Task-parallel Analysis of Molecular Dynamics Trajectories

Ioannis Paraskevakos
Rutgers University
Piscataway, New Jersey, USA

Andre Luckow
Ludwig-Maximilians-University
Munich, Germany

Mahzad Khoshlessan
Arizona State University
Tempe, Arizona, USA

George Chantzialexiou
Rutgers University
New Jersey, USA

Thomas E. Cheatham
University of Utah
Salt Lake City, Utah, USA

Oliver Beckstein
Arizona State University
Tempe, Arizona, USA

Geoffrey C. Fox
Indiana University
Bloomington, Indiana, USA

Shantenu Jha
Rutgers University and Brookhaven National Laboratory.

ABSTRACT

Different parallel frameworks for implementing data analysis applications have been proposed by the HPC and Big Data communities. In this paper, we investigate three task-parallel frameworks: Spark, Dask and RADICAL-Pilot with respect to their ability to support data analytics on HPC resources and compare them to MPI. 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 a physical system of $O(100k)$ atoms over $\mu secs$ 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 leaflet identification. We evaluate Spark, Dask and RADICAL-Pilot with respect to their abstractions and runtime engine capabilities to support these algorithms. We provide a conceptual basis for comparing and understanding different frameworks that enable users to select the optimal system for each application. We also provide a quantitative performance analysis of the different algorithms across the three frameworks.

KEYWORDS

Data analytics, MD Simulations Analysis, MD analysis, task-parallel

ACM Reference Format:
Ioannis Paraskevakos, Andre Luckow, Mahzad Khoshlessan, George Chantzialexiou, Thomas E. Cheatham, Oliver Beckstein, Geoffrey C. Fox, and Shantenu Jha. 2018. Task-parallel Analysis of Molecular Dynamics Trajectories. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225128

1 INTRODUCTION

Frameworks for parallel data analysis have been created by the High Performance Computing (HPC) and Big Data communities [17]. MPI is the most used programming model for HPC resources. It assumes a SPMD execution model where each process executes the same program. It is highly optimized for high-performance computing and communication, which along with synchronization need explicit implementation. Big Data frameworks utilize higher-level MapReduce [7] programming models avoiding explicit implementations of communication. In addition, the MapReduce [7] abstraction makes it easy to exploit data-parallelism as required by many analysis applications. Several recent publications applied HPC techniques to advance traditional Big Data applications and Big Data frameworks [17].

Task-parallel applications involve partitioning a workload into a set of self-contained units of work. Based on the application, these tasks can be independent, have no inter-task communication, or coupled with varying degrees of data dependencies. Big Data applications exploit task parallelism for data-parallel parts (e. g., `map` operations), but also require coupling, for computing aggregates (the `reduce` operation). The MapReduce [7] abstraction popularized this execution pattern. Typically, a reduce operation includes shuffling intermediate data from a set of nodes to node(s) where the reduce executes. There is a recognized need to optimize communication intensive parts of Big Data frameworks using established HPC techniques for interprocess, e. g. shuffle operations [18] and other forms of communication [9, 16].

Spark [43] and Dask [30] are two Big Data frameworks. Both provide 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. In addition, Spark offers in-memory capabilities allowing caching data that are read multiple times, making it suited for interactive analytics and iterative machine learning algorithms. Dask also provides a MapReduce API (Dask Bags). Furthermore, Dask's API is more versatile, allowing custom workflows and parallel vector/matrix computations.

In this paper, we investigate the data analysis requirements of Molecular Dynamics (MD) applications. MD simulations are significant consumers of computing cycles, producing immense amounts of data. A typical μsec MD simulation of physical system of $O(100k)$ atoms can produce from $O(10)$ to $O(1000)$ GBs of data [4]. In addition to being the prototypical HPC application, there is increasingly a need for the analysis to be integrated with simulations and drive the next stages of execution [2]. The analysis phase must be performed quickly and efficiently in order to steer the simulations.

We investigate three task-parallel frameworks and their suitability for implementing MD trajectory analysis algorithms. In addition to Spark and Dask, we investigate RADICAL-Pilot [25], a Pilot-Job [20] framework designed for implementing task-parallel applications on HPC. We utilize MPI4py [6] to provide MPI equivalent implementations of the algorithms. The task-parallel implementations performance and scalability compared to MPI is the basis of our analysis. MD trajectories are time series of atoms/particles positions and velocities, which are analyzed using different statistical methods to infer certain properties, e. g. the relationship between distinct trajectories, snapshots of a trajectory etc. As a result, they can be considered as a representative set of scientific datasets that are organized as time series and their analysis algorithms.

The paper makes the following contributions: i) it characterizes and explains the behavior of different MDAnalysis algorithms on these frameworks, and ii) provides a conceptual basis for comparing the abstraction, capabilities and performance of these frameworks.

The paper is organized as follows: Section 2 discusses the MD analysis algorithms investigated, and provides a brief characterization based on the Big Data Ogres classification ontology [10]. Section 3, describes the different frameworks that were used for evaluation. Section 4 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 5 reviews different MD analysis frameworks 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 6.

2 MOLECULAR DYNAMICS ANALYSIS APPLICATIONS

Some commonly used algorithms for analyzing MD trajectories are Root Mean Square Deviation (RMSD), Pairwise Distances (PD), and Sub-setting [27]. Two more advanced algorithms are Path Similarity Analysis (PSA) [33] and Leaflet Identification [26]. RMSD is used to identify the deviation of atom positions between frames. PD and PSA methods calculate distances based on different metrics either between atoms or trajectories. Sub-setting methods are used to isolate parts of interest of MD simulation. Leaflet Identification provides information about groups of lipids by identifying the lipid leaflets in a lipid bilayer. All these methods, in some way, read and process the whole physical system generated via simulations. The analysis reduces the data to either a number or a matrix.

We discuss two of these methods, a Path Similarity Analysis (PSA) algorithm using the Hausdorff distance and a Leaflet Identification method, and their implementations in MDAnalysis [26, 29]. In addition, we implemented the PSA algorithm using CPPTraj [31]. Furthermore, we explore the applications' Ogres Facets and Views [10], which provide a more systematic characterization.

Big Data Ogres [10] are organized into four classes, called *views*. The possible features of a view are called *facets*. A combination of facets from all views defines an Ogre. The Views are: 1) execution - describes aspects, such as I/O, memory, compute ratios, whether computations are iterative, and the 5 V's of Big Data (Volume, Velocity, Value, Variety and Veracity), 2) data source & style - discusses input data collection, storage and access, 3) processing - describes algorithms and kernels used for computation, and 4) problem architecture - describes the application architecture.

