames[start:stop]
    for ∀iframe in g do
        results[iframe] ← RMSD(g, ref)
    end for
    return results
end procedure

MPI Init
rank ← rank ID
index ← indices of mobile atom group
xref0 ← Reference atom group's position
out ← Block_RMSD(topology, trajectory, xref0, start=start, stop=stop)

Gather(out, RMSD_data, rank_ID=0)
MPI Finalize
```

3.1.2. Dihedral Featurization

As a real-world compute-bound task we investigated **Dihedral featurization** [15] (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 (Algorithm 2) is $T \times N$. ***oliver: Write out the calculation, based on the 4 atoms ***mahzad: What calculations? how residues are converted to dihedrals? or the procedure we perform in the algorithm? that one is explained in algorithm 2

Algorithm 2 MPI-parallel Multi-frame Dihedral Featurization Algorithm

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 $\cos x$, $\sin x$
end procedure

procedure *residues_to_dihedrals*(residues)
    List angles
    for $\forall res$ in *residues* **do**
        Append ($\phi(res)$, $\psi(res)$) in angles
    end for
    return angles
end procedure

procedure *featurize_dihedrals*(dihedrals)
    List angles
    for $\forall dihedral$ in *dihedrals* **do**
        Append value of *dihedral* in angles
    end for
    return `angle2sincos(angles)`
end procedure

procedure *Block_Dihedral*(topology, trajectory, ref, start, stop)
    u ← Universe(topology, trajectory)
    g ← u.frames[start:stop]
    List *results*
    for $\forall frame$ in g **do**
        D_{angles} ← *featurize_dihedrals*(*dihedrals*)
        Append D_{angles} in *results*
    end for
end procedure

MPI Init
residues ← residues of mobile atom group
dihedrals ← residues_to_dihedrals(residues)
rank ← rank ID
index ← indices of mobile atom group
xref0 ← Reference atom group's position
out ← Block_Dihedral(topology, trajectory, xref0, start=start, stop=stop)

Gather(out, RMSD_data, rank_ID=0)
MPI Finalize

4. Benchmark Environment

The test system contained the protein adenylate kinase with 214 amino acid residues and 3341 atoms in total [16]. The trajectory [17] was in Gromacs XTC format trajectory ("600x" in Khoshlessan et al. [4]) with a size of about 30 GB and 2,512,200 time frames (corresponding to 602.4 $\sim \mu s$ 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*, *PSC Bridges* and *SuperMIC*. SDSC Comet is a 2.7 PFlop/s cluster with 6,400 compute nodes in total. The standard compute nodes consist of Intel Xeon E5-2680v3 processors, 128 GB DDR4 DRAM (64 GB per socket). The network topology is 56 Gbps FDR Infini-Band (IB).

PSC Bridges is a 1.35 PFlop/s cluster with four types of computational nodes. Bridges computational nodes supply 1.3018 PFlop/s and 274 TiB RAM. The Regular Shared Memory (RSM) nodes consist of Intel Haswell (E5-2695 v3) processors, 128 GB DDR4 DRAM (64 GB per socket). The network topology is 12.37 Gbps Omni-Path Architecture (OPA).

LSU SuperMIC offers 360 compute nodes with Ivy Bridge Intel processors (E5-2680). Each node has 64 GBs DDR3 RAM. SUperMIC's nodes are connected 56 Gbps Infiniband Network. In addition, it offers 20 hybrid nodes which provide an NVIDIA GPU. SuperMIC's peak performance is measured at 557 TFlop/s. All the experiments are performed using standard compute nodes on Comet, PSC Bridges and SuperMIC respectively in the present study.

Cluster	Nodes	Number	CPUs	RAM	Network Topology	Scheduler and Resource Manager
SDSC Comet	Compute	6400	2 Intel Xeon (E5-2680v3) CPUs 12 cores/CPU, 2.5 GHz	128 GB DDR4 DRAM	56 Gbps IB	SLURM
PSC Bridges	RSM	752	2 Intel Haswell (E5-2695 v3) CPUs 14 cores/CPU, 2.3 GHz	128 GB, DDR4-2133Mhz	12.37 Gbps OPA	SLURM
SuperMIC	Standard	360	2 Intel Ivy Bridges (E5-2680) CPUs 10 cores/CPU, 2.8GB GHz	64 GB, DDR3-1866Mhz	56Gbps IB	PBS

Table 1: List of benchmarked clusters and their system configuration

4.1. Installing libraries on multi-user HPC systems

***oliver: Include table with software and versions; add a paragraph in which you mention software, summarize how it was compiled, and reference the table.

***mahzad: Are the following enough? Different packages and libraries are used in the present study in order to achieve the desired performance. However, getting scientific libraries installed can be a very challenging task. Lack of documentation and good software engineering practices, non-standard installation procedures, and lots of dependencies can contribute to the challenge of getting the libraries to work [???]. Scientists mostly care about the science and they are often not software engineers or system administrators. Therefore, the libraries and tools they need should be easily accessible. This is not always the case, though.

In fact, many domain specific packages are not available through package manager in supercomputers and as a result, we spent considerable amount of time getting packages dependencies to work in the process of our performance study. As a result, we provide detail information on how we managed to build these libraries. This will let future works spend least amount of time for this purpose. Detailed information regarding the version of each library, its dependencies, the quality of its documentation, the time necessary for building and installing the packages are given in Table 2.

From the libraries given in Table 2, *MPI4py*, *H5py*, *MDAnalysis* were the easiest to build. Users are able to get these packages installed through the *Conda* package support. *OpenMPI*, *GCC*, can be easily configured and installed. The configure script supports a lot of different command line options, but the support for these libraries is very strong, they are widely used and the excellent documentation and discussion mailing lists provide users with great resources for consult, troubleshooting and tracking issues.

Global Array, and *PHDF5* have much slower installation especially because the libraries are built from source and variable configure options are required to support specific network interconnects and back-end run-time environments.

Package	Version	Description	Time to Result	Documentation	Installation	Dependencies
GCC	4.9.4	GNU Compiler Collection	Fast	Excellent	via configuration files, environment or command line options, minimal configuration is required	–
OpenMPI	1.10.7	MPI Implementation	Fast	Excellent	via configuration files, environment or command line options, minimal configuration is required	–
Global Array	5.6.1	Global Array	Slow	Good	via configuration files, environment or command line options, several optional configuration settings available	MPI 1.x/2.x/3.x implementation like MPICH or Open MPI built with shared/dynamic libraries, GCC
MPI4py	3.0.0	MPI for Python	Very Fast	Excellent	Conda Installation	Python 2.7, or above, MPI 1.x/2.x/3.x implementation like MPICH or Open-MPI built with shared/dynamic libraries, Cython
GA4py	1.0	Global Array for Python	Fast	Average	Python Setuptools	Python 2.7, or above, MPI 1.x/2.x/3.x implementation like MPICH or Open-MPI built with shared/dynamic libraries, Cython, MPI4py, Numpy
PHDF5	1.10.1	Parallel HDF5	Slow	Excellent	via configuration files, environment or command line options, several optional configuration settings available	MPI 1.x/2.x/3.x implementation like MPICH or Open-MPI, GNU, MPIF90, MPICC, MPICXX
H5py	2.7.1	Pythonic wrapper around the HDF5	Very Fast	Excellent	Conda Installation	Python 2.7, or above, PHDF5, Cython
MDAnalysis	0.17.0	Python library to analyze trajectories from MD simulations	Very Fast	Excellent	Conda Installation	Python >=2.7, or <3, Cython, GNU, Numpy

Table 2: Detailed comparison on the dependency and installation of different packages used in the present study on multi-user HPC systems

GA4py has only one release, and does not provide users with strong documentation and there are still room for improvement for this package.

We performed our benchmark on several HPC resources and therefore, we had to install all the related packages and tools on all resources. However, there are always differences in the resources because their set up and architectures differ from each other. For example, on SuperMIC although tool installation was done in the same way as Comet and also passed initial testing, the execution did not distribute the processes to all nodes. This was due to the fact that our custom OpenMPI installation did not correctly parse the node list offered by SuperMIC to our job. Thus, we had to manually pass the node lists to MPIRUN. In addition, we found that the loaded modules, along with library path changes, did not propagate to all nodes from our OpenMPI installation. OpenMPI's execution engine could access the correct libraries and was not able to launch the processes correctly. Reinstalling OpenMPI with enabling the flag to use the Open Run-Time Environment (ORTE) by default and including the

OpenMPI installation to BASHRC allowed for correct execution.

Overall, the installation was successful on all clusters and we were able to observe similar performances (will be discussed in the result section) which shows the applicability of the libraries for achieving near ideal scaling.

5. Methods

5.1. MPI for Python mpi4py

MPI for Python (*mpi4py*) is a Python wrapper written over Message Passing Interface (MPI) standard and allows any Python program to employ multiple processors [6, 7]. 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. 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 [6, 7], the performance degradation due to using *mpi4py* is not prohibitive and the overhead introduced by *mpi4py* is far smaller than the overhead associated to the use of interpreted versus compiled languages [8]. 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 [6].

5.2. Applications of Global Array

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.

Global array (GA) toolkit allows manipulating physically distributed dense multi-dimensional arrays without explicitly defining communication and synchronization between processes. Global Arrays in NumPy (GAiN) extends GA to python through Numpy [8]. 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 (*ga_distribution*, *ga_scatter*), 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 [8].

The global array itself is physically located in the local memory space of each process [18]. User can get a pointer to this memory by using ga_access function or one of its variants. Using this pointer it is possible to directly modify the data that is local to each process. The GA library keeps track of all these memory locations by recording a list of them when a global array is created. When a process tries to access a block of data, it first does a decomposition of the request into individual blocks representing the contribution to the total request from the data held locally on each processor [19]. The requesting processor then makes individual requests to each of the remote processors. The requests are implemented using the native one-sided semantics inside the the infini-band Verbs library. OpenIB/infiniband uses sys5 shmem within the node and will use the infiniband network card to communicate between nodes. Algorithm 3 describes RMSD algorithm in combination with the global array. In this algorithm, we use global array instead of message passage paradigm to see if we can reduce communication cost.

5.3. 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

Algorithm 3 MPI-parallel Multi-frame RMSD using Global Arrays

Input:*size*: Total number of frames assigned to each rank N_b
g_a: Initialized global array
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
Include: *Block_RMSD* from Algorithm 1

