Large Scale String Analytics in Arkouda

Zhihui Du, Oliver Alvarado Rodriguez, and David A. Bader
Department of Data Science
New Jersey Institute of Technology
Newark, New Jersey, US
{zhihui.du,oaa9,bader}@njit.edu

2021 IEEE High Performance Extreme Computing Conference (HPEC) | 978-1-6654-2369-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/HPEC49654.2021.9622810

***Abstract*—Large scale data sets from the web, social networks, and bioinformatics are widely available and can often be represented by strings and suffix arrays are highly efficient data structures enabling string analysis. But, our personal devices and corresponding exploratory data analysis (EDA) tools cannot handle big data sets beyond the local memory. Arkouda is a framework under early development that brings together the productivity of Python at the user side with the high-performance of Chapel at the server-side. In this paper, an efficient suffix array data structure design and integration method are given first. A suffix array algorithm library integration method instead of one single suffix algorithm is presented to enable runtime performance optimization in Arkouda since different suffix array algorithms may have very different practical performances for strings in various applications. A parallel suffix array construction algorithm framework is given to further exploit hierarchical parallelism on multiple locales in Chapel. A corresponding benchmark is developed to evaluate the feasibility of the provided suffix array integration method and measure the end-to-end performance. Experimental results show that the proposed solution can provide data scientists an easy and efficient method to build suffix arrays with high performance in Python. All our codes are open source and available from GitHub (https://github.com/Bader-Research/arkouda/tree/string-suffix-array-functionality).**

***Index Terms*—exploratory data analysis, large scale string sets, suffix array construction algorithm, Arkouda**

I. Introduction

Suffix trees [11] allow for particularly fast implementations of many important string operations, such as locating a substring, searching the longest common substring, and so on [5]. These speedups come at a cost: storing a string's suffix tree typically requires significantly more space than storing the string itself. The construction of such a tree for a string S of length n takes $O(n)$ in time and a total of $2n$ nodes (n leaves, $n-1$ internal non-root nodes, 1 root) in space.

The suffix array [27] was invented in 1990 by Manber and Myers as a new space-efficient data structure. Suffix arrays with additional tables, such as the longest common prefix (LCP) array, can reproduce the full functionality of suffix trees preserving the same time and memory complexity [3].

Just like suffix trees, suffix arrays can be widely employed to solve many problems that can be modeled as a string processing problem, such as those found in bioinformatics, web information search and analysis, and lossless compression (Burrows-Wheeler transform [16], [26], [34]).

Arkouda [30], [37] is under early development as a framework that brings together the productivity of Python with world-class high-performance computing. It is built on Python and Chapel, a modern parallel processing compiler for high-performance computing solutions. Together, Arkouda+Chapel allows Python-trained programmers to readily use HPC resources, lowering the barrier allowing data scientists to be more productive at solving exploratory data analysis (EDA) problems on large scales.

In this paper, we provide the solution for integrating the suffix array into Arkouda for interactive data science at scale. The major contributions are as follows.

1) An efficient suffix array data structure enabling interactive large string analysis at the Python front-end and high-performance data processing at the Chapel back-end are proposed and developed in Arkouda. The presented data structures are the foundation of high-level and high-performance large-scale string analysis.
2) A suffix array construction algorithm library and its building method in Arkouda are provided. Such a library is necessary for Arkouda to dynamically optimize its performance by selecting suitable suffix array construction algorithms at run-time based on the features of different input strings.
3) All the proposed methods have been implemented and integrated into Arkouda and a corresponding benchmark has been developed to evaluate the end-to-end performance. Experimental results show that the proposed method can build suffix arrays with negligible overhead in Arkouda. This work sets up the basic methods and software tools for interactive string-based data science at scale.

II. Overview of Arkouda

Arkouda is a software package that allows a user to interactively issue massively parallel computations on distributed data using functions and syntax that mimic NumPy and Pandas, the underlying computational libraries used in the vast majority of Python data science work-flows.

To enable exploratory data analysis on large-scale data sets in Python, Arkouda divides its data into two physical sections. The first section is the metadata which only includes attribute information and occupies very little memory space. The second section is the raw data which includes the actual big data sets to be handled by the back-end. Yet, from the view of the Python programmers, all data is directly available just like on their local laptop device. This is why Arkouda can break the limit of local memory capacity, while at the same time bringing traditional laptop users powerful computing capabilities that could only be provided by supercomputers.

