Guidelines on 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. Peta-scale molecular dynamics simulations provide a powerful tool for investigating biological systems. The exponential growth in the computational power of these simulations has lead to these large output files.

MDAnalysis is an open source Python data analysis library for post-processing the MD output data and provides optimized classes and functions for important components of scientific code such as multidimensional arrays or linear algebra routines. *MDAnalysis*; however, misses effective use of high performance computing (HPC) resources for efficiently analyzing these trajectories and more importantly achieving linear scaling still remains a big challenge.

Present work aims to provide insights, guidelines and strategies to the community on how to take advantage of the available HPC resources to gain the best possible performance inside *MDAnalysis*. We investigated a single program multiple data (SPMD) execution model where each process executes the same

*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)

program, to parallelize the Map-Reduce Root Mean Square Distance (RMSD) and Dihedral Featurization algorithms for analysis of MD trajectories in the MDAnalysis library. We employ the Python language because it is widely used in the bio-molecular simulation field and focus on an MPI-based implementations. We notice that straggler tasks negatively impact the performance and act as scalability bottlenecks.

Straggler tasks are a very common problem in distributed environments and are significantly slower ($\approx 6\times$) 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 a single node. We introduce two important performance parameters $t_{Compute}/t_{IO}$ and $t_{Compute}/t_{Communication}$ which determines whether we observe any stragglers.

In addition, we show that I/O and communication lead to stragglers. 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

Molecular dynamics simulations are the most powerful and fundamental tool among a wide variety of techniques that enable scientists to achieve detailed info regarding the function of bio-molecules in living cells [1]. MD is widely used

to understand structure-to-function relationships of complex protein systems. Data analysis tools and libraries have been developed to extract the desired information from these MD simulations [2–7]. *MDAnalysis* [2, 3] is an open-source object-oriented Python library for structural and temporal analysis of molecular dynamics simulation trajectories and individual protein structures.

Development of powerful parallel supercomputers has significantly increased the physical and time scales of MD simulations. Present simulation times are close to biologically relevant ones and this has lead to rapid increase in the amount of data produced by MD simulations. As a result, typical trajectory sizes from MD simulations range from Gigabytes to Terabytes [8]. Therefore, parallel algorithms and frameworks are the key to meeting the scalability and performance requirements for analyzing these trajectory files.

MDAnalysis does not provide parallel analysis of these trajectories. Thus, post-processing and analyzing these trajectories can become a very tedious process and requires great amount of time when performed serially. As a result, we are trying to look for state of the art HPC tools (MPI and OpenMP) or Big Data ecosystem to speed up post processing task of MD trajectories. However, efficiently programming for a parallel environment and achieving near ideal scaling performance can be a very challenging task.

In our previous study, we used a parallel map-reduce approach to study the performance of RMSD task [9, 10]. We previously looked at the *Dask* library [11], 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 [12, 13]. 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 processes 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. 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 analyzing molecular dynamics (MD) trajectories, in order to speed-up post-processing of MD trajectories in *MDAnalysis* library.

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 [14]. Based on our initial analysis, there are two important performance parameters, $t_{Compute}/t_{IO}$ and $t_{Compute}/t_{Communication}$ which are 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 and the ratio of computational to communication load respectively. 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 a single node. For the algorithms with small $t_{Compute}/t_{IO}$, we need to come up with strategies to improve scaling and overcome straggler problems. In addition, when $t_{Compute}/t_{Communication}$ plays an important role on scaling and performance. The details on these are given in the results section. 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 show that I/O time does not scale beyond a single node. In order to improve I/O scaling, we used two dif-

ferent approaches: MPI-based approach using Parallel HDF5 [15], 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

Long Tail phenomena, whereby a small proportion of task stragglers significantly impede job completion time is a big challenge toward achieving improved performance [16]. Over the past few years a significant amount of research has been done trying to detect and mitigate stragglers [17–21]. There are also works to find the stragglers root-cause in many big data processing systems [22–26].

Straggler root-cause has both internal and external factors. Internal factors include heterogeneous capacity of worker nodes, and resource competition due to other tasks running on the same worker node. External factors include resource competition due to co-hosted applications, input data skew, remote input or output source being too slow and faulty hardware [19].

Spark introduces stragglers for a number of different reasons: 1) garbage collections [27, 28], 2) JVM positioning to cores [27], 2) the delay's introduced while the task moves from the scheduler to executing [29], 3) Disk I/O during shuffling, 4) Java's just-in-time compilation [28], and 5) output skew [28]. In addition to these reasons, stragglers on Spark have been attributed to the overall performance of the worker or competition between resources [30].

Tuning resource allocation and tuning parallelism are other methods to improve Spark job performance. However, manual tuning are laborious and imprecise. Many frameworks suggest breaking the workload into many small tasks that are dynamically scheduled at runtime [17]. This approach is only effective in systems with high-throughput, low-latency task schedulers and efficient data materialization though. In general, finer-grained splitting produces fewer stragglers; however, at some point the overhead of a large number of splits starts to

dominate.

Some frameworks will identify the straggler nodes using a slow Node-Threshold. They identify straggler node when its performance score is less than the average performance score of all nodes and therefore prevent to launch any tasks on these slow nodes. The straggler node can be due to the faulty hardware or mis-configuration [18]. However, this solution only addresses hardware-based stragglers [19].

For most of the factors causing stragglers, speculative execution and its improved versions are an effective way to solve the straggler problem. Speculative execution (back-up tasks) is a replication-based reactive straggler mitigation technique that spawns redundant copies of the slow running tasks, hoping a copy will reach completion before the original. This is the most prominently used in production clusters at Facebook and Microsoft Bing and is also implemented by both Hadoop and Google's MapReduce [18]. Google reported that speculative execution can improve job running times by 44% [18]. However, without any additional information, such reactive techniques can not differentiate between nodes that are inherently slow and nodes that are temporarily overloaded (i.e. input data skew or data imbalance) [19, 20].

Sampling or similar techniques can help estimate the data distribution to split the data more evenly. However, such statistics are often expensive to collect, insufficiently precise, or obsolete. Moreover, stragglers can happen even if the data distribution is balanced, due to unbalanced processing complexity for different parts of the data [9].

SkewTune is another technique for addressing data imbalance. SkewTune identifies the task with the greatest expected remaining processing time when a node becomes idle. The un-processed input data of this straggling task is then pro-actively repartitioned in a way that fully utilizes the nodes in the cluster and preserves the ordering of the input data so that the original output can be reconstructed by concatenation [21]. SkewTune can significantly reduce job runtime and adds little to no overhead in the absence of skew; however, the cost of fully reading the rest of a straggler's input can be prohibitive, in particular,

making the technique meaningless if the straggler was already dominated by the cost of reading the input.

Garraghan et al. [16] studied stragglers on Virtualized Cloud Data-centers. Through statistical analysis they found that the most frequent reasons were high CPU utilization, disk utilization, unhandled I/O access requests and network package loss. About 3% to 6.5% of the tasks were stragglers resulting to a 37.8% to almost 50% of jobs negatively impacted. In this study, task stragglers are defined as tasks whose execution is 150% of the median duration of all tasks within the same job and median task duration is used instead of mean task duration. This is especially because median job execution duration is less affected by extreme execution times caused by stragglers.

The analysis performed on the trace data from Microsoft Bing's production cluster shows that 80% of stragglers have a uniform probability of delay between $150-250\%$ compared to the median task duration, with 10% exhibiting a delay 1000% greater than median task duration [31].

Cloud data-flow deals with stragglers problem using dynamic work rebalancing. However, this method has its own technical challenges which includes data consistency, intense systematic testing, as well as a careful design, and predicting how a task will progress over time [32]. In addition, this approach is only effective in systems with high-throughput, low-latency task schedulers and efficient data materialization [17].

Blocked time analysis is used to measure how long each task spends blocked on a given resource. These approach provides a per-task measurements and allows the understanding of straggler root-causes by correlating slow tasks with long blocked times [33]. However, the work by [33] does not look at a vast range of workloads nor a wide range of cluster sizes.

Intelligent scheduling [34] is another approach to remove stragglers. Several components of task scheduling affect the scalability. The first is knowledge of a worker's past task execution times. This allows to schedule tasks preferentially to the most performant machines. The second involves task replication to remove the effect of straggling workers. By monitoring the state of the work-

ers either busy or idle (w_i) and the number of tasks waiting to be assigned a worker, task replication can be engaged when $w_i > 0$.

All of these previous studies are trying to mitigate stragglers through a wide variety of approaches. While a comprehensive solution will involve tricks to avoid causing stragglers. This work focuses on the stragglers root-causes that prevent us from achieving near ideal scaling in *MDAnalysis* beyond a single compute node. In the present study, we are trying to find the root-cause of the straggler tasks and solutions through which we can improve performance and scaling.

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, with time critical code in C/Cython. *MDAnalysis* supports many common file formats of simulation packages including CHARMM, Gromacs, Amber, and NAMD and the Protein Data Bank format and enables accessing data stored in trajectories. *MDAnalysis* utilizes fast numeric/algebraic libraries such as ATLAS, LAPACK, or MKL to improve speed and powerful Python libraries such as NumPy and SciPy.

