Combining HPC and Big Data Infrastructures in Large-Scale Post-Processing of Simulation Data: A Case Study

Yu Li
School of Computing
Clemson University
Clemson, SC
yli25@g.clemson.edu

Xiaohong Zhang
Chemical and Biomolecular Engineering
Clemson University
Clemson, SC
xiaohoz@g.clemson.edu

Ashwin Srinath
Clemson Computing & Information Technology
Clemson University
Clemson, SC
atrikut@g.clemson.edu

Rachel B Getman
Chemical and Biomolecular Engineering
Clemson University
Clemson, SC
rgetman@clemson.edu

Linh B Ngo
Clemson Computing & Information Technology
Clemson University
Clemson, SC
lngo@clemson.edu

ABSTRACT

Advances in scientific software and computing infrastructure have enabled researchers across disciplines to simulate and model highly complex systems. At the same time, these increases in simulation duration and scale have led to significant growths in the sizes of output data, which can be as much as hundreds of gigabytes or more. While there exist solutions to assist with most standard post-simulation analytics, researchers must develop their own code to support customized analytical tasks. Given the nature of these output data, most naive in-house sequential codes end up being inefficient, and in most cases, time-consuming. In this paper, we propose a solution to this issue by transparently combining the strengths of a high-performance computing cluster and a big data infrastructure to support an end-to-end scientific workflow. More specifically, we present a case study around the design of a research computing environment at Clemson University where these two computing systems are integrated and accessible from one another. This environment allows simulation data to be automatically transferred across systems and complex analytical tasks on these data to be developed using the Hadoop/Spark frameworks. Results show that a hybrid workflow for molecular dynamics simulation can provide significant performance improvements over a traditional workflow. Furthermore, code complexity of Hadoop/Spark solutions is shown to be less than that of a traditional solution.

CCS CONCEPTS

• **Computing methodologies → Simulation evaluation**;

KEYWORDS

HPC, Apache Hadoop, Apache Spark, Big Data, Molecular Dynamics Simulation

ACM Reference Format:
Y. Li, X. Zhang, A. Srinath, R. B. Getman, and L. B. Ngo. 2018. Combining HPC and Big Data Infrastructures in Large-Scale Post-Processing of Simulation Data: A Case Study. In *PEARC '18: Practice and Experience in Advanced Research Computing, July 22–26, 2018, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3219104.3229279

PEARC '18, July 22–26, 2018, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-6446-1/18/07…$15.00
https://doi.org/10.1145/3219104.3229279

1 INTRODUCTION

Since the advent of computers, computational methods have been used to solve problems across disciplines in science, engineering, and economics, among others. Such numerical methods can help researchers better predict the behavior of complex systems using computer simulations where experiments are not available or difficult to implement. For example, computational methods can be used to find new materials or predict the reaction pathways by studying the chemical reactions [1]. Most of the chemistry simulations can be performed using open source tools or commercial software, such as LAMMPS [10] and VASP [6], and are usually simulated through high-performance computing cluster. Since the simulation data are stored on network file systems, it is slow to query and post-process. Methods for these data analysis are done in R, Python, Matlab, etc., usually in sequential format. With the improvement of computing capability, larger data set is generated and the sequential code for the data analysis is insufficient and inefficient. Therefore, how to efficiently analyze the large data set is a key problem in speeding up the research progress.

Apache Hadoop/Spark is a powerful platform for parallel and distributed computing with a MapReduce programming model that can be used for intensive data analysis[5]. One challenge in using Hadoop/Spark for scientific researchers is how to easily integrate user's own sequential code with this platform. In this work, we demonstrate a hybrid workflow that combines a high-performance computing cluster to generate results from molecular dynamics

simulations and a Hadoop/Spark cluster to accelerate the processing of the resulting data. We show how existing, sequential, post-processing scripts written in Python can be integrated into a big data framework built using Spark [14]. This enables users to run queries on and post-process simulation data conveniently using Jupyter Notebook via the JupyterHub multi-user notebook server for Clemson University's Palmetto and Cypress cluster[4]. The remainder of this paper is structured as follows. In section 2, we describe the motivating scientific problem. The case study section starts with a description of Clemson University's research computing environment in subsection 3.1. The original sequential data processing code is shown in subsection 3.2. Next, subsection 3.3.1 illustrates an initial attempt to optimize this code using a popular open-source library. Finally, subsection 3.3.2 goes into details of the hybrid workflow that enables the integration of compute-intensive and data-intensive computing environments. The performance evaluation is discussed in section 4. Section 5 concludes the paper.