2.1 MDAnalysis

MDAnalysis is a Python library [26, 29] that provides a comprehensive environment for filtering, transforming and analyzing MD trajectories in all commonly used file formats. It provides a common object-oriented API to trajectory data and leverages existing libraries in the scientific Python software stack, such as NumPy [40] and Scipy [15].

2.1.1 Path Similarity Analysis (PSA): Hausdorff Distance. Path Similarity Analysis (PSA) [33] is used to quantify the similarity between trajectories considering their full atomic detail. The basic idea is to compute pair-wise distances (for instance, using the Hausdorff metric [12]) between members of an ensemble of trajectories and cluster the trajectories based on their distance matrix.

Each trajectory is represented as a two dimensional array. The first dimension corresponds to time frames of the trajectory, the second to the N atom positions, in 3-dimensional space.

Algorithm 1 Path Similarity Algorithm: Hausdorff Distance

```
procedure HausdorffDistance($T_1$,$T_2$)        ▷ $T_1$ and $T_2$ are a set of 3D points
    List $D_1$,$D_2$
    for $\forall frame_1$ in $T_1$ do
        for $\forall frame_2$ in $T_2$ do
            Append in $D_1$ $d_{RMS}(frame_1, frame_2)$
        end for
        $D_{t_1}$ append $min(D_1)$
    end for
    for $\forall frame_2$ in $T_2$ do
        for $\forall frame_1$ in $T_1$ do
            Append in $D_2$ $d_{RMS}(frame_2, frame_1)$
        end for
        $D_{t_2}$ append $min(D_2)$
    end for
    return $max\Big(max(D_{t_1}), max(D_{t_2})\Big)$
end procedure

procedure PSA($Traj$)        ▷ $Traj$ is a set of trajectories
    for $\forall T_1$ in $Traj$ do
        for $\forall T_2$ in $Traj$ do
            $D_{(T_1,T_2)}$=HausdorffDistance$\Big(T_1, T_2\Big)$
        end for
    end for
    return $D$
end procedure
```

Algorithm 1 describes the PSA algorithm with the Hausdorff metric over multiple trajectories. We apply a 2-dimensional data partitioning over the output matrix to parallelize, shown in algorithm 2. Our Hausdorff metric calculation is based on a naive algorithm. Recently, an algorithm was introduced that uses early break to speedup execution [34], although we are not aware of a parallel implementation of this algorithm.

The algorithm is embarrassingly parallel and uses linear algebra kernels for calculations. It has complexity $O(n^2)$ (problem architecture & processing views). 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

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 serially.
```

(execution view). Input data are produced by HPC simulations, and are stored on HPC storage systems, such as parallel filesystem like Lustre (data source & style view).

2.1.2 Leaflet Finder. Algorithm 3 describes the Leaflet Finder (LF) algorithm as presented in Ref. [26]. LF assigns particles to one of two curved but locally approximately parallel sheets, provided that the inter-particle distance is smaller than the distance between sheets. In biomolecular simulations of lipid membranes, consisting of a double layer of lipid molecules, LF is used to identify the lipids in the outer and inner leaflets (sheets). The algorithm consists of two stages: a) construction of a graph connecting particles based on threshold distance (cutoff), and b) computing the connected components of the graph, determining the lipids located on the outer and inner leaflets.

Algorithm 3 Leaflet Finder Algorithm