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 [35, 36] 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 $\mathcal{O}(T \times N^2)$ [37] where T is the number of frames in the trajectory and N the number of particles in a frame.

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** [38] (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 $\mathcal{O}(T \times N)$. ***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

3.2. 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 [12, 13]. 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, add to its attractiveness and clarifies why parallel programming models, such as MPI or MapReduce, should be supported. Based on the efficiency tests [12, 13], 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 [14]. 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 [12].

3.3. Applications of Global Array

Global array toolkit provides users with a language interface that allows them to distribute data while maintaining the type of global index space and programming syntax similar to what is available when programming on a single processor [39].

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 [14]. 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 [14].

When the global array is created (*ga_create*) each process will create an array of the same shape and size, physically located in the local memory space of that process [39]. 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 process [40]. The requesting process then makes individual requests to each of the remote processes. The requests are implemented using the native one-sided semantics inside the the infini-band Verbs library. OpenIB/infini-band uses SHMEM (Symmetric Hierarchical MEMory) parallel computing library [41, 42] within the node and will use the infini-band 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.

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
```

3.4. 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. In parallel HDF5, all file accesses are coordinated though the MPI library;

otherwise, multiple processes would compete over accessing over the same file on disk. HDF5 itself handles nearly all the details involved with coordinating file access when the shared file is opened through "mpio" driver. In addition, MPI communicator should be supplied as well and the users also need to follow some constraints for data consistency [15].

3.4.1. 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 [15]. 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 [15]. Other tasks and any type of data operations can be performed independently by processes. In the present study, we use independent operations.

4. Benchmark Environment

The test system contained the protein adenylate kinase with 214 amino acid residues and 3341 atoms in total [43]. The trajectory [44] was in Gromacs XTC format trajectory ("600x" in Khoshlessan et al. [9]) 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 of Nodes	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	56 Gbps IB	PBS

Table 1: List of benchmarked clusters and their system configuration

4.1. Installing libraries on multi-user HPC systems

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.

There are always a lot of dependencies for highly optimized libraries ***mahzad: Giaanis, Can you give references here or correct my sentences or add any if you think could be helpful to make it look better? . However, Lack of precise documentation can contribute to the challenge of getting the libraries to work.

In addition, many domain specific packages are not available through package manager in super-computers 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.

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

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. *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 not 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. 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_{comp} 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_{comp} 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 $t_N \equiv t_{RMSD} + t_{\mathrm{comm}}$.

5.2. 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 and efficiency (using MPI).

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

$$E = \frac{S}{N} \tag{2}$$

Additionally, we introduce two important performance parameters that determines whether we observe stragglers. We define ratio of compute time to I/O time. In the present study, we calculated this ratio using the serial version of our algorithms. However, one can run the algorithm using several cores and several frames of the trajectory to get an estimation of the compute to I/O ratio. This is because, ideally, compute time per frame and I/O time per frame should be the same for different runs using different processes and we have shown this on our previous study [9].

$$\frac{\bar{t}_{comp}^{frame}}{\bar{t}_{IO}^{frame}} \approx \frac{t_{\mathrm{comp}}}{t_{\mathrm{I/O}}} \tag{3}$$

and the ratio of compute to communication time:

$$\frac{\sum_1^{N_{frames}} t_{comp}^{frame}}{\bar{t}_{communication}} \approx \frac{t_{\mathrm{comp}}}{t_{\mathrm{comm}}} \tag{4}$$

Item	Definition
N_{frames}	N_{frames}^{total}/N
t_{end_loop}	$t_{L6} + t_{L7}$
$t_{opening\_trajectory}$	$t_{L2} + t_{L3}$
t_{comp}	$\sum_1^{N_{frames}} t_{comp}^{frame}$
$t_{I/O}$	$\sum_1^{N_{frames}} t_{I/O}^{frame}$
t_{all_frame}	$t_{L4} + t_{L5} + t_{L6}$
t_{RMSD}	$t_{L1} + ... + t_{L8}$
$t_{Communication_{MPI}}$	t_{L16}
$t_{Communication_{GA}}$	$t_{L5} + t_{L6} + t_{L7} + t_{L8}$
$t_{Overhead1}$	$t_{all\_frame} - t_{I/O\_final} - t_{comp\_final} - t_{end\_loop}$
$t_{Overhead2}$	$t_{RMSD} - t_{all\_frame} - t_{opening\_trajectory}$
t_N	$t_{RMSD} + t_{Communication}$
$\bar{t}_{communication}$	$\frac{\sum_1^N t_{communication}}{N}$
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. All the timings on the first row are per rank.

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 $t_{\mathrm{comp}}/t_{\mathrm{I/O}} \approx 0.3$). We showed previously that

the RMSD task only scaled well up to 24 cores, on a single compute node on *Comet*, *Stampede*, and *ASU computing resources*, using either Dask or MPI [9]. Here we focus on the MPI implementation (via *mpi4py* [12, 13]), in order to better understand the cause for the poor scaling across nodes.

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 1a, 1b for MPI.

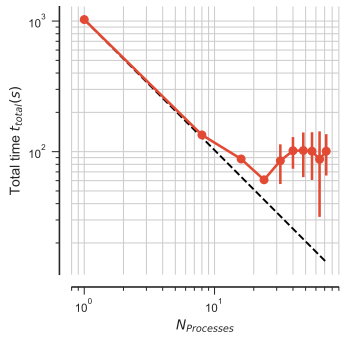
In the example run in Figure 1d, ten ranks out of 72 take almost 65 s whereas the remaining ranks only take about 40 s. The detailed breakdown of the time spent on each rank (Figure 1d) shows that time for the actual computation, t_{comp}, is fairly constant across ranks. The time spent on reading data from the shared trajectory file on the Lustre file system into memory, $t_{\mathrm{I/O}}$, 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 $t_{\mathrm{comm}} < 10$ s or is barely measurable. This initial analysis (especially Figure 1d) indicates that communication is a major issue.

Identification of Scalability Bottleneck

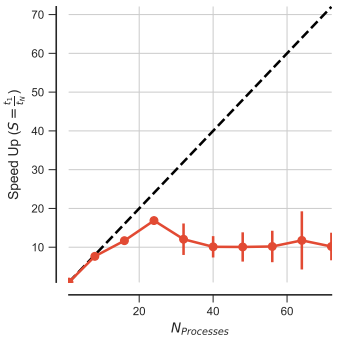
Figure 1c shows the scaling of t_{comp} and $t_{\mathrm{I/O}}$ individulally. As shown, t_{comp} scales very well; however, $t_{\mathrm{I/O}}$ does not show good scaling beyond a single node (24 cores) and that explains why we are seeing these variations in $t_{\mathrm{I/O}}$ across different ranks (Figure 1d). Considering the results in Figures 1 and 1c, we can conclude that communication and I/O are the root causes for stragglers.

Hardware

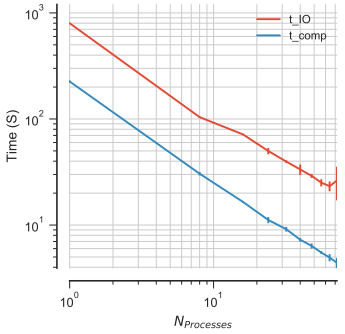
***giannis: Are the data in figure 2 from Comet and Stampede? Why Stampede and not SuperMic, Bridges, and Comet? Those are the system you say the experimentation was done. ***mahzad: the data is on Comet. We perform everything on Comet as a baseline and then make a comparison across



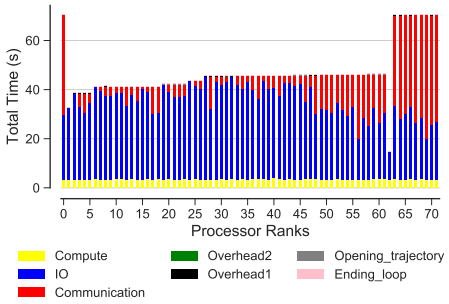
(a) Scaling total



(b) Speed-up



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



(d) Time comparison on different parts of the calculations per MPI rank

Figure 1: Performance of the RMSD task with MPI which is I/O-bound $t_{\mathrm{comp}}/t_{\mathrm{I/O}} \approx 0.3$ on SDSC Comet. 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. (a-c) The error bars show standard deviation with respect to mean. (d) Compute t_{comp}, 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. 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.

different clusters on the final section. We do not have any data from Stampede here ***giannis: You say that you observed stragglers on Comet, Stampede and an ASU cluster. You do not show any data from Stampede, so I suggest to remove it. I think there are some data with stragglers from Bridges or SuperMic. Including those will make your point stronger. ***mahzad: but I cited scipy

paper. all the data are shown in scipy paper. Why do we need to repeat it? We can just cite it. ***giannis: Sorry, Mahzad, but we cannot cite them. They were presented and discussed. Unless they have to offer something new, a different type of analysis than the SciPy paper, they should not be included apart from showing the motivation of this paper. That is my opinion and that is what I am doing in all my papers ***mahzad: I do not agree. Here we are saying that we are observing stragglers on different resources including the ones used in the present study and the ones used for scipy. Then we want to discuss ways for overcoming those stragglers. We did not discern any specific patterns that could be traced to the underlying hardware. Stragglers were observed on *SDSC Comet*, *TACC Stampede* and *ASU local computing resources* [9]. We also show in the present study that stragglers are observed on other XSEDE resources such as PSS Bridges and SuperMIC as well. 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 resources.

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 $t_{\mathrm{comp}}/t_{\mathrm{I/O}} \approx 100$), namely dihedral featurization on *Comet* with Infiniband and Lustre file system.

The system scales linearly and close to ideal (Figure 2a, 2b, 2c). Although, there is communication of large result arrays (e.g. $\approx 2GB$ with 8 processes and $\approx 0.24GB$ with 72 processes), which is costly for multiple ranks (Figure 3), the speed-up curve (Eq. 1) in Figure 2b 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 3). Based on Figure 3, 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 [12] 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.

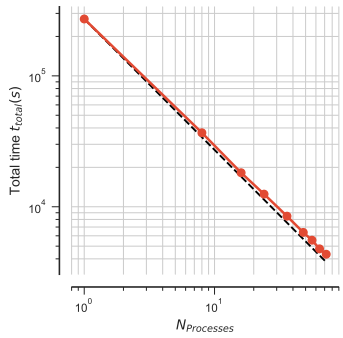
Overall, increasing the computational workload over I/O improves scaling. For large compute-bound workloads such as the dihedral featurization task, stragglers are eliminated and nearly ideal strong scaling is achieved. The fact that linear scaling is possible, even with expensive communications, makes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

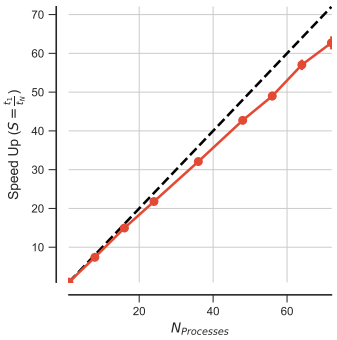