2 PROBLEM DESCRIPTION

The hydrogen bond (HB) is an important phenomenon in many research areas including chemistry, material science, biology, etc. It is an intermolecular interaction between two molecules, such as water-water, water-alcohols and water-protein, or an intramolecular interaction within one molecule. HB is directional and sufficiently strong that can determine the molecular conformation [12]. The development of computational modeling approach has attracted many attentions to study this phenomenon using simulation methods. A chemical engineering research lab at Clemson University has been studying aqueous phase chemical reactions using molecular dynamics simulation, where HB has been shown to be very important [3, 15].

The motivating scientific problem for this work studies the intermolecular HB between water (H_2O) and methanol decomposition intermediate (CH_2OH^*). Fig.1 shows an example of HB definition in this scenario. The molecular dynamics simulation was carried out using an open source tool called Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [10]. In this simulation, water molecules were allowed to move around which might form HB with the intermediate. After the simulation was finished, a trajectory file was generated. This file includes hundreds of thousands of frames, and each frame contains the particle coordinates at a certain time from which the HB criteria is checked (the criteria includes distance, angle, etc.). We calculated the HB number of each frame and then obtained the averaged HB (AvgHB) from all of the frames. A file with size of 30 GB and frame number of 4.5 million would take about 40 hours to finish using a sequential user code on one CPU core. It is inefficient and nearly intractable to process a larger file.

One thing to notice is that the HB calculation in each frame is independent of other frames, which means that we do not have to wait for the previous frame results in order to process next frames (sequential processing). Therefore, it is possible to develop a parallel method that can process each frame at the same time. In the next section, we use HB calculation as an example to discuss how the initial sequential user code has been developed and evolved and how this code can be transformed to work with the Hadoop/Spark infrastructure for large simulation data parallel analysis.

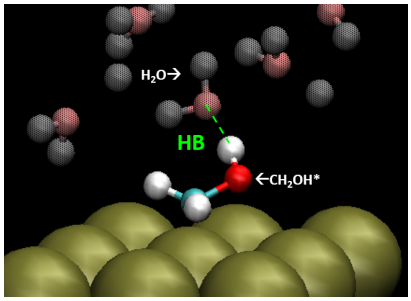

Figure 1: An example of HB definition

3 CASE STUDY

3.1 Research Computing Environment at Clemson

Clemson's computing resources include a high performance computing cluster, known as Palmetto, and a Big Data analytic cluster running Hadoop, known as Cypress. To promote access to Palmetto and Cypress, Clemson University's CCIT has deployed a web-based interface that utilizes Python's Jupyter Server. Through this interface, users can quickly request computing resources, create programming notebooks for languages such as Matlab, Python, and R, and launch terminal all inside a web browser.

The Palmetto cluster is designed and deployed using a traditional Beowulf model [2], which include a login component, a computing component consisting of 2021 compute nodes, and several large-scale storage systems including parallel and distributed file systems. As Palmetto follows a condominium funding model, the entire cluster is heterogeneous with multiple node phases, each of which represents an accumulation of new computing nodes. The network interconnecting across the cluster also differs, with the earlier phases using myrinet and 1-gigabit Ethernet connections, and the later phases using 10-gigabit and infiniband connections. Since Palmetto's target applications are traditional CPU-intensive scientific applications, the hardware configurations of individual compute nodes overtime emphasize memory and CPU rather than local storage. As a result, all Palmetto's phases prior to 2016 have less than half a terabyte of local disks.