```
1: procedure LeafletFinder($Atoms$, $Cutoff$)  ▷ $Atoms$ is a set of 3D points that represent the position of atoms in space. $Cutoff$ is an Integer Number
2:     Graph G =($V = Atoms$, $E = \emptyset$)
3:     for $\forall atom$ in $Atoms$ do
4:         $N = [a \in V : d(a, atom) \leq Cutoff]$
5:         Add edges $[(atoms, a) : a \in N]$ in G
6:     end for
7:     C = ConnectedComponents(G)
8:     return C
9: end procedure
```

The application stages have different complexities. The first stage identifies neighboring atoms. There are different implementation alternatives: i) computing the 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 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.

The application typically uses HPC nodes as the execution environment, and NumPy arrays (execution view). It uses matrices to represent the physical system and the distance matrix. The output data representation is a graph. Leaflet Finder can be efficiently implemented using the MapReduce abstraction (problem architecture view). It uses graph algorithms and linear algebra kernels (processing view facets). The data source & style view facets are the same as the PSA algorithm.

2.2 CPPTraj

CPPTraj [31, 32] is a C++ MD trajectory analysis tool. It is parallelized via MPI and OpenMP. CPPtraj reads in parallel frames from a single trajectory file or ensemble members of the same trajectory from different files. The frames are equally distributed to the MPI processes. Computational intensive algorithms are further parallelized using OpenMP. It requires at least one MPI process per ensemble member, where each member is a single trajectory file.

3 BACKGROUND OF EVALUATED FRAMEWORKS

The landscape of frameworks for data-intensive applications is manifold [14, 17] and has been extensively studied in the context of scientific [13] applications. In this section, we investigate the suitability of frameworks such as Spark [43], Dask [30] and RADICAL-Pilot [25], for MD data analytics.

MapReduce [7] and its open source Hadoop implementation combined a simple functional API with a powerful distributed computing engine that exploits data locality to allow optimized I/O performance. However, MapReduce is inefficient for interactive workloads and iterative machine learning algorithms [8, 43]. Spark and Dask provide richer APIs, caching and other capabilities critical for analytics applications. Spark is considered the standard solution for iterative data-parallel applications. Dask is quickly gaining support in the scientific community, since it offers a fully python environment. RADICAL-Pilot is a Pilot-Job framework [20] that supports task-level parallelism on HPC resources. It successfully adds a parallelization level on top of HPC MPI-based applications.

As described in [14], these frameworks typically comprise of distinct layers, e. g.,cluster scheduler access, framework-level scheduling, and higher-level abstractions. On top of low-level resource management capabilities various higher-level abstractions can be provided, e. g., MapReduce-inspired APIs. These then provide the foundation for analytics abstractions, such as Dataframes, Datasets and Arrays. Figure 1 visualizes the different components of RADICAL-Pilot, Spark and Dask. The following describes each framework in detail.

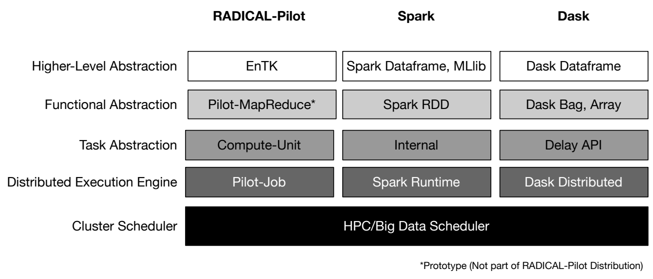

Figure 1: Architecture of RADICAL-Pilot, Spark and Dask: The frameworks share common architectural components for managing cluster resource, and tasks. Spark, Dask offer several high-level abstractions inspired by MapReduce.

3.1 Spark

Spark [43] extends MapReduce [7] providing a rich set of operations on top of the Resilient Distributed Dataset (RDD) abstraction [42]. RDDs are cached in-memory making Spark well suitable for iterative applications that need to cache a set of data across multiple stages. PySpark provides a Python API to Spark.

A Spark job is compiled of multiple stages; a stage is a set of parallel tasks executed without communicating (e. g., `map`) and an action (e. g., `reduce`). Each action defines new stage. The `DAGScheduler` is responsible for translating the workflow specified via RDD transformations and actions to an execution plan. Spark's distributed execution engine handles the low-level details of task execution. The execution of a Spark job is triggered by actions. Spark can read data from different sources, such as HDFS, blob storage, parallel and local filesystems. While Spark caches loaded data in memory, it offloads to disk when an executor does not have enough free memory to hold all the data of its partition. Persisted RDDs remain in memory, unless specified to use the disk either complementary or as a 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 an executor. Finally, Spark provides a set of actions that write text files, Hadoop sequence files or object files to local filesystems, HDFS or any filesystem that supports Hadoop. In addition, Spark supports higher-level data abstractions for processing structured data, such as dataframes, Spark-SQL, datasets, and data streams.

3.2 Dask

Dask [30] provides a Python-based parallel computing library, which is designed to parallelize native Python code written for NumPy and Pandas. In contrast to Spark, Dask also provides a lower-level task API (`delayed` API) that allows users to construct arbitrary workflow graphs. Being written in Python, it does not require to translate data types from one language to another like PySpark, which moves data between Python's interpreter and Java/Scala.

In addition to the low-level task API, Dask offers three higher-level abstractions: Bags, Arrays and Dataframes. Dask Arrays are a collection of 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.

Furthermore, 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 via threads and 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.

Dask's learning curve cannot be considered steep. Its API is well defined and documented. In addition, familiarity with Spark or MapReduce helps to minimize the learning curve even further. As a result, implementing MD analysis algorithms on Dask did not require significant engineering time. In addition, setting up a Dask cluster on a set of resources was relatively straightforward, since it provides all the binaries, e.g. (dask-ssh).

3.3 RADICAL-Pilot

RADICAL-Pilot [25] is a Pilot system that implements the pilot paradigm as outlined in Ref. [39]. RADICAL-Pilot (RP) is implemented in Python and provides a well defined API and usage modes. Although RP is vehicle for research in scalable computing, it also supports production grade science. Currently, it is being used by applications drawn from diverse domains, ranging from earth and biomolecular sciences to high-energy physics. RP can be used as a runtime system by workflow or workload management systems [3, 5, 35, 37, 38]. In 2017, RP was used to support more than 100M core-hours on US DOE, NSF resources (BlueWaters and XSEDE), and European supercomputers (Archer and SuperMUC).