```
bsize ← ceil(trajectory.number_frames / size)
g_a ← ga.create(ga.C_DBL, [bsize*size,2], "RMSD")
buf ← np.zeros([bsize*size,2], dtype=float)
out ← Block_RMSD(topology, trajectory, xref0, start=start, stop=stop)
ga.put(g_a, out, (start,0), (stop,2))
if rank == 0 then
    buf ← ga.get(g_a, lo=None, hi=None)
end if
```

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 [9].

5.3.1. 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 [9]. With HDF5, modifications to file metadata must be done collectively and although all processes perform the same task, they do not wait until the others catch up [9]. Other tasks and any type of data operations can be performed independently by processes. In the present study, we use independent operations.

5.4. 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 3. Inside the code, relevant probs were taken and stored. We will abbreviate the timings in the following as variables t_{Ln} where Ln refers to the line number in algorithm 1. Similar calculations can be used for all other algorithms.

We directly measured inside our code (in the function `block_rmsd()`) the "I/O" time for ingesting the data from the file system into memory, ($t_{I/O}^{frame} = t_{L4}$) and the "compute" time per trajectory frame to perform the computation ($t_{comp}^{frame} = t_{L5}$). $t_{\mathrm{I/O}}$ is the summation of "I/O" time per frame and t_{compute} is the summation of "compute" time per frame for all the frames assigned to each rank (N_{frames}). $t_{end\_loop} = t_{L6} + t_{L7}$ is the time delay between the end of

the last iteration and exiting the for loop. $t_{opening\_trajectory} = t_{L2} + t_{L3}$ is the time which data structures are initialized and topology and trajectory files are opened (problem setup).

$t_{Communication_{MPI}} = t_{L16}$ is the time "Shuffle" time to gather ("reduce") all data from all processor ranks to rank zero. The total time (for all frames) spent in `block_rmsd()` is $t_{\mathrm{RMSD}} = t_{L1} + ... + t_{L8}$. There are parts of the code in `block_rmsd()` that are not covered by the detailed timing information of t_{compute} and $t_{\mathrm{I/O}}$. To measure the un-accounted 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 $t_N \equiv t_{RMSD} + t_{\mathrm{comm}}$.

5.5. Performance Measurement

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).

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

Additionally, we introduce another important performance parameter that determines whether we observe stragglers. We define this parameter as the ratio of compute time to I/O time:

$$\frac{\bar{t}^{frame}_{comp}}{\bar{t}^{frame}_{IO}} \approx \frac{t_{\mathrm{compute}}}{t_{\mathrm{I/O}}} \tag{2}$$

Item	Definition
N_{frames}	N_{frames}^{total}/N
t_{end_loop}	$t_{L6} + t_{L7}$
$t_{opening\_trajectory}$	$t_{L2} + t_{L3}$
t_{comp}	$\sum_1^{N_{frames}} t_{comp}^{frame}$
t_{IO}	$\sum_1^{N_{frames}} t_{I/O}^{frame}$
t_{all_frame}	$t_{L4} + t_{L5} + t_{L6}$
t_{RMSD}	$t_{L1} + ... + t_{L8}$
$t_{Communication_{MPI}}$	t_{L16}
$t_{Communication_{GA}}$	$t_{L5} + t_{L6} + t_{L7} + t_{L8}$
$t_{Overhead1}$	$t_{all\_frame} - t_{IO\_final} - t_{comp\_final} - t_{end\_loop}$
$t_{Overhead2}$	$t_{RMSD} - t_{all\_frame} - t_{opening\_trajectory}$
t_N	$t_{RMSD} + t_{Communication}$
t_{total}	$\max t_N$

Table 3: Summary of the notation of our performance modeling. Relevant probes in the codes are taken and stored, which we will abbreviate in here as t_{Ln} where Ln refers to the line number in the corresponding algorithm. $t_{Communication_{MPI}}$ and $t_{Communication_{GA}}$ are both referred to $t_{Communication}$ in the text

6. Performance Study

6.1. RMSD Benchmarks

RMSD algorithm for the present test case, represents a task for which computational workload per frame is smaller than I/O workload per frame ($t_{compute}^{frame}$ = 0.09 ms, t_{IO}^{frame} = 0.3 ms, thus $t_{\mathrm{compute}}/t_{\mathrm{I/O}} \approx 0.3$). 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 [4]. 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* [6, 7]) 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 take 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 2a, 2b for MPI.

In the example run in Figure 2d, 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 2d) shows that time for the actual computation, t_{compute}, is fairly constant across ranks. The time spent on reading data from the shared trajectory file on the Lustre file system into memory, $t_{\text{I/O}}$, shows variability across different ranks. Stragglers, however, appear to be defined by occasional much larger *communication* times, t_{comm} (line 16 in 1), 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. This initial analysis (especially Figure 2d) indicates that communication is a major issue.

Identification of Scalability Bottleneck

Figure 2c shows the scaling of t_{compute} and $t_{\text{I/O}}$ individulally. As shown, t_{compute} scales very well; however, $t_{\text{I/O}}$ does not show good scaling beyond a single node (24 cores) and that explains why we are seeing these variations in $t_{\text{I/O}}$ across different ranks (Figure 2d). Considering the results in Figures 2 and 2c, we can conclude that communication and I/O are the root causes for stragglers.

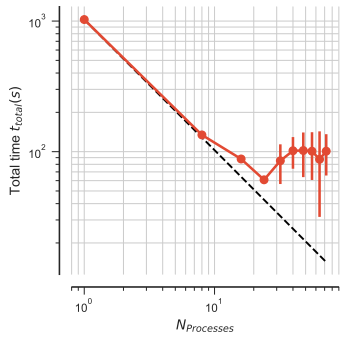
Hardware

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.

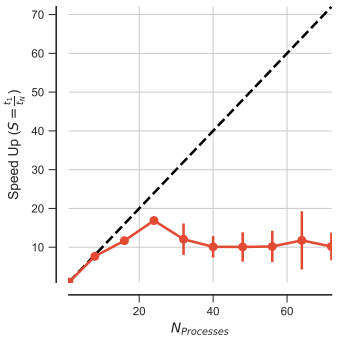
6.2. Dihedral Featurization Benchmarks

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

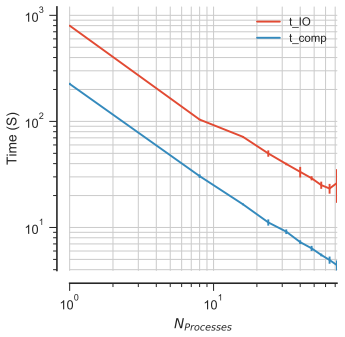
The system scales linearly and close to ideal (Figure 3a, 3b, 3c). Although, there is communication of large result arrays, which is costly for multiple ranks



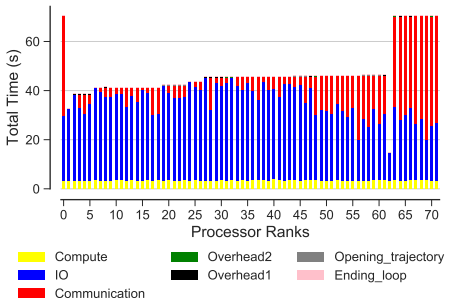
(a) Scaling total



(b) Speed-up



(c) t_{compute} and $t_{\mathrm{I/O}}$ scaling



(d) Compute t_{compute}, IO $t_{\mathrm{I/O}}$, 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). This is typical data from one run of the 5 repeats

Figure 2: Performance of the RMSD task with MPI which is I/O-bound $t_{\mathrm{compute}}/t_{\mathrm{I/O}} \approx 0.3$. Data are read from the file system (I/O included) and results are communicated back to rank 0 (communications included). Five repeats are performed to collect statistics. The error bars show standard deviation with respect to mean. MPI ranks 0 and 63 to 72 are stragglers, i.e., their total time far exceeds the mean of the majority of ranks.

***oliver: This does not look like t_comp/t_IO = 0.3 ? looks more like 2/30 = 0.06; the next page says 4/26=0.16. Please make sure that all numbers are consistent! ***mahzad: because t_comp/t_IO is per frame and it is also average, what I have reported is only average.

(Figure 4), the speed-up curve (Eq. 1) in Figure 3b demonstrates very good scaling with the number of cores (up to 72 cores on 3 nodes). The reason is that the communication cost (for $MPI.Gather()$-line 39 in 2) 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 [6] the overhead of the *mpi4py* package decreases as the array size to be communicated increases. The dihedral featurization workload has larger array size for all processor sizes (per task, a time series of feature vectors of size $N_{frames} \times (213 \times 2 \times 2)$ when compared to the RMSD workload (per task a float array of length N_{frames}) and therefore we are hypothesizing that the higher performance of *mpi4py* for larger array sizes has lead to better overall scaling. In addition, for higher computational workloads the competition over accessing the file is less severe as compared to lower computational workloads.

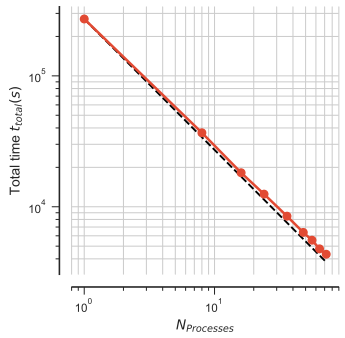
Overall, increasing the computational workload over I/O improves scaling. For large compute-bound workloads such as the dihedral featurization task, stragglers are eliminated and nearly ideal strong scaling is achieved. The fact that linear scaling is possible, even with expensive communications, makes 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).

6.3. Effect of $t_{compute}/t_{I/O}$ on Performance

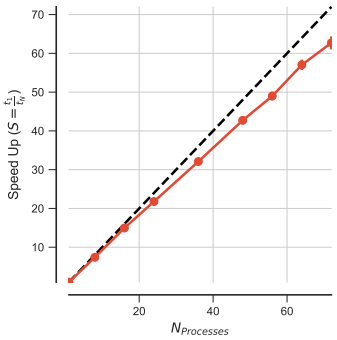