The computational heart of Arkouda is a Chapel [7] interpreter that accepts a predefined set of commands from a client (currently implemented in Python) and uses Chapel's built-in machinery for multi-locale and multi-threaded execution to evaluate computations at scale [14] [15]. EDA operations in Arkouda currently scale to hundreds of HPC nodes comprising tens of thousands of cores and hundreds of terabytes of memory.

When users are analyzing their data, if only the metadata section is needed, then the operations can be completed locally. These actions are carried out just like in previous Python data processing workflows. If the operations have to be executed on raw data, the Python program will automatically generate an internal message and send the message to Arkouda's message processing pipeline for external and remote help. Arkouda's message processing center (ZeroMQ) is responsible for exchanging messages between its front-end and back-end. When the Chapel back-end receives the operation command from the front-end, it will execute the analyzing task quickly using HPC resources on the corresponding raw data and return the required information back to the front-end. Through this, Arkouda can support Python users to locally handle, on their devices, large-scale data sets residing on powerful back-end servers without knowing all the detailed operations at the back-end.

The final goal of this research is to support interactive data science at scale. We will focus on the fundamental and parallel algorithm design, development, and integration with Arkouda to enable productive data analytics. Besides suffix array, we will further investigate other basic constructs such as trees, graphs, and matrices as well as their algorithms to support a wider range of applications.

In this paper, we will introduce the method and solution to integrate suffix arrays into Arkouda which is the first step of the research and development road map.

III. Proposed Method

Our solution for integrating suffix arrays into Arkouda is shown in Fig. 1. Arkouda provides a complete framework and mechanism for the message exchange between the front-end and the back-end, so our focus is developing the basic suffix array building blocks and a pipeline to provide a complete solution by connecting the data structures and functionalities together at both the front-end and the back-end.

A. Data structure design and implementation

To enable Python programmers to operate on suffix arrays as if all data are available locally on his/her device, we design our suffix array data structure as shown in Fig. 2.

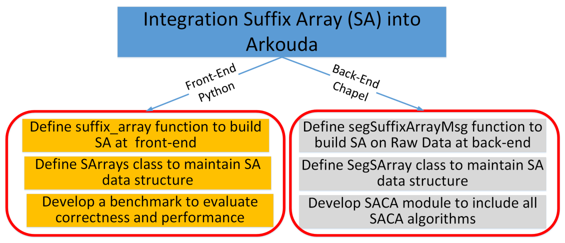
Integration Suffix Array (SA) into Arkouda
Front-End Python
Back-End Chapel
Define suffix_array function to build SA at front-end
Define SArrays class to maintain SA data structure
Develop a benchmark to evaluate correctness and performance
Define segSuffixArrayMsg function to build SA on Raw Data at back-end
Define SegSArray class to maintain SA data structure
Develop SACA module to include all SACA algorithms

Fig. 1. Proposed method to integrate suffix arrays into Arkouda.

The symbol mapping table created and maintained in Chapel is the basic mechanism for mapping the name ID given by Python to the corresponding original data in Chapel. Each entry of the table is a $\langle key, value \rangle$ pair. The *value* part includes all necessary information of an array in Chapel, such as the data type, size, number of dimensions, element size, shape, starting position and distribution of given array. The *key* part is the unique ID of the corresponding array. Given a *key*, we can search in the symbol mapping table to get its *value* and then we will know the raw data represented by the *key*. When a new array is created at the Chapel back-end, to make it available to the Python front-end, we need to build the $\langle key, value \rangle$ pair and insert it into the symbol mapping table. The ID string will be sent to the front-end as a part of a Python data structure. In this way, a simple ID string in the Python front-end is connected with the raw data of the Chapel back-end.

Two classes *Strings* and *SegString* have been defined in Python and Chapel respectively to model a group of strings. Accordingly, we define two classes *SArrays* and *SegSArray* in Python and Chapel respectively to describe the suffix arrays of given strings.

An array is the basic data structure that supports parallel operations in Arkouda. For a group of strings, the raw data are stored in one large array in the Chapel back-end. All strings are stored in the large array one by one without any gap. So the total length of the strings is the length of the large array. This array is called a "value array" of given strings. To access a specific string, another array is built to store the starting position or offset of each string in the value array. This array is called an "offset array". The *SegString* includes the two kinds of arrays to describe a group of strings.

Correspondingly, we define a new class *SegSArray* in Chapel to describe the suffix array data structure of given strings. *SegSArray* also has its offset array and value array that are defined as an *offsets* and a *values SymEntry* instance. When suffix arrays have been built for a group of strings, both their value array and their offset array description information will be added to the symbol mapping table for future access. Given *offsetName* and *valueName*, the *offsets* and *offsets* object instances can be accessed by looking up the symbol mapping table.