RADICAL-Pilot allows concurrent task execution on HPC resources. The user defines a set of Compute-Units (CU) - the abstraction that defines a task along with its dependencies - which are submitted to RADICAL-Pilot. RADICAL-Pilot schedules these CUs to be executed under the acquired resources. It uses the existing environment of the resource to execute tasks. Any data communication between tasks is done via an underlying shared filesystem, e.g., Lustre. Task execution coordination and communication is done through a database (MongoDB).

RADICAL-Pilot's learning curve can be quite steep at the beginning, at least until the user becomes familiar with the concept and usability of Pilots and CUs. Once the user is comfortable with RADICAL-Pilot's API, she can easily develop new algorithms.

	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 [3]	Dataframe, ML Pipeline, MLlib [23]	Dataframe, Arrays for block computations
Resource Management	Pilot-Job	Spark Execution Engines	Dask Distributed Scheduler
Scheduler	Individual Tasks	Stage-oriented DAG	DAG
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 1: Frameworks Comparison: Dask and Spark are designed for data-related task, while RADICAL-Pilot focuses on compute-intensive tasks.

3.4 Comparison of Frameworks

Table 1 summarizes the properties of these frameworks with respect to abstractions and runtime provided to create and execute parallel data applications.

API and Abstractions. RADICAL-Pilot provides a low-level API for executing tasks onto resources. While this API can be used to implement high-level capabilities, e. g. MapReduce [21], they are not provided out-of-the box. Both Spark and Dask provide such capabilities. Dask's API is generally lower level than Spark's , e. g., it allows specifying arbitrary task graphs. Although, data partition size is automatically decided, in many cases it is necessary to tune parallelism by specifying the number of partitions.

Another important aspect is the availability of high-level abstractions. High-level tools for RADICAL-Pilot, such as Ensemble Toolkit [3], are designed for workflows involving compute-intensive tasks. Spark and Dask already offer a set of high-level data-oriented abstractions, such as Dataframes.

Scheduling. Both Spark and Dask create a Direct Acyclic Graph (DAG) based on operations over data, which is then executed using their execution engine. Spark jobs are separated into stages. When a stage is completed, the scheduler executes 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 execution. Dask tasks are executed when their dependencies are satisfied, starting from leaf tasks. When a task is reached with unsatisfied dependencies, the scheduler executes the dependent task first. Dask's scheduler does not rely on synchronization points that Spark's stage-oriented scheduler introduces. RADICAL-Pilot does provide a DAG and requires the execution order and synchronization to be specified by the user.

Suitability for MDAnalysis Algorithms. Trajectory analysis methods are often embarrassingly parallel. So, they are ideally suited for task management and MapReduce APIs. PSA-like 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 be expressed as a bag of tasks using a task management API or a map-only application in a MapReduce-style API.

Leaflet Finder is more complex and requires two stages: a) the edge discovery stage, and b) the connected components stage. It is possible to implement Leaflet Finder with a simple task-management API, although the MapReduce programming model allows more efficient implementation with a `map` for computing and filtering distances and a `reduce` for finding the components. The shuffling required between map and reduce is medium as the number of edges is a fraction of the input data.

4 EXPERIMENTS AND DISCUSSION

In this section, we characterize the performance of RADICAL-Pilot, Spark and Dask compared to MPI4py. In section 4.1 we evaluate the task throughput using a synthetic workload. In sections 4.2 and 4.3 we evaluate the performance of two algorithms from MDAnalysis: PSA and Leaflet Finder using different real-world datasets. We investigate: 1) which capabilities and abstractions of the frameworks are needed to efficiently express these algorithms, 2) what architectural approaches can be used to implement these algorithms with these frameworks, and 3) 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 has 24 Haswell hyper-threading enabled cores/node and 128 GB memory/node (120 nodes). Experiments were carried using RADICAL-Pilot and Pilot-Spark [19] extension, which allows to efficiently manage Spark on HPC resources through a common resource management API. We utilize a set of custom scripts to start the Dask cluster. We used RADICAL-Pilot 0.46.3, Spark 2.2.0, Dask 0.14.1 and Distributed 1.16.3. The data presented are means over multiple runs; error bars represent the standard deviation of the sample. We employed up to 10 nodes in Comet and Wrangler.

4.1 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 simultaneously. RADICAL-Pilot's backend database was running on the same node to avoid large 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 by the Distributed scheduler. We used Wrangler and Comet for this experiment.

Figure 2 shows the results. 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 as the number of tasks increased. RADICAL-Pilot showed the worst throughput and scalability mainly due to some architectural limitations. It relies on a MongoDB to communicate between Client and Agent, as well as several components that allow RADICAL-Pilot to move data and introduce delays in the execution of the tasks. Thus, we were not able to scale RADICAL-Pilot to 32k or more tasks.

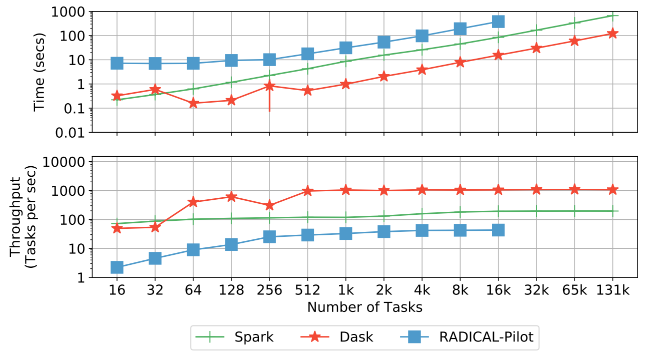

Figure 2: Task Throughput by Framework (Single Node): Time/Throughput executing a given number of zero-workload tasks on Wrangler. Dask performs best; Dask and Spark have very small delays for few tasks. RADICAL-Pilot offers the smallest throughput.

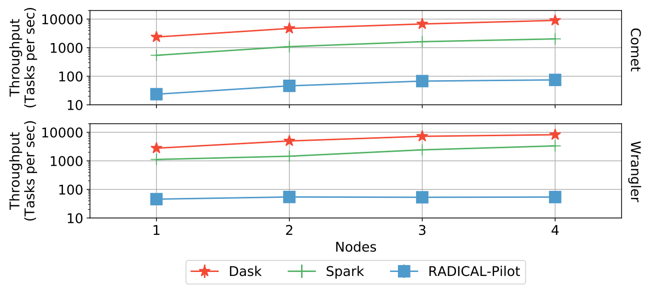

Figure 3: Task Throughput by Framework (Multiple Nodes): Task throughput for $100k$ zero-workload tasks on different numbers of nodes for each framework. Dask has the largest throughput, followed by Spark and RADICAL-Pilot.

Figure 3 illustrates the throughput when scaling to multiple nodes measured by submitting $100k$ tasks. 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 $100 task/sec$. Wrangler and Comet show a comparable performance with Comet slightly outperforming Wrangler.

4.2 Path Similarity Analysis: Hausdorff Distance