The RMSD task turned out to be I/O bound, i.e.,

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

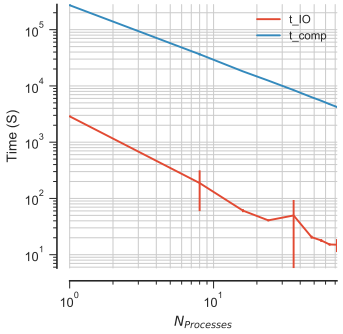
and we were not able to achieve good scaling above a single node. However, Dihedral featurization turned out to be compute bound and we were able to achieve near ideal scaling. We therefore, hypothesized that decreasing the relative I/O load with respect to compute load would also reduce the stragglers.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



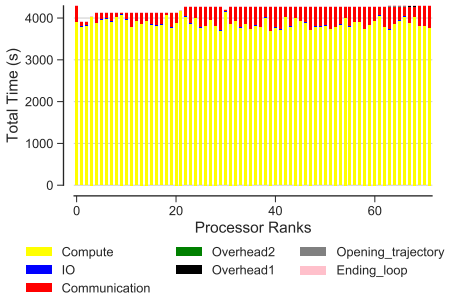
(a) Scaling total



(b) Speed-up



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

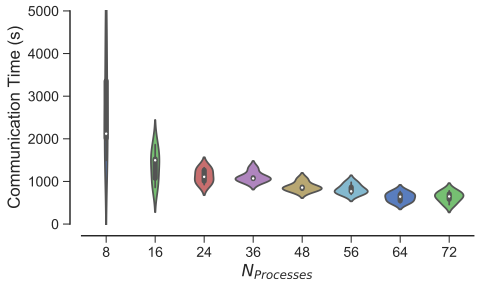


(d) Time comparison on different parts of the calculations per MPI rank

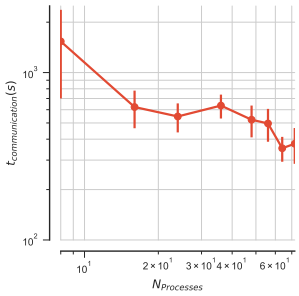
Figure 2: Performance for the **dihedral featurization** workload, which is compute-bound $t_{\mathrm{comp}}/t_{\mathrm{I/O}} \approx 100$ on SDSC Comet. Data are read from the file system (I/O included) and results are communicated back to rank 0 (communications included). Five repeats are performed to collect statistics. (a-c) The error bars show standard deviation with respect to mean. (d) Compute t_{comp}, 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. No straggler is observed.

$$\frac{t_{\mathrm{comp}}}{t_{\mathrm{I/O}}} \gg 1.$$

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



(a) Communication time distribution for different number of processes



(b) Communication time scaling

Figure 3: Comparison of communication cost for different number of processes over 5 repeats for the **dihedral featurization** workload performed on SDSC Comet (a) Communication time distribution shown as violin plots [45] with kernel density estimates as implemented in *seaborn* for different number of processes; the white dot indicates the median. (b) Scaling communication time (mean and standard deviation). ***giannis: I am really confused with this figure. How many GBs of data did you move and there was a delay of 1.5 hours? Was it more 10TBs? (Assuming you were able to use 20Gbps from IB $20Gbps * 5000sec = 10^{14} = 12500000000000B = 12500000000000/2^{40}TB \simeq 11.6TB$) ***mahzad: with 8 processes size of the data to be communicated is 2 GB per process. But communication time is not uniform across different processes and for some processes it is in the order of 1000 for another processes is 0.86 and the other is 4 s. Thus there are stragglers due to communication but because communication is small with respect to compute it does not affect the performance. ***giannis: This is very weird. Are you sure that the IB network was used correctly? Over 1k seconds for 2GB is way too much. I am afraid that the IB network was not used correctly, either from the implementation of RMSD or from OpenMPI's custom installation. I will also ask AndreM, who has worked a bit with OpenMPI. Either way, it does not make any sense to me. ***mahzad: I do not think there is any problem with IB usage. I ran dihedral featurzation at two different time (one several months after the other) and every time I did five repeats. Both gave me the same pattern and results. ***giannis: Then something is very wrong with the setup of our experiments. 2GBs on a network of 56 Gbps should take a few seconds or worst case scenario minutes to transfer not hours. Are you sure these are not milliseconds? ***mahzad: Please refer to the email discussion for my answer to this

6.3.1. Increased workload (RMSD)

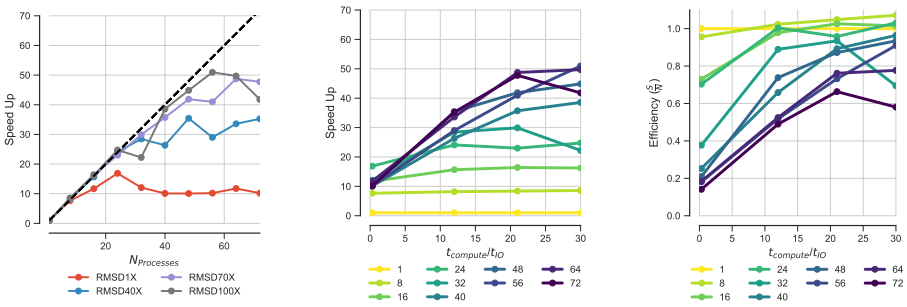
The RMSD workload was artificially increased forrty-fold ("40×"), seventy-fold ("70×"), and hundred-fold ("100×") and we measured performance as before. These workloads correspond to $t_{\text{comp}}/t_{\text{I/O}}$ ratio of 12, 21, 30 respectively as shown in Table 4. We performed this experiment to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

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

Table 4: Change in $t_{\mathrm{comp}}/t_{\mathrm{I/O}}$ 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.

As the $t_{\mathrm{comp}}/t_{\mathrm{I/O}}$ ratio increases, speed-up and performance improves and show overall better scaling than the I/O-bound workload, i.e. 1× RMSD (Figure 4a). When $t_{\mathrm{comp}}/t_{\mathrm{I/O}}$ ratio increases, the RMSD calculation consistently scales up to larger numbers of cores ($N = 56$ for 70× RMSD). Figures 4b and 4c shows the improvement in performance more clearly. In fact, as the $t_{\mathrm{comp}}/t_{\mathrm{I/O}}$ ratio increases, the values of speed-up and efficiency get closer to their ideal value for each number of processor count.



(a) Speed-Up (b) Speed-Up (c) Efficiency

Figure 4: Effect of $t_{\mathrm{comp}}/t_{\mathrm{I/O}}$ ratio on performance of the RMSD task with MPI performed on SDSC Comet. We tested performance for $t_{\mathrm{comp}}/t_{\mathrm{I/O}}$ ratios of 0.3, 12, 21, 30 which correspond to 1× RMSD, 40× RMSD, 70× RMSD, and 100× RMSD respectively (communication is included). (a) Effect of $t_{\mathrm{comp}}/t_{\mathrm{I/O}}$ on the Speed-Up (b) Change in the Speed-Up with respect to $t_{\mathrm{comp}}/t_{\mathrm{I/O}}$ for different processor counts (c) Change in the efficiency with respect to $t_{\mathrm{comp}}/t_{\mathrm{I/O}}$ for different processor counts

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 irregular scaling due to the fluctuations.

Given the results for Dihedral featurization and RMSD algorithms (Algorithms 2, and 1) and $X\times$ RMSD (Figure 4) 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 $t_{\mathrm{comp}}/t_{\mathrm{I/O}}$ 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

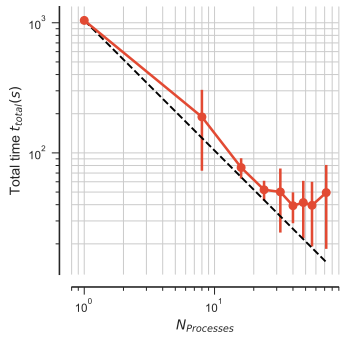
As discussed in the previous sections, Figure 1d, for small $t_{\mathrm{comp}}/t_{\mathrm{I/O}}$ communication acts as the scalability bottleneck. In fact, when the processes communicate result arrays back to the master process (rank 0), some processes take much longer as compared to other processes. 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 5b) and total time scaling (Figure 5a) global array improves strong scaling performance. Although communication time has significantly decreased using global array (compare Figure 5d to Figure 1d), 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 5c). 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 communica-

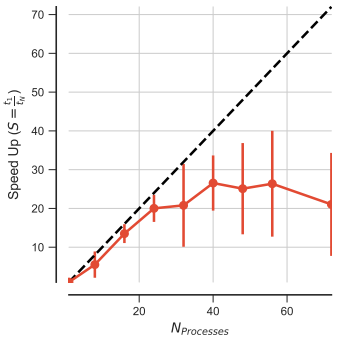
tions were performed using global arrays, scaling is still far from ideal as a result of slow processes due to I/O variation and the delay in opening the trajectory. ***oliver: In Fig 8c it is the trajectory opening step that creates stragglers. This was not a problem before. Is this now ALWAYS the problem 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.

6.5. I/O Cost

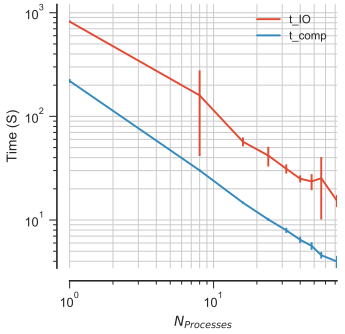
We showed previously that the I/O system can have a large effect on the parallel performance of the RMSD task [9], especially because the average time to perform the computation t_{comp} (about 0.09 ms) is about three times smaller than the I/O time $t_{\text{I/O}}$ (about 0.3 ms) (Figures 1c and 1). 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 1c and 1). 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 $t_{\text{comp}}/t_{\text{I/O}}$ 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.



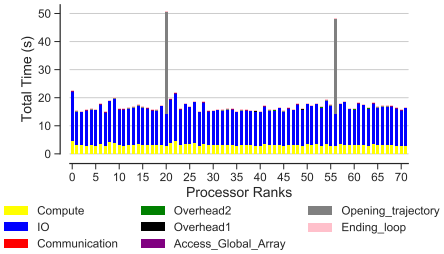
(a) Scaling total



(b) Speed-up



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



(d) Time comparison on different parts of the calculations per MPI rank

Figure 5: Performance of the RMSD task with MPI using global array ($t_{\text{comp}}/t_{\text{I/O}} \approx 0.3$) on SDSC Comet. Data are read from the file system (I/O included). All ranks update the global array (ga_{put}) and rank 0 accesses the whole RMSD array through the global memory address (ga_{get}) (communications included). Five repeats are performed to collect statistics. (a-c) The error bars show standard deviation with respect to mean. (d) Compute t_{comp}, IO $t_{\text{I/O}}$, communication t_{comm}, access to the whole global array by rank 0 $t_{Access\_Global\_Array}$, 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 repeatsMPI ranks 20 and 56 are stragglers, i.e., their total time far exceeds the mean of the majority of ranks.

6.5.1. Splitting the Trajectories

In all the previous benchmarks all processes were using a shared trajectory file. In order to test whether *I/O and communication compete over the network resources with small* $t_{comp}/t_{I/O}$ *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

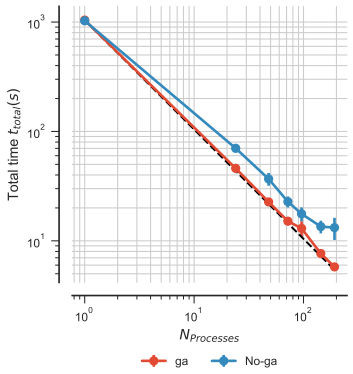
have N_b frames in it. Through this approach, each process will have access to its own segment and there will be no competition over file accesses. For reference, the necessary time for splitting the trajectory file is given in Appendix A.

Performance without Global Array

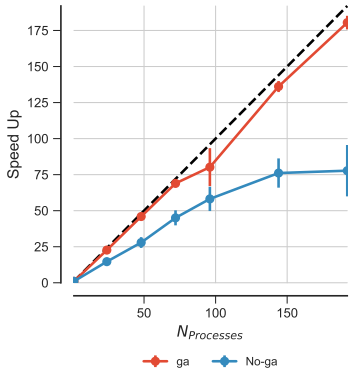
We ran a benchmark up to 8 nodes (192 cores) and, we observed rather better scaling behavior with efficiencies above 0.6 (Figure 6a and 6b) and the delay time for stragglers has also reduced from 65s to about 10s for 72 processes. However, the scaling is still far from ideal due to the communication. Although the delay due to communication is much smaller as compared to RMSD with a shared trajectory file (Compare Figure 6c to Figure 1d), it is still delaying several processes and as a result leads to longer job completion time (Figure 6c). These delayed tasks impact performance as well and hence the speed-up is not still close to ideal scaling (Figure 6b).

Performance using Global Array

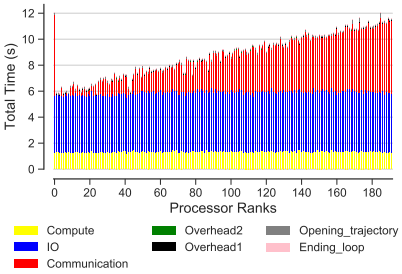
***giannis: This is very confusing. In a previous paragraph you say that GA passes data over the IB network. Here you say there is no congestion? Don't they, MPI communication, MPI I/O, and GA communication, use the same network? Also, I do not think you have the data to support that clearly, unless you know how many Bytes are moved around, how many packets and their type. If that is the case include it. Or are you using the Local filesystem on the nodes and IO has nothing to do with the IB? ***mahzad: I think there is something different in the way GA use IB that when we use GA there is no contention. That is mainly because there is no synchronization and that reduces the traffic a lot. GA uses IB but the way it uses is very different from MPI Gather. it defines a pointer and that is very different from what happens in MPI for communication. ***giannis: So there are no similarities between GA communication and asynchronous MPI communication? Asynchronous MPI communication does not block anything and there is no need for synchronization when done correctly. It is still confusing for me at least. ***mahzad: No, there



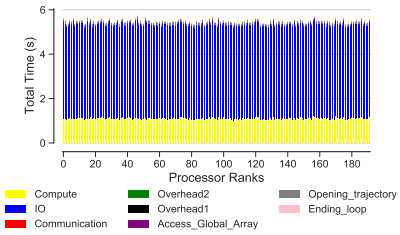
(a) Scaling total



(b) Speed-up



(c) Time comparison on different parts of the calculations per MPI rank without global array