Analogous data structures are defined in Python at the front-end. Based on the existing *Strings* class, we define a new class *SArrays* to describe a group of suffix arrays. It is very similar to the *Strings* class except that its element data is a suffix array

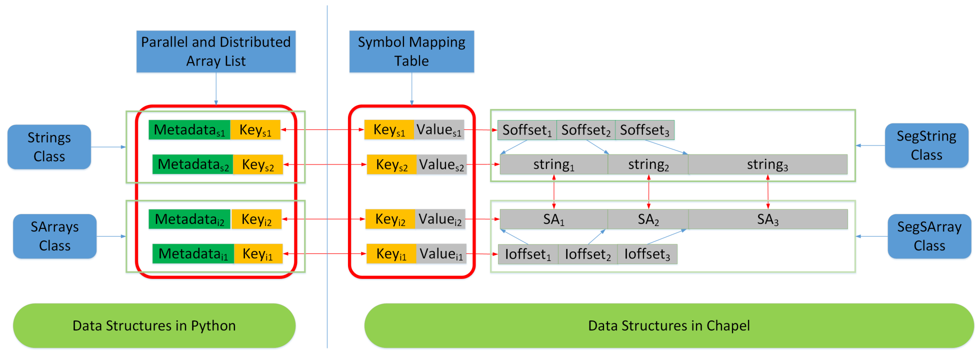
Parallel and Distributed Array List
Symbol Mapping Table
Strings Class
SArrays Class
Metadata_{s1} Key_{s1}
Metadata_{s2} Key_{s2}
Metadata_{i2} Key_{i2}
Metadata_{i1} Key_{i1}
Key_{s1} Value_{s1}
Key_{s2} Value_{s2}
Key_{i2} Value_{i2}
Key_{i1} Value_{i1}
Soffset_1 Soffset_2 Soffset_3
string_1 string_2 string_3
SA_1 SA_2 SA_3
Ioffset_1 Ioffset_2 Ioffset_3
SegString Class
SegSArray Class
Data Structures in Python
Data Structures in Chapel

Fig. 2. Data structure design of suffix array in Arkouda.

instead of a string.

B. Algorithm Design and Integration

After the first suffix array construction algorithm with $O(n \log n)$ time complexity was proposed in 1990 by Manber and Myers [27], it took about thirteen years to achieve the time complexity of $O(n)$ in 2003 [18]–[20]. However, such algorithms need additional $O(n)$ working space during the building of the suffix array. Then, it took another thirteen years to achieve the $O(1)$ working space or in-place and $O(n)$ time complexity algorithm for integer alphabets in 2016 [25]. The suffixes' distribution and the length of different inputs are two important aspects that can affect the practical performance of existing Suffix Array Construction Algorithms (SACAs) and better time complexity does not necessarily mean faster execution time. Antonitio *et al.* discussed the inconsistency between algorithm time complexity and practical execution time [4]. Different parallel SACAs [1], [8]–[10], [21]–[23], [26], [39], [40], [42] have been developed on shared-memory multiprocessors, distributed memory clusters, and many threads GPUs to exploit the HPC technology to further improve the practical performance.

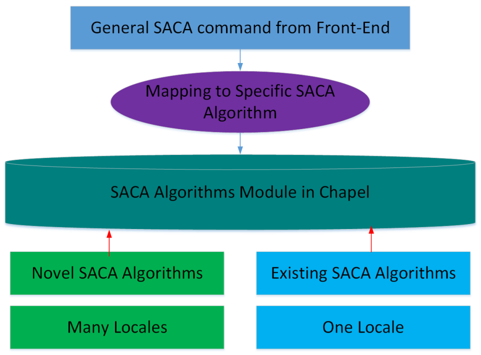
General SACA command from Front-End
Mapping to Specific SACA Algorithm
SACA Algorithms Module in Chapel
Novel SACA Algorithms
Existing SACA Algorithms
Many Locales
One Locale

Fig. 3. Algorithm design and implementation method in Arkouda.

We use Chapel to develop scalable parallel suffix array algorithms to achieve portable performance on different parallel hardware. The basic idea is given in Fig. 3. For a given suffix array request from the Python front-end, our method can select the suitable suffix array construction algorithm based on the available hardware, and the input strings automatically. We need to develop different kinds of suffix array construction algorithms to generate an available algorithm library. Here, we divide the algorithms into two types based on if they can run on many locales (distributed memory systems) in Chapel. We will not only design our novel algorithms but also rewrite some existing excellent algorithms in Chapel to take advantage of the advanced features of Chapel to improve their portability and scalability. At the same time, we can integrate some existing algorithms into Arkouda quickly but, the restriction is that they can only run on one locale.

