Task-parallel Analysis of Molecular Dynamics Trajectories

Ioannis Paraskevakos[1], Andre Luckow[2,3], George Chantzialexiou[1], Mahzad Khoshlessan[4], Oliver Beckstein[4], Geoffrey C. Fox[5] , Shantenu Jha[1]

[1] *Rutgers, the State University of New Jersey, Piscataway, NJ 08854, USA*
[2] *Institute of Computer Science, Ludwig-Maximilians-University, Munich, Germany*
[3] *School of Computing, Clemson University, Clemson, SC 29634, USA*
[4] *Department of Physics, Arizona State University, AZ 85281, USA*
[5] *School of Informatics, Computing and Engineering, Indiana University Bloomington, IN 470405, USA*

***Abstract*—Different frameworks for implementing parallel data analytics applications have been proposed by the HPC and Big Data communities. In this paper, we investigate three frameworks: Spark, Dask and RADICAL-Pilot with respect to their ability to support data analytics requirements on HPC resources. We investigate the data analysis requirements of Molecular Dynamics (MD) simulations which are significant consumers of supercomputing cycles, producing immense amounts of data: a typical large-scale MD simulation of physical systems of O(100,000) atoms can produce from O(10) GB to O(1000) GBs of data. We propose and evaluate different approaches for parallelization of a representative set of MD trajectory analysis algorithms, in particular the computation of path similarity and the identification of connected atom. We evaluate Spark, Dask and RADICAL-Pilot with respect to the provided abstractions and runtime engine capabilities to support these algorithms. We provide a conceptual basis for comparing and understanding the different frameworks that enable users to select the optimal system for its application. Further, we provide a quantitative performance analysis of the different algorithms across the three frameworks using different high-performance computing resources.**

***Index Terms*—Data analytics, MD Simulations Analysis, Parallel MD analysis, task-paralell**

I. Introduction

Different frameworks for implementing parallel data analytics have been developed by the HPC (MPI, OpenMP) and Big Data communities (Spark, Dask) [1]. MPI is the most widely used programming abstraction on HPC resources. MPI assumes a SPMD execution model where each process executes the same program; communication and synchronization need explicit implementation. Big Data frameworks like Spark utilize higher-level MapReduce-style programming models avoiding explicit implementations of communication. While MPI is well-suited for tightly-coupled, compute-centric algorithms, Big Data frameworks are designed for loosely-coupled, data-parallel algorithms.

Task-parallel applications are parallelized by partitioning the work into a set of self-contained units of compute, which require small amounts of coordination. Depending on the application, these tasks can be independent, i. e., with no interprocess communication, or coupled with varying degrees of data and compute dependencies. Big Data applications exploit task parallelism for data-parallel parts (e. g., `map` operations), but also require some coupling, viz., for computation of aggregates (the `reduce` operation). The MapReduce [2] abstraction popularized this important execution pattern. Typically, the reduce operation includes shuffling of intermediate data from a set of nodes to node(s) where the reduce operation executes. Big Data frameworks started adopting different HPC techniques to efficiently support interprocess communication, e. g. efficient shuffle operations [3] and other forms of communication [4].

Spark [5] and Dask [6] are two well-known Big Data frameworks. Both provide high-level MapReduce abstractions and are optimized for parallel processing of large data volumes, interactive analytics and machine learning. Their runtime engines can automatically partition data, generate parallel tasks, and execute them on a cluster of nodes. In addition, Spark offers in-memory capabilities that allows the caching of data that is read multiple times, making it particularly suited for interactive analytics and iterative machine learning algorithms. Dask also provides a MapReduce API (Dask Bags). Further, Dask's API is more versatile and allows custom task DAGs and parallel vector/matrix computations.

The attempt is not to determine which approach is better (Big Data versus HPC), but how to provide the "best of both" to a diverse set of applications [4], [7]. Several recent publications have applied HPC techniques to advance traditional Big Data applications and Big Data analytic frameworks [1]. The converse problem, viz., the application of Big Data frameworks to traditional HPC analytics applications, has received less attention. In this paper, we investigate the data analysis requirements of Molecular Dynamics (MD) applications. MD simulations are significant consumers of supercomputing cycles, producing immense amounts of data: a typical MD simulation of physical systems of O(100,000) atoms can produce from O(10) to O(1000) GBs of data [8]. In addition to being the prototypical HPC application, there is increasingly a need for the analysis to be integrated with the simulations to drive the next stages of the simulation (analysis- driven-simulation) [9].

In this paper, we investigate three frameworks and their suitability for implementing MD trajectory analysis algorithms. In addition to Spark and Dask, two Big Data frameworks, we investigate RADICAL-Pilot [10], a Pilot-Job [11] framework designed for implementing task-parallel applications on HPC. MD trajectories are time series of atoms/particles positions and velocities, which are analyzed using different statistical meth-

ods to infer certain properties, e. g. the relationship between distinct trajectories, frames within a trajectory etc. Many of these algorithms can be expressed using simple task abstractions or MapReduce [12]. Thus, the selected frameworks are promising candidates for MD analysis applications.

The paper makes the following contributions: it (i) characterizes and explains the behaviour of different MDAnalysis algorithms on these frameworks, (ii) evaluates the performance differences with respect to the architectures of these frameworks, and (iii) provides a conceptual basis for comparing the abstraction, capabilities and performance of these frameworks.

This paper is organized as follows: Section II discusses the Molecular Dynamics analysis algorithms under investigation, and tries to characterize them using the Big Data ogres classification ontology [13]. Section III, provides a description of the different frameworks that were used for evaluation. Section IV provides a description of the implementation of the MD algorithms on top of RADICAL-Pilot, Spark and Dask, as well as a performance evaluation and a discussion of findings. Section V reviews different of MD analysis frameworks in particular with respect to their ability to support scalable analytics of large volume MD trajectories. The paper concludes with a summary and discussion of future work in section VI.

II. Molecular Dynamics Analysis Applications

Some of the commonly used algorithms in the analysis of MD trajectories are Root Mean Square Deviation (RMSD), Pairwise Distances (PD), Path Similarity (PS), Sub-setting, "Leaflet Identification". RMSD is used to identify the deviation of atoms' positions between frames produced by a MD simulation. PD and PS methods calculate distances based on different metrics either between atoms or trajectories. Subsetting methods are used to isolate parts of interest of MD simulation. Leaflet identification methods provide information about groups of atoms in space; named after the common use case of identifying the two lipid leaflets in lipid bilayer. All these methods, in some way, read and process the whole physical system generated via simulations. The analysis done reduces the data to either a number or matrix.

We discuss in more detail two of these methods and their implementation in MDAnalysis [14], [15]. Specifically, we discuss a Path Similarity algorithm using the Hausdorff distance and a Leaflet Identification method. In addition, we explore the applications' Ogres Facets and Views [13] which will provide a more systematic characterization.

A. MDAnalysis

MDAnalysis is a Python library [14], [15] to analyze time series of atom coordinates. It is based on existing PyData tools, such as NumPy, and provides a comprehensive environment for filtering, transforming and analyzing MD trajectories.

