Enabling Exploratory Large Scale Graph Analytics through Arkouda

Zhihui Du, Oliver Alvarado Rodriguez, and David A. Bader
Department of Data Science
New Jersey Institute of Technology
Newark, New Jersey, USA
{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/HPEC49654.2021.9622860

***Abstract*—Exploratory graph analytics helps maximize the informational value from a graph. However, increasing graph sizes makes it impossible for existing popular exploratory data analysis tools to handle dozens of terabytes or even larger data sets in the memory of a common laptop/personal computer. 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, we present our initial work on overcoming the memory limit and high-performance computing coding roadblocks for high-level Python users to perform large graph analyses. Based on a simple and succinct graph data structure, a high-level Chapel-based graph algorithm, Breadth-First Search (BFS), is presented to show the scalable and parallel graph algorithm development method in a productive way through Arkouda. The reverse Cuthill–McKee (RCM) algorithm is implemented in Chapel to relabel the vertices of a graph as a preprocessing step to improve the performance of BFS and one low-level BFS algorithm is also developed to compare with the performance of high-level method. Both synthetic graphs and typical graph benchmarks are used to evaluate the performance of the provided graph algorithms. The experimental results show that, based on the proposed high-level algorithm framework, the performance of BFS can be improved significantly and easily by simply selecting suitable Chapel high-level data structures and parallel constructs. Our code is open source and available from GitHub (https://github.com/Bader-Research/arkouda).**

***Index Terms*—Exploratory graph analysis, Parallel graph algorithms, Breadth-First Search, High-Performance Computing**

I. Introduction

A graph is a well-defined mathematical model used to express the relationship between different objects and is widely used in many fields such as social sciences, biological systems, and information systems. The edge distributions of many large-scale real-world problems follow a power-law distribution [2], [11], [21]. Dense graph data structures and algorithms consume too much memory to efficiently analyze very large sparse graphs. Therefore, parallel algorithms for sparse graphs [18] has become an important research topic.

Exploratory data analysis (EDA) [5], [12], [14] was proposed by Tukey [22] and his associates in the early 1960s. The basic idea of EDA is observing the data in as many ways as possible until a succinct story of the data is apparent. Tukey linked EDA with "detective work" to summarize the main characteristics of data sets. Instead of checking a given hypothesis with data, EDA primarily is for seeing what the data can tell us beyond the formal modeling or hypotheses testing task. In this way, EDA tries to maximize the value of data in example use cases such as COVID-19 and Twitter [9] [15].

Popular EDA methods and tools, which often run on laptops or common personal computers, cannot hold terabyte or even larger sparse graph data sets. Arkouda [16], [19] is an EDA framework under early development that combines the productivity of Python with world-class high-performance computing. Arkouda allows data scientists to make use of the advantages of both laptop computing and cloud/supercomputing together. Until now, Arkouda did not support graph analysis.

In this work, we provide a preliminary solution for integrating sparse graphs and their methods into Arkouda. The major contributions are as follows.

1) A distributed parallel Breadth-First Search (BFS) graph algorithm is developed using the high-level parallel language Chapel. High productivity in algorithm design and quick optimization in performance improvement are the two major advantages of our Chapel based graph algorithm development methodology.
2) The reverse Cuthill–McKee (RCM) algorithm, working as a preprocessing step, is developed to further improve the performance of the proposed BFS algorithm.
3) Experimental results show that the proposed method can enable Arkouda to handle graph analytics problems at scale. Based on the same algorithm framework, quick and small tuning in high-level data structure and parallel construct selection can achieve more than 8 times speedup.

II. Arkouda Framework for Data Science

As a high-level exploratory data analysis framework, Arkouda aims to support not only flexible but also high-performance large-scale data analysis. Python [20] is an interpreted, high-level and general-purpose programming language. Python consistently ranks as one of the most popular programming languages and has an ever-growing community. Python has become a very powerful EDA tool. However, performance and very large-scale data processing are two challenges of

Python. Chapel [7] is a high-level programming language designed for productive parallel computing at scale with the portability and open-source nature of Python.

Arkouda integrates its front-end Python with its back-end Chapel with a middle, communicative part ZeroMQ [13]. ZeroMQ is used for the data and instruction exchanges between Python users and back-end services.