The PSA algorithm is embarrassingly parallel and can be implemented 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 executed in a `map` function. In Dask, the tasks are defined as `delayed` functions. In MPI, each task is executed by a process.

The experiments were executed on Comet and Wrangler. The dataset used consists of three different atom count trajectories: small (3341 atoms/frame), medium (6682 atoms/frame) and large (13364 atoms/frame), and 102 frames. We used 128 and 256 trajectories of each size.

Figure 4 shows the runtime for 128 and 256 trajectories on Wrangler. Figure 5 compares the execution times on Comet and Wrangler for 128 large trajectories. We see that the frameworks have similar performance on both systems. Furthermore, we see that Wrangler gives smaller speedup than Comet. Although, we used the same number of cores, we see that utilizing half the nodes due to hyper-threading results to smaller speedup.

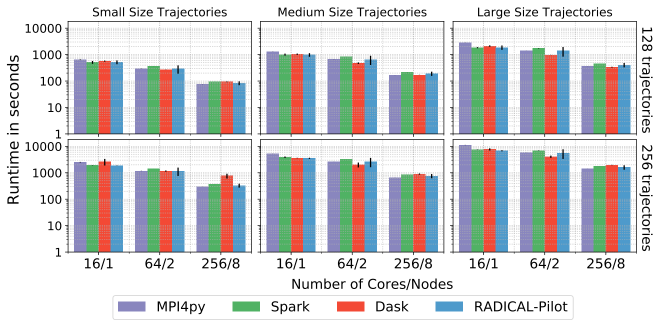

Figure 4: Hausdorff Distance on Wrangler using RADICAL-Pilot, Spark and Dask: Runtimes over different number of cores, trajectory sizes, and number of trajectories. All frameworks scaled by a factor of 6 from 16 to 256 cores.

MPI4py, RADICAL-Pilot, Spark and Dask have similar performance when used to execute embarrassingly parallel algorithms. All frameworks achieved similar speedups as the number of cores increased, which are lower than MPI4py. Although, the frameworks' overheads are comparably low in relation to the overall runtime, they were significant to impact their speedup. RADICAL-Pilot's large deviation is due to sensitivity to communication delays with the database. In summary, all three frameworks provide appropriate abstractions and runtime performance, compared to MPI, for embarrassingly parallel algorithms. In this case aspects such as programmability and integrate-ability are more 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, which are based on other languages.

CPPTraj [32] provides an optimized C++ implementation of the 2D-RMSD, which is Algorithm 1 with no $min - max$ operations. The 2D-RMSD between trajectories was executed in parallel. The results were gathered and the Hausdorff distance was calculated. CPPTraj [32] was compiled with GNU C++ compiler and no optimizations, and with Intel's compiler O3 optimization enabled. An experiment was run with 20-core Haswell nodes and 128 small trajectories; number of cores ranging from 1 up to 240. Figure 6 shows the runtimes and speedup. MPI C++ provides lower execution times. However, we are interested in scalable solutions, that may offer worse performance in absolute numbers, but allows easier integration, i.e., less lines of code, and/or less engineering time.

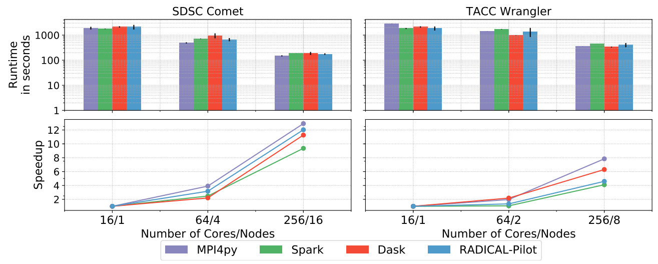

Figure 5: Hausdorff Distance on Comet and Wrangler: Runtime and Speedup for 128 large trajectories.

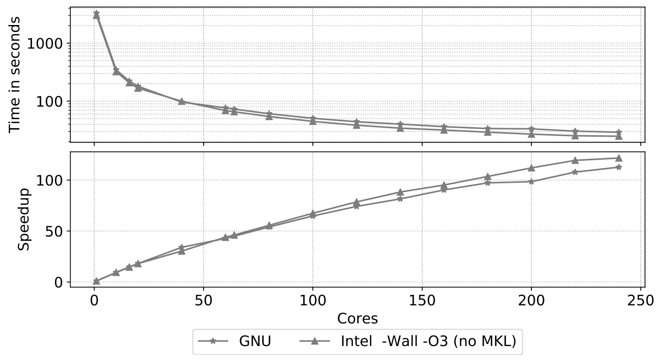

Figure 6: Hausdorff Distance using CPPTraj: Runtimes and Speedup over different number of cores,

4.3 Leaflet Finder

In this section, we investigate four different approaches for implementing the Leaflet Finder algorithm using RADICAL-Pilot, Spark, Dask, and MPI4py (see Table 2):

1) **Broadcast and 1-D Partitioning:** The physical system is broadcast and partitioned through a data abstraction. Use of RDD API (broadcast), Dask Bag API (scatter), and MPI Bcast to distribute data to all nodes. A `map` function calculates the edge list using `cdist` from SciPy [15] – realized as a loop for MPI. The list is collected to the master process (gathered to rank 0) and the connected components are calculated.
2) **Task API and 2-D Partitioning:** Data management is done without using the data-parallel API. The framework is used for task scheduling. Data are pre-partitioned in 2-D partitions and passed to a `map` function that calculates the edge list using `cdist`– realized as a loop for MPI. The list is collected (gathered to rank 0) and the connected components are calculated.
3) **Parallel Connected Components:** Data are managed as in approach 2. Each `map` task performs edge list and connected components computations. The reduce phase joins the calculated components into one, when 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.

	Broadcast and 1-D (Approach 1)	**Task API and 2-D** (Approach 2)	**Parallel Connected Components** (Approach 3)	**Tree-Search** (Approach 4)
Data Partitioning	1D	2D	2D	2D
Map	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	Edge List ($O(E)$)	Edge List ($O(E)$)	Partial Connected components ($O(n)$)	Partial Connected components ($O(n)$)
Reduce	Connect Components	Connected Components	Joined Connected Components	Joined Connected Components

Table 2: MapReduce Operations used by Leaflet Finder

A tree 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 with $896k$, $1.75M$, $3.52M$, and $44.6M$ edges in their graphs. Experimentation was conducted on Wrangler where we utilized up to 256 cores. Data partitioning results into 1024 partitions for each approach, thus 1024 `map` tasks. Due to memory limitations from using `cdist` – uses double precision floating point – Approach 3 data partitioning of the $4M$ atom dataset resulted to $42k$ tasks for both Spark and MPI4py.

Figure 7 shows the runtimes for all datasets for Spark, Dask and MPI4py. RADICAL-Pilot's performance is illustrated in Figure 9. We continue by analyzing the performance of each architectural approach and used framework in detail.

4.3.1 Broadcast and 1-D Partitioning. Approach 1 utilizes a broadcast to distribute the data to all nodes, which is supported by Spark, Dask and MPI. All nodes maintain a complete copy of the dataset. Each `map` task computes 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 for Spark and Dask. MPI broadcast is a fraction of the overall execution time and significantly smaller than Spark and Dask. MPI's broadcast times increase linearly as the number of processes increases, while Spark's and Dask's remain relatively constant for each dataset, due to more elaborate broadcast algorithms compared to MPI. Broadcast times are about 3% – 15% of the edge discovery time for Spark, 40% – 65% for Dask, and < 1% – 10% for MPI4py. Spark offers a more efficient communication subsystem compared to Dask. In addition, Dask broadcast partitions the dataset to a list where each element represents a value from the initial dataset. This did not allow broadcasting the $524k$ atom dataset. Nevertheless, the limited scalability of this approach due to transmitting the entire dataset renders it only usable for small datasets. It shows the worst performance and scaling of all approaches for Spark, Dask and MPI4py.