The RMSD task turned out to be I/O bound, i.e.,

$$\frac{t_{\mathrm{compute}}}{t_{\mathrm{I/O}}} \ll 1.$$

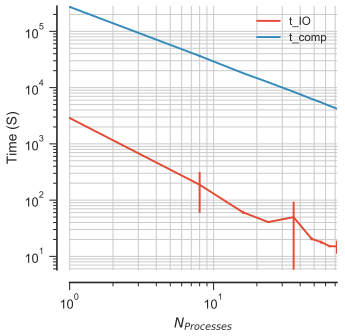
and we were not able to achieve good scaling above a single node. However, Dihedral featurization turned out to be compute bound and we were able to achieve near ideal scaling. We therefore, hypothesized that decreasing the relative I/O load with respect to compute load would also reduce the stragglers. We therefore increased the computational load so that the work became compute



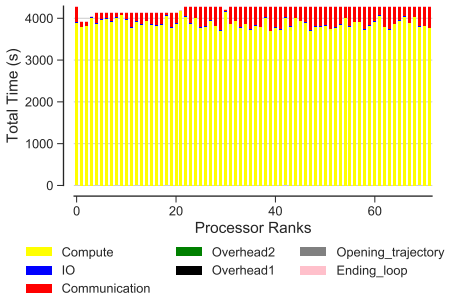
(a) Scaling total



(b) Speed-up



(c) t_{compute} and $t_{\text{I/O}}$ scaling



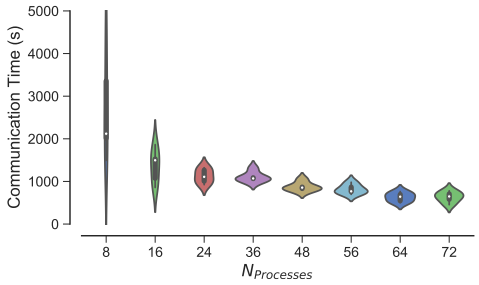
(d) Compute t_{compute}, IO $t_{\text{I/O}}$, communication t_{comm}, ending the for loop t_{end_loop}, opening the trajectory $t_{opening\_trajectory}$, and overheads $t_{Overhead1}$, $t_{Overhead2}$ per MPI rank (as described in methods). This is typical data from one run of the 5 repeats

Figure 3: Performance for the **dihedral featurization** workload, which is compute-bound $t_{\text{compute}}/t_{\text{I/O}} \approx 100$. Data are read from the file system (I/O included) and results are communicated back to rank 0 (communications included). Five repeats are performed to collect statistics. The error bars show standard deviation with respect to mean. No straggler is observed.

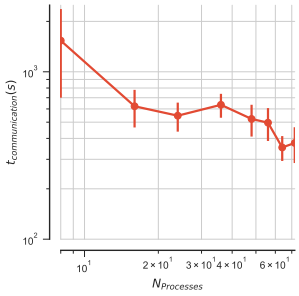
bound, i.e.,

$$\frac{t_{\text{compute}}}{t_{\text{I/O}}} \gg 1.$$

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



(a) Communication time distribution for different number of processes (shown as violin plots [20] 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*).

computational load we repeated the same RMSD calculation (line 10, algorithm 1) 40, 70 and 100 times in a loop respectively.

6.3.1. Increased workload (RMSD)

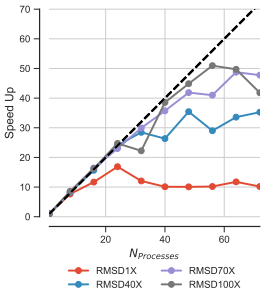
The RMSD workload was artificially increased forrty-fold ("40×"), seventy-fold ("70×"), and hundred-fold ("100×") and we measured performance as before. These workloads correspond to $t_{\mathrm{compute}}/t_{\mathrm{I/O}}$ ratio of 12, 21, 30 respectively as shown in Table 4. We performed this experiment to show the effect of $t_{\mathrm{compute}}/t_{\mathrm{I/O}}$ ratio on performance (Figure 5). On average, each rank's workload is $N_{frames} \times t_{\mathrm{I/O}}$ (where $N_{frames} = N_{frames}^{total}/N$ 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, and $X \times N_{frames} \times t_{\mathrm{compute}}$ for the RMSD calculation. X is the factor by which we increase the RMSD compute workload in our experiment.

As the $t_{\mathrm{compute}}/t_{\mathrm{I/O}}$ ratio increases, speed-up and performance improves and show overall better scaling than the I/O-bound workload, i.e. 1× RMSD (Figure 5a). When $t_{\mathrm{compute}}/t_{\mathrm{I/O}}$ 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

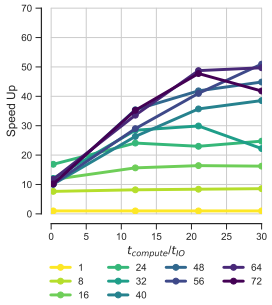
Workload	$t_{\text{compute}}/t_{\text{I/O}}$
RMSD 1×	0.3
RMSD 40×	12
RMSD 70×	21
RMSD 100×	30

Table 4: Change in $t_{\text{compute}}/t_{\text{I/O}}$ ratio with change in the RMSD workload. We artificially increased the RMSD workload in order to examine the effect of compute to I/O ratio on the performance.

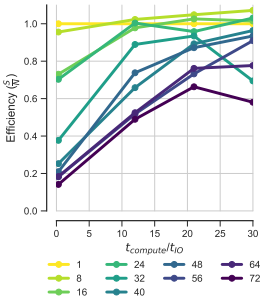
$t_{\text{compute}}/t_{\text{I/O}}$ ratio increases, the values of speed-up and efficiency get closer to their ideal value for each number of processor count.



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



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



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

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

Even for moderately compute-bound workloads such as the 40× and 70× RMSD tasks, increasing the computational workload over I/O reduced the impact of stragglers even though they still contribute to large variations in timing across different ranks and thus to somewhat erratic scaling.

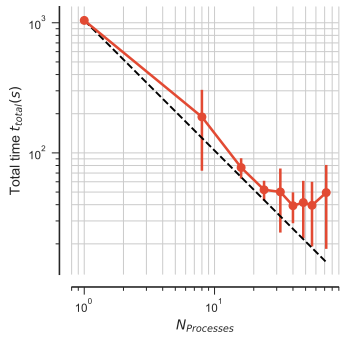
Given the results for Dihedral featurization and RMSD algorithms (Algorithms 2, and 1) and $X\times$ RMSD (Figure 5) we hypothesize that MPI competes with Lustre on the same network interface, which would explain why communication appears to be primarily a problem in the presence of I/O when $t_{\text{compute}}/t_{\text{I/O}}$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

load should open up the network for communication.

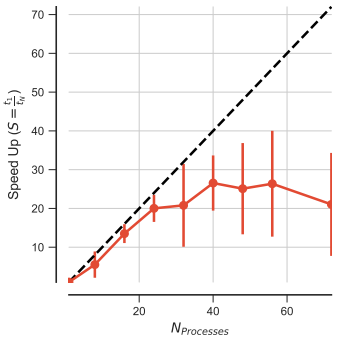
6.4. Communication Cost and Application of Global Array

As discussed in the previous sections, Figure 2d, for small $t_{\text{compute}}/t_{\text{I/O}}$ communication acts as the scalability bottleneck. In fact, when the processes communicate result arrays back to the master process (rank 0), some processes take much longer as compared to other processes. Now we want to know what strategies can be used to avoid communication cost.

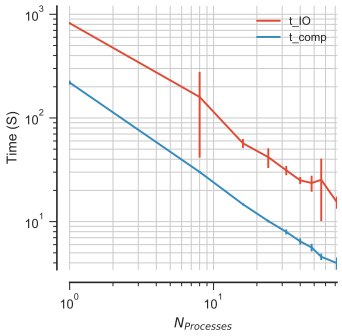
We used global array instead of collective communication in MPI and examined the change in the performance. In global array, we define one large RMSD array, and each MPI rank updates its associated block in the global RMSD array using *ga_put*. At the end, when all the processes exit `block_rmsd()` function and update their local block in the global array, rank 0 will access the whole global array using *ga_access*. In fact, in global arrays the time for communication is $t_{ga\_put} + t_{ga\_access}$. Given the speed up plots (Figure 6b) and total time scaling (Figure 6a) global array improves strong scaling performance. Although communication time has significantly decreased using global array (compare Figure 6d to Figure 2d), the existing variation in the dominant I/O part of the calculation plus two delayed MPI ranks due to the delay in opening the trajectory would still prevent ideal scaling (Figure 6c). This figure shows only one repeat out of 5 we performed for our benchmark study. Opening the trajectory was not a problem in other repeats. In fact, although communications were performed using global arrays, scaling is still far from ideal as a result of slow processes due to I/O variation and the delay in opening the trajectory. ***oliver: In Fig 8c it is the trajectory opening step that creates stragglers. This was not a problem before. Is this now ALWAYS the problem when using GA? Needs to be discussed. ***mahzad: no only happened in one repeat. Does not seem to me to be a problem with GA These slow processes take about 50 s, which are slower than the mean execution time of all ranks, i.e. 17 s.



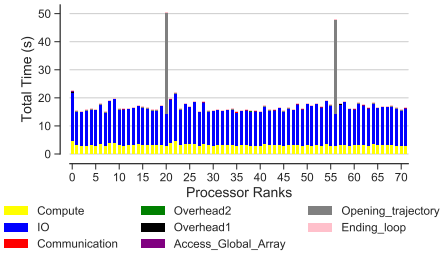
(a) Scaling total



(b) Speed-up



(c) t_{compute} and $t_{\text{I/O}}$ scaling



(d) Compute t_{compute}, IO $t_{\text{I/O}}$, 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). This is typical data from one run of the 5 repeats

Figure 6: Performance of the RMSD task with MPI using global array ($t_{\text{compute}}/t_{\text{I/O}} \approx 0.3$). Data are read from the file system (I/O included). All ranks update the global array and rank 0 accesses the whole RMSD array. Five repeats are performed to collect statistics. The error bars show standard deviation with respect to mean. MPI ranks 20 and 56 are stragglers, i.e., their total time far exceeds the mean of the majority of ranks.

6.5. I/O Cost

We showed previously that the I/O system can have a large effect on the parallel performance of the RMSD task [4], especially because the average time to perform the computation t_{compute} (about 0.09 ms) is about three times smaller than the I/O time $t_{\text{I/O}}$ (about 0.3 ms) (Figures 2c and 2). In fact, poor I/O performance is 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. The test trajectory has about $2,512,200$ frames in total and as a result there will be $2,512,200$ file access requests. Thus, 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 (Figures 2c and 2). Read throughput might be limited by the available bandwidth on the Infini-band network interface that serves the Lustre file system and access to files might be throttled. We show that depending on the cluster and its capabilities the throughput might become a problem for achieving good performance and we also show ways to overcome this problem and improve performance. In fact, we need to come up with ways and strategies to avoid the competition over file access across different ranks when $t_{\mathrm{compute}}/t_{\mathrm{I/O}}$ ratio is small. 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 trajectory file into as many segments as number of processes and MPI-based Parallel HDF5. We discuss these two approaches in detail in the following sections.