1) Path Similarity Analysis (PSA): Hausdorff Distance: Path Similarity Analysis (PSA) [16] is used to quantify the similarity between two trajectories based on the properties of a distance metric. The Hausdorff Distance is an example of such a metric. Each trajectory is represented as a two dimensional array. The first dimension is a frame of the trajectory and the second is position of the atoms, in 3-dimensional space.

Algorithm 1 Path Similarity Algorithm: Hausdorff Distance

```
1: procedure HausdorffDistance(T1,T2)   ▷ T1 and T2 are a set of 3D points
2:     List D1, D2
3:     for ∀frame1 in T1 do
4:         for ∀frame2 in T2 do
5:             Append in D1 d_RMS(frame1, frame2)
6:         end for
7:         D_t1 append min(D1)
8:     end for
9:     for ∀frame2 in T2 do
10:        for ∀frame1 in T1 do
11:            Append in D2 d_RMS(frame2, frame1)
12:        end for
13:        D_t2 append min(D2)
14:    end for
15:    return max(max(D_t1), max(D_t2))
16: end procedure
17:
18: procedure PSA(Traj)                 ▷ Traj is a set of trajectories
19:    for ∀T1 in Traj do
20:        for ∀T2 in Traj do
21:            D_(T1,T2)=HausdorffDistance(T1, T2)
22:        end for
23:    end for
24:    return D
25: end procedure
```

Algorithm 2 Two Dimensional Partitioning

1: Initially, there are N^2 distances, where N is the number of trajectories. Each distance defines a computation task.
2: Map the initial set to a smaller set with $k = N/n_1$ elements, where n_1 is a divisor of N, by grouping n_1 by n_1 elements together.
3: Execute over the new set with k^2 tasks. Each task is the comparisons between n_1 and n_1 elements of the initial set. They are executed with a double for loop.

Algorithm 1 shows the PSA: Hausdorff Distance algorithm over multiple number of trajectories. We apply a 2-dimensional data partitioning over the output matrix in order to parallelize this algorithm. The partitioning is shown in algorithm 2. The Hausdorff metric we employ is based on a brute-force algorithm. Recently, an algorithm that uses early break was introduced [17] to speedup the execution. However, we are not aware of a parallel implementation of this algorithm.

2) Leaflet Finder: Algorithm 3 describes the Leaflet Finder algorithms as presented in [14]. It can be used to identify the outer and inner leaflets of a lipid membrane of arbitrary shape from trajectory information provided by simulations. The algorithm consists of two stages: first, a graph connecting particles based on threshold distance (the cutoff) is constructed. In the second stage the connected components of the graph are computed. The largest and second largest subgraph are the leaflets.

B. Application Characterization Using Big Data Ogres

The Big Data Ogres [13] provide a framework for understanding and characterizing the data-intensive applications. Ogres are organized into 4 classification dimensions, the *views*. The possible features of a view are called *facets*. The combination of the facets from different views define an Ogre. Ogres comprise of four views: 1) execution, 2) data source & style, 3) processing and 4) problem architecture. The execution view describes aspects, such as the I/O, memory and compute ratios, whether computations are iterative, and the 5 V's of Big Data. The data source & style view discusses the way input

Algorithm	Execution View	Data Source & Style	Problem Architecture View	Processing View
PSA	$O(n^2)$, Large Input/Small Output, NumPy & Python Arithmetic Libraries, Runtime	HPC Simulations, Permanent Data, Lustre, produced & stored by simulations	Embarrassingly Parallel	Linear Algebra Kernels
Leaflet Finder	Pairwise Distance: $O(n^2)$. Tree-based Nearest Neighbor Construction: $O(n \log n)$, Search: $O(log\, n)$. Connected Components: $O(\|V\| + \|E\|)$. Similar I/O, Small intermediate Data, NumPy & Python Arithmetic Libraries, Runtime, Graphs	HPC Simulations, Permanent Data, Lustre, produced & stored by simulations	MapReduce	Graph Algorithm, Linear Algebra Kernels

TABLE I: Mapping of the MDAnalysis algorithms to Big Data Ogres.

Algorithm 3 Leaflet Finder Algorithm

```
procedure LeafletFinder(Atoms, Cutoff)  ▷ Atoms is a set of 3D points that represent the position of atoms in space. Cutoff is an Integer Number
    Graph G =(V = Atoms, E = ∅)
    for ∀atom in Atoms do
        N = [a ∈ V : d(a, atom) ≤ Cutoff]
        Add edges [(atoms, a) : a ∈ N] in G
    end for
    C = ConnectedComponents(G)
    return C
end procedure
```

data are collected, stored and accessed. The processing view describes algorithms and kernels used for computation. The problem architecture view, describes the application architecture needed to support the application.

Table I summarizes the characteristics of the two MDAnalysis algorithms. The PSA Hausdorff Distance is embarrassingly parallel and uses some linear algebra kernels for calculations. It has complexity $O(n^2)$ (problem architecture and processing views). Its input data volume is medium to large while the output is small. Specific execution environments, such as HPC nodes, and Python arithmetic libraries, e.g., NumPy, are used (execution view). The input data are produced by HPC simulations. It is typically stored on HPC storage systems, such as parallel filesystem like Lustre (data source & style view).

The Leaflet Finder execution view can be described as follows. The application comprises of multiple stages with different complexities: The first stage is the identification of the neighboring atoms. There are different implementation alternatives: (i) computation of the pairwise distance between all atoms ($O(n^2)$); (ii) utilizing a tree-based nearest neighbor (Construction: $O(n \log n)$, Query: $O(\log n)$). In both alternatives the input data volume is medium size and the output of this stage is smaller than the input. The complexity of connected components is: $O(|V| + |E|)$ (V: Vertices, E: Edges), i. e. it greatly depends on the characteristics of the graph (in particular its sparsity).

The application typically uses HPC nodes as the execution environment, NumPy arrays, and runtime as performance metric. It uses matrices to represent the physical system and the distance matrix. The output data representation is a graph, i. e., a set of connected atoms that make up the leaflet. The Leaflet Finder can be efficiently implemented using the MapReduce abstraction (problem architecture view). The map phase determines adjacent atoms, and the reduce phase computes the connected components. Furthermore, it uses graph algorithms and linear algebra kernels, characteristics which are processing view facets. The data source & style view facets are the same as the PSA algorithm.

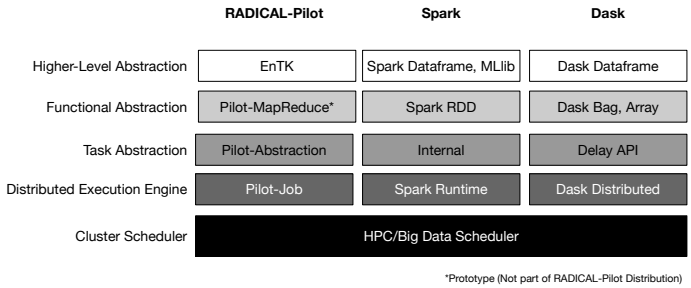

Fig. 1: **Architecture of RADICAL-Pilot, Spark and Dask:** The frameworks share common architectural components for managing cluster resource, managing task. Spark and Dask offer several high-level abstractions inspired by MapReduce.