Furthermore, this approach only scales up to $262k$ atoms for Dask, and $524k$ atoms for Spark and MPI4py on Wrangler. Spark's performance is comparable to MPI4py for the $262k$, and $524k$ datasets. It also shows better performance for the smallest core count in the $524k$ case. Dask is at least two times slower than our MPI implementation.

4.3.2 Task-API and 2-D Partitioning. Approach 2 tries to overcome the limitations of approach 1, especially broadcasting and 1-D partitioning. A 2-D block partitioning is essential, as it evenly distributes the compute and more efficiently utilizes the available memory. 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 Dask's array supports 2-D block partitioning, it was not used for this implementation. We return the adjacency list of the graph instead of an array to fully use the capabilities of the abstraction. Thus, each task works on a 2-D pre-partitioned part of the input data.

Figure 7 shows the runtimes of approach 2 for Spark, Dask, MPI4py and 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. Dask's execution time was smaller by at least a factor of two. However, we were not able to scale this implementation to the 4M dataset, due to memory requirements of `cdist`. For RADICAL-Pilot we observed significant task management overheads (see also section 4.1). 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 become available, i.e. more than 64 cores, the performance improves dramatically.

Furthermore, Spark and Dask did not scale as well as MPI, which achieved linear speedups of ~ 8 when using 256 cores. Spark and Dask achieved maximum speedups of 4.5 and ~ 5 respectively. Despite this fact, both frameworks had similar performance on 32 cores for the $262k$ and $524k$ datasets.

4.3.3 Parallel Connected Components. Communication between the edge discovery and connected components stages is another important aspect. The edge discovery phase output for the $524k$ atoms dataset is ≈100 MB. To reduce the amount of data that need to be shuffled, we refined the algorithm to compute the graph components on the partial dataset in the `map` phase. The partial components are then merged in a `reduce` phase. This reduces the amount of shuffle data by more than 50% (e. g., to 12MB for Spark and 48MB for Dask). Figure 7 shows the improvements in runtime, by $\sim 20\%$ for Spark and Dask, but not MPI4py. Further, we were able to run very large datasets, such as the 4M dataset, using this architectural approach using Spark and MPI4py. Dask was restarting its worker processes because their memory utilization was reaching 95%.

Spark, and Dask have comparable performance with MPI on 32 cores, which utilizes a single node on Wrangler. However, the MPI4py implementation scales almost linearly for all datasets, Spark and Dask cannot, reaching a maximum of ~ 5 for the three smaller datasets. In addition, Spark is able to scale almost linearly for the $4M$ atoms dataset providing comparable performance to MPI4py.

4.3.4 Tree-Search. A bottleneck of approaches 1, 2 and 3 is the edge discovery via the naive 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 [28]. The algorithm: (1) constructs of a tree, and (2) queries for neighboring atoms. Using tree-search, the computational complexity can be reduced from n^2 to log. We use a BallTree as offered by Scikit-Learn [1] for our implementation.

Figure 7 illustrates the performance of the implementation. For small datasets, i. e., $131k$ and $262k$ atoms, approach 3 is faster than the tree-based approach, since the number of points is too small.

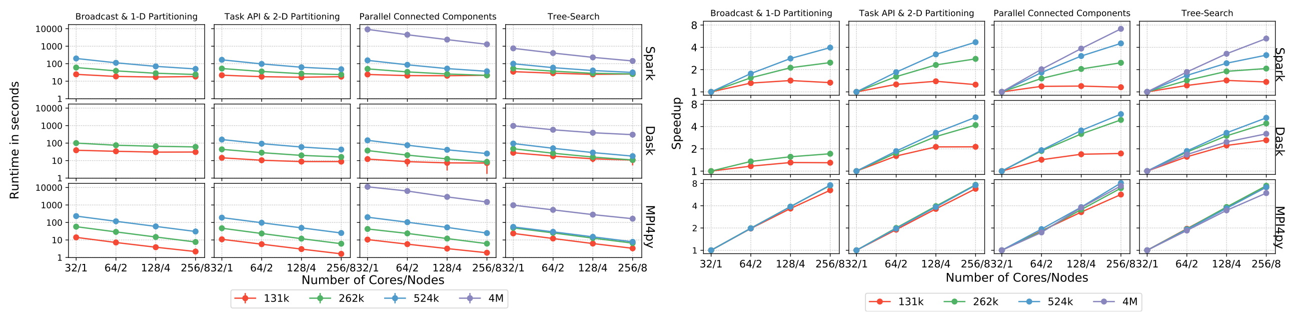

Figure 7: Leaflet Finder: Performance of Different Architectural Approaches for Spark & Dask: Runtimes and Speedups for different system sizes over different number of cores for all approaches and frameworks.

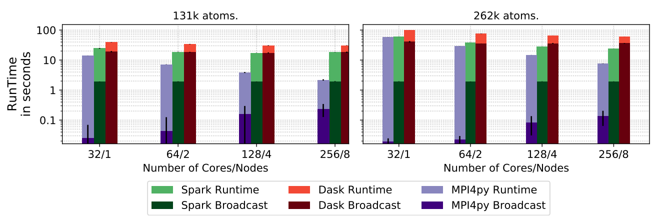

Figure 8: Broadcast and 1-D Partitioned Leaflet Finder (Approach 1): Runtime for multiple system sizes on different number of cores for Spark, Dask and MPI4py.

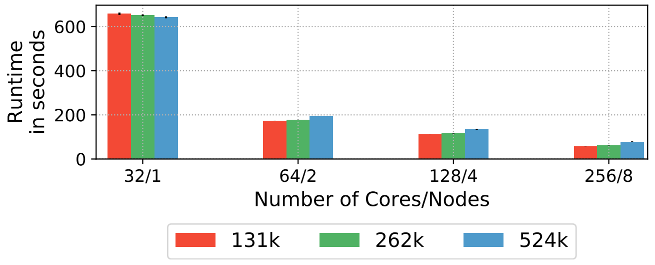

Figure 9: RADICAL-Pilot Task API and 2-D Partitioned Leaflet Finder (Approach 2): Runtime for multiple system sizes over different number of cores. Overheads dominate since execution times are similar despite the system size.

For the large datasets, the tree approach is faster. In addition, the tree has a smaller memory footprint than `cdist`. This allowed to scale to larger problems, e. g., a $4M$ atoms and $44.6M$ edges dataset without changing the total number of tasks.

Dask shows better scaling than Spark for $131k$, $262k$, and $524k$ atoms. This is not true for $4M$ atoms, indicating that Dask's communication layer is not able to scale as well as Spark's. Spark shows similar performance with MPI4py for the largest dataset due to minimal shuffle traffic. Thus, MPI's efficient communication does not become relevant.

4.4 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., compute and I/O). It is important to understand both the properties of the application and Big Data frameworks. Table 3 illustrates the criteria of this conceptual framework and ranks the three frameworks.

	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 3: Decision Framework: Criteria and Ranking for Framework Selection. - : Unsupported or low performance +: Supported, ++: Major Support, and o:Minor support.

4.4.1 *Application Perspective.*

We showed that we can implement MD trajectory data analysis applications using all three frameworks, as well as MPI4py. Implementation aspects, such as computational complexity, and shuffled data size influence the performance greatly. For embarrassingly parallel applications with coarse grained tasks, such as PSA, the choice of the framework does not significantly influence performance (Figures 4 and 5). In addition, the performance difference against MPI4py was not significant (Figures 4 and 5). Thus, 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 (Figures 7, 9). If coupling is introduced, i. e. task communication is required (e. g., reduce), using Spark becomes advantageous (Approaches 3 & 4). MPI4py outperformed Dask, and Spark, despite both frameworks scaling for the larger datasets. Especially Spark was able to provide linear speedup for approach 3 of Leaflet Finder (Figure 7). Integrating with frameworks that provide higher level abstractions provides scalable solutions for more complex algorithms. However, 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.

Dask had the smallest learning curve of all three frameworks. As a result, it allows for faster prototyping compared to RADICAL-Pilot and Spark. RADICAL-Pilot's learning curve is more steep, but is more versatile than Dask and Spark, by offering the lowest level abstraction. Spark had the slowest learning curve. It required tuning to get the number of tasks correctly, as well as argument passing to map and reduce functions.

4.4.2 *Framework Perspective.*

