21 Interactive Graph Analytics at Scale in Arkouda

Zhihui Du, Oliver Alvarado Rodriguez, Joseph Patchett, and David A. Bader
New Jersey Institute of Technology

CONTENTS

DOI: 10.1201/9781003033707-26

Data from many real-world applications often can be abstracted as graphs. However, the increasing graph size makes it impossible for existing popular exploratory data analysis tools to handle large 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 work, a succinct double-index data structure is designed to build a static graph and the sketch of a graph stream with much less memory footprint. Two typical graph algorithms, Breadth-First Search (BFS) and triangle counting algorithms, are developed to evaluate the efficiency of the proposed graph analytics workflow. Experimental results show that our method can take advantage of distributed resources to handle large graphs. This work provides the large and rapidly growing Python community a powerful way to handle terabyte and beyond graph data using their laptops. All our methods and code have been implemented in Arkouda and are available from GitHub (https://github.com/Bader-Research/arkouda/tree/streaming).

21.1 INTRODUCTION

A graph is a well-defined mathematical model to formulate the relationship between different objects and is widely used in numerous domains. The edge distributions of many large scale real world problems tend to follow a power-law distribution [20, 1, 48]. More and more emerging applications, such as social networks, cybersecurity and bioinformatics, have data that often comes in the form of real-time graph streams [36]. Over its lifetime, the sheer volume of a stream could be petabytes or more like network traffic analysis in the IPv6 network which has 2^{128} nodes. Dense graph data structures and algorithms will consume much more memory and cannot analyze very large sparse graphs efficiently. Therefore, these applications motivate the challenging problem of designing succinct data structures and highly efficient parallel algorithms to handle large graphs and even larger graph streams.

Exploratory data analysis (EDA) [9, 23, 27] is a critical method in data science. Instead of checking results given a hypothesis with data, EDA primarily is for seeing what the data can tell us beyond the formal modeling or hypotheses testing tasks in an interactive way. In this way, EDA tries to maximize the value of data. Popular EDA methods and tools, which often run on laptops or common personal computers, cannot hold terabyte or even larger graph data sets, let alone produce highly efficient analysis results. Arkouda [26, 25, 39, 38, 46] is an EDA framework under early development that brings together the productivity of Python with world-class high-performance computing. If a graph algorithm can be integrated into Arkouda, it means that data scientists can take advantage of both laptop computing and cloud/supercomputing to do interactive data analysis at scale.

In this chapter, we provide the preliminary solution on integrating sparse graph analysis into Arkouda. The major contributions are as follows:

1. An efficient and succinct Double-Index (DI) data structure, which can be used to build both a static graph and a sketch of a graph stream, is developed in this paper. The DI data structure can achieve $\mathcal{O}(1)$ time complexity in searching incident vertices of a given edge or the adjacency list of a given vertex.
2. Based on the proposed DI data structure, two typical graph algorithms, Breadth-First Search (BFS) algorithm and triangle counting algorithm, are developed based on the high-level parallel language Chapel to evaluate the efficiency of the proposed graph and graph stream analytics workflow and their practical end-to-end performance.
3. All the proposed methods have been integrated into an open-source framework Arkouda. Experimental results show that the proposed DI data structure and algorithms can

support Arkouda to handle different kinds of large graphs and graph streams. This work can help the large and popular data analytics community that exists within Python to conduct interactive graph analytics easily and efficiently on terabytes and beyond of graph data.

21.2 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 [47] 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 bottlenecks of Python. Chapel [15] is a high-level programming language designed for productive parallel computing at scale. It has the same advantages such as being portable and open-source like in Python. Furthermore, it has the scalable and fast features that Python lacks.

Arkouda integrates its front-end Python with its back-end Chapel with a middle, communicative part ZeroMQ [24]. ZeroMQ is used for the data and instruction exchanges 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 bringing 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 allow 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.

21.3 SUCCINCT DOUBLE-INDEX DATA STRUCTURE

In this section, we introduce the details of our double-index data structure. There are three major parts:(1) The detailed description of the edge and vertex arrays. (2) The time and space complexity analysis based on the proposed data structure. (3) A comparison with the CSR data structure.

21.3.1 EDGE INDEX AND VERTEX INDEX

We propose a Double-Index (DI) data structure to support quick searching from a given edge to its incident vertices or from a given vertex to its adjacency list. The two index arrays are called the edge index array and vertex index array. Furthermore, our DI data structure requires a significantly smaller memory footprint for sparse graphs.

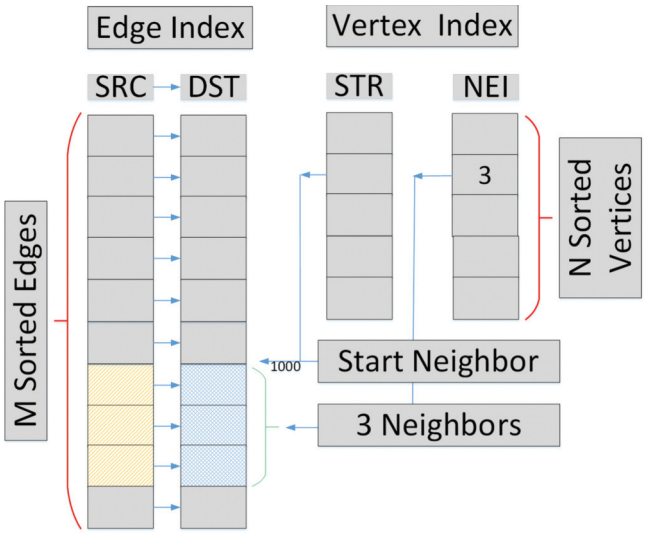


Figure 21.1 Double-Index data structure for graph analytics.

