Exploratory Large Scale Graph Analytics in Arkouda

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

Michael Merrill and William Reus
mhmerrill@mac.com
reus@post.harvard.edu
Department of Defense
USA

ABSTRACT

Exploratory graph analytics helps maximize the informational value for a graph. However, the increasing graph size 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, the preliminary work on overcoming the memory limit and high performance computing coding roadblock for high level Python users to perform large graph analysis is presented. A simple and succinct graph data structure design and implementation at both the Python front-end and the Chapel back-end in the Arkouda framework are provided. A typical graph algorithm, Breadth-First Search (BFS), is used to show how we can use Chapel to develop high performance parallel graph algorithm productively. Two Chapel based parallel Breadth-First Search (BFS) algorithms, one high level version and one corresponding low level version, have been implemented in Arkouda to support analyzing large graphs. Multiple graph benchmarks are used to evaluate the performance of the provided graph algorithms. Experimental results show that we can optimize the performance by tuning the selection of different Chapel high level data structures and parallel constructs. Our code is open source and available from GitHub (https://github.com/Bader-Research/arkouda).

KEYWORDS

Exploratory graph analysis, Parallel graph algorithms, Breadth-First Search, Real-world graph data sets, High level parallel language

ACM Reference Format:
Zhihui Du,Oliver Alvarado Rodriguez and David A. Bader and Michael Merrill and William Reus. 2021. Exploratory Large Scale Graph Analytics in Arkouda. In *Proceedings of CHIUW '21: Chapel Implementers and Users Workshop (CHIUW '21).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

CHIUW '21, June 04, 2021, Virtual

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A graph is a well defined mathematical model to formulate the relationship between different objects and is widely used in numerous domains such as social sciences, biological systems, and information systems. The edge distributions of many large scale real world problems tend to follow a power-law distribution [1, 11, 26]. Dense graph data structures and algorithms will consume much more memory and cannot analyze very large sparse graphs efficiently. Therefore, parallel algorithms for sparse graphs [23] have become an important research topic to efficiently analyze the large and sparse graphs from different real-world problems.

Exploratory data analysis (EDA) [6, 13, 15] was proposed by Tukey [27] and his associates early in 1960s. The basic idea of EDA is listening to the data in as many ways as possible until a plausible 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 and has been widely used in different applications, such as COVID-19 [10], Twitter [19] and so on.

Popular EDA methods and tools, which often run on laptops or common personal computers, cannot hold terabyte or even larger sparse graph data sets, let alone produce highly efficient analysis results. Arkouda [21, 24] is an EDA framework under early-development that brings together 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. Currently, Arkouda cannot support graph analysis. In this work we provide the preliminary design and implementation on extending Arkouda's data structures and parallel algorithms to support large scale sparse graph analytics.

In this paper we provide the preliminary solution on integrating sparse graphs into Arkouda. The major contributions are as follows.

(1) An efficient and succinct Double-Index (DI) graph data structure and large sparse graph partition method are developed to support parallel and distributed graph algorithm design.
(2) Two distributed parallel Breadth-First Search (BFS) graph algorithms are developed based on 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 method.
(3) Experimental results show that the proposed graph data structure and algorithm can support Arkouda to handle different kinds of large graphs. 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.

2 ARKOUDA FRAMEWORK FOR DATA SCIENCE

As a high level exploratory data analytics framework, Arkouda aims to support not only flexible, but also high performance large scale data analysis. Python [25] is an interpreted, high-level and general-purpose programming language. Python consistently ranks as one of the most popular programming languages and has ever growing community. Python has become a very powerful EDA tool. However, performance and very large scale data processing are two bottlenecks of Python. Chapel [8] is a high level programming language designed for productive parallel computing at scale. It has the same advantages such as portable and open-source like in Python. Furthermore, it has the scalable and fast features that Python lacks.

So, Arkouda integrates its front-end Python with its back-end Chapel with a middle, communicative part ZeroMQ [14]. ZeroMQ is used for the data and instruction exchangs between Python users and back-end services. In this way, Arkouda can provide flexible and high performance large scale data analysis capability.