RADICAL-Pilot is well suited for HPC applications, e. g., ensembles (up to 50k tasks) of parallel MPI applications, as shown in Ref. [24, 25]. It has limited scalability when supporting large numbers of short-running tasks, as often found in data-intensive workloads. The file staging implementation of RADICAL-Pilot is not suitable for supporting the data exchange patterns, i.e. shuffling, required for these applications. However, executing MPI and Spark applications alongside on the same resource makes RADICAL-Pilot particularly suitable when different programming models need to be combined.

Dask provides a highly flexible, low-latency task management and excellent support for parallelizing Python libraries. We established that Dask has higher throughput (Figures 2 and 3). However, Spark provides better speedups for the largest datasets compared to Dask (Figure 7). Dask's broadcast (Leaflet Finder approach 1) and shuffle (Leaflet Finder approaches 2- 4) performance is worse for larger problems compared to Spark. Thus, Dask's communication layer shows some weaknesses that are particularly visible during broadcast and shuffle. Spark needs to be particularly considered for shuffle-intensive applications. Its in-memory caching mechanism is particularly suited for iterative algorithms that maintain a static set of data in-memory and conduct multiple passes on that set.

5 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 frameworks are HiMach [36], Pteros 2.0 [41], MDTraj [22], and nMoldyn-3 [11]. We compare these frameworks with our approach over the parallelization techniques used.

HiMach [36] was developed by D. E. Shaw Research group to provide a parallel analysis framework for MD simulations, and 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 [41] 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 via 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 recompiling every time there is a change to the underlying resource, ensuring the application's execution.

MDTraj [22] 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 allows data parallelization on any level of the execution, not only in data read.

nMoldyn-3 [11] parallelizes the execution through a Master Worker architecture. The master 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. These frameworks provide higher level abstractions, e.g machine learning, so any integration with other data analysis methods can be fast and easier. In addition, resource acquisition and management is done transparently.

6 CONCLUSION AND FUTURE WORK

In this paper, we investigated the use of different programming abstractions and frameworks for implementing a range of algorithms for MD trajectory analysis. We conducted an in-depth analysis of applications' characteristics and assessed the architectures of RADICAL-Pilot, Spark and Dask. We provide a conceptual framework that enables application developers to qualitatively evaluate task parallel frameworks with respect to application requirements. Our benchmarks enable them to quantitatively assess framework performance as well as the performance of different implementations. Our method can be used for any application which data are represented as time series of simulated systems, e. g. weather forecast, and earthquakes.

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 has several advantages. It is easier to use and efficiently support common data exchange patterns, e. g. shuffling between `map` and `reduce` stages. In our benchmarks, Spark outperforms Dask in communication -intensive tasks, such as broadcasts and shuffles. Further, the in-memory RDD abstraction performs well for iterative algorithms. Dask provides more versatile low and high level APIs and integrates better with python frameworks. RADICAL-Pilot does not provide a MapReduce API, but is well suited for coarse-grained task-level parallelism [24, 25], and when HPC and analytics frameworks need to be integrated. We also identified a limitation in Dask and Spark: while both frameworks provide some support for linear algebra - both provide a distributed array abstractions - it proved inflexible for an efficient all-pairs pattern implementation. They required workarounds and utilization of out-of-framework functions to read and partition data (Table 2). Although, none of these frameworks outperformed MPI, their scaling capabilities along with their high-level APIs create a strong case on utilizing them for data analytics of HPC applications.

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, optimizing filesystem usage. To better support PyData tools in RADICAL-Pilot, we plan to extend the Pilot-Abstraction to support Dask and other Big Data frameworks. Thus, providing a system that allows MPI simulations along with Big Data frameworks on the same resources. Further, we will refine the RADICAL-Pilot task execution engine to meet the requirement of data analytics applications and create strategies that mitigate issues occurring at large scale, e. g. stragglers. Another area of research is dynamic resource management and 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 for useful discussions. This work is funded by NSF 1443054 and 1440677. Computational resources were provided by NSF XRAC awards TG-MCB090174 and TG-MCB130177.*Software and Data*Source scripts, data and analysis scripts can be found at: Scripts: http://github.com/radical-cybertools/midas, Experiments and Data:http://github.com/radical-experiment/midas-exps/

REFERENCES

[1] 2016. Scikit-Learn: Nearest Neighbors. http://scikit-learn.org/stable/modules/neighbors.html. (2016).

[2] V. Balasubramanian, I. Bethune, A. Shkurti, E. Breitmoser, E. Hruska, C. Clementi, C. Laughton, and S. Jha. 2016. ExTASY: Scalable and flexible coupling of MD simulations and advanced sampling techniques. In *2016 IEEE 12th International Conference on e-Science (e-Science)*. 361–370.

[3] Vivek Balasubramanian, Matteo Turilli, Weiming Hu, Matthieu Lefebvre, Wenjie Lei, Guido Cervone, Jeroen Tromp, and Shantenu Jha. 2018. Harnessing the Power of Many: Extensible Toolkit for Scalable Ensemble Applications. *IPDPS 2018 (accepted)* (2018). https://arxiv.org/abs/1710.08491.

[4] T. Cheatham and D. Roe. 2015. The impact of heterogeneous computing on workflows for biomolecular simulation and analysis. *Computing in Science Engineering* 17, 2 (2015), 30–39.

[5] Jumana Dakka and et al. 2017. High-throughput Binding Affinity Calculations at Extreme Scales. *accepted Computational Approaches for Cancer Workshop, SC'17* (2017). http://arxiv.org/abs/1712.09168.

[6] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. 2005. MPI for Python. *J. Parallel and Distrib. Comput.* 65, 9 (2005), 1108 – 1115.

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

[8] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. 2010. Twister: A Runtime for Iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10)*. ACM, New York, NY, USA, 810–818.