III. Background of Evaluated Frameworks

The landscape of frameworks for data-intensive applications is manifold [7], [1] and has been extensively studied in the context of scientific [18] applications. In this section, we investigate the suitability of frameworks, such as Spark [5], Dask [6] and RADICAL-Pilot [10], for molecular dynamics data analytics.

MapReduce [2] and the open source Hadoop implementation pioneered the concept of combining a simple functional API with a powerful distributed computing engine that exploits data locality to allow optimal I/O performance. It is widely recognized that MapReduce is inefficient for interactive workloads and iterative machine learning algorithms [5], [19]. Spark and Dask are more modern systems providing richer APIs, caching and other capabilities critical for analytics applications. RADICAL-Pilot is a Pilot-Job framework [11] that supports task-level parallelism on HPC resources. It has been successfully utilized for adding a parallelization level on top of HPC applications based on MPI.

As described in [7], these frameworks typically comprise of several distinct layers, e. g., for accessing the cluster scheduler, for framework-level scheduling and higher-level abstractions. On top of these low-level resource management capabilities various higher-level abstractions can be provided, e. g., high-level MapReduce-inspired functional APIs. These then provide the foundation for analytics abstractions, such as Dataframes and machine learning capabilities. Figure 1 visualizes the different components of RADICAL-Pilot, Spark and Dask. In the following, we describe each framework in detail.

A. Spark

Spark [5] is a distributed computing framework that extends the MapReduce programming model [2] providing a richer set

of transformations on top of the resilient distributed dataset (RDD) abstraction [20]. RDDs are cached in-memory making Spark well suitable for iterative machine learning applications that need to cache a working set of data across multiple stages. PySpark provides a comprehensive Python API to Spark.

A Spark job consists of multiple stages; a stage comprises of a set of tasks that can be executed without communication (e. g., `map`) and an action (e. g., `reduce`, `groupby`). After each action a new stage is started. The `DAGScheduler` is responsible for translating the logical execution plan specified via RDD transformation to a physical plan.

The Spark distributed execution engine handles the low-level details of task execution based on the specified lineage of RDD transformations. The framework handles parallelism (e. g., by RDDs partitioning) and generates dataflow graph (referred to as DAG) that is executed according to the specified dependencies. Transformation are not scheduled for execution until the data they generate are needed by an action. Actions are Spark instructions that do an operation over a dataset. Execution of jobs is triggered by actions.

Spark can read data from different data sources, e. g., HDFS, blob storage (such S3), parallel and local filesystems. While Spark caches loaded data in memory, it is capable of offloading to disk when an executor has not sufficient memory available to hold all the data of its assigned partition. Persisted RDDs remain in memory unless it is specified to use the disk either complementary or as the single target. In addition, Spark writes to disk data that are used in a shuffle. As a result, it allows quick access to those data when transmitted to another executor. Finally, Spark provides a set of actions that allow to directly write text files, Hadoop sequence files or object files to the local filesystem, HDFS or any other filesystem that supports Hadoop. In addition to RDDs, Spark supports higher-level API for processing structured data, such as dataframes, Spark-SQL, datasets, and for streaming data.

B. Dask

Dask [6] provides a Python-based parallel computing library, which is designed to inter-operate and to parallelize native Python code written for NumPy and Pandas. In contrast to Spark, Dask also provides a lower-level task API (the `delayed` API) that allows the user to construct arbitrary graphs. Dask is written in Python and does not require a JVM avoiding some of the inefficiencies of PySpark, e. g., the necessity to move data from the Java/Scala to Python interpreter.

In addition to the low-level task API, Dask offers three higher-level abstractions: bags, arrays and dataframes. Dask Arrays are collection of independent NumPy arrays organized as a grid. Dask Bags are similar to Spark RDDs and are used to analyze semi-structured data, like JSON files. Dask Dataframe is a distributed collection of Pandas dataframes that can be analyzed in parallel.

In addition, Dask offers three schedulers: multithreading, multiprocessing and distributed. The multithreaded and multiprocessing schedulers can be used only on a single node and the parallel execution is done through threads or processes respectively. The distributed scheduler creates a cluster with a scheduling process and multiple worker processes. A client process creates and communicates a DAG to the scheduler. The scheduler assigns tasks to workers.

	RADICAL-Pilot	**Spark**	**Dask**
Languages	Python	Java, Scala, Python, R	Python
Task Abstraction	Task	Map-Task	Delayed
Functional Abstraction	-	RDD API	Bag
Higher-Level Abstractions	EnTK [21]	Dataframe, ML Pipeline, MLlib [22]	Dataframe, Arrays for block computations
Resource Management	Pilot-Job	Spark Execution Engines	Dask Distributed Scheduler
Scheduler	Individual Tasks	Stage-oriented DAG	DAG
Caching	-	RDD are in-memory cached	-
Shuffle	-	hash/sort-based shuffle	hash/sort-based shuffle
Limitations	no shuffle, filesystem-based communication	high overheads for Python tasks (serialization)	Dask Array can not deal with dynamic output shapes

TABLE II: **Frameworks Comparison:** Dask and Spark are designed for data-related task, while RADICAL-Pilot focuses on compute-intensive tasks.

C. RADICAL-Pilot

RADICAL-Pilot [10] is a Pilot-Job framework [11], implemented in Python, which allows concurrent task execution on HPC resources. The user can define a set of Compute-Units (CU) - the abstraction used to define a task along with its dependencies - which are submitted to RADICAL-Pilot. RADICAL-Pilot then schedules these CUs to be executed under the acquired resources. RADICAL-Pilot uses the existing environment of the resource to execute tasks. Any data communication between tasks is done via the use of the underlying parallel filesystem, e.g., Lustre. Task execution coordination and communication is done through a database (MongoDB).

D. Discussion

Table II summarizes the properties of these frameworks with respect to abstractions and runtime properties provided to create and execute parallel data applications.

API and Abstractions: RADICAL-Pilot provides a low-level API for scheduling tasks onto resources. While this API can be used to implement high-level capabilities, e. g. MapReduce [23], they are not provided out-of-the box. Both Spark and Dask provide higher-level abstractions: Dask's APIs are generally lower-level than the comparable Spark APIs, e. g., it allows the specification of arbitrary task DAGs. Further, the data partition size needs to be manually specified (via the block size). However, in many cases it is also necessary to fine-tune the parallelism in Spark by specifying the number of partitions.

Another important consideration is the availability of high-level abstractions. Higher-level tools for RADICAL-Pilot, such as the Ensemble Toolkit (EnTK) [21] are designed for workflows involving compute-intensive tasks. For Spark and Dask a set of high-level data-oriented abstractions exist, such as Dataframes and machine learning APIs.

Scheduling: Both Spark and Dask create a Direct Acyclic Graph (DAG) based on the specified transformations, which is then executed using the execution engine. Spark jobs are separated into stages. Once all tasks in a stage are completed, the scheduler moves to the next stage.

Dask's DAGs are represented by a tree where each node is a task. Leaf tasks do not depend on other task for the execution. Dask starts executing leaf tasks; tasks are executed as soon as their dependencies are satisfied. When a task is reached with unsatisfied dependencies, the scheduler executes the dependent task first. In contrast to Spark, the Dask scheduler does not rely on synchronizations points that the Spark stage-oriented scheduler introduces. RADICAL-Pilot does provide a DAG execution engine and requires the execution order to be specified explicitly, i. e. the user needs to describe, submit and synchronize tasks.