To break the data volume limit of Python, Arkouda provides a virtual data view for its Python users. However, the real or raw data is stored in Chapel on the back-end supercomputer. Python users can use the metadata to access the actual big data sets at the back-end. 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 local memory capacity limit, while at the same time bring traditional laptop users powerful computing capabilities that could only be provided by supercomputers.

III. Distributed and Parallel BFS Algorithm

A. Graph Data Structure

We borrow the basic idea of existing data structures and propose our Double-Index (DI) , edge index and vertex index, sparse graph data structure to enable directly locating vertices from a given edge or locating edges from a given vertex. DI has been described in detail in our other work [10] and here we give a brief introduction.

DI has two important features: (1) significant memory savings for large sparse graphs; (2) supporting easy and high-level array operators.

The compressed sparse row (CSR) or compressed row storage (CRS) or Yale format has been widely used to represent a sparse matrix with much smaller space. Our double-index data structure has some similarities with CSR. The major differences between CSR and DI are as follows. (1) CSR is a vertex-oriented data structure but DI is an edge-vertex combined data structure. CSR cannot support edge ID-based searching and edge-based graph partition efficiently. However, our DI data structure can. Many real-world graphs follow power-law distributions and their degrees are highly skewed. Vertex-based graph partitions will cause serious load balancing problems. Using the DI data structure, we can support edge-based graph partitions to avoid the load balancing problem and support quick edge ID-based searching to improve the analysis performance. (2) In order to reduce the space cost as much as possible, CSR introduces irregularity and ambiguity in its data structure. DI is a balanced design and keeps its regularity and semantic clarity with only a small additional space cost. To save space, CSR assigns its row index array with two roles: a) indicating the first non-zero value position in the value array for a given row, and b) indicating the number of neighbors of the giving row. The multiple roles of the CSR index array will make the parallel programming codes hard to understand and cause a performance problem. In the DI data structure, we introduce an additional neighbor index array to explicitly express the number of neighbors of a given vertex v. In this way, we can make our vertex index array have a clear and unique meaning. The regularity and semantic clarity of the DI data structure allows it to support the standard array or array section-based parallel operators easily and efficiently.

B. High-Level Multi-Locale BFS Algorithm

We select one typical graph algorithm, Breadth-First Search, to show how we can implement exploratory large graph analytics in Arkouda. Two significant features of our parallel BFS algorithm design are different from the existing BFS algorithm design: (1) Our BFS algorithm can exploit parallelism in graph search easily and efficiently based on the proposed DI graph data structure. (2) We employ the high-level parallel language Chapel to develop the BFS algorithm so we can significantly improve the productivity of parallel algorithm design.

Very large graphs cannot be held in one shared memory computer; however, they can be handled with distributed memory computers, such as compute clusters to execute the BFS in parallel. In Chapel, the locale type refers to a unit of the machine resources on which one's program is running. Locales have the capability to store variables and to run Chapel tasks. In practice for a standard distributed memory architecture, a single multicore/SMP node is typically considered a locale. Shared memory architectures are typically considered a single locale.

The standard level by level BFS algorithm works as follows. For each vertex at the current level or frontier, we will search its unvisited next level vertices. When all the vertices at the current level have been expanded, we will switch the next level vertices to the current level and repeat the search until no vertices can be expanded in the current level.

The basic idea of our algorithm is that we take advantage of the multi-locale feature of Chapel to handle very large graphs in distributed memory. The distributed data are processed at their locales or their local memory. Furthermore, each shared memory computing node can process its owned data also in parallel. Our multi-locale BFS algorithm can exploit the following features. (1) The edges of the DI graph data have been distributed evenly onto the distributed memory to balance the load. (2) Each distributed node only expands the vertices it owns in the current frontier. This can be done in distributed memory in parallel.

Our algorithm, shown in Alg. 1, uses high-level structures in Chapel to implement a parallel breadth-first search algorithm. The *DistBag* data structure can be used to hold the current and the next frontier set easily and efficiently. At the same time, the *coforall* and *forall* parallel constructs can express the parallel expansion in a very efficient way. In line 8 multiple locales can execute the search in parallel. In line 10 different vertices in the current frontier can be expanded in parallel. In line 13, different expanded vertices can be added into the next frontier in parallel. We can exploit the parallelism in a hierarchical way to improve the total performance.