With the gradual maturity of the Hadoop ecosystem for Big Data analytics [9], researchers at Clemson University have begun to explore this framework. At the beginning, a dynamic deployment of Hadoop [7] using Palmetto was used. However, given the limited local disks on the compute nodes, this deployment model is limited in term of how much data can be supported. Furthermore, the movement of data between a dynamic Hadoop Distributed File System (HDFS) and a traditional Linux-based file system also takes up valuable scheduling time and network bandwidth, inadvertently defeat the purpose of Hadoop, which is to promote data locality. As a result, in 2015, Clemson University began to invest in Cypress, a dedicated computing resource to support Big Data analytics using Hadoop. The Cypress cluster uses the Hortonworks Data Platform

distribution of Hadoop which include components such as MapReduce and Spark to support data intensive computing and analytics. Cypress consists of 40 worker nodes, each has 256 GB of RAM. Out of these 40 nodes, 16 nodes have 12 1-terabyte local disks each, and 24 nodes have 24 6-terabyte local disks each. In total, the global Hadoop Distributed File System (HDFS) has up to 3.64 PB (petabyte) of storage available for big data analytic tasks.

Between 2015 and 2016, even though Cypress and Palmetto reside on the same subnet, each functions independently and only share the large-scale storage systems. These file systems provide a common zone for users who want to move data between Cypress and Palmetto. As a result, Cypress is usable only for a small number of applications that needs to analyze pre-existing massive amount of data. For applications running on Palmetto that generate massive amount of data, users will have to develop a separate workflow to move these data over to Cypress for post-processing analytic. To expedite this process, in 2017, the Clemson University's Cyberinfrastructure and Technology Integration (CITI) group has managed to setup shared Hadoop libraries and security on Palmetto. This enables Palmetto's compute nodes to interact with Cypress' HDFS and YARN scheduler, allowing researchers to design and deploy complex workflows that mix high performance computing and big data analytics.

3.2 Original User Code

The original sequential user code to analyze HB was developed by a chemical engineering Ph.D. student [13] in a research lab at Clemson University and has been used by the lab in its simulation workflow since then. It is written in Python, and originally can only calculate HB of a single frame (illustrated in Fig. 2) file on disk. To process multi frames file, users have to extract the frames that are needed from the trajectory file. The original code have gone through three stages of evolution to address the problem of involving more frames in the calculation to get better statistical data.

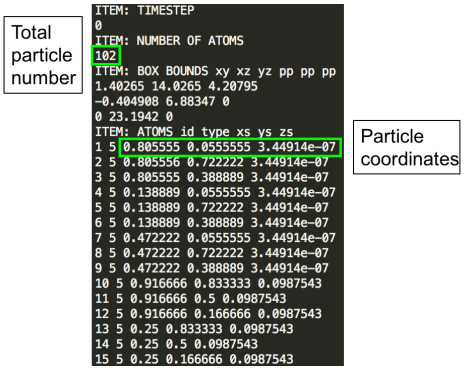

Figure 2: An example of a single frame from trajectory file

A. Analyzing HB of a few frames: Previous research only studied HB information from a small number of frames [3]. Thus the original Python code was used to analyze 10 separate frames extracted from the trajectory file. The code was executed 10 times to get the HB information from each frame and then AvgHB was obtained. Each calculation took only several seconds to finish and the total execution time was not long. Later on, the research needed better sampling where the 10 frames sampling was far away from sufficient. If this naive extracting-executing method was used, more single-frame should be extracted and the code had to be executed multiple times which was very inefficient.

B. Intense reading/writing from disk: In order to obtain more samplings and improve the efficiency, the original Python code was modified in order to avoid extracting each frame explicitly and executing the code repetitively. At this stage, the code was executed only once by sequentially writing a temporary single frame file to the disk for HB analysis. When the analysis on the first frame was finished, the temporary frame file was deleted and the second frame was written to the disk and analyzed until the last frame. When a small trajectory file (3 MB) was tested, it took several minutes to finish the analysis on all of the frames. But when testing on a larger trajectory file (4.5 GB), this method was not feasible, since there were massive amount of I/O operations during the analysis that was very time-consuming.