Suitability for MDAnalysis Algorithms: Trajectory analysis methods are often embarrassingly or pleasingly parallel, i. e., they are ideally suited for task management and functional MapReduce APIs. For example, PS methods typically require a single pass over the data and return a set of values that correspond to a relationship between frames or trajectories. They can easily be expressed as a bag of independent tasks using a task management API and a map-only application in a MapReduce-style API.

The Leaflet Finder is more complex and requires two stages: (i) in the edge discovery stage atoms that have a distance smaller than a defined cutoff are connected to a graph, and (ii) in the connected components stage it computes the connected components on this graph. While it is possible to implement Leaflet Finder with a simple task-management API, the MapReduce programming model enables the developer to express this problem more efficiently with a `map` for computing and filtering the distance and a `reduce` phase for computing the graph. The data movement or shuffling required between the map and reduce phase is medium as the number of edges represents a fraction of the input data.

IV. Experiments and Discussion

In this section, we characterize the performance of RADICAL-Pilot, Spark and Dask. In section IV-A we evaluate the task throughput using a synthetic workload. In sections IV-B and IV-C we evaluate the performance of two algorithms from the MDAnalysis library: Hausdorff Distance and LeafletFinder using different real-world datasets. We investigate: (i) what capabilities and abstractions of the frameworks are needed to efficiently express these algorithms, (ii) what architectural approaches can be used to implement these algorithms with these frameworks, and (ii) the performance trade-offs of these frameworks.

The experiments were executed on the XSEDE Supercomputers: Comet and Wrangler. SDSC Comet is a 2.7 PFlop/s cluster with 24 Haswell cores/node and 128 GB memory/node (6,400 nodes). TACC Wrangler is a cluster optimized for data-intensive computing; it has 48 cores/node and 128 GB memory/node (120 nodes). Experiments were carried using

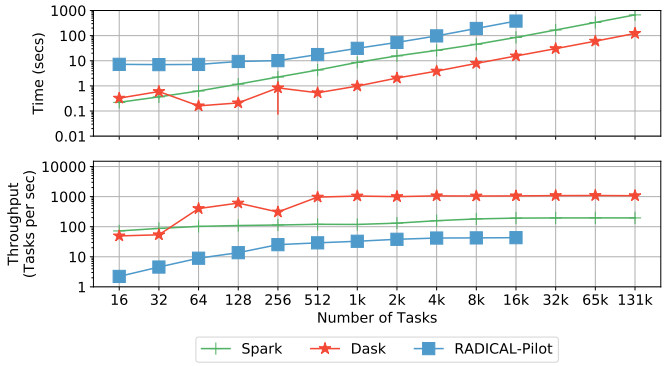

Fig. 2: **Task Throughput by Framework (Single Node)::** Time/Throughput for execution of a given number of zero-workload tasks on Wrangler. Dask performs best. Dask & Spark has very small delays for few tasks. RADICAL-Pilot offers the smallest throughput.

RADICAL-Pilot and the Pilot-Spark [24] extension, which allows the efficient management of Spark clusters on HPC infrastructures using a common interface for resource management. For Dask, we utilize a set of custom scripts to start the Dask cluster. We used RADICAL-Pilot 0.46.3rc, Spark 2.2.0, Dask 0.14.1 and Dask Distributed 1.16.3.

A. Frameworks Evaluation

As data-parallelism often involves a large number of short-running tasks, task throughput is a critical metric to assess the different frameworks. To evaluate the throughput we use zero workload tasks (`/bin/hostname`), We submit an increasing number of such tasks to RADICAL-Pilot, Spark and Dask and measure the execution time.

For RADICAL-Pilot, all tasks were submitted as bulk CUs. The RADICAL-Pilot backend database was running on the same node to avoid larger communication latencies. For Spark, we created an RDD with as many partitions as the number of tasks – each partition is mapped to a task by Spark. For Dask, we created tasks using `delayed` functions that were executed using the Distributed scheduler. We utilized Wrangler and Comet for this experiment.

The results are shown in Figure 2. Dask needed the least time to schedule and execute the assigned tasks, followed by Spark and RADICAL-Pilot. Dask and Spark quickly reach their maximum throughput, which is sustained while the number of tasks is increasing. RADICAL-Pilot showed the worst throughput and scalability mainly due to some architectural limitations, e. g., the reliance on MongoDB to communicate between the Pilot-Manager and Agents. Thus, we were not able to scale RADICAL-Pilot to more 32k or more tasks.

Figure 3 illustrate the throughput when scaling to multiple nodes measured by submitting $100,000$. Dask's throughput on all three resources increases almost linearly to the number of nodes. Spark's throughput is an order of magnitude lower than Dask's. RADICAL-Pilot's throughput plateaus at below 100 task/sec.

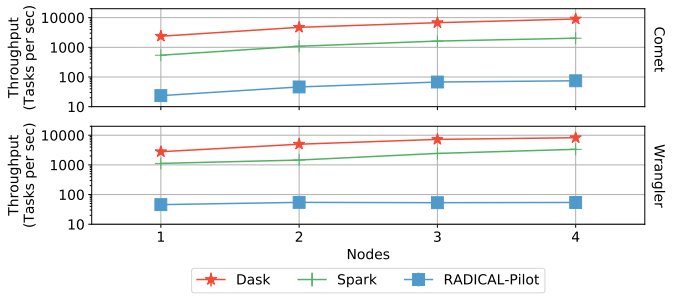

Fig. 3: **Task Throughput by Framework (Multiple Nodes):** Task throughput for $100k$ zero-workload tasks on different numbers of nodes for each framework. Consistent with the previous experiments: Dask has the largest throughput, followed by Spark and RADICAL-Pilot. Wrangler and Comet show a comparable performance with Comet slightly outperforming Wrangler.

B. Path Similarity: Hausdorff Distance

The Hausdorff distance algorithm is embarrassingly parallel and can be implement using simple task-level parallelism or a map only MapReduce application. The input data, i. e. a set of trajectory files, is equally distributed over the cores, generating one task per core. Each task reads its respective input files in parallel, executes and writes the result to a file.

For RADICAL-Pilot we define a Compute-Unit for each task and execute them using a Pilot-Job. For Spark, we create an RDD with one partition per task. The tasks are executes in a `map` function. In Dask the tasks are defined as `delayed` functions.

The experiments were executed on Comet and Wrangler. The dataset used consists of three different atom count trajectories: small (3,341 atoms/frame), medium (6,682 atoms/frame) and large (13,364 atoms/frame). Each trajectory has 102 frames and 128 trajectories of each size were used.

Figure 4 shows the runtime for 128 trajectories and 256 trajectories on Wrangler. Figure 5 compares the execution times between Comet and Wrangler for the large trajectories. We see that the frameworks have similar performance on both systems. As a result, moving the execution between resources will not affect the performance of the frameworks apart from the expected difference due to hardware differences.