[9] Geoffrey Fox, Judy Qiu, Shantenu Jha, Supun Kamburugamuve, and Andre Luckow. 2015. 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)*. 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Shenzhen, China.

[10] Geoffrey C. Fox, Shantenu Jha, Judy Qiu, and Andre Luckow. 2014. Towards an Understanding of Facets and Exemplars of Big Data Applications. In *Proceedings of Beowulf'14*. ACM, Annapolis, MD, USA.

[11] Konrad Hinsen, Eric Pellegrini, Sławomir Stachura, and Gerald R. Kneller. 2012. nMoldyn 3: Using task farming for a parallel spectroscopy-oriented analysis of molecular dynamics simulations. *Journal of Computational Chemistry* 33, 25 (2012), 2043–2048.

[12] Daniel P. Huttenlocher, Gregory A. Klanderman, and William J Rucklidge. 1993. Comparing images using the Hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 15, 9 (1993), 850–863.

[13] Shantenu Jha, Daniel S. Katz, Andre Luckow, Neil Chue Hong, Omer Rana, and Yogesh Simmhan. 2017. Introducing distributed dynamic data-intensive (D3) science: Understanding applications and infrastructure. *Concurrency and Computation: Practice and Experience* (2017), e4032–n/a. e4032 cpe.4032.

[14] Shantenu Jha, Judy Qiu, André Luckow, Pradeep Kumar Mantha, and Geoffrey Charles Fox. 2014. A Tale of Two Data-Intensive Paradigms: Applications, Abstractions, and Architectures. *Proceedings of 3rd IEEE Internation Congress of Big Data* abs/1403.1528 (2014).

[15] Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001–. SciPy: Open source scientific tools for Python. (2001–).

[16] Supun Kamburugamuve, Geoffrey Fox, Pulasthi Wickramasinghe, Govindarajan Kannan, and Vibhatha Abeykoon. 2018. Twister:Net - Communication Library for Big Data Processing in HPC and Cloud Environments. (03 2018).

[17] Supun Kamburugamuve, Pulasthi Wickramasinghe, Saliya Ekanayake†, and Geoffrey C. Fox. 2017. Anatomy of machine learning algorithm implementations in MPI, Spark, and Flink. In *Technical Report*. Indiana University, Bloomington.

[18] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda. 2016. High-Performance Design of Apache Spark with RDMA and Its Benefits on Various Workloads.

[19] Andre Luckow, Ioannis Paraskevakos, George Chantzialexiou, and Shantenu Jha. 2016. Hadoop on HPC: Integrating Hadoop and Pilot-Based Dynamic Resource Management. *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2016), 1607–1616.

[20] Andre Luckow, Mark Santcroos, Andre Merzky, Ole Weidner, Pradeep Mantha, and Shantenu Jha. 2012. P*: A model of pilot-abstractions. *IEEE 8th International Conference on e-Science* (2012), 1–10. http://dx.doi.org/10.1109/eScience.2012.6404423.

[21] Pradeep Kumar Mantha, Andre Luckow, and Shantenu Jha. 2012. Pilot-MapReduce: an extensible and flexible MapReduce implementation for distributed data. In *Proceedings of third international workshop on MapReduce and its Applications (MapReduce '12)*. ACM, New York, NY, USA, 17–24.

[22] Robert T. McGibbon, Kyle A. Beauchamp, Matthew P. Harrigan, Christoph Klein, Jason M. Swails, Carlos X. Hernández, Christian R. Schwantes, Lee-Ping Wang, Thomas J. Lane, and Vijay S. Pande. 2015. MDTraj: A Modern Open Library for the Analysis of Molecular Dynamics Trajectories. *Biophysical Journal* 109, 8 (2015), 1528 – 1532.

[23] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7.

[24] Andre Merzky, Matteo Turilli, Manuel Maldonado, and Shantenu Jha. 2018. Design and Performance Characterization of RADICAL-Pilot on Titan. *in preparation* (2018). https://arxiv.org/abs/1801.01843.

[25] Andre Merzky, Matteo Turilli, Manuel Maldonado, Mark Santcroos, and Shantenu Jha. 2018. Using Pilot Systems to Execute Many Task Workloads on Supercomputers. (2018). http://arxiv.org/abs/1512.08194.

[26] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. 2011. MDAnalysis: A toolkit for the analysis of molecular dynamics simulations. *Journal of Computational Chemistry* 32, 10 (2011), 2319–2327.

[27] Cameron Mura and Charles E. McAnany. 2014. An introduction to biomolecular simulations and docking. *Molecular Simulation* 40, 10-11 (2014), 732–764.

[28] Stephen M. Omohundro. 1989. *Five Balltree Construction Algorithms*. Technical Report.

[29] 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. 2016. 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.). 98 – 105.

[30] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra (Eds.). 130 – 136.

[31] Daniel R. Roe and III Thomas E. Cheatham. 2013. PTRAJ and CPPTRAJ: Software for Processing and Analysis of Molecular Dynamics Trajectory Data. *Journal of Chemical Theory and Computation* 9, 7 (2013), 3084–3095. PMID: 26583988.

[32] Daniel R. Roe and III Thomas E. Cheatham. 2018. Parallelization of CPPTRAJ Enables Large Scale Analysis of Molecular Dynamics Trajectory Data. *Journal of Computational Chemistry* (2018). in press.

[33] Sean L. Seyler, Avishek Kumar, M. F. Thorpe, and Oliver Beckstein. 2015. Path Similarity Analysis: A Method for Quantifying Macromolecular Pathways. *PLoS Comput Biol* 11, 10 (10 2015), 1–37.

[34] A. A. Taha and A. Hanbury. 2015. An Efficient Algorithm for Calculating the Exact Hausdorff Distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 37, 11 (Nov 2015), 2153–2163.

[35] A. Treikalis, A. Merzky, H. Chen, T. S. Lee, D. M. York, and S. Jha. 2016. RepEx: A Flexible Framework for Scalable Replica Exchange Molecular Dynamics Simulations. In *2016 45th International Conference on Parallel Processing (ICPP)*. 628–637.

[36] Tiankai 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. 2008. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[37] M. Turilli, Y. N. Babuji, A. Merzky, M. T. Ha, M. Wilde, D. S. Katz, and S. Jha. 2017. Evaluating Distributed Execution of Workloads. In *2017 IEEE 13th International Conference on e-Science (e-Science)*. 276–285.

[38] Matteo Turilli, Andre Merzky, Vivek Balasubramanian, and Shantenu Jha. 2018. A Building Blocks Approach towards Domain Specific Workflow Systems? *Short Paper (IEEE/ACM CCGrid 2018)* (2018). http://arxiv.org/abs/1609.03484.

[39] Matteo Turilli, Mark Santcroos, and Shantenu Jha. 2017. A Comprehensive Perspective on Pilot-Jobs. *ACM Computing Surveys (accepted, in press), arXiv preprint arXiv:1508.04180v3* (2017). https://arxiv.org/abs/1508.04180.

[40] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering* 13, 2 (2011), 22–30.

[41] Semen O. Yesylevskyy. 2015. Pteros 2.0: Evolution of the fast parallel molecular analysis library for C++ and python. *Journal of Computational Chemistry* 36, 19 (2015), 1480–1488.

[42] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. 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 (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2.

[43] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10.