C. Reading from memory: In order to solve the intensive I/O problem, the code was modified again so that each frame was read into memory at this stage instead of to the disk. A file with size of 10 GB was tested but the memory usage exceeded 120 GB. In this case, the whole file could not be read into memory at one time and had to be read iteratively through frame blocks, where each block contained around 3000 frames. After the analysis on the first block was finished, the code moved to the second block until all frame blocks were analyzed.

After the third evolution, the original Python code has reached a point where it is possible to analyze consecutive frames at scale in research production [15]. However, the execution time on the 10 GB file still took approximately 12 hours to finish on Phase 12 on the Palmetto cluster, which includes compute nodes with 24-core CPU and 128GB of memory where only one CPU core was used by this sequential user code. A larger file will not be tractable using this sequential code. Therefore, a better approach for the big data file analysis is needed.

3.3 Optimizing User Code

To address the I/O issue previously discussed, we examine two approaches. The first approach uses an existing public framework written in Python, and the second approach requires the development of a post-processing data pipeline that leverages Hadoop and Spark for big data analytics.

3.3.1 MDTraj Pipeline. MDTraj is a modern, open library for the analysis of molecular dynamics trajectories [8]. MDTraj comes with a built-in function that supports efficient loading of LAMMPS trajectory files into memory. The library also has a number of existing template functions for HB calculation. In examining MDTraj, we want to see if this library can be used in our HB calculation and whether the run time could be faster than our original user code.

Although MDTraj has its own HB calculation functions, it is developed to be used in a different simulation system (biology system). To use this library in our case, we need to build an additional topology file while loading the trajectory file, which is not required when using the original user code. Besides, the HB criteria (bond,

angle, etc.) is a little bit different from our case. In MDTraj, the code studies the intramolecular HB while our research studies the intermolecular HB. Thus, we need to modify some functions to meet our requirements. The calculation function in MDTraj is called Baker-Hubbard which can identify the intramolecular HB. We modified this function and several other functions called by Baker-Hubbard function to make sure that the HB criteria was the same as that in our case. After a lot of modifications to MDTraj code, we finally got the same result as the original user code, which provided a comparable basis for performance evaluation purposes.

For our baseline test data, which is 6.3 megabytes, MDTraj finished the job in 17 minutes, while the user code can do it in 18 seconds. For another one-gigabyte test data, the user code completed the job in around 25 minutes but MDTraj finished it in 25 hours. There are a number of possible reasons to explain why MDTraj is slower than the original code. Firstly, the way MDTraj calculates the bond information creates redundant data. The Baker-Hubbard function computes all possible HB pairs among all particles which took most of the computation time in the program. In the sequential user code, only the needed HB pairs are classified and stored in the calculation which is more efficient than MDTraj program. Secondly, the way we modified the MDTraj program may affect its original performance to some extent.

At this point in the investigation process, we have come to a conclusion that MDTraj is a more general method for the analysis of molecular dynamics trajectories, and it would take a great effort to modify and optimize MDTraj code to suit our existing workflow. By doing so, we also risk making MDTraj code become more similar to our own code and loosing MDTraj's existing optimization, which defeats the purpose of using MDTraj.

3.3.2 Big Data Pipeline. The main reason for our long analytic time is the single-threaded implementation of the calculations in both the original code and the MDTraj code. While this implementation works for small dataset, it has become too slow to process large-scale simulation results. To parallelize the process of calculating HB across multiple frames, we have decided to use Apache Spark, a big data analytic framework [14] that has been proven to be capable of rapidly processing big data and is faster than Hadoop MapReduce [11]. The dataset model of Spark, called resilient distributed datasets (RDDs), is a read-only collection of records and is partitioned across multiple computing nodes to enable parallel processing of individual records. This concept lends itself nicely to the structure of LAMMPS simulation results, where a simulation file consists of multiple segments. Each segment represents a single simulation frame. On the other hand, in using Spark, there is a chance that significant efforts are needed as we will need to develop new code for loading LAMMPS simulation data into Spark and implementing HB calculation using Spark's MapReduce-based operations. The Spark framework is available as a component of Cypress, the Hadoop cluster. To minimize the learning curve for the development of the new code, the Python API of Spark is used instead of the native Scala interface.