RADICAL-Pilot, Spark and Dask have similar performance when they are used to execute algorithms that are pleasingly parallel. In all cases the overheads are comparable low in relation to the overall runtime. All frameworks achieved similar speedups as the number of cores increased. RADICAL-Pilot had the largest overheads because it creates links to files for moving data. In addition, RADICAL-Pilot large deviation is due to sensitivity to communication delays with the database. In summary, all three frameworks provide appropriate abstractions and runtime performance for embarrassingly parallel algorithms. In this case aspects such as programmability and integrate-ability are the most important considerations, e. g., both RADICAL-Pilot and Dask are native Python frameworks making the integration with MDAnalysis easier and more efficient than with other frameworks.

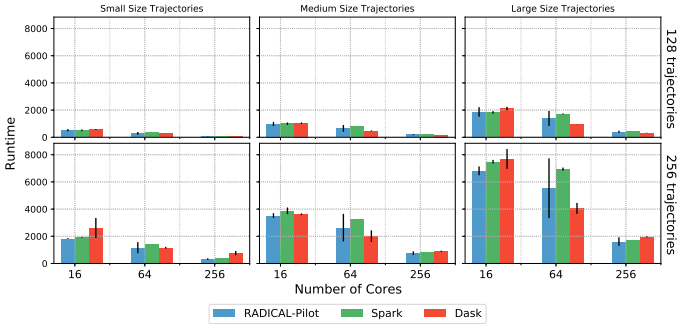

Fig. 4: **Hausdorff Distance on Wrangler using RADICAL-Pilot, Spark and Dask:** Runtimes over different number of cores, trajectory sizes and number of trajectories. All frameworks scaled by a factor of 6 from 16 to 256 cores.

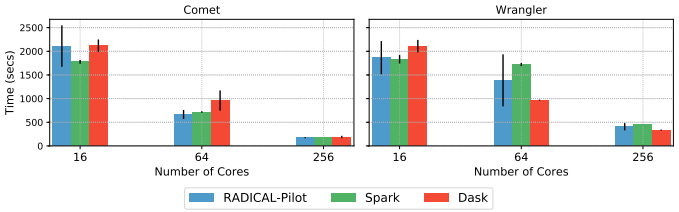

Fig. 5: **Hausdorff Distance on Comet and Wrangler:** Runtime for 128 large trajectories.

C. Leaflet Finder

In this section, we investigate four different approaches for implementing the Leaflet Finder algorithm using RADICAL-Pilot, Spark and Dask (see Table III and Figure 6):

1) **Broadcast and 1-D Partitioning:** Use of the RDD API (broadcast) and Dask Bag API (scatter) to distribute data to all nodes. The physical system is broadcasted and partitioned through a RDD. A `map` function calculates the edge list using the `cdist` function of SciPy [25]. This list is collected by the master process and the connected components are calculated.
2) **Task API and 2-D Partitioning:** Data management is done outside of the data-parallel framework, i. e., outside of Spark and Dask. The framework is used solely for task scheduling. The data are pre-partitioned in 2-D partitions and passed to a `map` function that calculates the pairwise distance and the edge list. Results are collected and the connected components are calculated
3) **Parallel Connected Components (Parallel-CC):** Data are managed as in approach 2. Each map task performs the pairwise-distance and connected components computations. The reduce phase joins the calculated graph components into one, if there is at least one common node.
4) **Tree-based Nearest Neighbor and Parallel-Connected Components (Tree-Search):** This approach is different to approach 3 only on the way edge discovery in the map phase is implemented. A tree-structure containing all atoms is created which is then used to query for adjacent atoms.

We use four physical systems with $131k$, $262k$, $524k$ and $4M$ atoms withand $896k$, $1.75M$, $3.52M$ and $44.6M$ edges in

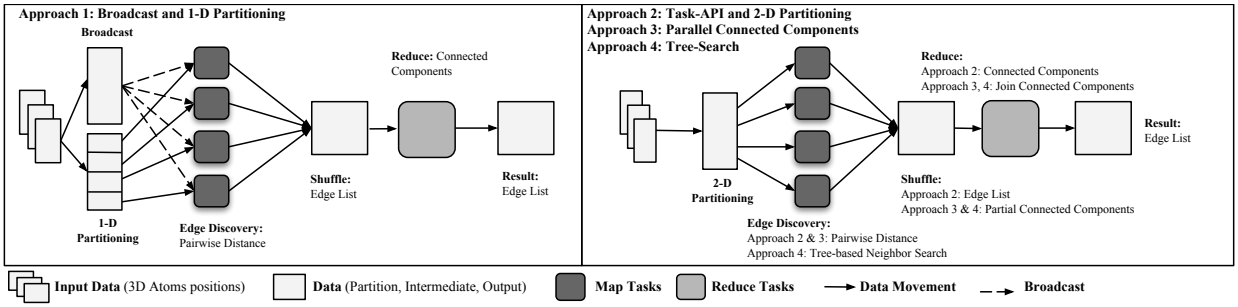

Fig. 6: **Leaflet Finder Architectural Approaches:** (1) Higher-Level Functional API used for data distribution in the map phase and subsequent reduce for connected components, (2) using the Task-API and framework-external data management and a reduce function, (3) optimization of the shuffle traffic by computing partial connected components in the map phase, and (4) optimizing edge discovery processing using tree-search algorithm.

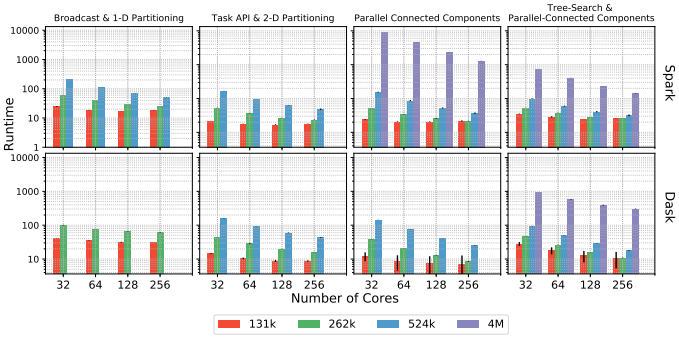

Fig. 7: **Leaflet Finder: Performance of Different Architectural Approaches for Spark & Dask:** Runtimes for different system sizes over different number of cores for all approaches and frameworks.

the graphs respectively. The experiments were conducted on Wrangler where we utilized up to 256 cores. The data is partitioned into 1024 parts so that 1024 tasks are used during the edge discovery phase. The only exception is with Approach 3 and the $4M$ atoms dataset. Due to `cdist`'s memory footprint – uses double precision floating point internally – the data is partitioned into $42k$ parts, and thus tasks.

Figure 7 shows the runtimes for all datasets for Spark and Dask. The RADICAL-Pilot performance is illustrated in Figure 9. In the following we analyze the performance of the architectural approaches and used frameworks in detail.