In line 8, we use *coforall* instead of *forall* to implement distributed parallel computing on each distributed memory computing node. In line 11, we just select the vertices owned by the current locale. In this way, we can increase the access

Algorithm 1: high-level Chapel based parallel BFS for distributed memory supercomputers

Input: A graph G and the starting vertex *root*
Output: An array *depth* to show the different visiting level for each vertex

```
1  depth = −1 // initialize the visiting level of all the vertices
2  depth[root] = 0 // set starting vertex's level is 0
3  cur_level = 0 //set current level
4  SetCurF = new DistBag(int, Locales) // allocate a distributed bag to hold vertices in the current frontier
5  SetNextF = new DistBag(int, Locales) // allocate another bag to hold vertices in the next frontier
6  SetCurF.add(root) //insert the starting vertex into the current vertices bag
7  while !SetCurF.isEmpty() do
8      coforall loc in Locales do
9          // parallel search on each locale
10         forall i in SetCurF do
11             if i is on current locale then
12                 SetNeighbor = ∪k, k is a neighbor of i
13                 forall j in SetNeighbor do
14                     if depth[j] == −1 then
15                         SetNextF.add(j)
16                         depth[j] = current_level + 1
17                     end
18                 end
19             end
20         end
21     end
22     SetCurF <=> SetNextF // exchange values
23     SetNextF.clear()
24     current_level+ = 1
25 end
26 return depth
```

locality and avoid expanding the same edges on multiple locales. Yet, we cannot avoid idle threads when the current locale spawns a thread for the vertex owned by other locales.

IV. Optimization and Comparison

In this section, we introduce the reverse Cuthill–McKee (RCM) method [3], [8] that is taken as a preprocessing procedure to improve the locality of BFS. At the same time, based on the same algorithm framework, we replace the high-level Chapel data structure implementation with a low-level implementation to compare the performance of the two versions. In this way, we can show the efficiency of the Chapel high-level data structures.

A. RCM Preprocessing

Many sparse matrix computations can be accelerated by reordering the matrix to reduce its bandwidth. So, reordering vertices of a graph can minimize data size, maximize data locality and improve the performance of graph algorithms. However, optimal reordering with a minimal bandwidth is NP-complete [17]. Hence, heuristic reordering algorithms are used in practice.

The Cuthill–McKee algorithm (CM) is a heuristic algorithm which permutes a sparse matrix that has a symmetric sparsity pattern into a band matrix form with a small bandwidth. Given a symmetric $n \times n$ adjacency matrix, the Cuthill–McKee algorithm relabels the vertices of the graph to reduce the bandwidth of the adjacency matrix. The reverse Cuthill–McKee algorithm (RCM) is the same algorithm but with the resulting index numbers reversed. In practice, this generally results in less fill-in than the CM ordering when Gaussian elimination is applied. So RCM is widely used in practical applications.

In this paper, we employ the RCM method as a preprocessing step to relabel the vertices. In this way. we can improve the BFS performance.

The basic idea of our heuristic RCM algorithm is as follows. (1) Start from a vertex with the smallest degree and push the vertex into a stack; (2) Search the vertices level by level using the same method as in Alg. 1 and push the vertices in the next frontier into the stack in increasing order of their degrees; (3) Pop all the vertices from the stack to an RCM array and relabel all the vertices using the RCM array's index.

B. Low Level Multi-Locale BFS Algorithm

In the high-level BFS algorithm, we use two *DistBag* classes to store all the vertices in the current frontier and the next frontier. The communication between different locales is implicit. This makes our parallel program simple and easy. To evaluate the performance of such high-level data structures in Chapel, we directly use arrays to hold the vertices in the current and next frontiers and explicitly implement the corresponding communication between different locales. In this way, we can check if the high-level data structure *DistBag* introduces significant performance overhead.

To optimize the performance, we use a distributed array *curFAry* to clearly distinguish the frontier elements owned by different locales. We also use a distributed array *recvAry* to hold the expanded vertices from different locales. In this way, we can exploit the locality and optimize the communication during the graph search. The low-level algorithm is given in Alg. 2.