To break the data volume limit of Python, Arkouda provides a virtual data view for its Python users. However, the real or raw data are stored in Chapel. 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.

When users are exploring their data, if only the metadata section is needed, then the operations can be completed locally and quickly. 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 analysis tasks quickly on the powerful HPC resources and large memory to handle 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 personal devices, large scale data sets residing on powerful back-end servers without knowing all the detailed operations at the back-end.

3 DOUBLE-INDEX SPARSE GRAPH DATA STRUCTURE AND PARTITION

For sparse matrix representations, compressed sparse row or column are two very important data structures for sparse graphs. We borrow the basic idea of such existing data structures and propose our Double-Index (*DI*), edge index and vertex index, sparse graph data structure to enable directly locating vertices from given edge or locating edges from given vertex. *DI* has two features: (1) significant memory savings for large sparse graphs; (2) supporting easy and high level array operators. The proposed *DI* data structure is new innovation to support large sparse graph exploratory data analytics.

3.1 Directed Graphs

For directed graphs, the edge index array consists of two arrays with the same shape. One is the source vertex array and the other is the destination vertex array. If there are a total of M edges and N vertices, we will use the numbers from 0 to $M-1$ to identify different edges and the numbers from 0 to $N-1$ to identify different vertices.

For example, given edge $e =< i, j >$, we will let $SRC[e] = i$ and $DST[e] = j$ where SRC is the source vertex array and DST is the destination vertex array; e is the edge ID number. Both SRC and DST have the same size M. When all edges are stored into SRC and DST, we will sort them based on their vertex ID value and remap the edge ID from 0 to $M-1$. Based on the sorted edge index array, we can build the vertex index array, which also consists of two of the same shape arrays. For example, we let edge e_{1000} have ID 1000. If $e_{1000} =< 50, 3 >$, $e_{1001} =< 50, 70 >$ and $e_{1002} =< 50, 110 >$ are all the edges starting from vertex 50, then we will assign the entry of one vertex index array $STR[50] = 1000$ and another vertex index array $NEI[50] = 3$. This means that for given vertex 50, the edges starting with vertex 50 are stored in edge index array starting at position 1000 and there are totally 3 such edges. If there are no edges from vertex i, we will let $STR[i] = -1$ and $NEI[i] = 0$. In this way, we can directly locate all the connected edges and neighbors of any given vertex.

For a given array A, we use $A[i..j]$ to express the elements in A from $A[i]$ to $A[j]$. $A[i..j]$ is also called an array section of A. So, for a given vertex with index i, it will have $NEI[i]$ neighbours and their vertex IDs are from $DST[STR[i]]$ to $DST[STR[i]+NEI[i]-1]$. This can be expressed as an array section $DST[STR[i]..STR[i] + NEI[i] - 1]$ (here we assume the out degree of i is not 0). For any vertex i, its adjacency list can be directly expressed as $< i, x >$ where x in $DST[STR[i]..STR[i] + NEI[i] - 1]$.

Fig. 1 shows M sorted edges represented by the SRC and DST arrays. Any one of the N vertices n_k can find its neighbours using NEI and STR arrays with $O(1)$ time complexity. This figure shows how NEI and STR arrays can help us locate neighbours and adjacency lists quickly.

3.2 UnDirected Graphs

For undirected graphs, an edge $< i, j >$ means that we can also arrive at i from j. We can use the data structures SRC, DST, STR, NEI to search the neighbours of j in SRC and create the adjacency list. However, this search cannot be done in $O(1)$ time complexity. To achieve $O(1)$ search time complexity for an undirected graph, we introduce another four arrays called reversed index arrays $SRCr, DSTr, STRr, NEIr$. For any edge $< i, j >$ in SRC and DST, we will have the corresponding reverse edge $< j, i >$ in $SRCr$ and $DSTr$, where $SRCr$ has the exact same elements as in DST and $DSTr$ has the exact same elements as in SRC. $SRCr$ and $DSTr$ are also sorted and $NEIr$ and $STRr$ are the index array of the number