1) Broadcast and 1-D Partitioning: Approach 1 utilizes a broadcast to distribute the data to all nodes, which is only supported by Spark and Dask. All nodes maintain a complete copy of the dataset. Each `map` task will then compute the pairwise distance on its partition. We use 1-D partitioning. Figure 8 shows the detailed results: as expected the usage of a broadcast has severe limitations. It only scales up to $262k$ atoms for Dask and $524k$ atoms for Spark on Wrangler. Another observation is that Dask's broadcast implementation is slower than the one of Spark. The main reason is that Spark is the more mature framework with a more optimized communication subsystem. Nevertheless, the limited scalability of this approach due to the need to retain the entire dataset in memory renders it only usable for small datasets. It shows the

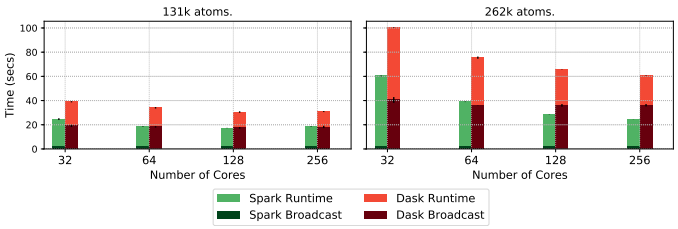

Fig. 8: **Broadcast and 1-D Partitioned Leaflet Finder (Approach 1):** Runtime for multiple system sizes on different number of cores for Spark and Dask. Spark provides a more efficient broadcast implementation. Broadcast times are about 3% – 15% of the edge discovery time for Spark, while 40% – 65% for Dask.

worst performance of all four approaches.

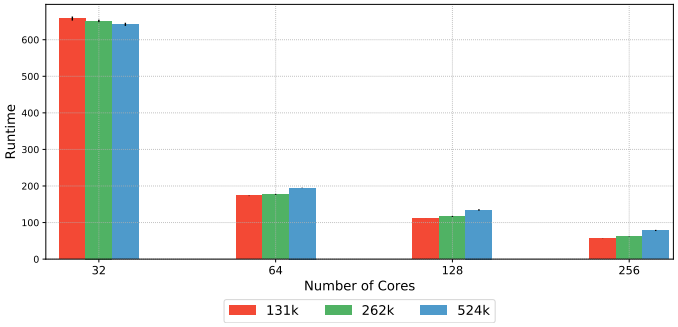

Fig. 9: **RADICAL-Pilot Task API and 2-D Partitioned Leaflet Finder (Approach 2):** Runtime for multiple system sizes over different number of cores using RADICAL-Pilot. RADICAL-Pilot is in the overheads since execution times for the pairwise distance are similar despite the system size.

2) Task-API and 2-D Partitioning: Approach 2 attempts to overcome the limitations of approach 1, in particular the broadcast and the 1-D partitioning. A 2-D block partitioning is essential, as it evenly distributes the compute and more efficiently utilizes the available memory. Unfortunately, 2-D partitioning is not well supported by Spark and Dask. Spark's RDDs are optimized for data-parallel applications with 1-D partitioning. While the Dask distributed array supports 2-D

block partitioning, it cannot be used for the Leaflet finder. The main reason is that Dask's `map_block` is always required to return an array of the same size, i. e., the pairwise distance cannot be computed without storing n^2 points in memory and then running a reduce which requires the shuffling of these points. Thus, we implemented a 2-D partitioning outside of RADICAL-Pilot, Spark and Dask. Each task will work on a 2-D pre-partitioned, rectangular part of the input data.

The runtimes of approach 2 are shown in Figure 7 for Spark and Dask and in Figure 9 for RADICAL-Pilot. As expected this approach overcomes the limitations of approach 1 and can easily scale to larger datasets (e. g., 524k atoms) while improving the overall runtime. However, we were not able to scale this implementation to the 4M dataset. For RADICAL-Pilot we observed significant task management overheads (see also section IV-A). This is a limitation of RADICAL-Pilot with respect to managing large numbers of tasks. This is particularly visible when the scenario was run on a single node with 32 cores. As more resources for the RADICAL-Pilot agent become available in the scenarios with more than 64 cores the performance improves dramatically.

3) Parallel Connected Components: Another important aspect is the communication between the edge discovery and the connected components stage. For the $524k$ atoms dataset the output of the edge discovery phase is $\approx$100 MB. To reduce the amount of data that needs to be shuffled, we refined the algorithm to compute the connected components on the partial dataset in the `map` phase. The connected components are then merged in the `reduce` phase. This reduces the amount of shuffle data by more than 50% (e. g., to 12 MB for Spark and 48 MB for Dask). Figure 7 shows the improvements in the runtimes, by $\approx 20\%$ for both Spark and Dask. Further, we were able to run very large datasets, such as the 4M dataset, using this architectural approach.

4) Tree-Search: Another bottleneck of the previous approaches is the edge discovery via brute-force calculation of the distances between all pairs of atoms. In approach 4 we replace the pairwise distance function with a tree-based, nearest neighbor search algorithm, in particular BallTree [26]. The algorithm has two stages: (i) construction of a tree and (ii) the querying tree for neighboring atoms. Using the tree-search the complexity of the problem can be reduced from n^2 to a log complexity. For our implementation we use the BallTree implementation of Scikit-Learn [27].

Figure 7 illustrates the performance of the implementation. For small datasets, i. e., $131k$ and $262k$ atoms, approach 2 is faster than the tree-based approach, since the number of points is too small. For the large dataset, the tree approach is faster. In addition, the tree algorithm has a smaller memory footprint than `cdist`. This allows us to scale to larger problems, e. g., a $4M$ atoms and $44.6M$ edges dataset without changing the total number of tasks or other optimizations.

D. Conceptual Framework and Discussion

In this section we provide a conceptual framework that allows application developers to carefully select a framework according to their requirements (e. g., computational and I/O characteristics). Thus, it is important to understand both the properties of the application and Big Data frameworks: Table III summarizes how the two algorithms can be implemented using the well-known MapReduce abstraction. Table IV illustrates the criteria of the conceptual framework and ranks the three frameworks.

1) Application Perspective: We have shown that it is possible to implement analytics applications for MD trajectory data using all three frameworks. The performance critically depends on implementation aspects, such as the computational complexity, and the amount of data that needs to be shuffled across the network. For embarrassingly parallel applications, such as the path similarity analysis, with coarse grained tasks, the choice of the framework does not have a large influence on the performance. Thus, other aspects, such as programmability and integrate-ability become more important. For fine-grained data parallelism, a Big Data framework, such as Spark and Dask, clearly outperforms RADICAL-Pilot. If some coupling is introduced, i. e. communication between the tasks is required, e. g., a reduce, the usage of Spark becomes advantageous. However, the integrating Spark with other tools needs to be carefully considered: the integration of Python tools, e. g. MDAnalysis, often causes overheads due to the frequent need for serialization and copying data between the Python and Java space.

2) Framework Perspective: RADICAL-Pilot is particularly suited for HPC applications, e. g., ensembles of parallel MPI applications. It has some scalability limitations with respect to supporting large numbers of tasks as often found in data-intensive workloads. Further, the file staging implementation of RADICAL-Pilot is not suitable for supporting the data exchange patterns, in particular shuffling, required for these applications. However, with the ability to efficiently run MPI and Spark applications alongside on the same resource makes RADICAL-Pilot particularly suitable for scenarios where different programming paradigms need to be combined.