From line 1 to line 6, the low-level BFS algorithm is just like the high-level BFS algorithm except that we replace *SetCurF* with *curFAry* and replace *SetNextF* with *recvAry*. From line 7 to line 37 we implement an optimized breadth-first search procedure. The basic algorithm structure is similar to Alg. 1. From lines 8 to 32, we will complete one level vertex expansion. A set data structure *SetNextFLocal* is created to hold the expanded elements owned by the current locale (line 9) and the elements can be added in parallel. If the expanded elements do not belong to the current locale, we create another set data structure *SetNextFRemote* to hold such elements (line 10). Instead of parallel search on all the current frontier vertices, in the low-level version, each locale will first get its owned vertices (line 11). Then, each locale uses the parallel

construct *coforall* to expand the next frontier in parallel (line 12). The difference with Alg. 1 is that we put the expanded elements into different sets. If they are local, we put them into the *SetNextFLocal* set in parallel (from line 16 to line 18). If they are not owned by the current locale, we will put them into *SetNextFRemote* (from lines 19 to 21). After the vertex expansion at each level, each locale will scatter the next frontier elements in the *SetNextFRemote* to their owners (lines 26 to 28). At the same time, the elements of the next frontier owned by the current locale *SetNextFLocal* will be merged into the distributed array *curFAry* (lines 29 to 31). All the above vertex operations can be done in parallel without data races. The parallel construct *coforall* has an implicit synchronization mechanism. So, after line 32, we can make sure that all data communication has been completed, and we can safely use the data in *recvAry*. From lines 33 to 35, each locale will combine the next frontier elements generated by the current locale and the other locales to form the current frontier. The low-level BFS algorithm can exploit locality, avoid idle parallel threads, and use an aggregation method to optimize the communication performance. However, we have to take care of the data distribution and data communication.

Algorithm 2: Low level parallel BFS for distributed memory supercomputers

```
   Input: A graph G and the starting vertex root
   Output: An array depth to show the different visiting
           level for each vertex
1  depth = −1 // initialize the visiting level of all the
     vertices
2  depth[root] = 0 // set starting vertex's level is 0
3  cur_level = 0 //set current level
4  Create distributed array curFAry to hold current
     frontier of each locale
5  Create distributed array recvAry to receive expanded
     vertices from other locales
6  put root into curFAry
7  while !curFAry.isEmpty() do
8      coforall loc in Locales do
9          create SetNextFLocal to hold expanded vertices
             owned by current locale
10         create SetNextFRemote to hold expanded
             vertices owned by other locales
11         myCurF ← current locale's frontier in curFAry
             and then clear curFAry
12         coforall i in myCurF do
13             SetNeighbor = ∪k, k is a neighbor of i
14             forall j in SetNeighbor do
15                 if depth[j] == −1 then
16                     if j is local then
17                         SetNextFLocal.add(j)
18                     end
19                     else
20                         SetNextFRemote.add(j)
21                     end
22                     depth[j] = current_level + 1
23                 end
24             end
25         end
26         if !SetNextFRemote.isEmpty() then
27             scatter elements in SetNextRemote to
                 recvAry
28         end
29         if !SetNextFLocal.isEmpty() then
30             move elements in SetNextLocal to curFAry
31         end
32     end
33     coforall loc in Locales do
34         curFAry ← collect elements from recvAry
35     end
36     current_level+ = 1
37 end
38 return depth
```

V. Experiments

A. Experimental Setup

To evaluate the results of the proposed integrated solution, we used two kinds of graphs. The first is the R-MAT method [6] to generate the synthetic graphs. The other kinds of graphs are from standard benchmarks. We develop a simple *bsf.py* Python testing program to drive the experiments.

For the R-MAT graphs, we set the vertices count of four different graphs as the following values: 32768, 65536,131072, and 262144. The possibility of the dense edges area is set as 0.75. All other three parts' possibilities share the remainder 0.25 equally. The average number of edges per vertex is two so we will generate 65536, 131072, 262144, and 524288 edges for different R-MAT graphs. We will generate both directed and undirected R-MAT graphs.

The graph benchmarks utilized for testing include the Delaunay, Kronecker (notation as KRON in the following part), and Random Geometric graphs (notation as RGG in the following part) from the 10th DIMACS Implementation Challenge [4]. The number of edges and vertices will be approximately doubled to reach the next graph in the same benchmark series.

Testing of the methods was conducted in an environment composed of a 32 node cluster with an FDR Infiniband between the nodes in the cluster. Each node has two 10-core Intel Xeon E5-2650 v3 @ 2.30GHz and 512GB DDR4 memory. Infiniband connections between nodes are commonly found in high-performing computers. Due to Arkodua being designed primarily for data analysis in an HPC setting, an architecture setup that aptly fits an HPC scenario was chosen for testing.