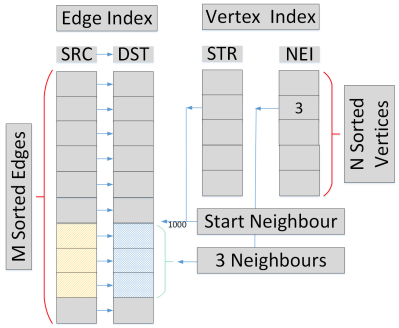

Figure 1: Double Index data structure for sparse graph.

of neighbours and the index array of the starting neighbour index just like in the directed graph. So for a given edge $< i, j >$ of an undirected graph, the neighbours of vertex i will include the elements in $DST[STR[i]..STR[i] + NEI[i] - 1]$ and the elements in $DSTr[STRr[i]..STRr[i]+NEIr[i]-1]$. The adjacency list of the vertex i should be $< i, x >$ where x in $DST[STR[i]..STR[i]+NEI[i]-1]$ or $< i, x >$ where x in $DSTr[STRr[i]..STRr[i] + NEIr[i] - 1]$.

3.3 Graph Partition

For real-world power law graphs, the edge and vertex distributions are highly skewed. Few vertices will have very large degrees but many vertices have very small degrees. If we partition the graph evenly based on the vertices, it will be easy to cause load balancing problem because the processor which holds the vertices that have a large number of edges will often have very heavy load. So, we equally divide the total number of edges into different processors/computing nodes instead.

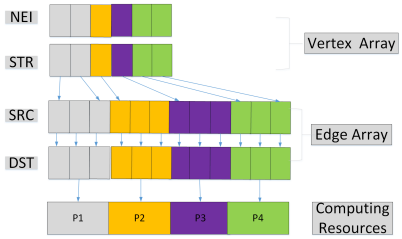

Figure 2: Edge based Sparse Graph Partition.

Fig. 2 shows the basic idea of our sparse graph partition method. When we partition an edge's vertex entry in index array NEI and STR to the same processors, this method can increase the locality when we search from edge to vertex or from vertex to edge. However, this requires us to distribute NEI and STR in an irregular way since the edge index array and the vertex index array have different shapes. Currently we cannot implement this distribution in existing Chapel methods easily, so we just partition NEI and STR arrays evenly as the edges.

4 PARALLEL 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 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.

For especially large graphs, one cannot be held in one shared memory computer, however, it can be handled with distributed memory computers, such as computing clusters to execute the BFS in parallel. In Chapel, the locale type refers to a unit of the machine resources on which your 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. We have developed two versions of parallel BFS algorithms in Arkouda. The first is the high level multi-locale version and the second is the corresponding low level version. We will give the details in the following subsections.

4.1 High Level Multi-Locale BFS Algorithm

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 method is described in Alg. 1. Line 1 initializes the return array. Line 2 sets the starting vertex's search level as 0. Line 3 initializes the current search level as 0. Lines 4 and 5 create two distributed bag classes to manage the current and next search frontiers. Line 6 adds the starting search vertex into the current frontier. The parallel code is very simple and easy. From line 7 to line 25 we will continue the standard loop if the current search frontier is not empty. From line 8 to line 21 we use the *coforall* parallel construct to execute the search on each locale in parallel. From line 10 to line 20 we will execute parallel search on each locale. On each locale, we will check each vertex in the current frontier but only the vertices on the current locale will be expanded (line 11). In this way, we will expand the current frontier in parallel on all the locales without any overlapping. In line 12 we will build the set of neighbours $SetNeighbour$ of the current vertex i. Since some neighbours have been visited before (line 14), we will only expand the unvisited vertices and add them into the next frontier set $SetNextF$ (line 15). At the same time, we will assign the visiting level to the expanded vertices with $current\_level + 1$ (line 16). After

Algorithm 1: High level Chapel based parallel BFS for distributed memory supercomptuers

```
   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                 SetNeighbour = {k|k is the neighbour of i}
13                 forall (j in SetNeighbour) 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
```

all locales have expanded their vertices in the current frontier, we will exchange the value of the current frontier and the next frontier (line 22), clean the vertices in the next frontier (line 23), add the *current_level* to next level (line 24) . Then the next loop will begin from the new frontier. When all vertices have been visited, we will return the search array *depth* as the final search result.