At a glance, loading LAMMPS simulation data into Spark seems to be a daunting task, as we need to separate adjacent frame segments consisting of multiple lines into individual records of a RDD. However, due to the structured attributes of the simulation data (each frame has exactly the same structure to describe the bounding box and the positions of the atoms), this turns out to be fairly straight forward. We find that the first line of each frame can be a unique seperator to distinguish one frame. Based on this idea, a default Hadoop library called *TextInputFormat* can be used to instruct Spark to automatically extract and store each simulation frame, which is seperated by the same delimiter, into individual records of a RDD. In this case, the phrase 'ITEM: TIMESTEP', which appears only once at the beginning of each simulation frame, is used as the delimiter. The code of loading LAMMPS file is:

```
read_rdd = sc.newAPIHadoopFile(
   indirectory + "/ch2oh.1g.lammpstrj",
   'org.apache.hadoop.mapreduce.lib.input.TextInputFormat',
   'org.apache.hadoop.io.LongWritable',
   'org.apache.hadoop.io.Text',
   conf = {'textinputformat.record.delimiter':'ITEM:
        TIMESTEP'})
```

After loading LAMMPS data into Spark, the effort needed for implementing the HB calculation is also fairly minimal. With Spark's MapReduce programming model, the sequential implementation can be reused during the mapping phase. One required modification is the encapsulation of the calculation code into a function whose input matches the input format of a record, which contains the entire collection of text lines for a simulation frame. The function also returns a Key/Value pair structure containing information regarding the results of a frame's HB calculations. The remaining modification to the code is the implementation of a reduce function, which gathers the results from the mapping function and calculate the final result. The key Python Spark code showing the whole process is as followings:

```
frames = read_rdd.values().
    filter(lambda x: x != "").
    map(parse_data).
    cache()
avgHB = frames.
    map(lambda x: int(x[1])).
    reduce(lambda x,y:x+y)/frames.count()
```

In the first statement of the above code segment, we first filter out any RDD record containing no simulation information. Next, we apply the *parse_data* function on each individual RDD record. As each RDD record is a single simulation frame, *parse_data* is an almost identical conversion of the original sequential code, which calculate the HB information on a single simulation frame. A minor modification is needed to make *parse_data* returns a complex tuple with the four elements: time step ID, number of HB appearances, list of configuration for each HB appearance, and the overall coordinates of all atoms associated with each HB appearance. The improvements in data processing speed mainly happen during the execution of this statement as the HB calculation process for the frames are inherently parallized via Spark's implementation of the MapReduce programming paradigm. The second statement of the above code segment demonstrates a simple post-processing analysis, in which we calculate the average number of HB appearing throughout the entire simulation duration. To accomplish this, first

we perform a mapping function which extracts the number of HB appearances from each frame, then use a reduce function to add all of these values together. Dividing the results over the total number of frames, which also represents the simulation time steps, will give us the final answer. The overall workflow describing the entire process is shown in Fig.3.

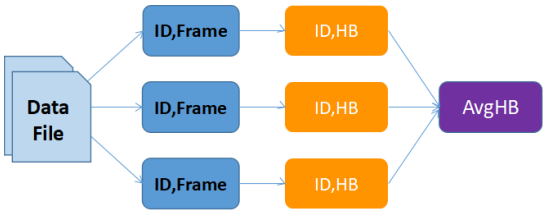

Figure 3: Big Data Pipeline

Once the new Spark code to calculate HB has been developed and tested, the integration of Palmetto and Cypress clusters allows the creation of a big data pipeline. On Palmetto, a job script is first developed to manage the LAMMPS simulation process. Once the simulation is completed, the dump file containing relevant information is directly moved into the Hadoop Distributed File System of Cypress rather than to the scratch file systems. Next, the Spark code is called from Palmetto, which will launch a Spark cluster inside Cypress to process this dump file. The final results are stored inside HDFS, and can be moved back to the *home* directory of the user.