For R-MAT, Delaunay, KRON, and RGG graphs, we will first build the graphs into distributed memory based on our

partition method and then execute the parallel BFS algorithm with different numbers of locales to check their performance (we will cancel some tests if the execution time is too long). For R-MAT graphs, we implement the R-MAT algorithm to generate the R-MAT graph in parallel each time. For the benchmark graphs, each locale will read the graph file in parallel using Chapel file IO and just select the data that should be stored at its locale. After the graph data are ready in memory, we will sort the edges and organize the graph based on our DI data structure. Furthermore, for the high-level multi-locale algorithm, we will show how a simple replacement in the data structure and parallel construct can affect the performance significantly.

B. Experimental Results

1) Graph Building: The experimental results from Fig. 1 to Fig. 8 show the graph building time and the building efficiency for different graphs in Arkouda. We can see that for Fig. 5 and Fig. 7, the building time will increase linearly with the number of edges, no matter how many locales are used. However, it will take more time when handling the same number of edges with more locales. The reason lies in the data movement overhead among locales. More locales means that more data movement between distributed memories will be needed. The building efficiency in Fig. 6 and Fig. 8 also have perfect flat lines. The flat line means that each core will have the same efficiency no matter how many edges or how many cores are used. Our experiments show that the best construction efficiency of the RGG graph is 1736 edges/second/core. The lowest building efficiency is 255 edges/second/core for the largest R-MAT graph because the R-MAT graph will need additional time to generate the graph.

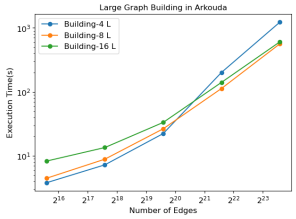

Fig. 1. Graph building time (R-MAT).

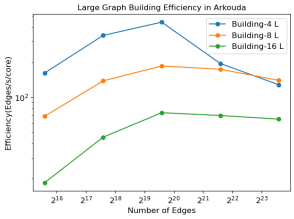

Fig. 2. Graph building efficiency (R-MAT).

We model the graph building time with the following multivariate nonlinear equation. Let E be the number of edges in the graph and L be the number of locales that will be used to build the graph. The building time will be

$$T(E, L) = a \times E/L + b \times E \times L + c$$

This model means that we assume that the computing time will increase linearly with E/L and the communication time will increase linearly with the product $E \times L$. Table I gives the regression results based on our experimental data.

For all the results, the Root Mean Square Error (RMSE) is less than 390 and the R-squared value is larger than

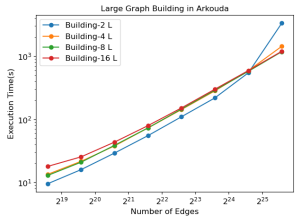

Fig. 3. Graph building time (Delaunay).

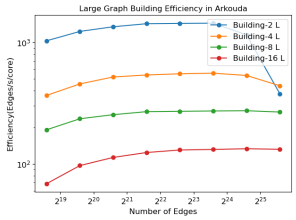

Fig. 4. Graph building efficiency (Delaunay).

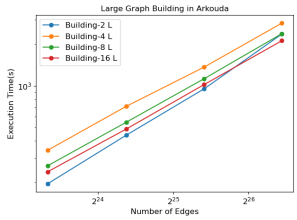

Fig. 5. Graph building time (KRON).

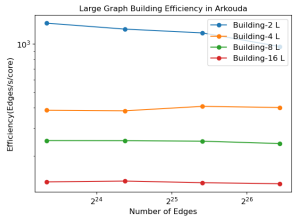

Fig. 6. Graph building efficiency (KRON).

0.79 which means more than 79% of the observed variation can be explained by the model's inputs. We can use the models to do some predictions. For example, for the com-friendster.ungraph.txt graph [1] which has 1,806,067,135 edges, the predicted building time on 2 locales will be 8.31 hours if we use the RGG data. In reality it takes 8.5 hours to build the graph in memory and the predicted value is very close to the practical value. However, if we use the data of Delaunay and KRON that have less edges, the predicted building time will be much longer. The experimental results in Fig. 1 and Fig. 3 can help us see what happens for different graphs. We can see in Fig. 1, locales with 8 and 16 have a good linear growth trend. However, the curve with locale 4 will increase very fast when the number of edges becomes larger although the total number of edges is less than the number in the KRON

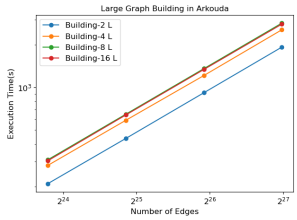

Fig. 7. Graph building time (RGG).