From this algorithm description, we can see it is very simple and natural to describe the level by level parallel method to implement the BFS algorithm in Chapel. 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 construct can express the parallel expansion in a very efficient way. At line 8 multiple locales can execute the search in parallel. At line 10 different vertices in the current frontier can be expanded in parallel. At 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.

At line 8, we use *coforall* instead of *forall* to implement distributed parallel computing on each distributed memory computing node. At line 11, we just select the vertices owned by the current locale. In this way, we can increase the access locality and avoid expanding the same edges on multiple locales.

4.2 Low Level Multi-Locale BFS Algorithm

In the high level BFS algorithm, we use two *DistBag* classes to hold all the current frontier and the next frontier. The communication between different locales is implicit. This can make our parallel program become simple and easy. To evaluate the performance of such high level data structures in Chapel, we directly use arrays to hold the vertices in 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 significantly 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*. The basic algorithm structure is similar to the Alg. 1. From line 8 to line 32 we will finish 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 are not owned by 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, in the low level version, each locale will first get its owned vertices (line 11). Then for each locale, it 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 *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 line 19 to line 21). After the vertex expansion at each level, each locale will scatter the next frontier elements in the *SetNextFRemote* to their owners (line 26 to line 28). At the same time, elements in the next frontier owned by the current locale *SetNextFLocal* will be merged into the distributed array *curFAry* (line 29 to line 31). All the above vertex operations can be done in parallel without data races. Parallel construct *coforall* has 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 line 33 to line 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, use an aggregation method to optimize the communication performance. However, we have to take care of the data distribution and data communication. Even though, this optimization cannot beat the high level data structure implementation for our Delaunay benchmark test (see section 5.2.2). This comparison shows the advantage of Chapel's high level data structure in easy programming and high performance.

5 EXPERIMENTAL RESULTS

5.1 Experimental Setup

To evaluate the results of the proposed integrated solution, we used two kind of graphs. The first is the R-MAT method [7] to generate the testing graphs. The other kind 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' possibility share the remainder 0.25 equally. Each vertex has 2 edges so we will generate 65536, 131072, 262144, and 524288 edges

Algorithm 2: Low level parallel BFS for distributed memory supercomptuers

```
   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             SetNeighbour = {k|k is the neighbour of i}
14             forall (j in SetNeighbour) 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
```

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 tenth DIMACS implementation challenge [3]. The number of edges and vertices will be approximately doubled to reach the next graph in the same benchmark series. All data for these graphs can be found online. For the intents and purposes of this paper, Table 1 summarizes some important information on the graphs selected and utilized for testing. All benchmark graphs utilized were undirected, and some were weighted. The number of connected components are listed as well under the CCs column. For those files where the number of connected components exceeded 1, 99%+ of the number of vertices found in the graph, were also found in the largest component. The diameter pictured is a rough estimate taken by iterating over the first 100 all-pairs shortest paths created by the NetworkX python graph tool. The actual diameter of these graphs may be bigger that what is shown.

Testing of the methods was conducted in an environment composed of a 32 node cluster with a 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 is commonly found in high performing computers. Every node is made up of 512GB of RAM and two Xeon E5-2650 processors with 20 cores total between the two processors. Due to Arkodua being designed primarily for data analysis in a HPC

Table 1: Important parameters for each graph benchmark file utilized.