4 EVALUATION AND DISCUSSION

We carried out two experiments to evaluate the speedup of the pipeline. In the first case, we tested the same modeling system but scaling out the number of frames for the trajectory file. In the second case, we scaled up the computation complexity of the modeling system but kept the number of frames fixed for the trajectory file.

Experiment 1: Scaling out

Three trajectory files were generated from a same modeling system, shown in Fig.4, and the frame number for each trajectory file was varied. The file sizes and frame numbers for each trajectory file are listed in Table 1. Since the modeling systems are same for all the trajectory files, the file size is in linear relationship with the frame number.

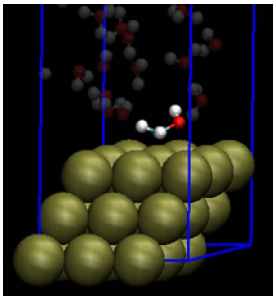

Figure 4: Modeling system for scaling out experiment

Table 1: File information for scaling out experiment

File index	1	2	3
File size	1GB	10GB	30GB
Frame number	150K	1.5M	4.5M

Experiment 2: Scaling up

Four trajectory files were generated with different modeling systems, shown in Fig.5, and the number of frames in each trajectory file was kept the same. The file sizes and frame numbers for each trajectory file are listed in Table 2. Since the modeling systems are different for the trajectory files, the modeling complexity increases from file a to file d, and the HB calculation time for each frame should be different for these files.

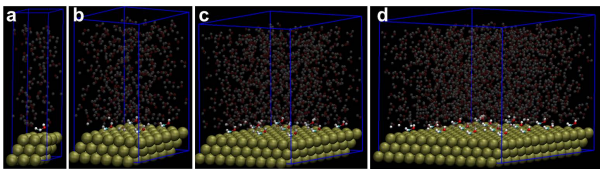

Figure 5: Modeling systems for scaling up experiment

Table 2: File information for scaling up experiment

File index	a	b	c	d
File size	0.5GB	2.4GB	4.8GB	8.6 GB
Frame number	90K	90K	90K	90K

Hardware Configuration:

To evaluate changes in performance of the Spark program, we first created a baseline measurement by running the sequential HB analysis on the above experiments using Palmetto compute nodes from phase 12 or higher. During these runs, we reserved a complete node, which has 24-core CPU and 128GB of memory. It should be noted that due to the sequential nature of the original code, only one core is utilized. For the Spark program, each run was performed with a cluster configuration consisting of one Spark driver with 15GB of memory and six executor instances, each has twelve processing cores and 60GB of memory.

Results

The scaling out experiment results are shown in Fig.6. In this experiment, the modeling systems are the same and thus the run time of the HB calculation for each single frame should be the same for all three trajectory files. The run time of the three HB calculations are about 25 minutes, 13 hours and 41 hours for the baseline runs of the original code (orange bar). Based on these measurements, it seemed that the run time of the sequential code was dependent on the file size/frame number. However, the run times also increased nonlinearly with the file size, suggesting that except for the HB calculation in each frame, extra reading time for large size file was needed. For the Spark program, the run times were about 5 minutes, 7 minutes and 16 min, which represented a

significant speedup (purple bar) comparing to the based line. We also notice that the scaling of the Spark program's run times were closer to a linear form, suggesting that it has the potential to scale linearly as the file size/frame number increases.

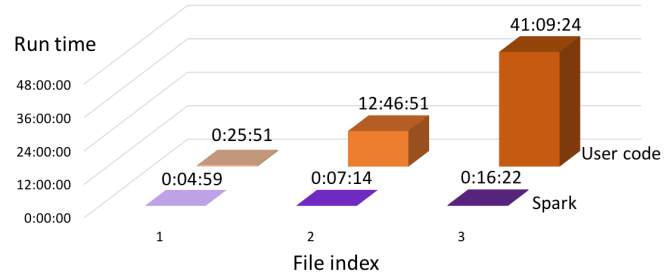

Figure 6: Scaling out experiment speedup

The scaling up experiment results are shown in Fig.7. In this experiment, even though the frame numbers are same for all of the trajectory files, the modeling systems are different and thus the run time of the HB calculation for each single frame should increase from file a to d, depending on the modeling complexity. The run time of the four HB calculations are about 21 minutes, 3.5 hours, 13 hours and 37 hours for user code (orange bar). It is shown that the run time of the user code also depends on the file size which increases nonlinearly with the file size. The run time of these calculations are about 5 minutes, 13 minutes, 26 minutes and 45 minutes for Spark test, which is also a huge speedup (purple bar). The Spark speedup is approximately in a linear relationship with the file size, suggesting that Spark work flow depends a lot on modeling complexity, where in this case complexity is realized through the file size.