A Chapel *SACA* module is developed to include all such SACAs. Currently, we have integrated two algorithms into the *SACA* module. One is the self-developed *skew* algorithm [18] in Chapel. The advantage of the *skew* algorithm is that it is conceptually simple and has low $O(n)$ time complexity.

The basic idea of the *skew* algorithm is: (1) Divide the given string into substrings (the length is 3). (2) Recursively sort the $\frac{2}{3}$ part of the substrings. (3) Sort the $\frac{1}{3}$ substrings based on the sorted $\frac{2}{3}$ substrings and finally merge the two parts together to form the sorted suffix array. The recursive sort for the $\frac{2}{3}$ substrings, linear induction algorithm for the $\frac{1}{3}$ substrings based on the sorted $\frac{2}{3}$ substrings and linear merge method are the key for *skew* algorithm to achieve linear time complexity.

The other is an open-source algorithm. We take advantage of Chapel's *C* interoperability to integrate the existing algorithm quickly. Currently, the existing fastest sequential suffix array algorithm is divsufsort [32] (it can support OpenMP, but it is sequential in major.) and it is selected as the first open-source suffix array implementation method. The implementation method involves using "Extern Declarations" to make the *C* codes available for Chapel. Here, we use the explicit instead of implicit strategy to integrate the *C* codes.

C. Suffix array construction pipeline

To integrate the suffix array into Arkouda, we need to add separate suffix array building functions at both the front-end and back-end. The role of the front-end function is to provide the descriptive information of a group of strings and issue the suffix array building command. The role of the back-end function is to locate the position of all the strings, build their suffix arrays, and return the offset and value IDs of built suffix arrays to the front-end.

Based on the *SArrays* class, we can introduce the suffix array building function *suffix_array* based on the given *Strings* object. This function will build suffix arrays for given $strings$ object. The built suffix arrays will reside on the back-end, but returned IDs of the raw data will be kept in the *Pdarray* object as well as other attributes (metadata).

For the *suffix_array* function, we only need to pass the basic information about the *Strings* object, including the data type *strings.objtype*, the offset array ID *strings.offsets.name* that can be used to access the raw offset array at the back-end, and the original strings array (value array) ID *strings.bytes.name* used to access the raw string at the back-end. Then, we compose a message including all the above information and call ZeroMQ to send the message to the back-end. ZeroMQ will return the resulting message to Python, which includes all the newly built suffix arrays descriptive information, and we can use this information to build a *SArrays* object to access the suffix arrays in the future.

There are three big steps to build suffix arrays at the back-end. First, we need to get all the raw data of the given strings. Just as in the definition of class *Strings*, *size* means the total number of strings. *nBytes* means the total length of the strings. *length* is the length of each string and *offsets* is the starting index of each string. We use *startposition* and *endposition* to indicate each string. Second, when we know all the necessary information of each string, we can call the specific suffix array algorithm to build the suffix array of a given string. Different suffix array construction procedures can be executed in parallel. We use the *forall* parallel structure in Chapel to build the suffix array of different strings in parallel. Third, after all the suffix arrays are built, the offset information about all of the suffix arrays is kept in *sasoff* and all the indices of all the suffix arrays are kept in *sasval*. Two symbol entry objects will be created to describe offset array and value array information. Such information will be added to the symbol table to use its string ID to further access the original data. The IDs of the two symbol entry objects will be returned back to the front end.

D. Parallel Suffix Array Construction Algorithm Framework

A parallel algorithm is necessary for very large suffix array construction. To enable large-scale data parallelism, we propose the parallel framework as in Alg. 1.

Let n be the total length of the given string and p be the total number of parallel execution units. For simplicity, we can assume n can be evenly divided by p. The basic idea is we first divide a large string into the same length independent substrings. Then, we can employ any parallel or sequential suffix array algorithm to build p partial suffix arrays. Since no communication is needed at this step, it can achieve linear speedup and it is also very easy to implement.

In the third step, we can employ any parallel multi-way merge algorithm to merge different partial suffix arrays into one complete suffix array. Just as we mentioned before, since the properties of strings, such as the average Longest Common Prefix (LCP) length, can significantly affect the practical performance of a suffix array construction algorithm, this framework is helpful for a specific application to employ the best suitable algorithm to improve its performance.