Name	Vertices	Edges	Weighted	CCs	Biggest CC Size	Diameter(≥)
delaunay_n17	131072	393176	0	1	131072	163
delaunay_n18	262144	786396	0	1	262144	226
delaunay_n19	524288	1572823	0	1	524288	309
delaunay_n20	1048576	3145686	0	1	1048576	442
delaunay_n21	2097152	6291408	0	1	2097152	618
delaunay_n22	4194304	12582869	0	1	4194304	861
delaunay_n23	8388608	25165784	0	1	8388608	1206
delaunay_n24	16777216	50331601	0	1	16777216	1668
rgg_n_2_21_s0	2097148	14487995	0	4	2097142	1151
rgg_n_2_22_s0	4194301	30359198	0	2	4194299	1578
rgg_n_2_23_s0	8388607	63501393	0	4	8388601	2129
rgg_n_2_24_s0	16777215	132557200	0	1	16777215	3009
kron_g500-logn18	210155	10583222	1	8	210141	4
kron_g500-logn19	409175	21781478	1	27	409123	4
kron_g500-logn20	795241	44620272	1	45	795153	4
kron_g500-logn21	1544087	91042010	1	94	1543901	4

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 number locales to check their performance (we will cancel some tests if the execution time is too long to keep the experiments in reasonable time arrangement). 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.

5.2 Experimental Results

For large scale graph analytics, there are two major steps. The first is building the graph into memory at the Arkouda back-end. The second step is conducting different analysis methods on the graph in memory to gain insight from the given graph. Here we use a parallel BFS algorithm to demonstrate how we can conduct analysis on large graphs. In this section, we will provide the experimental results of our graph building and graph analyzing methods.

5.2.1 Graph Building. The experimental results from Fig. 3 to Fig. 10 show the graph building time and the building efficiency of different graphs in Arkouda. We can see that for Fig. 7 and Fig. 9, the building time will increase linearly with the number of edges, no matter how many locales we use. However, it will take more time when handling the same amount of edges with more locales. The reason lies in the data movement overhead among locales. More locales mean that more data movement between distributed memory will be needed. The building efficiency Fig. 8 and Fig. 10 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.

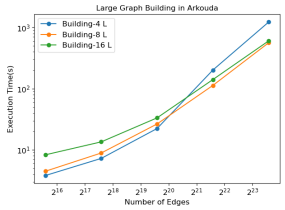

Figure 3: Graph building time (R-MAT).

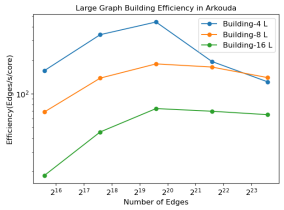

Figure 4: Graph building efficiency (R-MAT).

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 $E \times L$. For all the results, the RMSE (Root Mean Square Error) is less than 390 and the R-squared value is larger than 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 prediction. For examples, for the com-friendster.ungraph.txt [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. It really 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. 3 and Fig. 5 can help us see what happened for different graphs. We can see in Fig. 3, 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 is becoming larger although the total number of edges is less than the number in KRON and RGG benchmark. The major reason is that compared with the benchmark method, 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. 5, 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 resource will also cause loss in performance and efficiency. Beyond the resource bound is the major reason why the graph building efficiency will decrease in Fig. 4 and Fig. 6. 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.

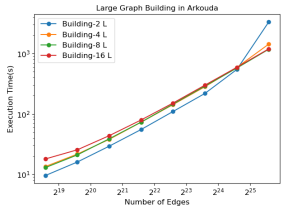

Figure 5: Graph building time (Delaunay).

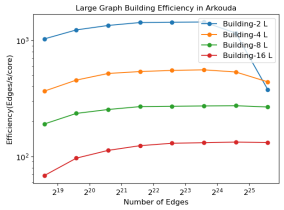

Figure 6: Graph building efficiency (Delaunay).

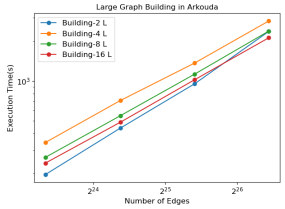

Figure 7: Graph building time (KRON).

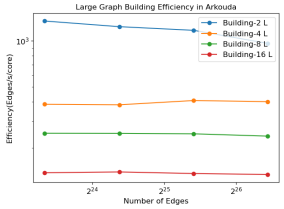

Figure 8: Graph building efficiency(KRON).

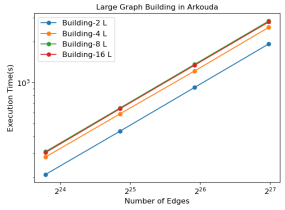

Figure 9: Graph building time (RGG).