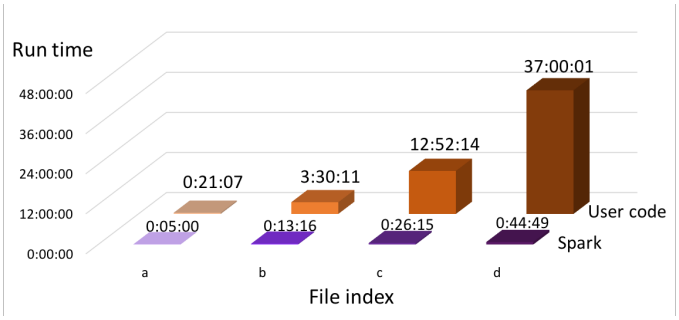

Figure 7: Scaling up experiment speedup

From the above results, Spark can be 4 to 150 times faster than the sequential user code, depending on the speedup scheme. It should be noted that in order to use Spark, we need to move simulation results from Palmetto to Cypress and also to load and convert these results into Spark's RDD. The time of moving the output data from Palmetto to Cypress is measured to range from 13 seconds for 0.5 GB of data to 7 minutes for 30 GB data. To convert the trajectory file on Cypress into Spark'RDD format, it takes 3, 7, 14 minutes in scaling out experiment for file 1, 2 and 3, respectively, and 3, 12, 24, 42 minutes in scaling up experiment for file a, b, c, d, respectively. When we compare this time to the simulation time (Fig. 6 and Fig. 7), we can see that the converting time takes about 90% of the simulation time. As a result, it is reasonable to state that these additional overheads, which does not exist in sequential implementations, are acceptable comparing to the achievable speedup shown in these experiments.

Discussion

In this section, we discuss problems encountered while implementing the big data pipeline and also elaborate on the possible reasons. One problem is how to tune the parameters of the Spark job well to get a better performance. Further understanding of the job configurations on different dataset will help researchers fully understand the configuration functions and relationship. We did not put too much effort on tuning the job configuration since it is time consuming and not the main purpose of this work. However, we are still able to have the desired results. We hypothesize that better performance may be obtained if the configurations are further optimized. Secondly, Jupyter Notebook is an easy interface to carry out Spark simulation from Cypress cluster and in our case, and the simulation time is less than one hour in our tests which is acceptable using Jupyter Notebook. However, if we are running a longer simulation, we will need to develop a bash script to run a Spark job on Cypress.

In general, Spark is easy to use and debug through Jupyter Notebook. By using cache function, which can cache the data in memory, the data analysis becomes faster after the first test since it has several operations on one RDD. The partitioning function is another operation that can help speeding up the program. Coalesce function, a built-in operation of Spark RDD, was used in this work to split the original data into smaller pieces. One can also choose repartition function, but it may be slower since it needs shuffle which costs extra running time.

There are many ways to implement the file readings in Spark framework. If each frame has the same format and length throughout one trajectory file, users can choose a built-in library, *NLinesInputFormat*, from Hadoop in Spark, which reads file N lines by N lines. Even though this library can be used in this work, we want to make this workflow suitable for all the cases by considering potential different kinds of dataset. As a result, the data was read by blocks, using a self-defined text delimiter in this work. Users can also override the *RecordReader* class from Hadoop library in Spark to achieve the same goal. In this case, the *sliding* function of RDD in machine learning library of Spark can be used as well, which can turn the text into array elements and frames can be read into elements with a self-defined offset, which is similar to the *NLinesInputFormat* method.