Algorithm 1: Parallel Suffix Array Construction

```
input : String $S$ of length $n$
output: Suffix Array $SA$
1 Divide $S$ into $p$ substrings $S_0, ..., S_{p-1}$ with length
    $\lceil \frac{n}{p} \rceil$;
2 Employ optimized suffix array construct algorithm on
    each substring in parallel and generate partial suffix
    array $SA_0, ..., SA_{p-1}$;
3 Employ partial suffix array merging on
    $SA_0, ..., SA_{p-1}$ in parallel and generate suffix array
    $SA$;
4 return SA
```

IV. Experimental results

A. Testing method

To evaluate the results of the proposed integrated solution, we develop a simple "sa.py" Python test benchmark. Two kinds of strings are generated as inputs. The first is randomly generated strings and the second is real-world benchmark strings. In order to check the results, we transfer all suffix arrays to the front end (this is not necessary for actual situations). Furthermore, we test the end-to-end suffix arrays generation time to evaluate the performance. All the experiments are executed in a server with an 8-core Intel(R) Xeon(R) E5-2680 2.70GHz CPU. The total memory is 16MB. In a Jupyter Notebook environment, our experiments show that large-scale string analysis can be done in Arkouda easily with high performance.

B. Experimental results

First, the correctness evaluation is done. The workflow is as follows. (1) Generate random strings. (2) Build suffix arrays for the strings. (3) The values of the string and the suffix array are transmitted to the front end for correctness comparison.

The end-to-end performance evaluation is done to show the effect of our integration. For random strings, we start the timer before calling the *suffix_array* function and stop it when the function returns. We execute the same function 10 times and calculate the average execution time. We test with random strings with length from 1M to 4M characters (see Fig. 4). The execution time (left plot) increases linearly with the number of strings. We define the sorting efficiency as the average number of bytes that can be sorted in one second. The efficiency (right plot) shows that the sorting efficiency of 1MB is much better than 4MB (more than 5 times).

To properly demonstrate how Arkouda can be utilized by data scientists in the field, we also present execution results of generating suffix arrays using real data sets. The data sets

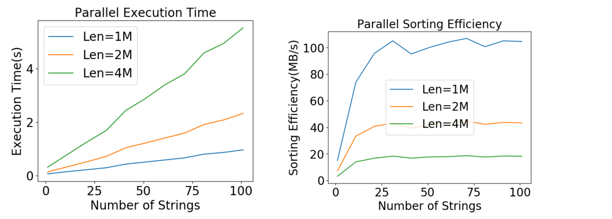

Fig. 4. The execution time and efficiency of parallel *forall* suffix array sorting for ramdom strings.

TABLE I
REAL-WORLD DATA SETS

Name	A. Size	B. Size	Descriptions
mj	20	448779	Protein sequence for M. jannaschii.
hi	20	509519	Protein sequence for H. influenzae.
pi	10	1000000	The first million digits of pi.
sc	20	2900352	Protein sequence for S. cerevisiae.
hs	20	3295751	Protein sequence for H. sapiens.
bible	63	4047392	The entire King James Bible.
ecoli	4	4638690	Genomic sequence for E. coli.
us_dns	59	20526804	List of DNS names for the U.S.

utilized include book text [35], genomic and protein sequences [2], and domain name system (DNS) addresses [43]. In these data sets, suffix arrays can be utilized to solve problems like computing the maximal repeated pairs, supermaximal repeats, or maximal unique matches [3]. Solving these problems can yield important information such as repeating substrings of protein sequences in different diseased cells or quickly searching through a list of DNS names looking for a specific malicious DNS name. Furthermore, not only was the Arkouda server being hosted, but also a Jupyter server to enable the use of Jupyter notebooks. With an SSH tunnel, we were able to connect our local device to our HPC server and run Arkouda jobs from the local Jupyter GUI. The environment setup for this is visualized in Fig. 5.

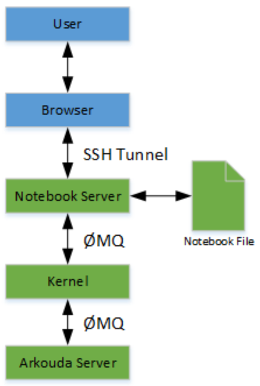

Fig. 5. Jupyter notebook based real-word data set testing setup.

Various data were selected from varying domains to properly demonstrate the interdisciplinary applications that suffix arrays may have ranging from biology to computer networking. Table I summarizes the important information of the selected data sets. A name is given to each set that is then used in following figures with results. Further, the alphabet size is shown as A. size. This size is equivalent to the number of unique characters found in the file. For example, the file named *ecoli* contains the full genetic sequence for the ecoli bacterium which is made up of only the characters *a, c, t,* and *g*. Therefore, its A. Size is 4. The B. Size column holds how long the file is in bytes. This is the same as the length of the file in characters.