6.5.1. Splitting the Trajectories

In all the previous benchmarks all processes were using a shared trajectory file. In order to test our hypothesis that *I/O and communication compete over the network resources with small* $t_{compute}/t_{I/O}$ *ratio*, 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 accesses. For reference, the necessary time for splitting the trajectory file is given 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 7a and 7b) and the delay time for stragglers has also reduced from 65 to about 23 (Compare Figure 7c to 2d).

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 (Compare Figure 7c to Figure 2d), it is still delaying several processes and as a result leads to longer job completion time (Figure 7c). These delayed tasks impact performance as well and hence the speed-up is not still close to ideal scaling (Figure 7b).

Performance using Global Array

Previously, we showed that global array significantly reduces the communication cost (Section 6.4). We want to see how the performance looks like if we split our trajectory file and take advantage of global array as well. Again, we ran our benchmark up to 8 nodes (192 cores) and, we observed near ideal scaling behavior with efficiencies above 0.9 (Figure 7a and 7b) with no straggler tasks (Figure 7d). 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 interferes with the communications when $t_{\text{compute}}/t_{\text{I/O}} \ll 1$ (i.e. with a I/O bound workload).

6.5.2. Chain-Reader

6.5.3. 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 $2,512,200$ frames into HDF5 format was about 5400 s in our local resources with network file system (NFS). Again, we ran our benchmark up to 8 nodes (192 cores) and, we observed near ideal scaling behavior with efficiencies above 0.8 (Figure 9a and 9b) with no straggler tasks (Figure 9d). When we split our trajectory, scaling is better as compared to that of parallel I/O (Compare Figure 9b to Figure 7b). However, both methods scale very well up to 8 nodes and have comparable performance.

7. Performance Comparison of the RMSD task on different clusters

In this section we try to compare the performance of RMSD task on different HPC resources to examine the robustness of the methods we used for our performance study. HPC resources used for this purpose and their system configuration are given in Table 1. We perform these comparisons for several cases discussed previously. These cases include: Splitting the trajectories with collective communications in MPI, Splitting the trajectories with global array for communications, and MPI-based Parallel HDF5.

7.1. Splitting the Trajectories

Figure 10 shows how RMSD task scales with the increase in the number of cores on different HPC resources. When we split the trajectories the scaling follows the same pattern on both Comet and SuperMIC with global array and without global array. Both Comet and SuperMIC scales very well using Global array. RMSD task still scales on both clusters without global array; however, scaling is far from ideal scaling due to the communication cost (Refer to section 6.5.1 and Figure 7c). Overal, the scaling of the RMSD task is better on SuperMIC and the performance gap increases as we increase the number of cores. This is expected for the following reasons: First, CPU speed on SuperMIC is 2.8 GHz vs 2.5 GHz on Comet. This is $\approx 12\%$ performance difference in favor of SuperMIC. Second, for the same network speed the number of cores per node on SuperMIC (20 cores per node) is smaller than Comet (24 cores per node). Therefore, access to memory per core should be factor of $12/10 \approx 20\%$ faster on SuperMIC.

7.2. MPI-based Parallel HDF5

Figure 11 shows how RMSD task scales with the increase in the number of cores on different HPC resources using MPI-based parallel HDF5. The scaling follows the same pattern on both Comet and SuperMIC. Both Comet and SuperMIC scales very well. Overal, the scaling of the RMSD task is better on SuperMIC for the reasons that discussed above. Bridge performance is different

from Comet and SuperMIC. Bridge has 28 cores per compute node and when we use all cores for our calculations there is no scaling. However, decreasing the number of cores per node and having uniform workload distributions across all nodes can lead to significant improvements in the scaling as shown in Figure 11. The reason for performance difference between Comet and Bridge can be explained as below: First, CPU speed on Bridge is 2.3 GHz vs 2.5 GHz on Comet. This is $\approx 8.6\%$ performance difference in favor of Comet. Second, for the same network speed the number of cores per node on Bridge (28 cores per node) is larger than Comet (24 cores per node). Therefore, access to memory per core should be factor of $14/12 \approx 16\%$ faster on Comet. The memory access effect is obvious on the timing distribution across different rank shown for one run of the 5 repeats (Figure 12). Based on Figure 12 the I/O time distribution is pretty small and uniform across all ranks on Comet and SuperMIC (Figures 12b & 9d). However, on Bridge the I/O time is on average about two and a half times larger and the I/O time distribution is also erratic across different ranks (Figure 12a).

7.3. Comparison of Compute & I/O Scaling Across Different Clusters

Comparison of compute & I/O scaling across different clusters for different test cases and algorithms is shown in Table 5. The corresponding plots for compute and I/O scaling are shown in related sections. These plots together with Table 5 allow both quantitative and qualitative comparison of the compute and I/O time scaling. As can be seen in Table 5 for MPI-based parallel HDF5, both compute and I/O time on Bridge is larger than its corresponding value on Comet and SuperMIC. For example, with one core the corresponding compute and I/O time are about (387, 1318) versus (225, 423) and (273, 503) on Comet and SuperMIC respectively. This performance difference becomes more obvious as the number of cores increases. When the trajectories are splitted and global array is used for communication both Comet and SuperMIC show similar performance at small number of cores and the their performance difference increases as the number of cores increases. The reason why we see these

performance differences is explained in previous sections.

Cluster	Calculation	Load Ratio	Gather	File Access	Time	$N_{Processes}$							
						1	Comet: 24 Bridges: 24 SuperMIC: 20	Others: 48 Bridges: 48 SuperMIC: 40	Comet: 72 Bridges: 60 SuperMIC: 80	Comet: 96 Bridges: 78	Comet: 144 Bridges: 84 SuperMIC: 160	Comet: 192	Comet: 384 SuperMIC: 320
Comet	Dihedral Featurization	100	MPI	Single	$t_{I/O}$	2880 ± 0	40 ± 1.63	20 ± 1.22	15 ± 3.91	–	–	–	–
					t_{compute}	272000 ± 0	12440 ± 200.78	6305 ± 38.13	4225 ± 83.41				
Comet	RMSD 1X	0.3	MPI	Single	$t_{I/O}$	799 ± 5.22	49 ± 3.45	29 ± 1.3	26 ± 9.19	–	–	–	–
					t_{compute}	225 ± 5.4	11 ± 0.75	6 ± 0.35	4 ± 0.48				
Comet	RMSD 1X	0.3	GA	Single	$t_{I/O}$	820 ± 18.49	41 ± 8.99	23 ± 4.14	15 ± 2.06	–	–	–	–
					t_{compute}	219 ± 9.8	10 ± 0.3	5 ± 0.48	3 ± 0.54				
Comet	RMSD 1X	0.3	MPI	Splitting	$t_{I/O}$	799 ± 5.22	37 ± 1.22	18 ± 0.18	12 ± 0.14	9 ± 0.3	6 ± 0.66	4 ± 0.23	–
					t_{compute}	225 ± 5.4	11 ± 0.31	5 ± 0.07	3 ± 0.04	3 ± 0.11	2 ± 0.23	1 ± 0.07	
SuperMIC	RMSD 1X	0.3	MPI	Splitting	$t_{I/O}$	1013.75 ± 2.8	39.99 ± 0.36	19.18 ± 0.25	9.61 ± 0.28	–	4.83 ± 0.06	–	–
					t_{compute}	304.26 ± 2.55	12.41 ± 0.22	5.99 ± 0.09	3.08 ± 0.13		1.5 ± 0.01		
Comet	RMSD 1X	0.3	GA	Splitting	$t_{I/O}$	820 ± 18.5	36 ± 0.78	17 ± 0.3	11 ± 0.23	10 ± 1.7	5 ± 0.14	4 ± 0.07	–