5 CONCLUSION

Development of computer hardware/software has facilitated computational study in many research areas. The increase in the simulation data size and complexity requires more efficient data processing method, where the existing method may not be efficient to customize user code. In this work, we proposed a solution to this issue by combining a high-performance computing cluster and a big

data infrastructure to support an end-to-end scientific workflow. Taking molecular dynamics simulation as a case study, we integrated the sequential user code with two computing environment at Clemson University, Palmetto cluster (HPC) and Cypress cluster (Hadoop/Spark), and successfully parallelized the big data analysis. Results show that this hybrid workflow can significantly improve the data analysis performance compared to a traditional workflow, and the speedup of the Hadoop/Spark workflow can be as high as 150 times of the sequential user code, suggesting a potential application of Spark in similar research fields. This workflow is easy to use and the user code can be directly used in Spark with slight configurations. Besides, the Jupyter Notebook gives a straightforward interface for users to debug.

ACKNOWLEDGMENTS

This research was funded by the National Science Foundation under grant numbers CBET-1554385. Simulations were performed on the Palmetto Supercomputer Cluster and Cypress Cluster, which are maintained and supoprted by Clemson University's CCIT Department.

REFERENCES

[1] Paul A Bash, Martin J Field, RC Davenport, Gregory A Petsko, D Ringe, and Martin Karplus. 1991. Computer simulation and analysis of the reaction pathway of triosephosphate isomerase. *Biochemistry* 30, 24 (1991), 5826–5832.

[2] Donald J Becker, Thomas Sterling, Daniel Savarese, John E Dorband, Udaya A Ranawak, and Charles V Packer. 1995. BEOWULF: A parallel workstation for scientific computation. In *Proceedings, International Conference on Parallel Processing*, Vol. 95. 11–14.

[3] Cameron J Bodenschatz, Sapna Sarupria, and Rachel B Getman. 2015. Correction to "Molecular-Level Details about Liquid H2O Interactions with CO and Sugar Alcohol Adsorbates on Pt (111) Calculated Using Density Functional Theory and Molecular Dynamics". *The Journal of Physical Chemistry C* 120, 1 (2015), 801–801.

[4] ClemsonCiti. 2018. cypress-pyspark-kernel. (2018). Retrieved May 7, 2018 from https://github.com/clemsonciti/cypress-pyspark-kernel

[5] Max Klein, Rati Sharma, Chris H Bohrer, Cameron M Avelis, and Elijah Roberts. 2017. Biospark: scalable analysis of large numerical datasets from biological simulations and experiments using Hadoop and Spark. *Bioinformatics* 33, 2 (2017), 303–305.

[6] G Kresse. 1996. Software vasp, vienna, 1999; g. kresse, j. furthmüller. *Phys. Rev. B* 54, 11 (1996), 169.

[7] Sriram Krishnan, Mahidhar Tatineni, and Chaitanya Baru. 2011. myHadoop-Hadoop-on-Demand on traditional HPC resources. *San Diego Supercomputer Center Technical Report TR-2011-2, University of California, San Diego* (2011).

[8] 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. https://doi.org/10.1016/j.bpj.2015.08.015

[9] J Yates Monteith, John D McGregor, and John E Ingram. 2013. Hadoop and its evolving ecosystem. In *5th International Workshop on Software Ecosystems (IWSECO 2013)*. 50.

[10] Steve Plimpton. 1995. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics* 117, 1 (1995), 1–19.

[11] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. 2015. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment* 8, 13 (2015), 2110–2121.

[12] Thomas Steiner. 2002. The hydrogen bond in the solid state. *Angewandte Chemie International Edition* 41, 1 (2002), 48–76.

[13] Tianjun Xie. 2018. hbonds. (2018). Retrieved May 3, 2018 from https://github.com/tianjunxie/hbonds

[14] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[15] Xiaohong Zhang, Torrie E Sewell, Brittany Glatz, Sapna Sarupria, and Rachel B Getman. 2017. On the water structure at hydrophobic interfaces and the roles of water on transition-metal catalyzed reactions: A short review. *Catalysis Today* 285 (2017), 57–64.