A suffix array was generated for each file and the elapsed time was recorded. The start timer was called right before the invocation of the function that generates a suffix array from a file and the stop timer was called right after this invocation was terminated. For 7 of the 8 files, the suffix array was generated 100 times and the average time was calculated. Due to the immense size of the *us_dns* data set, suffix array generation was only invoked 10 times, and the average of the elapsed time of the 10 invocations was taken.

Figures 6 shows the sorting efficiency for different data sets. As string size increased, the sorted bytes per second decreased.

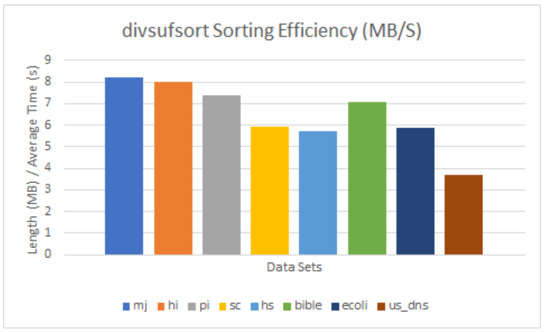

Fig. 6. Number of megabytes sorted in unit time with divsufsort.

To check the message exchanging overhead, as we did for the random strings, we ran the *suftest* executable generated by a local compilation of the *libdivsufsort* library. The code of *suftest* was edited to capture the start time before file reading instead of directly before the function that generates the suffix array. The purpose of this test is to see how much the execution time increases with *divsufsort* being utilized in Arkouda. The increase in execution time should be negligible and we are able to see that in Table II. Rather, for the largest data set, of 20,526,804 bytes, there is actually an execution time decrease (speed increase) by utilizing Arkouda. This is due to the compilation of the Arkouda server with the Chapel fast flag. Enabling this flag turns off all the runtime checks, optimizes the compiled C code, and specializes the executable to the underlying CPU architecture, if previously specified by the user. As is also evident in Table II, the largest speed decreases had to do with the smallest *mj* and *hi* files whereas once the file is over one million bytes, the speed decrease is almost nonexistent or there is actually a speed increase as shown with *us_dns*, the list of United States domain names.

Lastly, it is important to note the memory-wise limitations of Arkouda (because our testing hardware platform is not

TABLE II
PERCENT INCREASE IN EXECUTION TIME FROM LIBDIVSUFSORT TO DIVSUFSORT COMPILED BY ARKOUDA

Name	B. Size	% Increase
mj	448779	32.14
hi	509519	44.82
pi	1000000	9.01
sc	2900352	1.68
hs	3295751	4.86
bible	4047392	7.16
ecoli	4638690	2.93
us_dns	20526804	-1.91

powerful enough) when it comes to executing large string files. Arkouda can handle 20MB, 66MB, and 100MB files well. However, once you get over the 100MB limit, dependent on the hardware resources available and the amount of memory the *divsufsort* algorithm requires, the server is unable to allocate enough memory for the operations and the operation fails. The largest file used for testing was 100MB; when attempting to create a suffix array for a 500MB file, the execution failed. The file size limit for our Arkouda setup can therefore be said to be somewhere between 100MB and 500MB. Table III shows the execution times for generating suffix arrays from files composed of varying domain names.

TABLE III
EXECUTION TIMES FOR 20MB, 66MB, AND 100MB FILES COMPOSED OF VARYING DNS NAMES.

File Size (MB)	Execution Time (s)
20	5.29
66	35.48
100	62.43

V. RELATED WORK

Research on suffix arrays has increased since Manber and Myers [27] introduced this data structure as an alternative to suffix trees in the early 1990s. The efforts on suffix array algorithm design has three important directions: linear time complexity algorithms, succinct or light weight algorithms and parallel algorithms.

The research in [18]–[20] were the early achievements that reduced the time complexity from $O(n \log n)$ to $O(n)$. Research [13], [28] can achieve $O(n)$ working space. The survey paper [36] described many time and space complexity research in this field.