					t_{compute}	219 ± 9.5	9 ± 0.22	4 ± 0.07	3 ± 0.04	2 ± 0.4	1 ± 0.05	1 ± 0.02	
SuperMIC	RMSD 1X	0.3	GA	Splitting	$t_{I/O}$	1027.62 ± 10.32	39.62 ± 0.2	19.66 ± 0.1	9.57 ± 0.1	–	4.86 ± 0.05	–	–
					t_{compute}	305.78 ± 3.47	12.16 ± 0.1	6.01 ± 0.007	2.97 ± 0.1		1.51 ± 0.03		
Comet	RMSD 1X	0.3	MPI	PHDF5	$t_{I/O}$	423 ± 5.88	19 ± 0.3	9 ± 0.13	6 ± 0.06	5 ± 0.12	3 ± 0.2	3 ± 0.25	1.57 ± 0.29
					t_{compute}	225 ± 6.55	10 ± 0.12	5 ± 0.1	3 ± 0.04	2 ± 0.05	1 ± 0.04	1 ± 0.03	0.76 ± 0.09
Bridges	RMSD 1X	0.3	MPI	PHDF5	$t_{I/O}$	1318.87 ± 10.42	67.93 ± 0.52	37.37 ± 0.2	30.35 ± 0.15	24.16 ± 0.89	22.5 ± 0.17	–	–
					t_{compute}	387.8 ± 5.51	21.97 ± 0.38	12.12 ± 0.34	9.79 ± 0.24	7.72 ± 0.03	7.18 ± 0.08		
SuperMIC	RMSD 1X	0.3	MPI	PHDF5	$t_{I/O}$	503.69 ± 2.57	12.96 ± 0.06	6.46 ± 0.02	3.2 ± 0.01	–	1.64 ± 0.01	–	0.82 ± 0.004
					t_{compute}	273.54 ± 4.7	23.44 ± 0.29	12.22 ± 0.43	7.3 ± 0.85		4.59 ± 0.96		1.55 ± 0.009

Table 5: Comparison of the compute and I/O scaling for different test cases and number of processes. Five repeats are performed to collect statistics. The mean value and the standard deviation with respect to mean are reported for each case.

8. Guidelines on the Parallel Analysis of Three Dimensional Time Series

Here we provide practical guidelines for parallel analysis of the three dimensional time series (MD trajectories) on HPC resources.

Rule 1 Calculate the value of $t_{\text{compute}}/t_{\text{I/O}}$ to see whether the task is compute bound ($\frac{t_{\text{compute}}}{t_{\text{I/O}}} \gg 1$) or IO bound ($\frac{t_{\text{compute}}}{t_{\text{I/O}}} \ll 1$). As discussed in Section 6.3 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.

Rule 2 For $\frac{t_{\text{compute}}}{t_{\text{I/O}}} \geqslant 100$ the task is compute bound and the task will scale very well as we showed in Section 6.2. However, if the size of the data to be communicated to rank 0 using the blocking collective communication ($MPI.Gather()$) is small, one might consider using global arrays to achieve near ideal scaling behavior (Section 6.4). In fact the overhead of *mpi4py* is large with respect to C for small array size [6]. 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).

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

Rule 3.1 If there is access to HDF5 format the recommended way will be to use MPI-based Parallel HDF5 (Section 6.5.3). 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. MD trajectories are often re-analyzed and therefore we suggest to incorporate trajectory conversion into the beginning of standard workflows for MD simulations. Alternatively, it will be a good idea to keep the trajectories in smaller chunks, e.g. when running simulations on HPC resources using Gromacs [??], users can run their simulations with "-noappend" option which will automatically store the output trajectories in small chunks.

Rule 3.2 If there is not access to HDF5, the trajectory file should be splitted into as many trajectory segments as the number of processes. Splitting the trajectories is fast and does not take much time (Appendix A).

Rule 3.3 The appropriate parallel implementation along with *Global Array* should be used on the trajectory segments (Section 6.5.1) to achieve near ideal scaling.

9. Conclusion

***mahzad: Add chain-reader later There are currently many freely available libraries for the analysis and processing of three-dimensional time series. However, dramatic increases in the 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.

To this aim, we tested our benchmark on RMSD (I/O bound) and Dihedral featurization (compute bound) algorithms in *MDAnalysis*. Our initial analysis showed that for sufficiently large per-frame workloads ($t_{\text{compute}}/t_{\text{I/O}} \approx 100$), close to ideal scaling was achievable (Figure 4). However the I/O bound workload ($t_{\text{compute}}/t_{\text{I/O}} \approx 0.3$) does not scale due to the appearance of *stragglers*.

This means that the ratio between compute load and I/O load has a crucial effect on the performance.

Different factors like opening the trajectory file, or other sources of overheads can be responsible for observing *stragglers* for I/O bound workload. But, for the I/O bound workload, both communication and I/O appeared to be the main scalability bottlenecks when using a shared file. Our data suggest that *stragglers are due to the competition between MPI and the Lustre file system on the shared Infini-band interconnect.* ***oliver: We/I put this hypothesis forward originally but I don?t think that we really have enough evidence to corroborate it. We know that we have a problem with multiple ranks accessing the same file but we don?t have this problem with splitting or parallel I/O. I think that?s really all we can say right now. ***mahzad: I think we have enough evidences, I tried to explain it in a better way. please have a look and let me know if it is still not clear from my reasoning ***oliver: Can you look in the literature to see what is known about Lustre and accessing a single file? Papers on parallel I/O should discuss this problem. ***mahzad: Will have a look

This is because when we remove I/O, communication does not appear to be the scalability bottleneck anymore (data not shown here). In fact, communication time, t_{comm}, could take much longer for *stragglers* than for "normal" MPI ranks when I/O has to be performed through a shared trajectory file (Figure 2d).

Additionally, the number of I/O requests is a function of number of frames in the trajectory. For I/O bound task and compute bound task with the same number of frames per trajectory the frequency of sending the I/O requests makes a big difference. For sufficiently large per-frame compute workload, the I/O requests interfere much less often with communication than an I/O bound task. This is why both communication and I/O appear to prevent us from achieving the near ideal scaling for an I/O bound task.

It should be also noted that, the effect of communication was less pronounced when the work became more compute-bound and this is because with compute-

bound tasks there is less competition over accessing the shared trajectory file. We showed this effect by changing the ratio between compute load and I/O load and studying its impact on the performance.

Therefore, for I/O bound tasks we needed to come up with solution to overcome *stragglers*. We were able to achieve much better performance in our RMSD benchmark when we used global array toolkit instead of message-passing interface for communication. Using global array, we did not observe any delayed task due to communication (Figure 6) and it significantly reduced the communication cost. However, reducing communication cost was not enough for achieving near ideal scaling because I/O is more dominant for an I/O bound task.

We showed several approaches to improve I/O scaling. We were able to improve I/O through splitting the shared trajectory file and MPI-based parallel I/O through HDF5 file (Figures 7 and 9). In both cases we were able to achieve near ideal scaling. With splitting the trajectories, effect of communication is still apparent on the performance; however together with global array toolkit we could achieve near ideal scaling (Figure 7). ***oliver: I actually do not understand why we need GA for splitting but not for parallel MPI. ***mahzad: I mentioned before in my presentation in Spidal meeting that I myself do not have an answer for this.

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 to tackle their problem depending on the workload being I/O bound or compute bound. The analysis indicates that splitting the trajectories in combination with global array or parallel I/O will make it feasible to run a I/O bound task on scalable computers up to 8 nodes and achieve near ideal scaling behavior. In addition, we have examined all these benchmarks on several HPC resources in order ensure the robustness of our approach.

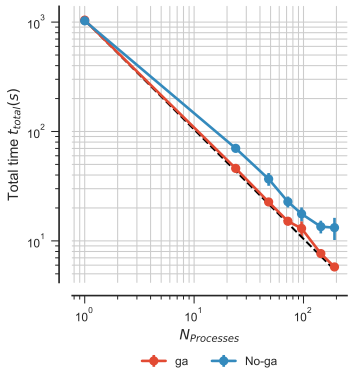
Acknowledgements Funding: This work was supported by the National Science Foundation [grant numbers 1443054,1440677]. Computational resources were provided by NSF XRAC awards TG-MCB090174 and TG-MCB130177.