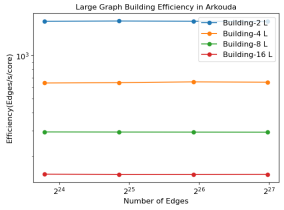

Figure 10: Graph building efficiency (RGG).

5.2.2 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.

To evaluate the performance of the proposed high level multi-locale BFS algorithm and the low level algorithm, we implement the algorithm with different data structures and parallel constructs provided in Chapel to show the performance difference.

We use four different Delaunay benchmark graphs to show the performance of our BFS algorithms. Fig. 11 will be used 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 using the *set* data structure, *SetG* is similar to *BagG*. *DomL* and *DomG* are just like *SetL* and *SetG* except we will replace *DistBag* with *Domain*. In our high level multi-locale BFS algorithm framework, *DistBag*, *Set* and *Domain*

[1] https://snap.stanford.edu/data/com-Friendster.html

can provide the same function to hold the current frontier and the next frontier elements. At the same time, they also have the same or similar method 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. 11 to Fig. 14, 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.

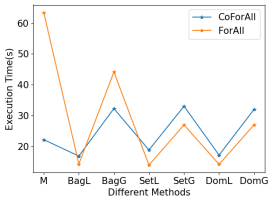

Figure 11: BFS time (*delaunay_n17*).

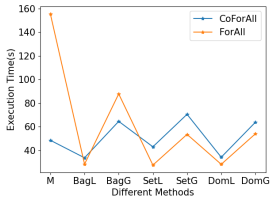

Figure 12: BFS time (*delaunay_n18*).

From the experimental results in Fig. 11 to Fig. 14, 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 better than the shared computing version (BagG, SetG and DomG). It is easy to understand that the shared computing will have a lot 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 if small. The performance of $forall$ will catch up when the graph size become larger (see Fig. 14). The reason is that our low level implementation can avoid idle threads and the number of

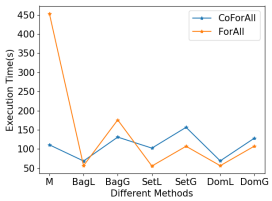

Figure 13: BFS time (*delaunay_n19*).

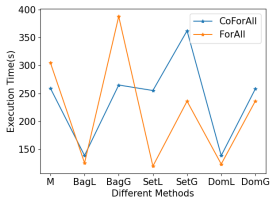

Figure 14: BFS time (*delaunay_n20*).

parallel threads created by $coforall$ is less than the high level implementation (about $\frac{1}{numLocales}$ of the size of 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 add an element into a set in parallel. Surprisingly, DistBag has not shown 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 by different locales independently to get 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 cost for such operations.

The reverse Cuthill–McKee algorithm (RCM) [9] can reduce the bandwidth of a sparse matrix and improve the data access locality. So we employ the RCM method as the pre-processing step to relabel the vertices, in this way we can improve the BFS performance. In Table 2 we give the experimental results without and with RCM pre-processing results. In the column of "RCM", "N" means without RCM pre-processing and "Y" means with RCM pre-processing.

We can see that the RCM method can substantially improve the performance in almost all cases. The best performance can be improved about 1.24 fold for the 4 different benchmarks. We can also see that the best performance is also different. For the performance without RCM pre-processing, the *Set* data structure together with the $forall$ parallel construct can achieve the best performance among all the cases. After RCM pre-processing, the *DistBag* data structure together with the $forall$ parallel construct can achieve the best performance. It seems that *DistBag* data structure is more sensitive to the data locality.

Our experimental results also show that for very large graphs, the $coforall$ parallel construct can cause runtime errors because of limited resources. To avoid this problem, we can set a threshold value to switch between $coforall$ and $forall$ based on the total number of threads and the available resources. 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.

6 RELATED WORK

Since BFS is a very basic graph algorithm, it has been investigated from different aspects. For shared-memory BFS algorithms, Leiserson et al. [17] proposed a data structure "bag" to replace shared

Table 2: 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

queue to improve the parallelism in expanding the next frontier of vertices. Of course, their "bag" is different from the "distbag" in Chapel. However, we share the same idea of employing efficient data structures to support parallel algorithm design. They optimized the implementation of the reducer and all their methods have been integrated into their Cilk++ compiler.