(d) Time comparison on different parts of the calculations per MPI rank using global array

Figure 6: Comparison on the performance of the RMSD task with MPI when the trajectories are splitted using global array and without global array ($t_{\text{comp}}/t_{\text{I/O}} \approx 0.3$) on SDSC Comet. 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}). Five repeats are performed to collect statistics. (a-b) The error bars show standard deviation with respect to mean. (c-d) Compute t_{comp}, IO $t_{\text{I/O}}$, communication t_{comm}, access to the whole global array by rank 0 $t_{Access\_Global\_Array}$, 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). When global array is not used, the performance is affected due to the non-uniform communication time across different ranks. However, when global array is used communication time has significantly reduced and scaling is close to ideal.

***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

are no similarities. MPI.Gather is collective communication and collective communications are blocking in MPI, right? ***giannis: For synchronous, yes you are right. I am asking about asynchronous MPI communication. ***mahzad: Maybe using Isend and Ireceive can be helpful but I do not think it will be com-

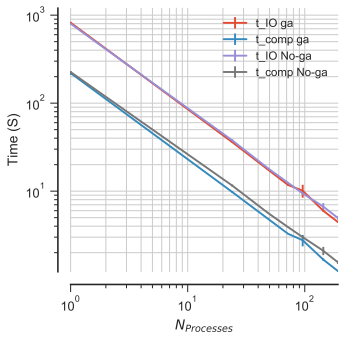


Figure 7: Comparison on the scaling of the t_{comp} and $t_{\text{I/O}}$ of the RMSD task when the trajectories are splitted with global array and without global array performed on SDSC Comet.

parable with GA. DO you volunteer to give it a shot and replace mpi.Gather with asynchronous communication and run it so that we can compare with GA?

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 6a and 6b) with no straggler tasks (Figure 6d). 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 $t_{\text{comp}}/t_{\text{I/O}} \ll 1$ (i.e. with a I/O bound workload).

6.6. Effect of t_{comp}/t_{comm} on Performance

In addition to the compute to I/O ratio we define another important performance parameter which is compute to communication ratio. Based on the results shown in the section **??**; although we overcame the I/O effect by splitting the trajectory, scaling is still far from ideal without global array. This means that the task remains communication bound (Figure 8b). Examples of the communication bound task is RMSD task with splitting the trajectories and without global array. When the task is communication bound, i.e.,

$$\frac{\bar{t}_{comp}}{\bar{t}_{comm}} \ll 1.$$

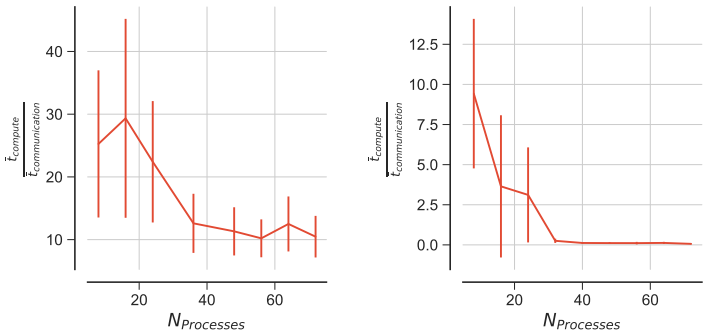
We are not able to achieve near ideal scaling even when I/O is not a bottleneck anymore (Figure 6). However, Dihedral featurization turned out to be compute bound and we were able to achieve near ideal scaling even when we did not use global array in our benchmark (Figure 2). This is mainly because both communication and I/O are very small with respect to compute time ($t_{\text{comp}}/t_{\text{I/O}} \approx 100$ and Figure 8a). Although, there are stragglers due to communication (Figure 9) their effect on performance is not crucial mainly because compute to communication ratio is very large. We therefore, conclude that decreasing the relative communication load with respect to compute load would also reduce the stragglers. If the computational load is more than communication load then the work became compute bound, i.e.,

$$\frac{\bar{t}_{comp}}{\bar{t}_{comm}} \gg 1.$$

It should be noted that size of the RMSD arrays to be communicated to rank 0 is much smaller than the resulted arrays in Dihedral featurization task (0.7 MB versus 2GB with 8 processes). As a result communication time is very small within a single node (24 cores) for RMSD task and that is why $t_{\text{comp}}/t_{\text{comm}} \gg 1$. However for more than a single node, communication overhead becomes a bottleneck and $t_{\text{comp}}/t_{\text{comm}} \ll 1$ (Figure 8b).

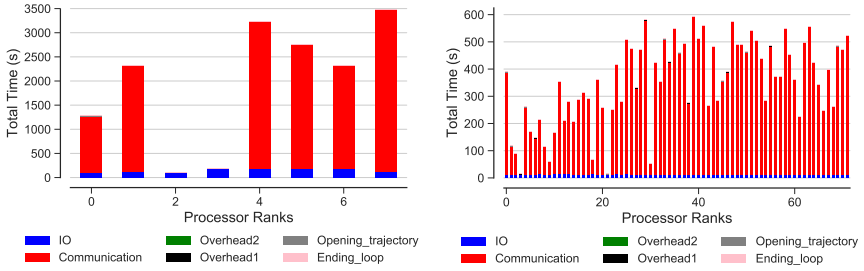
6.6.1. Chain-Reader

***giannis: Never heard about a chain reader, can you add a reference? Thank you! ***mahzad: It is the chain-reader method in mdanalysis https://www.mdanalysis.org/docs/documentation_pages/coordinates/chain.html



(a) Dihedral Featurization (b) RMSD

Figure 8: Average compute to communication ratio with number of processes for RMSD and Dihedral featurization tasks performed on SDSC Comet. Five repeats are performed to collect statistics and error bars show standard deviation with respect to mean.



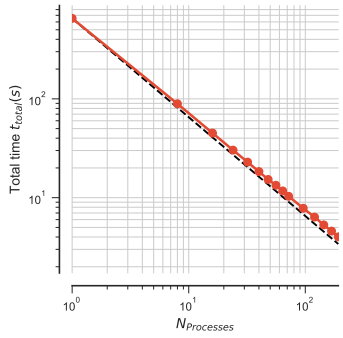
(a) 8 Processes (b) 72 Processes

Figure 9: Examples of timing per MPI rank for Dihedral featurization task ($t_{\mathrm{comp}}/t_{\mathrm{I/O}} \approx 100$) on SDSC Comet without showing the compute portion with 8 and 72 processes. I/O $t_{\mathrm{I/O}}$, 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 task is compute bound and the compute portion is not shown here so that we can observe the variation in communication time across different ranks. In fact, there are stragglers due to communication and communication time is not uniform across different ranks. However, the performance is not affected due to stragglers caused by communication because the task is compute bound. Five repeats are performed to collect statistics and this is typical data from one run of the 5 repeats.

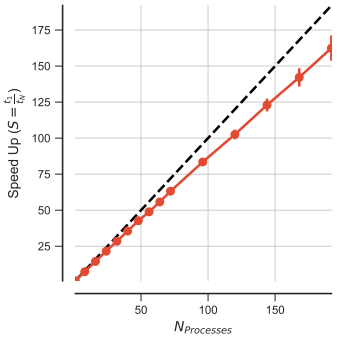
6.6.2. MPI-based Parallel HDF5

Another approach we examined to improve I/O scaling is MPI-based Parallel HDF5. We converted our XTC trajectory file into HDF5 format so that we can test the performance of parallel IO with HDF5 file format. The time it took to convert our XTC file with $2,512,200$ frames into HDF5 format was about

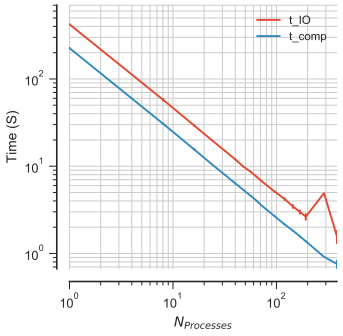
5400 s in our local resources with network file system (NFS). Again, we ran our benchmark up to 8 nodes (192 cores) and, we observed near ideal scaling behavior with efficiencies above 0.8 (Figure 10a and 10b) with no straggler tasks (Figure 10d). When we split our trajectory, scaling is better as compared to that of parallel I/O (Compare Figure 10b to Figure 6b). However, both methods scale very well up to 8 nodes and have comparable performance.



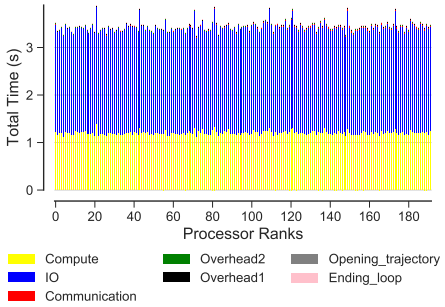
(a) Scaling total



(b) Speed-up



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



(d) Time comparison on different parts of the calculations per MPI rank

Figure 10: Performance of the RMSD task with MPI-based parallel HDF5 ($t_{\mathrm{comp}}/t_{\mathrm{I/O}} \approx 0.3$) on SDSC Comet. Data are read from the file system from a shared HDF5 file format instead of XTC format (Independent I/O) and results are communicated back to rank 0 (communications included). Five repeats are performed to collect statistics. (a-c) The error bars show standard deviation with respect to mean. (d) Compute t_{comp}, 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. No straggler is observed.

7. Performance Comparison of the RMSD task on different clusters

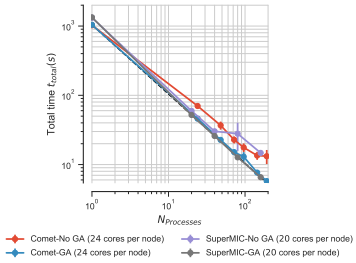
In this section we try to compare the performance of RMSD task on different HPC resources to examine the robustness of the methods we used for our performance study. HPC resources used for this purpose and their system configuration are given in Table 1. We perform these comparisons for several cases 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

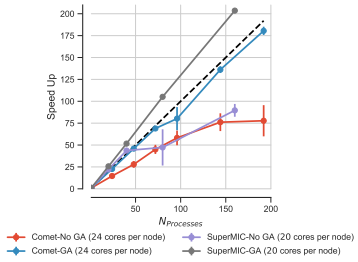
Figure 11 shows how RMSD task scales with the increase in the number of cores on different HPC resources. When we split the trajectories the scaling follows the same pattern on both Comet and SuperMIC with global array and without global array. Both Comet and SuperMIC scales very well using Global array. 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 **??** and Figure 6c). Overal, the scaling of the RMSD task is better on SuperMIC and the performance gap increases as we increase the number of cores. In global arrays, primary mechanisms provided by GA for accessing data are block copy operations that transfer data between layers of memory hierarchy, namely global memory of the node (distributed array) and local memory. Each process is able to directly access data held in a section of a Global Array that is locally assigned to that process [39]. This is an advantage over MPI collective communication where direct memory access is not allowed. In message passing, CPU is stopped in order to access data from the local memory; while in GA CPU would continue doing its computations and would not stop to accommodate any communication between nodes.

7.2. MPI-based Parallel HDF5

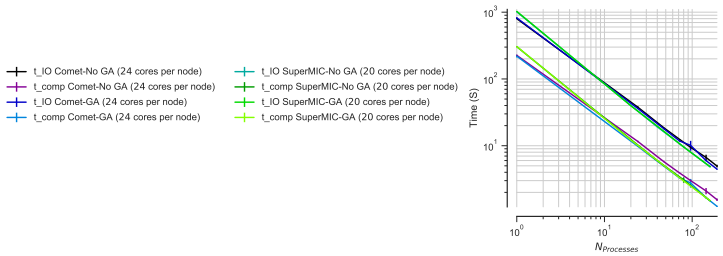
Figure 12 shows how RMSD task scales with the increase in the number of cores on different HPC resources using MPI-based parallel HDF5. The scaling



(a) Scaling total



(b) Speed-up



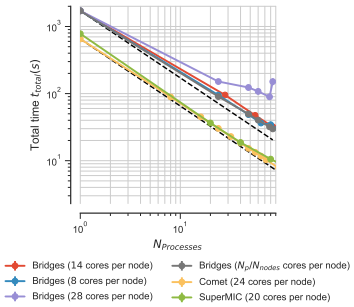
(c) Scaling of the t_{comp} and $t_{\mathrm{I/O}}$ of the RMSD task with MPI when the trajectories are splitted using global array and without global array.

Figure 11: Comparison of the performance of the RMSD task with MPI ($t_{\mathrm{comp}}/t_{\mathrm{I/O}} \approx 0.3$) when the trajectories are splitted using global array and without global array across different clusters (SDSC Comet, SuperMIC). 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}). Five repeats are performed to collect statistics. The error bars show standard deviation with respect to mean.

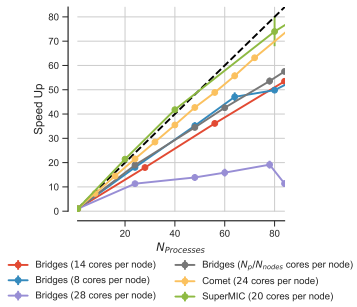
follows the same pattern on both Comet and SuperMIC. Both Comet and SuperMIC scales very well. Overal, the scaling of the RMSD task is better on SuperMIC. Bridge performance is different from Comet and SuperMIC. Bridge has 28 cores per compute node and when we use all cores for our calculations there 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 12. With less than 28 cores used, each core has access to proportionally more cache space and memory bandwidth, so the higher performance is obtained.

Based on Figure 13 the I/O time distribution is pretty small and uniform across all ranks on Comet and SuperMIC (Figures 13b & 10d). However, on

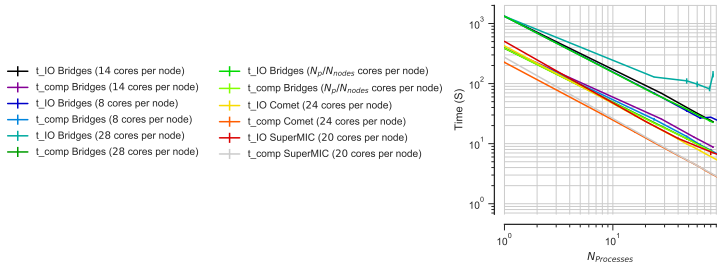
Bridges the I/O time is on average about two and a half times larger and the I/O time distribution is also erratic across different ranks (Figure 13a).



(a) Scaling total



(b) Speed-up

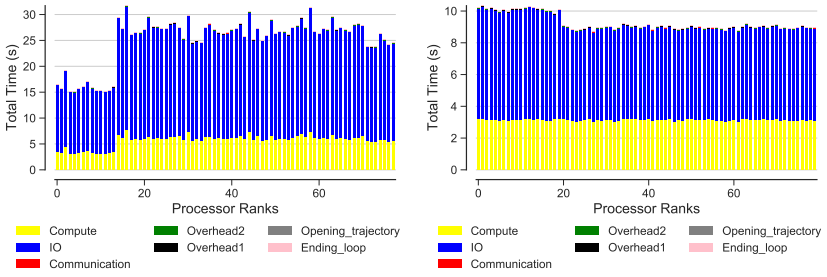


(c) Scaling of the t_{comp} and $t_{\mathrm{I/O}}$ of the RMSD task with MPI-based parallel HDF5 (Independent I/O)

Figure 12: Comparison of the performance of the RMSD task with MPI ($t_{\mathrm{comp}}/t_{\mathrm{I/O}} \approx 0.3$) across different clusters (SDSC Comet, PSC Bridge, SuperMIC). Data are read from a shared HDF5 file format 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.

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



(a) Bridge (b) SuperMIC