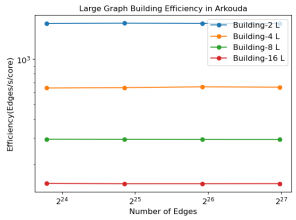

Fig. 8. Graph building efficiency (RGG).

TABLE I
ARKOUDA LARGE GRAPH BUILDING REGRESSION MODEL.

	Regression Model		
Benchmark	Delaunay	KRON	RGG
T(E, L)=	1.07e-04 *E/L + 1.22e-06 *E*L -6.47e+01	5.24e-05 *E/L +1.43e-06 *E*L +1.21e+02	2.76e-05 *E/L +1.34e-06 *E*L+ 1.52e+02
T(E,2)=	5.59e-05 *E -6.47e+01	2.90e-05 *E+ 1.21e+02	1.65e-05 *E + 1.52e+02
T(E,4)=	3.16e-05 *E -6.47e+01	1.88e-05 *E + 1.21e+02	1.23e-05 *E + 1.52e+02
T(E,8)=	2.31e-05 *E -6.47e+01	1.80e-05 *E + 1.21e+02	1.42e-05 *E + 1.52e+02
T(E,16)=	2.61e-05 *E -6.47e+01	2.61e-05 *E + 1.21e+02	2.31e-05 *E + 1.52e+02
RMSE	201.22	390.72	362.49
R-squared	0.90	0.79	0.83

and RGG benchmarks. The major reason is that compared with the benchmark method, the R-MAT graph building method will have additional graph generation time. When a graph touches the limit of its computing resources or the heavy workload watermark, it cannot maintain a linear trend. A heavy load will reduce the core's performance and efficiency. For Fig. 3, the Delaunay graph's building time with 2 locales will also increase fast when the number of edges is larger than 25,165,784. The reason is that Delaunay graphs have much more vertices than the other benchmarks. Therefore, for the same number of edges, the Delaunay benchmark will need more computation and memory. When the graph touches the suitable resource limit, the lack of hardware resources will also cause a loss in performance and efficiency. Going beyond the resource bound is the major reason why the graph building efficiency will decrease in Fig. 2 and Fig. 4. From all the graphs, we can conclude that the graph building efficiency curve will first increase, then stay almost the same, and finally reduce when the graph workload touches the computing or memory limit of the given platform.

2) BFS Performance: In this part, we will focus on the performance comparison of different BFS implementations. So we will use deterministic graph benchmarks instead of R-MAT graphs that can lead to different performance with the same method because of the randomness in the graphs.

We use four different Delaunay benchmark graphs to show the performance of our BFS algorithms. We use Fig. 9 as an example to explain the meaning of different algorithm implementations. On the x axis, *M* means the result of our manually optimized low-level Alg. 2. *BagL* is the result of high-level multi-locale Alg. 1. *BagG* means that we will remove line 11 of Alg. 1 and all locales will search on the whole frontier instead of the vertices owned by itself. *SetL* is the case that we just replace the high-level data structure *DistBag* with *Set* in Alg. 1. Except for using the *Set* data structure, *SetG* is similar to *BagG*. *DomL* and *DomG* are similar with *SetL* and *SetG* except we will replace *DistBag* with *Domain*. In our high-level multi-locale BFS algorithm framework, *DistBag*, *Set* and *Domain* can provide the same function to store the current frontier and the next frontier elements. At the same time, they also have the same or similar methods to use the data structure. For example, they all have the *add* function to add element into *DistBag*, *Set* or *Domain*.

For all the high-level multi-locale BFS methods in Fig. 9 to Fig. 12, the legend *ForAll* means we will use *forall* parallel construct to expand the vertex at line 10 in Alg. 1. The legend *CoForAll* means that we will use the *coforall* parallel construct to expand the vertices. However, for the manually optimized low-level method, the legend *CoForall* means that we will use the *coforall* parallel construct to expand the owned vertices by each locale at line 12 in Alg. 2. The legend *ForAll* means that we will use the *forall* parallel construct to expand the owned vertices by each locale.