There are many approaches that can exploit specific hardware architecture. For example, Bader et al. [4] employed the fine-grained, low-overhead synchronization Cray MTA-2 computer to develop a load-balanced BFS algorithm using thousands of hardware threads. Mizell et al. [22] implemented the BFS algorithm on the 128-processor Cray XMT system.

There are also many BFS algorithms on GPU. For examples, Luo et al. [18] used a hierarchical data structure at grid, block and warp levels to store and access the frontier vertices, and demonstrate that their algorithm is up to 10 times faster than the Harish-Narayanan algorithm on NVIDIA GPUs and low-diameter sparse graphs. Merrill et al. [20] used efficient prefix sum computations to deliver excellent performance on diverse graphs.

For very large graphs, distributed-memory BFS algorithms are necessary. Beamer et al. [5] proposed a hybrid strategy to combine the "top-down" and "bottom-up" expansions together. The basic idea is employing "top-down" expansion when the frontier has a small number of vertices. Otherwise the "bottom-up" expansion method will be used to avoid searching too many edges. Azad et al. [2] employed a variant of the standard breadth-first search algorithm, reverse Cuthill-McKee algorithm, to improve the performance. The Cuthill–McKee algorithm will relabel the vertices of the graph to reduce the bandwidth of the adjacency matrix. Jiang et al. [16] used both Reverse Cuthill-Mckee algorithm and SIMD executions to improve their BFS algorithm's performance. Fan et al. [12] employed several technologies, such as asynchronous virtual ring method, thread caching scheme and vertex ID reordering to improve the BFS performance.

The major difference between our idea and the existing BFS algorithms is high algorithm design productivity and quick optimization. The basic idea of our graph algorithms design in Chapel is taking advantage of the high level data structure provided by Chapel to simplify the algorithm design and employing the parallel constructs provided by Chapel to exploit the parallelism. Our experimental results show that with the same algorithm framework, small change in data structure or parallel construct can cause very significant performance differences. The purpose of Chapel based graph algorithm design in Arkouda is the productive algorithm design and quick performance optimization to support exploratory data analysis at scale.

7 CONCLUSION

Large graph analytics is a challenging problem and in this paper we present our preliminary work to show how we can use the open source framework Arkouda to handle this problem. The advantage of Arkouda lies in two aspects: high productivity and high performance. High productivity means that end users can use a popular EDA language such as Python to achieve insight from large scale graphs. High performance means that the end users can break the limit of their laptop and personal computer's capabilities in memory and calculation to handle very large graphs using Chapel at the server side. Taking advantage of the high level parallel programming language Chapel, the high performance solution is also highly productive. This means that we can design and develop high performance graph analysis algorithms using Chapel quickly and efficiently.

Based on the basic array data structure in Arkouda, we define a double index graph data representation to exploit the different kinds of array operators in Arkouda. At the same time, our graph representation can enable edge and vertex locating with $O(1)$ time complexity. We have implemented the double index graph data structure in Arkouda. We developed a typical graph algorithm breadth first search in the high level parallel language Chapel to support basic graph analysis in Arkouda. The proposed BFS algorithms show that we can develop parallel algorithms and optimize their performance based on the same algorithm framework in Chapel efficiently. Selecting suitable data structures and parallel constructs can significantly improve the algorithm performance in Chapel.

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.

ACKNOWLEDGMENTS

We appreciate the help from 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] Lada A Adamic, Bernardo A Huberman, AL Barabási, R Albert, H Jeong, and G Bianconi. 2000. Power-law distribution of the world wide web. *Science* 287, 5461 (2000), 2115–2115.

[2] Ariful Azad, Mathias Jacquelin, Aydin Buluç, and Esmond G Ng. 2017. The reverse Cuthill-McKee algorithm in distributed-memory. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 22–31.