Figure 13: Examples of timing per MPI rank for RMSD task with MPI-based parallel HDF5 ($t_{\text{comp}}/t_{\text{I/O}} \approx 0.3$) on (a) PSC Bridge and (b) SuperMIC (Buttomn). Five repeats are performed to collect statistics and this is typical data from one run of the 5 repeats. Compute t_{comp}, 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. I/O distribution is non-uniform on Bridge while it is more uniform on SuperMIC and this explains why SuperMIC has better scaling as compared to PSC Bridge.

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.

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

***mahzad: I believe adding a graph showing the workflow here can be much much better. I have some ideas, working on this .... Here we provide practical guidelines for parallel analysis of the three dimensional time series (MD trajectories) on HPC resources.

Heuristic 1 Calculate the value of $t_{\text{comp}}/t_{\text{I/O}}$ to see whether the task is compute bound ($\frac{t_{\text{comp}}}{t_{\text{I/O}}} \gg 1$) or IO bound ($\frac{t_{\text{comp}}}{t_{\text{I/O}}} \ll 1$). 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.

Heuristic 2 For $\frac{t_{\text{comp}}}{t_{\text{I/O}}} \geqslant 100$ 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 [12]. 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).

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

Heuristic 3.1 If there is access to HDF5 format the recommended way will be to use MPI-based Parallel HDF5 (Section 6.6.2). Since converting the XTC file to HDF5 is expensive if the trajectory file formats are not in HDF5 form then one might prefer to split the trajectories. 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.

Heuristic 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).

Heuristic 3.3 The appropriate parallel implementation along with *Global Array* should be used on the trajectory segments (Section **??**) 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 ($t_{\mathrm{comp}}/t_{\mathrm{I/O}} \approx 100$), close to ideal scaling was achievable (Figure 3). However the I/O bound workload ($t_{\mathrm{comp}}/t_{\mathrm{I/O}} \approx 0.3$) 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*. Since I/O traffic must compete with MPI messages and other traffic it can be difficult to achieve peak single-node I/O rates for an arbitrary file when the system is fully loaded with other jobs [46].

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 1d).

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 5) 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 6 and 10). 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 6). ***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. Ribeiro JV, Bernardi RC, Rudack T, Stone JE, Phillips JC, Freddolino PL, et al. Qwikmd-integrative molecular dynamics toolkit for novices and experts. Scientific Reports 2016;.
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. 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.
5. 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.
6. McGibbon RT, Beauchamp KA, Harrigan MP, Klein C, Swails JM, Hernández CX, et al. Mdtraj: A modern open library for the analysis of molecular dynamics trajectories. Biophysical Journal 2015;109(8):1528 –32. URL: //www.sciencedirect.com/science/article/pii/S0006349515008267. doi:http://dx.doi.org/10.1016/j.bpj.2015.08.015.
7. Mura C, McCrimmon CM, Vertrees J, Sawaya MR. An introduction to biomolecular graphics. PLoS Comp Biol 2010;6(8):e1000918. doi:10.1371/journal.pcbi.1000918.
8. 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.

9. 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.
10. Paraskevakos I, Luckow A, Khoshlessan M, Chantzialexiou G, Cheatham TE, Beckstein O, et al. Task-parallel analysis of molecular dynamics trajectories. In Proceedings of 47th International Conference on Parallel Processing; University of Oregon, Eugene, Oregon, USA: ICPP; 2018,.
11. 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.
12. 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.
13. 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.
14. DAILY JA. Gain: Distributed array computation with python. Master's thesis; School of Electrical Engineering and Computer Science, WASHINGTON STATE UNIVERSITY; 2009.
15. Collette A. Python and HDF5. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.; 2014.
16. Garraghan P OXYRea. Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters. IEEE Transactions on Services Computing 2016;.
17. Fine-Grained Micro-Tasks for MapReduce Skew-Handling. 2012.
18. Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters. In: OSDI'04 Sixth Symposium on Operating System Design and Implementation. 2004,.
19. Qi Chen CL, Xiao Z. Improving mapreduce performance using smart speculative execution strategy. In: IEEE TRANSACTIONS ON COMPUTERS. 2014,.
20. Bhandare A, George J, Deshpande S, Karle Y. Review and analysis of straggler handling techniques. International Journal of Computer Science and Information Technologies 2016;.
21. Kwon Y, Balazinska M, Howe B, Rolia J. Skewtune: Mitigating skew in mapreduce applications. In: SIGMOD'12. 2012,.
22. Ballani H, Costa P, Karagiannis T, Row-stron A. Towards predictable datacenter networks. In: SIGCOMM. 2011,.
23. Ananthanarayanan G, Hung MCC, Ren X, I. Stoica AW, Yu. M. Grass: Trimming stragglers in approximation analytics. In: In Proc. NSDI. 2014,.
24. Jeyakumar V, Alizadeh M, Mazieres D, Prab-hakar B, Kim C, Greenberg. A. Eyeq:

Practical network performance isolation at the edge. In: NSDI. 2013,.

25. Li H, Ghodsi A, Zaharia M, Shenker S, Stoica I. Reliable, memory speed storage for cluster computing frameworks. In: SoCC. 2014,.
26. Zaharia M, Chowdhury M, Das T, Dave A, J. Ma MM, Franklin MJ, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: NSDI. 2012,.
27. Kyong J, Jeon J, Lim SS. Improving scalability of apache spark-based scale-up server through docker container-based partitioning. In: Proceedings of the 6th International Conference on Software and Computer Applications - ICSCA '17. New York, New York, USA: ACM Press. ISBN 9781450348577; 2017, p. 176–80. URL: http://dl.acm.org/citation.cfm?doid=3056662.3056686. doi:10.1145/3056662.3056686.
28. Ousterhout K. Architecting for Performance Clarity in Data Analytics Frameworks 2017;URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-158.html.
29. Gittens A, Devarakonda A, Racah E, Ringenburg M, Gerhardt L, Kottalam J, et al. Matrix factorizations at scale: A comparison of scientific data analytics in spark and C+MPI using three case studies. In: 2016 IEEE International Conference on Big Data (Big Data). IEEE. ISBN 978-1-4673-9005-7; 2016, p. 204–13. URL: http://ieeexplore.ieee.org/document/7840606/. doi:10.1109/BigData.2016.7840606.
30. Yang H, Liu X, Chen S, Lei Z, Du H, Zhu C. Improving Spark performance with MPTE in heterogeneous environments. In: 2016 International Conference on Audio, Language and Image Processing (ICALIP). IEEE. ISBN 978-1-5090-0654-0; 2016, p. 28–33. URL: http://ieeexplore.ieee.org/document/7846627/. doi:10.1109/ICALIP.2016.7846627.
31. Ananthanarayanan G, Kandula S, Greenberg A, Stoica I, Lu Y, Saha B, et al. Reining in the outliers in map-reduce clusters using mantri. In: 9th USENIX Symposium on Operating Systems Design and Implementation. 2010,.
32. Schmidt E, DeMichillie G, Perry F, Akidau T, Halperin. D. Large-scale data analysis at cloud scale. In: Frontiers Big-Data Symposium. 2016,.
33. Ousterhout K, Rasti R, Ratnasamy S, Shenker S, Chun BG. Making sense of performance in data analytics frameworks. In: 12th USENIX Symposium on Networked System Design and Implementation. 2015,.
34. Abdul-Wahid B, Feng H, Rajan D, Costaouec R, Darve E, Thain D, et al. Awe-wq, fast-forwarding molecular dynamics using the accelerated weighted ensemble. Journal of Chemical Information and Modeling 2014;54:3033–43.
35. 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.

36. 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.

37. P L, K AD, L TD. Fast determination of the optimal rotational matrix for macromolecular superpositions. Journal of computational Chemistry 2010;31:1561–3.

38. 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.

39. 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.

40. Personal communication, pacific northwest national laboratory, https://www.pnnl.gov/. ????

41. https://pubs.cray.com/shmem, cray, document 004-2178-002, chapter 3. ????

42. Mellanox . Shmem is being used/proposed as a lower level interface for pgas implementations; 2012. New Accelerations for Parallel Programming.

43. 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.

44. 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.

45. 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.

46. Stone JE, Isralewitz B, Schulten K. Early experiences scaling vmd molecular visualization and analysis jobs on blue waters. IEEE; 2013,.

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	$t_{1/O}$ t_{comp}	2880 ± 0 272000 ± 0	40 ± 1.63 12440 ± 200.78	20 ± 1.22 6305 ± 38.13	15 ± 3.91 4225 ± 83.41	–	–	–	–
Comet	RMSD 1X	0.3	MPI	Single	$t_{1/O}$ t_{comp}	799 ± 5.22 225 ± 5.4	49 ± 3.45 11 ± 0.75	29 ± 1.3 6 ± 0.35	26 ± 9.19 4 ± 0.48	–	–	–	–
Comet	RMSD 1X	0.3	GA	Single	$t_{1/O}$ t_{comp}	820 ± 18.49 219 ± 9.8	41 ± 8.99 10 ± 0.3	23 ± 4.14 5 ± 0.48	15 ± 2.06 3 ± 0.54	–	–	–	–
Comet	RMSD 1X	0.3	MPI	Splitting	$t_{1/O}$ t_{comp}	799 ± 5.22 225 ± 5.4	37 ± 1.22 11 ± 0.31	18 ± 0.18 5 ± 0.07	12 ± 0.14 3 ± 0.04	9 ± 0.3 3 ± 0.11	6 ± 0.66 2 ± 0.23	4 ± 0.23 1 ± 0.07	–
SuperMIC	RMSD 1X	0.3	MPI	Splitting	$t_{1/O}$ t_{comp}	1013.75 ± 2.8 304.26 ± 2.55	39.99 ± 0.36 12.41 ± 0.22	19.18 ± 0.25 5.99 ± 0.09	9.61 ± 0.28 3.08 ± 0.13	–	4.83 ± 0.06 1.5 ± 0.01	–	–
Comet	RMSD 1X	0.3	GA	Splitting	$t_{1/O}$ t_{comp}	820 ± 18.5 219 ± 9.5	36 ± 0.78 9 ± 0.22	17 ± 0.3 4 ± 0.07	11 ± 0.23 3 ± 0.04	10 ± 1.7 2 ± 0.4	5 ± 0.14 1 ± 0.05	4 ± 0.07 1 ± 0.02	–
SuperMIC	RMSD 1X	0.3	GA	Splitting	$t_{1/O}$ t_{comp}	1027.62 ± 10.32 305.78 ± 3.47	39.62 ± 0.2 12.16 ± 0.1	19.66 ± 0.1 6.01 ± 0.007	9.57 ± 0.1 2.97 ± 0.1	–	4.86 ± 0.05 1.51 ± 0.03	–	–
Comet	RMSD 1X	0.3	MPI	PHDF5	$t_{1/O}$ t_{comp}	423 ± 5.88 225 ± 6.55	19 ± 0.3 10 ± 0.12	9 ± 0.13 5 ± 0.1	6 ± 0.06 3 ± 0.04	5 ± 0.12 2 ± 0.05	3 ± 0.2 1 ± 0.04	3 ± 0.25 1 ± 0.03	1.57 ± 0.29 0.76 ± 0.09
Bridges	RMSD 1X	0.3	MPI	PHDF5	$t_{1/O}$ t_{comp}	1318.87 ± 10.42 387.8 ± 5.51	67.93 ± 0.52 21.97 ± 0.38	37.37 ± 0.2 12.12 ± 0.34	30.35 ± 0.15 9.79 ± 0.24	24.16 ± 0.89 7.72 ± 0.03	22.5 ± 0.17 7.18 ± 0.08	–	–
SuperMIC	RMSD 1X	0.3	MPI	PHDF5	$t_{1/O}$ t_{comp}	503.69 ± 2.57 273.54 ± 4.7	12.96 ± 0.06 23.44 ± 0.29	6.46 ± 0.02 12.22 ± 0.43	3.2 ± 0.01 7.3 ± 0.85	–	1.64 ± 0.01 4.59 ± 0.96	–	0.82 ± 0.004 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.***giannis: The font is extremely small in this table. Also can you add lines between different experiment configurations? Haven't the number of cores per node played a role to the overall performance? ***mahzad: I corrected it. The number of cores per node has had effects and we need to have a discussion on that in text. Do you have any suggestion except from what I have disussed.

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]