Homann *et al.* [12] introduced the mkESA tool on multi-threaded CPUs, which is a parallelized version of the 'Deep-Shallow' algorithm of Manzini and Ferragina [29]. Mohamed and Abouelhoda [31] proposed a parallelized variant of the bucket pointer refinement (bpr) algorithm of Schürmann and Stoye [38] on multicore architectures, leveraging shared memory. Shun's problem-based benchmark suite (PBBS) [41] leverages the task-parallel Cilk Plus programming model in its parallel multicore skew implementation. Flick and Aluru's [9] parallel distributed memory SACA has a similar approach to LS method [24]. Nong *et al.* [22], [23] implement their linear and light-weight suffix sorting on a multicore computer.

Osipov [33] and Deo and Keely [8] have performed seminal work on developing highly parallel shared-memory GPU algorithms for suffix array construction. There are some following GPU Parallel suffix array algorithms, such as in [26], [42], [44], [45].

Disk sorting [6], [17] is another effort to solve the large scale string problem and our parallel suffix array construction algorithm framework is based on the out of core sorting method.

VI. CONCLUSION

The suffix array is a fundamental data structure in large scale string analysis. In this work, we provide the solution to integrate a suffix array into Arkouda to enable large scale string data analytics.

This work demonstrates that the increasing Python community can take advantage of the Arkouda framework to conduct very large string analysis just like on their laptops or PCs. High level Python users will have the capability to do large scale data analytics easily and efficiently. Specifically, our work shows that (1) The proposed suffix array integration method is feasible and can take advantage of the Chapel *forall* construct efficiently to provide coarse grain parallelism to improve a group of suffix arrays' building performance. This work provides the basic data structure support for large scale string analysis. (2) The Arkouda framework is easy to extend with new data structures and functions. This feature is very useful for a software framework under developement. At the same time, Arkouda's message exchange overhead between the Python front-end and the Chapel back-end can almost be ignored in large scale data analysis. This performance result means that Arkouda has the potential to support very efficient interactive exploratory data analysis at scale as it becomes a fully developed system.

In the future, we will further develop different parallel suffix array algorithms based the proposed parallel framework. Especially, we will develop multi-locale parallel algorithms that can exploit hierarchical parallelism. To achieve practical performance, we plan to develop and integrate different kinds of suffix array algorithms into Arkouda's algorithm library so the runtime will have more choices to select the best version for different applications. The dynamic algorithms selection, hierarchical parallelism and the codesign between application, algorithm and hardware will significantly improve the performance of our solution.

VII. ACKNOWLEDGEMENT

We appreciate the help from Michael Merrill, William Reus and the Arkouda community, as well as Brad Chamberlain, Elliot Joseph Ronaghan, Engin Kayraklioglu, David Longnecker and the Chapel community when we integrated the algorithms into Arkouda. This research was funded in part by NSF grant number CCF-2109988.

References

[1] Ahmed Abdelhadi, AH Kandil, and Mohamed Abouelhoda. Cloud-based parallel suffix array construction based on MPI. In *2nd Middle East Conference on Biomedical Engineering*, pages 334–337. IEEE, 2014.

[2] Jurgen Abel. Protein corpus.

[3] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86, 2004.

[4] A Antonitio, P Ryan, Bill Smyth, Andrew Turpin, and X Yu. New suffix array algorithms-linear but not fast? In *Proceedings of the 15th Australasian workshop on combinatorial algorithms (AWOCA)*, pages 148–156. Australian Computer Society, 2004.

[5] Alberto Apostolico. The myriad virtues of subword trees. In *Combinatorial algorithms on words*, pages 85–96. Springer, 1985.

[6] Timo Bingmann, Johannes Fischer, and Vitaly Osipov. Inducing suffix and LCP arrays in external memory. *Journal of Experimental Algorithmics (JEA)*, 21:1–27, 2016.

[7] Bradford L. Chamberlain. Chapel. In Pavan Balaji, editor, *Programming Models for Parallel Computing*, chapter 6, pages 129–159. MIT Press, November 2015.

[8] Mrinal Deo and Sean Keely. Parallel suffix array and least common prefix for the GPU. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 197–206, 2013.

[9] Patrick Flick and Srinivas Aluru. Parallel distributed memory construction of suffix and longest common prefix arrays. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2015.

[10] Amol Ghoting and Konstantin Makarychev. Indexing genomic sequences on the IBM Blue Gene. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 2009.

[11] Dan Gusfield. Algorithms on stings, trees, and sequences. 1997.

[12] Robert Homann, David Fleer, Robert Giegerich, and Marc Rehmsmeier. mkESA: enhanced suffix array construction tool. *Bioinformatics*, 25(8):1084–1085, 2009.

[13] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM Journal on Computing*, 38(6):2162–2178, 2009.

[14] Parry Husbands and Charles Isbell. The Parallel Problems Server: A Client-Server Model for Interactive Large Scale Scientific Computation. *Proceedings of VECPAR'98*, June 1998.