Dask provides a highly flexible, low-latency task management and excellent support for parallelization of Python libraries. While the communication subsystem of Dask showed some weaknesses in particular visible in the broadcast and shuffle performance: the broadcast (Leaflet Finder approach 1) and shuffle (Leaflet Finder approaches 2-4) performance for larger problems is significantly better for Spark than Dask.

Spark needs to be particularly considered for shuffle-intensive and machine learning applications: (i) the in-memory caching mechanism is particularly suited for iterative algorithms that maintain a static set of points in-memory and conduct multiple passes on the set, (ii) MLlib [22] provides several scalable, parallel machine learning algorithms.

V. Related Work

MD analysis algorithms were until recently executed serially and parallelization was not straightforward. During the last years several frameworks emerged providing parallel algorithms for analyzing MD trajectories. Some of those frame-

	Hausdorff Distance	Leaflet Finder Broadcast and 1-D Approach 1	Leaflet Finder Task API and 2-D Approach 2	Leaflet Finder Parallel Connected Components Approach 3	Leaflet Finder Tree-based Edge Discovery Approach 4
Data Partitioning	2D	1D	2D	2D	2D
Map	Path Similarity	Edge Discovery via Pairwise Distance	Edge Discovery via Pairwise Distance	Edge Discovery via Pairwise Distance and Partial Connected Components	Edge Discovery via Tree-based Algorithm and Partial Connected Components
Shuffle	no	Edge List ($O(E)$)	Edge List ($O(E)$)	Partial Connected components ($O(n)$)	Partial Connected components ($O(n)$)
Reduce	no	Connect Components	Connected Components	Joined Connected Components	Joined Connected Components

TABLE III: MapReduce Operations used by MD Applications

	RADICAL-Pilot	Spark	Dask
Task Management			
Low Latency	-	o	+
Throughput	-	+	++
MPI/HPC Tasks	+	o	o
Task API	+	o	++
Large Number of Tasks	–	++	++
Application Characteristics			
Python/native Code	++	o	+
Java	o	++	o
Higher-Level Abstraction	-	++	+
Shuffle	-	++	+
Broadcast	-	++	+
Caching	-	++	o

TABLE IV: **Decision Framework:** Criteria and Ranking for Framework Selection

works are CPPTRAJ [28], HiMach [29], Pteros 2.0 [30], MDTraj [31], and nMoldyn-3 [32]. We compare these frameworks with our approach over the parallelization techniques used. Any performance, functionality or usability comparison is more suited for a literature review. We show, to the best of our knowledge, the efforts being done for parallelizing MD analysis algorithms.

CPPTRAJ [28] provides several analysis algorithms parallelized through MPI and OpenMP. MPI is being used to parallelize the execution over the frame of a single trajectory or each trajectory in an ensemble of trajectories. OpenMP is used to parallelize the execution of compute intensive algorithms.

HiMach [29] was developed by D. E. Shaw Research group to provide a parallel analysis framework for MD simulations, extends Google's MapReduce. HiMach API defines trajectories, does per frame data acquisition (Map) and cross-frame analysis (Reduce). HiMach's runtime is responsible to parallelize and distribute Map and Reduce phases to resources. Data transfers are done through a communication protocol created specifically for HiMach.

Pteros-2.0 [30] is a open-source library that is used for modeling and analyzing MD trajectories, providing a plugin for each supported algorithm. The execution is done by a user defined driver application, which setups trajectory I/O and frame dispatch for analysis. It offers a C++ and Python API. Pteros 2.0 parallelizes computational intensive algorithms by using OpenMP and Multithreading. As a result, it is bounded to execute on a single node, making any analysis execution highly dependent on memory size. Through RADICAL-Pilot, Spark and Dask, we avoided the need to recompile every time there is a change to the underlying resource, ensuring the application's execution.

MDTraj [31] is a Python package for analyzing MD trajectories. It links MD data and Python statistical and visualization software. MDTraj proposes parallelizing the execution by using the parallel package of IPython as a wrapper along with an out-of-core trajectory reading method. Our approach support of data analysis frameworks allows data parallelization on any level of the execution, not only in data read.

nMoldyn-3 [32] parallelizes the execution through a Master/Worker architecture. The master or client defines analysis tasks, submits them to a task manager, which then are executed by the worker process. In addition, it provides adaptability allowing on-the-fly addition of resources, and execution fault tolerance when worker processes disconnect.

In contrast, our approach utilizes more general purpose frameworks for parallelization. Because they provide higher level abstractions, e.g machine learning, any integration with other data analysis methods can be fast and easier. In addition, resource acquisition and management is done transparently.

VI. Conclusion and Future Work

In this paper, we investigated the use of different programming abstractions and frameworks for the implementation of a range of algorithms for molecular dynamics trajectory analysis. We conducted an in-depth analysis of the application characteristics and assessed the architectures of RADICAL-Pilot, Spark and Dask. We provide a conceptual framework that enables application developers to qualitatively evaluate Big Data frameworks with respect to their application requirements. Our benchmarks enable them to quantitatively assess framework performance as well as the expected performance of different implementation alternatives.

While the task abstractions provided by all frameworks are well-suited for implementing all use cases, the high-level MapReduce programming model provided by Spark and Dask provides several advantages: it is easier to use and efficiently support common data exchange patterns, such as the shuffling of data between the `map` and `reduce` stage. In our benchmarks Spark outperforms Dask in communication-intensive tasks, such as broadcasts and shuffles. Further, the in-memory RDD abstraction is great for iterative algorithms (as many machine learning algorithms). Dask provides more versatile low-level and high-level APIs and integrates better with the PyData ecosystems. RADICAL-Pilot does not provide a MapReduce API, but is well suited for coarse-grained task-level parallelism and for cases where HPC and analytics framework need to be integrated. We also identified severe limitation in Dask and Spark: while both frameworks provide some support for

linear algebra – both provide abstractions for distributed array – these proofed not flexible enough for implementing the all-pairs patterns efficiently requiring significant workarounds in the implementation and the utilization of out-of-framework functions to read and partition the input data.

In the future, we will further improve the performance of the presented algorithms, e. g., by reducing the memory and computation footprint, data transfer sizes between stages, by optimizing filesystem usage. To better support PyData tools from RADICAL-Pilot, we plan to extend the Pilot-Abstraction to support Dask and other Big Data frameworks. Further, we will refine the RADICAL-Pilot task execution engine to meet the requirement of data analytics application and devise task execution strategies that can mitigate with issues occurring at large scale, such as stragglers. Another area of research, is dynamic resource management and the ability to dynamically scale the resource pool (e. g., by adding or removing nodes) to meet the requirements of a specific application stage.

Acknowledgements We thank Andre Merzky and Thomas Cheatham for useful discussions. This work is funded by NSF 1443054 and 1440677. Computational resources were provided by NSF XRAC award TG-MCB090174.

REFERENCES

[1] S. Kamburugamuve, P. Wickramasinghe, S. Ekanayake†, and G. C. Fox, "Anatomy of machine learning algorithm implementations in mpi, spark, and flink," in *Technical Report*, Indiana University, Bloomington, 2017.

[2] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. Berkeley, CA, USA: USENIX Association, 2004, pp. 137–150.

[3] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda, "High-performance design of apache spark with rdma and its benefits on various workloads," December 2016.