```

```
23      bsize = int(stop-start)
24      results = np.zeros([bsize,2], dtype=float)
25
980     for iframe, ts in enumerate(clone.trajectory[start:stop:step]):
27          results[iframe, :] = ts.time, rms.rmsd(g.positions, xref0, center=True,
                superposition=True)
28
29      return results
985 #---------------------------------------------------------------------
31  # Check the files in the directory
32  PSF = os.path.abspath(os.path.normpath(os.path.join(os.getcwd(),'files/adk4AKE.
        psf')))
33  longXTC = os.path.abspath(os.path.normpath(os.path.join(os.getcwd(),'newtraj.xtc
990     ')))
34  longXTC1 = os.path.abspath(os.path.normpath(os.path.join(os.getcwd(),'files/
        newtraj{}.xtc'.format(j))))
35
36  if rank == 0:
995     copyfile(longXTC, longXTC1)
38      u = mda.Universe(PSF, longXTC1)
39      print(len(u.trajectory))
40
41  MPI.COMM_WORLD.Barrier()
1000 #---------------------------------------------------------------------
43  u = mda.Universe(PSF, longXTC1)
44  mobile = u.select_atoms("(resid 1:29 or resid 60:121 or resid 160:214) and name
        CA")[1:147]
45  index = mobile.indices
1005 topology, trajectory = mobile.universe.filename, mobile.universe.trajectory.
        filename
47  ref0 = mobile
48  xref0 = ref0.positions-ref0.center_of_mass()
49
1010 # Create each segment for each process
51  frames_seg = np.zeros([size,2], dtype=int)
52  bsize = int(np.ceil(mobile.universe.trajectory.n_frames / float(size)))
53  for iblock in range(size):
54      frames_seg[iblock, :] = iblock*bsize, (iblock+1)*bsize
1015
56  d = dict([key, frames_seg[key]] for key in range(size))
57
58  start, stop = d[rank][0], d[rank][1]
59
1020 # Block-RMSD in Parallel
61  out = block_rmsd(index, topology, trajectory, xref0, start=start, stop=stop,
        step=1)
```

```
62
63 if rank == 0:
1025    data1 = np.zeros([size*bsize,2], dtype=float)
65 else:
66    data1 = None
67
68 comm.Gather(out[0], data1, root=0)
1030
70 if rank == 0:
71    data = np.zeros([size,5], dtype=float)
72 else:
73    data = None
1035
75 comm.Gather(np.array(out[1:], dtype=float), data, root=0)
76
77 MPI.Finalize
```