References

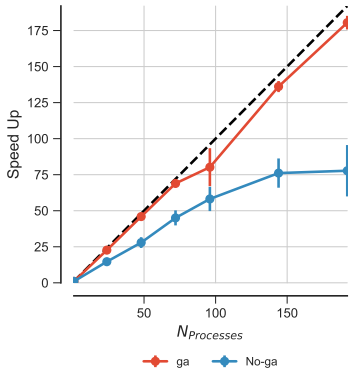
1. Cheatham T, Roe D. The impact of heterogeneous computing on workflows for biomolecular simulation and analysis. Computing in Science Engineering 2015;17(2):30–9. doi:10.1109/MCSE.2015.7.
2. Gowers RJ, Linke M, Barnoud J, Reddy TJE, Melo MN, Seyler SL, et al. MDAnalysis: A Python package for the rapid analysis of molecular dynamics simulations. In: Benthall S, Rostrup S, editors. Proceedings of the 15th Python in Science Conference. Austin, TX: SciPy; 2016, p. 102 –9. URL: http://mdanalysis.org.
3. Michaud-Agrawal N, Denning EJ, Woolf TB, Beckstein O. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. J Comp Chem 2011;32:2319–27. doi:10.1002/jcc.21787.
4. Khoshlessan M, Paraskevakos I, Jha S, Beckstein O. Parallel analysis in MDAnalysis using the Dask parallel computing library. In: Katy Huff , David Lippa , Dillon Niederhut , Pacer M, editors. Proceedings of the 16th Python in Science Conference. Austin, TX: SciPy; 2017, p. 64–72. doi:10.25080/shinma-7f4c6e7-00a.
5. Rocklin M. Dask: Parallel computation with blocked algorithms and task scheduling. In: Proceedings of the 14th Python in Science Conference. 130–136; 2015,URL: https://github.com/dask/dask.
6. Dalcín LD, Paz RR, Kler PA, Cosimo A. Parallel distributed computing using python. Advances in Water Resources 2011;34(9):1124 –39. doi:10.1016/j.advwatres.2011.04.013; new Computational Methods and Software Tools.
7. Dalcín L, Paz R, Storti M. MPI for python. Journal of Parallel and Distributed Computing 2005;65(9):1108 –15. doi:10.1016/j.jpdc.2005.03.010.
8. DAILY JA. Gain: Distributed array computation with python. Master's thesis; School of Electrical Engineering and Computer Science, WASHINGTON STATE UNIVERSITY; 2009.
9. Collette A. In: Python and HDF5. 2014,.
10. Roe DR, Thomas E. Cheatham I. Ptraj and cpptraj: Software for processing and analysis of molecular dynamics trajectory data. Journal of Chemical Theory and Computation 2013;9(7):3084–95. URL: http://dx.doi.org/10.1021/ct400341p. doi:10.1021/ct400341p. arXiv:http://dx.doi.org/10.1021/ct400341p; pMID: 26583988.
11. Tu T, Rendleman CA, Borhani DW, Dror RO, Gullingsrud J, Jensen MO, et al. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In: 2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis. 2008, p. 1–12. doi:10.1109/SC.2008.5214715.
12. Liu P, Agrafiotis DK, Theobald DL. Fast determination of the optimal rotational matrix

for macromolecular superpositions. J Comput Chem 2010;31(7):1561–3. doi:10.1002/jcc.21439.

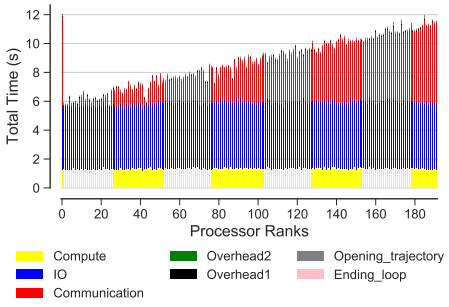
13. Theobald DL. Rapid calculation of RMSDs using a quaternion-based characteristic polynomial. Acta Crystallogr A 2005;61(Pt 4):478–80. doi:10.1107/S0108767305015266.
14. P L, K AD, L TD. Fast determination of the optimal rotational matrix for macromolecular superpositions. Journal of computational Chemistry 2010;31:1561–3.
15. Sittel F, Jain A, Stock G. Principal component analysis of molecular dynamics: on the use of cartesian vs. internal coordinates. J Chem Phys 2014;141(1):014111. doi:10.1063/1.4885338.
16. Seyler SL, Beckstein O. Sampling of large conformational transitions: Adenylate kinase as a testing ground. Molec Simul 2014;40(10–11):855–77. doi:10.1080/08927022.2014.919497.
17. Seyler S, Beckstein O. Molecular dynamics trajectory for benchmarking MDAnalysis. 2017. URL: https://figshare.com/articles/Molecular_dynamics_trajectory_for_benchmarking_MDAnalysis/5108170. doi:10.6084/m9.figshare.5108170.
18. Nieplocha J, Palmer B, Tipparaju V, Krishnan M, Trease H, Aprà E. Advances, applications and performance of the global arrays shared memory programming toolkit. The International Journal of High Performance Computing Applications 2006;20(2):203–31.
19. Personal communication, pacific northwest national laboratory ????;.
20. Hintze JL, Nelson RD. Violin plots: A box plot-density trace synergism. The American Statistician 1998;52(2):181–4. URL: http://www.tandfonline.com/doi/abs/10.1080/00031305.1998.10480559. doi:10.1080/00031305.1998.10480559. arXiv:http://www.tandfonline.com/doi/pdf/10.1080/00031305.1998.10480559.



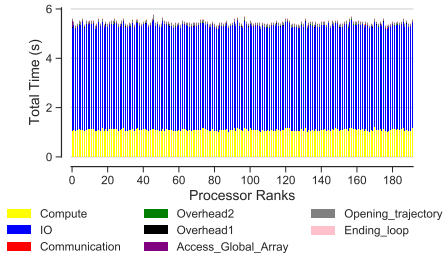
(a) Scaling total



(b) Speed-up



(c) Examples of timing per MPI rank without global array



(d) Examples of timing per MPI rank using global array

Figure 7: Comparison on the performance of the RMSD task with MPI when the trajectories are splitted using global array and without global array ($t_{\text{compute}}/t_{\text{I/O}} \approx 0.3$). Data are read from the file system (I/O included). In case of global array, all ranks update the global array (ga_{put}) and rank 0 accesses the whole RMSD array through the global memory address (ga_{get}).

***oliver: The compute/IO ratio is about 1:4 judging from the graph, i.e., 0.25 ? or did you calculate the ratio for the serial version? ***mahzad: Yes, this is based on serial version

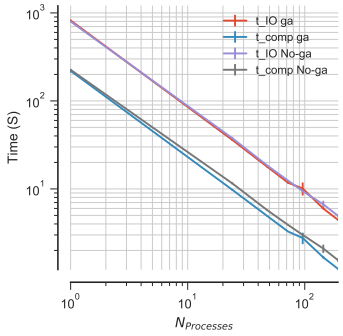
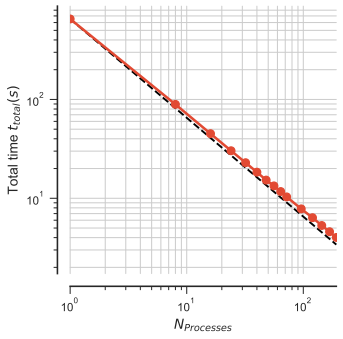
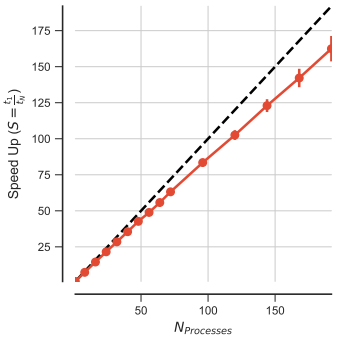


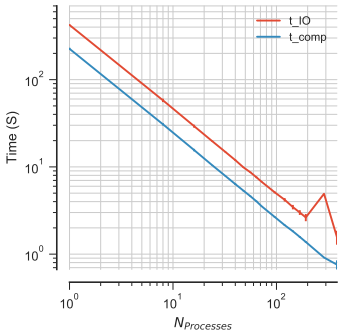
Figure 8: Comparison on the scaling of the t_{compute} and $t_{\text{I/O}}$ of the RMSD task when the trajectories are splitted with global array and without global array.



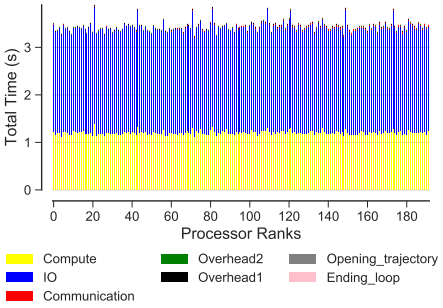
(a) Scaling total



(b) Speed-up

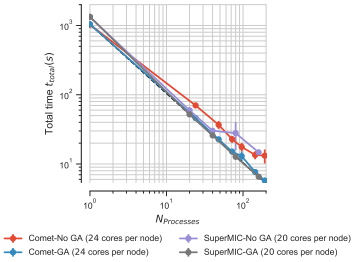


(c) t_{compute} and $t_{\text{I/O}}$ scaling

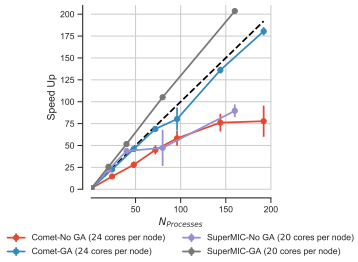


(d) Compute t_{compute}, IO $t_{\text{I/O}}$, communication t_{comm}, ending the for loop t_{end_loop}, opening the trajectory $t_{opening\_trajectory}$, and overheads $t_{Overhead1}$, $t_{Overhead2}$ per MPI rank (as described in methods).

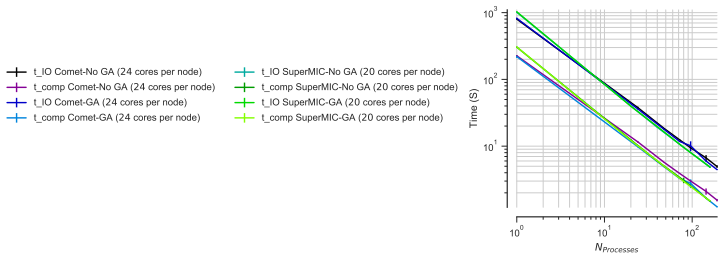
Figure 9: Performance of the RMSD task with MPI-based parallel HDF5 ($t_{\text{compute}}/t_{\text{I/O}} \approx 0.3$). Data are read from the file system from a shared HDF5 file instead of XTC format (Independent I/O).



(a) Scaling total

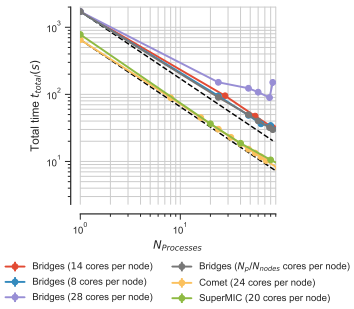


(b) Speed-up

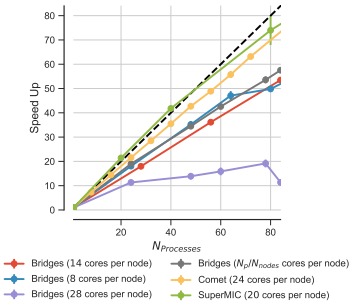


(c) Scaling of the t_{compute} and $t_{\mathrm{I/O}}$ of the RMSD task with MPI when the trajectories are splitted using global array and without global array.

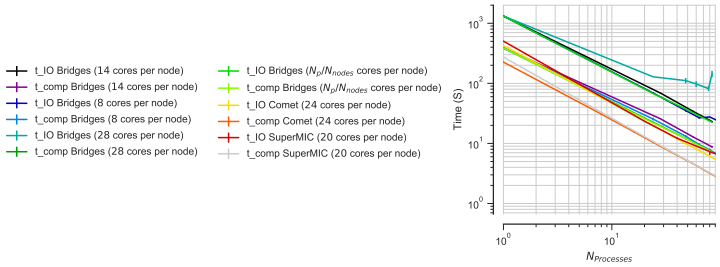
Figure 10: Comparison of the performance of the RMSD task with MPI ($t_{\mathrm{compute}}/t_{\mathrm{I/O}} \approx 0.3$) when the trajectories are splitted using global array and without global array across different clusters. Data are read from the file system. Five repeats are performed to collect statistics. The error bars show standard deviation with respect to mean.



(a) Scaling total

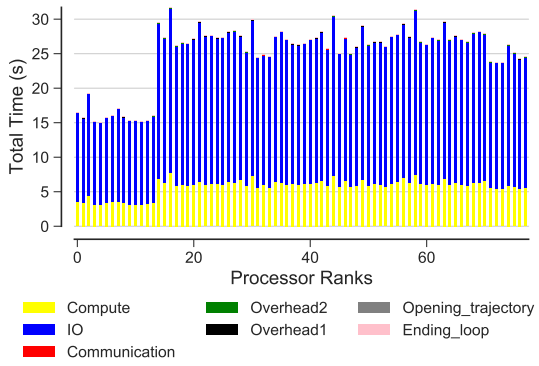


(b) Speed-up

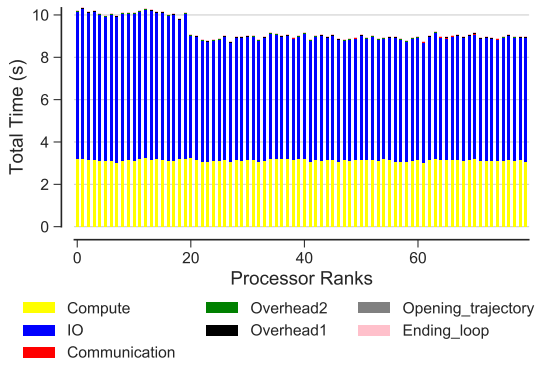


(c) Scaling of the t_{compute} and $t_{\mathrm{I/O}}$ of the RMSD task with MPI-based parallel HDF5 (Independent I/O)

Figure 11: Comparison of the performance of the RMSD task with MPI ($t_{\mathrm{compute}}/t_{\mathrm{I/O}} \approx 0.3$) across different clusters. Data are read from a shared HDF5 file instead of XTC format (Independent I/O) and results are communicated back to rank 0 (communications included). Five repeats are performed to collect statistics. The error bars show standard deviation with respect to mean.



(a) Bridge



(b) SuperMIC

Figure 12: Examples of timing per MPI rank for RMSD task with MPI-based parallel HDF5 ($t_{\mathrm{compute}}/t_{\mathrm{I/O}} \approx 0.3$) on PSC Bridge (Top) and SuperMIC (Buttomn). This is typical data from one run of the 5 repeats.

Appendix A. Detailed timing for splitting the trajectories

Table A.6 shows the necessary time for splitting the trajectory file using MPI on SDSC Comet.

Number of trajectory segments	N_p used for writing the segments	time (s)
24	24	89.92
48	48	46.79
72	72	33.7
96	96	25.12
144	144	43.7
196	196	13.49

Table A.6: The wall-clock time spent for writing N_p trajectory segments using N_p processes using MPI on SDSC Comet

Appendix B. Python codes used for the benchmark study

Appendix B.1. Python code used for RMSD *task with MPI for Python*