From the experimental results in Fig. 9 to Fig. 12, we have the following observations: (1) For all the data structures *DistBag*, *Set* and *Domain*, the performance of distributed parallel computing version (*BagL*, *SetL* and *DomL*) will be better than the shared computing version (*BagG*, *SetG* and *DomG*). It is easy to understand that the shared computing will have a significant amount of duplicated computations and the distributed resources cannot be used efficiently. (2) For most of the distributed parallel computing versions, the performance of the *forall* parallel construct is better than the *coforall* parallel construct. The reason is that the size of our frontier (from hundreds to thousands and beyond) is relatively larger than the parallel units (20 in our system). The *coforall* construct will generate many parallel threads but they cannot be run immediately. So the *forall* parallel construct that only generates the same number of threads as the maximum cores will be more efficient. However, for our manually optimized low-level version, the *coforall* parallel construct implementation has better performance when the graph size is small. The performance of *forall* will catch up when the graph size become larger (see Fig. 12). The reason is that our low-level implementation can avoid idle threads and the number of parallel threads created by *coforall* is less than the high-level implementation (about $1/numLocales$ of the size of the current frontier). (3) For different high-level data structures (*DistBag*, *Set*, and *Domain*), their optimized *forall* parallel performance is very close to each other. The major operation in our algorithm is to add an element into a set in parallel. Surprisingly, *DistBag* has not shown any obvious advantage in our preliminary tests. (4) Our manually optimized low-level algorithm cannot have better performance than the high-level algorithms. This means that Chapel's high-level data structures (*DistBag*, *Set*, and *Domain*) can implement the data insertion into a set and the communication among different locales with high performance.

The major advantage of our manually optimized low level implementation is that we can extend the vertices owned

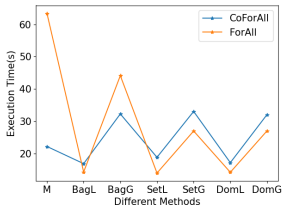

Fig. 9. BFS time ($delaunay\_n17$).

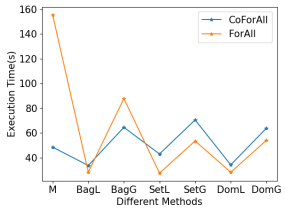

Fig. 10. BFS time ($delaunay\_n18$).

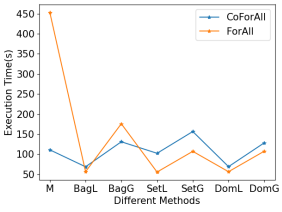

Fig. 11. BFS time ($delaunay\_n19$).

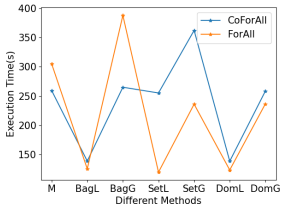

Fig. 12. BFS time ($delaunay\_n20$).

by different locales independently to get the next frontier without generating any idle threads. However, for the high-level method, we have to create the same number of threads on each locale to check if the element is owned by the local locale. If a vertex is not owned by the current locale, this thread will become idle. The disadvantage of our low-level method is that we have to create two additional sets to keep the local elements and remotes. And we need to send the remote elements to their owners. We will incur additional costs for such operations.

In Table II we present the experimental results without and with RCM preprocessing results. In the column of "RCM", "N" means without RCM preprocessing, and "Y" means with RCM preprocessing. We can see that the RCM method can substantially improve the performance in almost all cases. The best performance can be improved by about 1.24 fold for the four different benchmarks. Please note that these times are not inclusive of the preprocessing time.

TABLE II
EXECUTION TIME OF DIFFERENT BFS IMPLEMENTATIONS.

Graph	Parallel Construct	RCM	*M*	*BagL*	*BagG*	*SetL*	*SetG*	*DomL*	*DomG*
delaunay_n17	CoForall	N	22.20	16.87	32.28	18.84	33.05	17.18	32.06
		Y	14.90	14.77	26.68	16.94	29.11	14.42	26.65
	Forall	N	63.42	14.28	44.14	13.97	26.99	14.20	27.02
		Y	24.28	10.85	33.75	12.02	21.85	12.16	21.85
delaunay_n18	CoForAll	N	48.57	33.76	64.58	43.08	70.55	34.25	63.84
		Y	31.08	30.91	55.62	47.10	70.52	32.51	55.58
	ForAll	N	155.39	28.37	87.79	27.59	53.58	28.26	54.07
		Y	37.56	23.37	73.45	25.28	43.52	25.58	44.05
delaunay_n19	CoForAll	N	110.93	68.72	131.04	102.32	156.55	69.08	128.39
		Y	63.77	63.83	114.82	114.05	159.83	62.55	109.56
	ForAll	N	453.23	56.54	175.88	55.62	107.17	56.49	107.56
		Y	69.90	46.23	141.92	49.65	86.68	50.27	86.50
delaunay_n20	CoForAll	N	259.44	139.16	265.08	255.28	361.99	138.98	258.44
		Y	126.62	127.22	231.47	286.72	386.11	133.12	229.45
	ForAll	N	305.01	125.89	387.61	120.19	236.20	123.91	236.66
		Y	172.16	92.87	293.59	99.46	176.49	101.05	176.03