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

```
1  import sys
2  import numpy as np
1045 import MDAnalysis as mda
4  from MDAnalysis.analysis import rms
5  import time
6  from shutil import copyfile
7  import glob, os
1050 import mpi4py
9  from mpi4py import MPI
10 from ga4py import ga
11 from ga4py import gain
12 #---------------------------------------
1055 ga.initialize()
14 comm = gain.comm()
15
16 size = ga.nnodes()
17 rank = ga.nodeid()
1060 #---------------------------------------
19 j = sys.argv[1]
20
21 def block_rmsd(index, topology, trajectory, xref0, start=None, stop=None, step=
       None):
1065    clone = mda.Universe(topology, trajectory)
23    g = clone.atoms[index]
```

```
24
25      print("block_rmsd", start, stop, step)
26      bsize = int(stop-start)
1070    results = np.zeros([bsize,2], dtype=float)
28
29      for iframe, ts in enumerate(clone.trajectory[start:stop:step]):
30          results[iframe, :] = ts.time, rms.rmsd(g.positions, xref0, center=True, ↩
                superposition=True)
1075
32      return results
33  #--------------------------------------------------------------------------
34  PSF = os.path.abspath(os.path.normpath(os.path.join(os.getcwd(),'files/adk4AKE.↩
        psf')))
1080 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/↩
        newtraj{}.xtc'.format(j))))
37
1085 if rank == 0:
39     copyfile(longXTC, longXTC1)
40     u = mda.Universe(PSF, longXTC1)
41     print(len(u.trajectory))
42
1090 ga.sync()
44  #--------------------------------------------------------------------------
45  u = mda.Universe(PSF, longXTC1)
46  mobile = u.select_atoms("(resid 1:29 or resid 60:121 or resid 160:214) and name ↩
        CA")
1095 index = mobile.indices
48  topology, trajectory = mobile.universe.filename, mobile.universe.trajectory.↩
        filename
49  ref0 = mobile
50  xref0 = ref0.positions-ref0.center_of_mass()
1100 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)
54
55  # Create each segment for each process
1105 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):
59      frames_seg[iblock, :] = iblock*bsize, (iblock+1)*bsize
60
1110 d = dict([key, frames_seg[key]] for key in range(size))
62
63  start, stop = d[rank][0], d[rank][1]
```

```
64
65  # Block-RMSD in Parallel
66  out = block_rmsd(index, topology, trajectory, xref0, start=start, stop=stop,
        step=1)
67
68  ga.put(g_a, out[0][:,:], (start,0), (stop,2))
69
70  if rank == 0:
71     buf = ga.get(g_a, lo=None, hi=None)
72
73  if rank == 0:
74     data = np.zeros([size,5], dtype=float)
75  else:
76     data = None
77
78  comm.Gather(np.array(out[1:], dtype=float), data, root=0)
79
80  ga.destroy(g_a)
81  ga.terminate()
```