```
import sys
import numpy as np
import MDAnalysis as mda
from MDAnalysis.analysis import rms
import time
from shutil import copyfile
import glob, os
import mpi4py
from mpi4py import MPI
#---------------------------------------
MPI.Init

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
#---------------------------------------
j = sys.argv[1]

def block_rmsd(index, topology, trajectory, xref0, start=None, stop=None, step=
    None):
    clone = mda.Universe(topology, trajectory)
    g = clone.atoms[index]

```

```
    bsize = int(stop-start)
    results = np.zeros([bsize,2], dtype=float)

    for iframe, ts in enumerate(clone.trajectory[start:stop:step]):
        results[iframe, :] = ts.time, rms.rmsd(g.positions, xref0, center=True,
             superposition=True)

    return results
#--------------------------------------------------------------------
# Check the files in the directory
PSF = os.path.abspath(os.path.normpath(os.path.join(os.getcwd(),'files/adk4AKE.
     psf')))
longXTC = os.path.abspath(os.path.normpath(os.path.join(os.getcwd(),'newtraj.xtc
     ')))
longXTC1 = os.path.abspath(os.path.normpath(os.path.join(os.getcwd(),'files/
     newtraj{}.xtc'.format(j))))

if rank == 0:
   copyfile(longXTC, longXTC1)
   u = mda.Universe(PSF, longXTC1)
   print(len(u.trajectory))