[3] David A. Bader, Andrea Kappes, Henning Meyerhenke, Peter Sanders, Christian Schulz, and Dorothea Wagner. 2017. Benchmarking for Graph Clustering and Partitioning. *Encyclopedia of Social Network Analysis and Mining* (2017), 1–11. https://doi.org/10.1007/978-1-4614-7163-9_23-1

[4] David A Bader and Kamesh Madduri. 2006. Designing multithreaded algorithms for breadth-first search and st-connectivity on the Cray MTA-2. In *2006 International Conference on Parallel Processing (ICPP'06)*. IEEE, 523–530.

[5] Scott Beamer, Aydin Buluc, Krste Asanovic, and David Patterson. 2013. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 1618–1627.

[6] John T Behrens. 1997. Principles and procedures of exploratory data analysis. *Psychological Methods* 2, 2 (1997), 131.

[7] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 442–446.

[8] Bradford L Chamberlain, Elliot Ronaghan, Ben Albrecht, Lydia Duncan, Michael Ferguson, Ben Harshbarger, David Iten, David Keaton, Vassily Litvinov, Preston Sahabu, et al. 2018. Chapel comes of age: Making scalable programming productive. *Cray User Group* (2018).

[9] Elizabeth Cuthill and James McKee. 1969. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*. 157–172.

[10] Samrat K Dey, Md Mahbubur Rahman, Umme R Siddiqi, and Arpita Howlader. 2020. Analyzing the epidemiological outbreak of COVID-19: A visual exploratory data analysis approach. *Journal of medical virology* 92, 6 (2020), 632–638.

[11] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. 1999. On power-law relationships of the internet topology. *ACM SIGCOMM computer communication review* 29, 4 (1999), 251–262.

[12] Dongrui Fan, Huawei Cao, Guobo Wang, Na Nie, Xiaochun Ye, and Ninghui Sun. 2020. Scalable and efficient graph traversal on high-throughput cluster. *CCF Transactions on High Performance Computing* (2020), 1–13.

[13] Irving J Good. 1983. The philosophy of exploratory data analysis. *Philosophy of science* 50, 2 (1983), 283–295.

[14] Pieter Hintjens. 2013. *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.".

[15] Andrew T Jebb, Scott Parrigon, and Sang Eun Woo. 2017. Exploratory data analysis as a foundation of inductive research. *Human Resource Management Review* 27, 2 (2017), 265–276.

[16] Zite Jiang, Tao Liu, Shuai Zhang, Zhen Guan, Mengting Yuan, and Haihang You. 2020. Fast and Efficient Parallel Breadth-First Search with Power-law Graph Transformation. *arXiv preprint arXiv:2012.10026* (2020).

[17] Charles E Leiserson and Tao B Schardl. 2010. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. 303–314.

[18] Lijuan Luo, Martin Wong, and Wen-mei Hwu. 2010. An effective GPU implementation of breadth-first search. In *Design Automation Conference*. IEEE, 52–55.

[19] Theo Lynn, Pierangelo Rosati, Binesh Nair, and Ciáran Mac an Bhaird. 2020. An Exploratory Data Analysis of the# Crowdfunding Network on Twitter. *Journal of Open Innovation: Technology, Market, and Complexity* 6, 3 (2020), 80.

[20] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2015. High-performance and scalable GPU graph traversal. *ACM Transactions on Parallel Computing (TOPC)* 1, 2 (2015), 1–30.

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

[22] D Mizell and K Maschhoff. 2009. Early experiences with large-scale XMT systems. In *Proc. Workshop on Multithreaded Architectures and Applications (MTAAP'09)*.

[23] Michael J Quinn and Narsingh Deo. 1984. Parallel graph algorithms. *ACM Computing Surveys (CSUR)* 16, 3 (1984), 319–348.

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

[25] Guido Rossum. 1995. Python reference manual. (1995).

[26] Andrew T Stephen and Olivier Toubia. 2009. Explaining the power-law degree distribution in a social commerce network. *Social Networks* 31, 4 (2009), 262–270.

[27] John W Tukey. 1977. *Exploratory data analysis*. Vol. 2. Reading, MA.