[4] G. Fox, J. Qiu, S. Jha, S. Kamburugamuve, and A. Luckow, "Hpc-abds high performance computing enhanced apache big data stack," in *Proceedings of Workshop on Scalable Computing For Real-Time Big Data Applications (SCRAMBL'15)*. Shenzhen, China: 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2015.

[5] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[6] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Science Conference*, K. Huff and J. Bergstra, Eds., 2015, pp. 130 – 136.

[7] S. Jha, J. Qiu, A. Luckow, P. K. Mantha, and G. C. Fox, "A tale of two data-intensive paradigms: Applications, abstractions, and architectures," *Proceedings of 3rd IEEE Internation Congress of Big Data*, vol. abs/1403.1528, 2014.

[8] T. Cheatham and D. Roe, "The impact of heterogeneous computing on workflows for biomolecular simulation and analysis," *Computing in Science Engineering*, vol. 17, no. 2, pp. 30–39, 2015.

[9] V. Balasubramanian, I. Bethune, A. Shkurti, E. Breitmoser, E. Hruska, C. Clementi, C. Laughton, and S. Jha, "Extasy: Scalable and flexible coupling of md simulations and advanced sampling techniques," in *Accepted for IEEE International Conference on eScience*, 2016, https://arxiv.org/abs/1606.00093.

[10] A. Merzky, M. Santcroos, M. Turilli, and S. Jha, "Executing Dynamic and Heterogeneous Workloads on Super Computers," 2016, (under review) http://arxiv.org/abs/1512.08194.

[11] A. Luckow, M. Santcroos, A. Merzky, O. Weidner, P. Mantha, and S. Jha, "P*: A model of pilot-abstractions," *IEEE 8th International Conference on e-Science*, pp. 1–10, 2012, http://dx.doi.org/10.1109/eScience.2012.6404423.

[12] M. Khoshlessan, I. Paraskevakos, S. Jha, and O. Beckstein, "Parallel Analysis in MDAnalysis using the Dask Parallel Computing Library," in *Proceedings of the 16th Python in Science Conference*, K. Huff, D. Lippa, D. Niederhut, and M. Pacer, Eds., Austin, TX, 2017, pp. 64–72. [Online]. Available: http://conference.scipy.org/proceedings/scipy2017/mahzad_khoslessan.html

[13] G. C. Fox, S. Jha, J. Qiu, and A. Luckow, "Towards an understanding of facets and exemplars of big data applications," in *Proceedings of Beowulf'14*. Annapolis, MD, USA: ACM, 2014.

[14] N. Michaud-Agrawal, E. J. Denning, T. B. Woolf, and O. Beckstein, "Mdanalysis: A toolkit for the analysis of molecular dynamics simulations," *Journal of Computational Chemistry*, vol. 32, no. 10, pp. 2319–2327, 2011. [Online]. Available: http://dx.doi.org/10.1002/jcc.21787

[15] Richard J. Gowers, Max Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domański, David L. Dotson, Sébastien Buchoux, Ian M. Kenney, and Oliver Beckstein, "MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations," in *Proceedings of the 15th Python in Science Conference*, Sebastian Benthall and Scott Rostrup, Eds., 2016, pp. 98 – 105.

[16] S. L. Seyler, A. Kumar, M. F. Thorpe, and O. Beckstein, "Path similarity analysis: A method for quantifying macromolecular pathways," *PLoS Comput Biol*, vol. 11, no. 10, pp. 1–37, 10 2015. [Online]. Available: http://dx.doi.org/10.1371%2Fjournal.pcbi.1004568

[17] A. A. Taha and A. Hanbury, "An efficient algorithm for calculating the exact hausdorff distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 11, pp. 2153–2163, Nov 2015.

[18] S. Jha, D. S. Katz, A. Luckow, N. Chue Hong, O. Rana, and Y. Simmhan, "Introducing distributed dynamic data-intensive (d3) science: Understanding applications and infrastructure," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 8, pp. e4032–n/a, 2017, e4032 cpe.4032. [Online]. Available: http://dx.doi.org/10.1002/cpe.4032

[19] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 810–818. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851593

[20] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228301

[21] V. Balasubramanian, M. Turilli, W. Hu, M. Lefebvre, W. Lei, G. Cervone, J. Tromp, and S. Jha, "Harnessing the Power of Many: Extensible Toolkit for Scalable Ensemble Applications," *ArXiv e-prints*, Oct. 2017.

[22] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "Mllib: Machine learning in apache spark," *Journal of Machine Learning Research*, vol. 17, no. 34, pp. 1–7, 2016. [Online]. Available: http://jmlr.org/papers/v17/15-237.html

[23] P. K. Mantha, A. Luckow, and S. Jha, "Pilot-MapReduce: an extensible and flexible MapReduce implementation for distributed data," in *Proceedings of third international workshop on MapReduce and its Applications*, ser. MapReduce '12. New York, NY, USA: ACM, 2012, pp. 17–24. [Online]. Available: https://raw.github.com/saga-project/radical.wp/master/publications/pdf/pilot-mapreduce2012.pdf

[24] A. Luckow, I. Paraskevakos, G. Chantzialexiou, and S. Jha, "Hadoop on HPC: Integrating Hadoop and Pilot-based Dynamic Resource Management," *IEEE International Workshop on High-Performance Big Data Computing in conjunction with The 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2016)*, 2016.

[25] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[26] S. M. Omohundro, "Five balltree construction algorithms," Tech. Rep., 1989.

[27] "Scikit-Learn: Nearest Neighbors," http://scikit-learn.org/stable/modules/neighbors.html, 2016.

[28] D. R. Roe and I. Thomas E. Cheatham, "Ptraj and cpptraj: Software for processing and analysis of molecular dynamics trajectory data," *Journal of Chemical Theory and Computation*, vol. 9, no. 7, pp. 3084–3095, 2013, pMID: 26583988. [Online]. Available: http://dx.doi.org/10.1021/ct400341p

[29] T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. O. Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K. A. Stafford, and D. E. Shaw, "A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories," in *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2008, pp. 1–12.

[30] S. O. Yesylevskyy, "Pteros 2.0: Evolution of the fast parallel molecular analysis library for c++ and python," *Journal of Computational Chemistry*, vol. 36, no. 19, pp. 1480–1488, 2015. [Online]. Available: http://dx.doi.org/10.1002/jcc.23943

[31] R. McGibbon, K. Beauchamp, M. Harrigan, C. Klein, J. Swails, C. Hernández, C. Schwantes, L.-P. Wang, T. Lane, and V. Pande, "Mdtraj: A modern open library for the analysis of molecular dynamics trajectories," *Biophysical Journal*, vol. 109, no. 8, pp. 1528 – 1532, 2015. [Online]. Available: //www.sciencedirect.com/science/article/pii/S0006349515008267

[32] K. Hinsen, E. Pellegrini, S. Stachura, and G. R. Kneller, "nmoldyn 3: Using task farming for a parallel spectroscopy-oriented analysis of molecular dynamics simulations," *Journal of Computational Chemistry*, vol. 33, no. 25, pp. 2043–2048, 2012. [Online]. Available: http://dx.doi.org/10.1002/jcc.23035