The performance results show that for the same algorithm framework, we can select suitable data structures and parallel constructs to achieve much better performance in Chapel programming. So we can quickly optimize the performance and this is the basic reason why we can develop parallel graph algorithms in Chapel in a productive and efficient way.

VI. CONCLUSION

This work shows that Arkouda is a promising framework to support large-scale graph analytics. Of course, the reported work is the first step to evaluate the feasibility and performance of Arkouda based large graph analytics. In future work, we will provide more graph algorithms and further optimize the performance of our algorithms in Arkouda. At the same time, we will compare our method with other approaches.

VII. ACKNOWLEDGEMENT

We appreciate the help from the Arkouda co-creators Michael Merrill and William Reus, 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] Friendster social network dataset: Friends, https://archive.org/details/friendster-dataset-201107, 2011.
[2] Lada A Adamic, Bernardo A Huberman, AL Barabási, R Albert, H Jeong, and G Bianconi. Power-law distribution of the world wide web. *Science*, 287(5461):2115–2115, 2000.
[3] Ariful Azad, Mathias Jacquelin, Aydin Buluç, and Esmond G Ng. The reverse Cuthill-McKee algorithm in distributed-memory. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 22–31. IEEE, 2017.
[4] David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. Benchmarking for graph clustering and partitioning. *Encyclopedia of Social Network Analysis and Mining*, page 1–11, 2017.
[5] John T Behrens. Principles and procedures of exploratory data analysis. *Psychological Methods*, 2(2):131, 1997.
[6] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.
[7] Bradford L Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson, Ben Harshbarger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, et al. Chapel comes of age: Making scalable programming productive. *Cray User Group*, 2018.
[8] Elizabeth Cuthill and James McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172, 1969.
[9] Samrat K Dey, Md Mahbubur Rahman, Umme R Siddiqi, and Arpita Howlader. Analyzing the epidemiological outbreak of COVID-19: A visual exploratory data analysis approach. *Journal of medical virology*, 92(6):632–638, 2020.
[10] Zhihui Du, Oliver Alvarado Rodriguez, Joseph Patchett, and David A Bader. Interactive graph stream analytics in Arkouda. *Algorithms*, 14(8):221, 2021.
[11] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. *ACM SIGCOMM computer communication review*, 29(4):251–262, 1999.
[12] Irving J Good. The philosophy of exploratory data analysis. *Philosophy of science*, 50(2):283–295, 1983.
[13] Pieter Hintjens. *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.
[14] Andrew T Jebb, Scott Parrigon, and Sang Eun Woo. Exploratory data analysis as a foundation of inductive research. *Human Resource Management Review*, 27(2):265–276, 2017.
[15] Theo Lynn, Pierangelo Rosati, Binesh Nair, and Ciáran Mac an Bhaird. An exploratory data analysis of the #crowdfunding network on Twitter. *Journal of Open Innovation: Technology, Market, and Complexity*, 6(3):80, 2020.
[16] 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.
[17] Ch H Papadimitriou. The NP-completeness of the bandwidth minimization problem. *Computing*, 16(3):263–270, 1976.
[18] Michael J Quinn and Narsingh Deo. Parallel graph algorithms. *ACM Computing Surveys (CSUR)*, 16(3):319–348, 1984.
[19] 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.
[20] Guido Rossum. *Python reference manual*. CWI (Centre for Mathematics and Computer Science), 1995.
[21] Andrew T Stephen and Olivier Toubia. Explaining the power-law degree distribution in a social commerce network. *Social Networks*, 31(4):262–270, 2009.
[22] John W Tukey. *Exploratory data analysis*, volume 2. Reading, MA, 1977.