[15] Parry Husbands, Charles L. Isbell, and Alan Edelman. Interactive Supercomputing with MITMatlab. August 2001.

[16] Juha Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.

[17] Juha Kärkkäinen and Dominik Kempa. Engineering a lightweight external memory suffix array construction algorithm. *Mathematics in Computer Science*, 11(2):137–149, 2017.

[18] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *International colloquium on automata, languages, and programming*, pages 943–955. Springer, 2003.

[19] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2-4):126–142, 2005.

[20] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.

[21] Julian Labeit, Julian Shun, and Guy E Blelloch. Parallel lightweight wavelet tree, suffix array and FM-index construction. *Journal of Discrete Algorithms*, 43:2–17, 2017.

[22] Bin Lao, Ge Nong, Wai Hong Chan, and Yi Pan. Fast induced sorting suffixes on a multicore machine. *The Journal of Supercomputing*, 74(7):3468–3485, 2018.

[23] Bin Lao, Ge Nong, Wai Hong Chan, and Jing Yi Xie. Fast in-place suffix sorting on a multicore computer. *IEEE Transactions on Computers*, 67(12):1737–1749, 2018.

[24] N Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.

[25] Zhize Li, Jian Li, and Hongwei Huo. Optimal in-place suffix sorting. In *International Symposium on String Processing and Information Retrieval*, pages 268–284. Springer, 2018.

[26] Chi-Man Liu, Ruibang Luo, and Tak-Wah Lam. GPU-accelerated BWT construction for large collection of short reads. *arXiv preprint arXiv:1401.7457*, 2014.

[27] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[28] Michael A Maniscalco and Simon J Puglisi. Faster lightweight suffix array construction. In *Proc. of International Workshop On Combinatorial Algorithms (IWOCA)*, pages 16–29. Citeseer, 2006.

[29] Giovanni Manzini and Paolo Ferragina. Engineering a lightweight suffix array construction algorithm. *Algorithmica*, 40(1):33–50, 2004.

[30] Michael Merrill, William Reus, and Timothy Neumann. Arkouda: interactive data exploration backed by Chapel. In *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*, pages 28–28, 2019.

[31] Hisham Mohamed and Mohamed Abouelhoda. Parallel suffix sorting based on bucket pointer refinement. In *2010 5th Cairo International Biomedical Engineering Conference*, pages 98–102. IEEE, 2010.

[32] Yuta Mori. libdivsufsort, 2015. *URL: https://github.com/y-256/libdivsufsort*.

[33] Vitaly Osipov. Parallel suffix array construction for shared memory architectures. In *International Symposium on String Processing and Information Retrieval*, pages 379–384. Springer, 2012.

[34] Jacopo Pantaleoni. A massively parallel algorithm for constructing the BWT of large string sets. *arXiv preprint arXiv:1410.0562*, 2014.

[35] Matt Powell. The Canterbury Corpus, Jan 2001.

[36] Simon J Puglisi, William F Smyth, and Andrew H Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys (CSUR)*, 39(2):4–es, 2007.

[37] William Reus. CHIUW 2020 Keynote Arkouda: Chapel-Powered, Interactive Supercomputing for Data Science. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 650–650. IEEE, 2020.

[38] Klaus-Bernd Schürmann and Jens Stoye. An incomplex algorithm for fast suffix array construction. *Software: Practice and Experience*, 37(3):309–329, 2007.

[39] Julian Shun. Fast parallel computation of longest common prefixes. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 387–398. IEEE, 2014.

[40] Julian Shun and Guy E Blelloch. A simple parallel Cartesian tree algorithm and its application to parallel suffix tree construction. *ACM Transactions on Parallel Computing (TOPC)*, 1(1):1–20, 2014.

[41] Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 68–70, 2012.

[42] Weidong Sun. Using GPU to accelerate suffix array construction. In *2014 7th International Conference on Biomedical Engineering and Informatics*, pages 677–682. IEEE, 2014.

[43] Bohdan Turkynewych. Domains project: Processing petabytes of data so you don't have to, Jan 2020.

[44] Leyuan Wang, Sean Baxter, and John D Owens. Fast parallel suffix array on the GPU. In *European Conference on Parallel Processing*, pages 573–587. Springer, 2015.

[45] Leyuan Wang, Sean Baxter, and John D Owens. Fast parallel skew and prefix-doubling suffix array construction on the GPU. *Concurrency and Computation: Practice and Experience*, 28(12):3466–3484, 2016.