MPI.COMM_WORLD.Barrier()
#--------------------------------------------------------------------
u = mda.Universe(PSF, longXTC1)
mobile = u.select_atoms("(resid 1:29 or resid 60:121 or resid 160:214) and name
     CA")[1:147]
index = mobile.indices
topology, trajectory = mobile.universe.filename, mobile.universe.trajectory.
     filename
ref0 = mobile
xref0 = ref0.positions-ref0.center_of_mass()

# Create each segment for each process
frames_seg = np.zeros([size,2], dtype=int)
bsize = int(np.ceil(mobile.universe.trajectory.n_frames / float(size)))
for iblock in range(size):
    frames_seg[iblock, :] = iblock*bsize, (iblock+1)*bsize

d = dict([key, frames_seg[key]] for key in range(size))

start, stop = d[rank][0], d[rank][1]

# Block-RMSD in Parallel
out = block_rmsd(index, topology, trajectory, xref0, start=start, stop=stop,
     step=1)
```

```
815
63  if rank == 0:
64     data1 = np.zeros([size*bsize,2], dtype=float)
65  else:
66     data1 = None
820
68  comm.Gather(out[0], data1, root=0)
69
70  if rank == 0:
71     data = np.zeros([size,5], dtype=float)
825 else:
73     data = None
74
75  comm.Gather(np.array(out[1:], dtype=float), data, root=0)
76
830 MPI.Finalize
```

Appendix B.2. Python code used for RMSD *task using global array with MPI for Python.*

```
835 import sys
2   import numpy as np
3   import MDAnalysis as mda
4   from MDAnalysis.analysis import rms
5   import time
840 from shutil import copyfile
7   import glob, os
8   import mpi4py
9   from mpi4py import MPI
10  from ga4py import ga
845 from ga4py import gain
12  #---------------------------------------
13  ga.initialize()
14  comm = gain.comm()
15
850 size = ga.nnodes()
17  rank = ga.nodeid()
18  #---------------------------------------
19  j = sys.argv[1]
20
855 def block_rmsd(index, topology, trajectory, xref0, start=None, stop=None, step=
        None):
22      clone = mda.Universe(topology, trajectory)
23      g = clone.atoms[index]
```

```
24
860     print("block_rmsd", start, stop, step)
26      bsize = int(stop-start)
27      results = np.zeros([bsize,2], dtype=float)
28
29      for iframe, ts in enumerate(clone.trajectory[start:stop:step]):
865         results[iframe, :] = ts.time, rms.rmsd(g.positions, xref0, center=True,
                superposition=True)
31
32      return results
33  #--------------------------------------------------------------------------
870 PSF = os.path.abspath(os.path.normpath(os.path.join(os.getcwd(),'files/adk4AKE.
        psf')))
35  longXTC = os.path.abspath(os.path.normpath(os.path.join(os.getcwd(),'newtraj.xtc
        ')))
36  longXTC1 = os.path.abspath(os.path.normpath(os.path.join(os.getcwd(),'files/
875     newtraj{}.xtc'.format(j))))
37
38  if rank == 0:
39     copyfile(longXTC, longXTC1)
40     u = mda.Universe(PSF, longXTC1)
880    print(len(u.trajectory))
42
43  ga.sync()
44  #--------------------------------------------------------------------------
45  u = mda.Universe(PSF, longXTC1)
885 mobile = u.select_atoms("(resid 1:29 or resid 60:121 or resid 160:214) and name
        CA")
47  index = mobile.indices
48  topology, trajectory = mobile.universe.filename, mobile.universe.trajectory.
        filename
890 ref0 = mobile
50  xref0 = ref0.positions-ref0.center_of_mass()
51  bsize = int(np.ceil(mobile.universe.trajectory.n_frames / float(size)))
52  g_a = ga.create(ga.C_DBL, [bsize*size,2], "RMSD")
53  buf = np.zeros([bsize*size,2], dtype=float)
895
55  # Create each segment for each process
56  frames_seg = np.zeros([size,2], dtype=int)
57  bsize = int(np.ceil(mobile.universe.trajectory.n_frames / float(size)))
58  for iblock in range(size):
900     frames_seg[iblock, :] = iblock*bsize, (iblock+1)*bsize
60
61  d = dict([key, frames_seg[key]] for key in range(size))
62
63  start, stop = d[rank][0], d[rank][1]
```

```
# Block-RMSD in Parallel
out = block_rmsd(index, topology, trajectory, xref0, start=start, stop=stop, step=1)

ga.put(g_a, out[0][:,:], (start,0), (stop,2))

if rank == 0:
   buf = ga.get(g_a, lo=None, hi=None)

if rank == 0:
   data = np.zeros([size,5], dtype=float)
else:
   data = None

comm.Gather(np.array(out[1:], dtype=float), data, root=0)

ga.destroy(g_a)
ga.terminate()
```

Appendix B.3. Python code used for Dihedral Featurization *task with MPI for Python*

```
import MDAnalysis as mda
import numpy as np
import glob
from itertools import chain
import time
from shutil import copyfile
import glob, os
import mpi4py
from mpi4py import MPI
import sys
#---------------------------------------
MPI.Init

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
#---------------------------------------
j = sys.argv[1]

def angle2sincos(x):
    """Convert angle x to (cos x, sin x).
```

```
22
950     Parameters
24      ----------
25      x : float or array_like
26
27      Returns
955     -------
29      feature_vector : array
30          1D feature vector ``[cos(x[0]), sin(x[0]), cos(x[1]), sin(x[1]), ...]``.
31      """
32      x = np.deg2rad(x)
960     return np.ravel(np.transpose([np.cos(x), np.sin(x)]))
34
35  def residues_to_dihedrals(residues):
36      """Return list of [phi1, psi1, phi2, psi2, ...] dihedral objects"""
37      return list(chain.from_iterable(
965             (res.phi_selection().dihedral, res.psi_selection().dihedral) for res
                    in residues))
39
40  def featurize_dihedrals(dihedrals):
41      angles = [dihedral.value() for dihedral in dihedrals]
970     return angle2sincos(angles)
43
44  def block_dihedrals(index, topology, trajectory, start=None, stop=None, step=
        None):
45      start00 = time.time()
975     clone = mda.Universe(topology, trajectory)
47      g = clone.atoms[index]
48      residues = g.residues[1:-1]
49      dihedrals = residues_to_dihedrals(residues)
50
980     print("block_rmsd", start, stop, step)
52      print(len(clone.trajectory))
53      bsize = stop-start
54      results = []
55
985     for iframe, ts in enumerate(clone.trajectory[start:stop:step]):
57          results.append(featurize_dihedrals(dihedrals))
58
59      return np.array(results)
60  #-----------------------------------------------------------------------
990 PSF = os.path.abspath(os.path.normpath(os.path.join(os.getcwd(),'files/adk4AKE.
        psf')))
62  longXTC = os.path.abspath(os.path.normpath(os.path.join(os.getcwd(),'newtraj.xtc
        ')))
```

```
63  longXTC1 = os.path.abspath(os.path.normpath(os.path.join(os.getcwd(),'files/
995       newtraj{}.xtc'.format(j))))
64
65  if rank == 0:
66      copyfile(longXTC, longXTC1)
67      u = mda.Universe(PSF, longXTC1)
1000    print(len(u.trajectory))
69
70  MPI.COMM_WORLD.Barrier()
71  #--------------------------------------------------------------------
72  u = mda.Universe(PSF, longXTC1)
1005 mobile = u.select_atoms("protein")
74  index = mobile.indices
75
76  topology, trajectory = mobile.universe.filename, mobile.universe.trajectory.
         filename
1010 bsize = int(np.ceil(mobile.universe.trajectory.n_frames / float(size)))
78
79  # Create each segment for each process
80  frames_seg = np.zeros([size,2], dtype=int)
81  bsize = int(np.ceil(mobile.universe.trajectory.n_frames / float(size)))
1015 for iblock in range(size):
83      frames_seg[iblock, :] = iblock*bsize, (iblock+1)*bsize
84
85  d = dict([key, frames_seg[key]] for key in range(size))
86
1020 start, stop = d[rank][0], d[rank][1]
88
89  out = block_dihedrals(index, topology, trajectory, start=start, stop=stop, step
         =1)
90
1025 if rank == 0:
92     data1 = np.zeros([size*bsize,848], dtype=float)
93  else:
94     data1 = None
95
1030 comm.Gather(out[0], data1, root=0)
97
98  if rank == 0:
99      data = np.zeros([size,5], dtype=float)
100 else:
1035     data = None
102
103 comm.Gather(np.array(out[1:], dtype=float), data, root=0)
104
105 MPI.Finalize
```

1040