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

```
1  import MDAnalysis as mda
2  import numpy as np
3  import glob
4  from itertools import chain
5  import time
6  from shutil import copyfile
7  import glob, os
8  import mpi4py
9  from mpi4py import MPI
10 import sys
11 #---------------------------------------
12 MPI.Init
13
14 comm = MPI.COMM_WORLD
15 size = comm.Get_size()
16 rank = comm.Get_rank()
17 #---------------------------------------
18 j = sys.argv[1]
19
20 def angle2sincos(x):
21     """Convert angle x to (cos x, sin x).
```

```
22
23      Parameters
24      ----------
1160    x : float or array_like
26
27      Returns
28      -------
29      feature_vector : array
1165        1D feature vector ``[cos(x[0]), sin(x[0]), cos(x[1]), sin(x[1]), ...]``.
31      """
32      x = np.deg2rad(x)
33      return np.ravel(np.transpose([np.cos(x), np.sin(x)]))
34
1170  def residues_to_dihedrals(residues):
36      """Return list of [phi1, psi1, phi2, psi2, ...] dihedral objects"""
37      return list(chain.from_iterable(
38              (res.phi_selection().dihedral, res.psi_selection().dihedral) for res
                    in residues))
1175
40    def featurize_dihedrals(dihedrals):
41      angles = [dihedral.value() for dihedral in dihedrals]
42      return angle2sincos(angles)
43
1180  def block_dihedrals(index, topology, trajectory, start=None, stop=None, step=
       None):
45      start00 = time.time()
46      clone = mda.Universe(topology, trajectory)
47      g = clone.atoms[index]
1185    residues = g.residues[1:-1]
49      dihedrals = residues_to_dihedrals(residues)
50
51      print("block_rmsd", start, stop, step)
52      print(len(clone.trajectory))
1190    bsize = stop-start
54      results = []
55
56      for iframe, ts in enumerate(clone.trajectory[start:stop:step]):
57          results.append(featurize_dihedrals(dihedrals))
1195
59      return np.array(results)
60    #-----------------------------------------------------------------------
61    PSF = os.path.abspath(os.path.normpath(os.path.join(os.getcwd(),'files/adk4AKE.
       psf')))
1200  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/
        newtraj{}.xtc'.format(j))))
64
1205 if rank == 0:
66      copyfile(longXTC, longXTC1)
67      u = mda.Universe(PSF, longXTC1)
68      print(len(u.trajectory))
69
1210 MPI.COMM_WORLD.Barrier()
71  #--------------------------------------------------------------------
72  u = mda.Universe(PSF, longXTC1)
73  mobile = u.select_atoms("protein")
74  index = mobile.indices
1215
76  topology, trajectory = mobile.universe.filename, mobile.universe.trajectory.
        filename
77  bsize = int(np.ceil(mobile.universe.trajectory.n_frames / float(size)))
78
1220 # 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)))
82  for iblock in range(size):
83      frames_seg[iblock, :] = iblock*bsize, (iblock+1)*bsize
1225
85  d = dict([key, frames_seg[key]] for key in range(size))
86
87  start, stop = d[rank][0], d[rank][1]
88
1230 out = block_dihedrals(index, topology, trajectory, start=start, stop=stop, step
        =1)
90
91  if rank == 0:
92     data1 = np.zeros([size*bsize,848], dtype=float)
1235 else:
94     data1 = None
95
96  comm.Gather(out[0], data1, root=0)
97
1240 if rank == 0:
99      data = np.zeros([size,5], dtype=float)
100 else:
101     data = None
102
1245 comm.Gather(np.array(out[1:], dtype=float), data, root=0)
104
105 MPI.Finalize
```