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 integers from 0 to $M-1$ to identify different edges and the integers from 0 to $N-1$ to identify different vertices. For example, given edge $e = \langle i, j \rangle$, 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 combined vertex ID value $(SRC[e], DST[e])$ 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, in Figure 21.1, we let edge e_{1000} have ID 1000. If $e_{1000} = \langle 50, 3 \rangle$, $e_{1001} = \langle 50, 70 \rangle$ and $e_{1002} = \langle 50, 110 \rangle$ are all the edges starting from vertex 50 (a directed graph), then we will let one vertex index array $STR[50] = 1000$ and another vertex index array $NEI[50] = 3$. This means that for given vertex ID 50, the edges starting with vertex 50 are stored in edge index array starting at position 1000 and there are a total of three 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 search the neighbors or adjacency list of any given vertex. Our DI data structure can also support graph weights. If each vertex has a different weight, we use an array V_WEI to express the weight values. If each edge has a weight, we use an array E_WEI to store the different weights.

21.3.2 TIME AND SPACE COMPLEXITY ANALYSIS

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 $NEI[i]$ neighbors 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 given vertex i, the adjacency list of vertex i can be easily expressed as $\langle i, x \rangle$ where x in $DST[STR[i]..STR[i] + NEI[i] - 1]$. Based on the NEI and STR vertex index arrays, we can find

the neighbor vertices or the adjacency list in $\mathcal{O}(1)$ time complexity. For given edge $e = \langle i, j \rangle$, it will be easy for us to find its incident vertices i in array $SRC[e]$ and j in array $DST[e]$ also in $\mathcal{O}(1)$ time complexity.

Regarding the storage space, if the graph has M edges and N vertices, we will need $2M$ memory to store all the edges. Compared with the dense matrix data structure which needs N^2 memory to store all the edges, this is much smaller. To improve the adjacency list search performance, we use $2N$ memory to store the NEI and STR arrays.

Figure 21.1 shows M sorted edges represented by the SRC and DST arrays. Any one of the N vertices n_k can find its neighbors using NEI and STR arrays with $\mathcal{O}(1)$ time complexity. For example, given edge $\langle i, j \rangle$, if vertex j's starting index in SRC is 1000, it has three adjacency edges then such edges can be found starting from index position 1000 in array SRC and DST using NEI and STR arrays directly. This figure shows how the NEI and STR arrays can help us locate neighbors and adjacency lists quickly.

For an undirected graph, an edge $\langle i, j \rangle$ means that we can also arrive at i from j. We may use the data structures SRC, DST, STR, NEI to search the neighbors of j in SRC. However, this search cannot be done in $\mathcal{O}(1)$ time complexity. To improve the search performance, we introduce another four arrays called reversed arrays $SRCr, DSTr, STRr, NEIr$. For any edge $\langle i, j \rangle$, where its i vertex is in SRC and j vertex in DST, we will have the corresponding reverse edge $\langle j, i \rangle$ in $SRCr$ and $DSTr$, where $SRCr$ has the exactly same elements as in DST and $DSTr$ has the exactly same elements as in SRC. $SRCr$ and $DSTr$ are also sorted and $NEIr$ and $STRr$ are the array of the number of neighbors and the array of the starting neighbor index just like the directed graph. So, for a given vertex i of an undirected graph, the neighbors 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 $\langle i, x \rangle$ where x in $DST[STR[i]..STR[i] + NEI[i] - 1]$ or $\langle i, x \rangle$ where x in $DSTr[STRr[i]..STRr[i] + NEIr[i] - 1]$.

Given a directed graph with M edges and N vertices, our data structure will need $2(M + N)$ integer (64 bits) storage, or $\frac{M+N}{4}$ bytes. For an undirected graph, we will need twice the storage of a directed graph. For weighted vertices and weighted edges, additional N, M integer storage will be needed, respectively.

21.3.3 COMPARISON WITH CSR

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 value array of CSR is just like the edge weight array in DI; the column array of CSR is the same as the DST array in DI; the row array of CSR is very close to the STR in DI. CSR is a vertex-oriented data structure, and it can support quick search from any vertex to its adjacency list. DI has the same function like CSR.

The major difference between DI and CSR is that DI provides the explicit mapping between an edge ID to its incident vertices, but CSR does not. This difference has two effects. (1) DI can support another kind of quick search, from any edge ID to its incident vertices, however, CSR cannot because CSR does not provide the source vertex of a given edge ID. (2) DI can support edge oriented graph partition (see Subsection 21.5.1) based on the explicit edge index arrays to achieve load balance, however, CSR cannot.

Another difference is that in DI we use an array NEI to explicitly represent the number of neighbors of given vertex to remove the ambiguous meaning of the row index array in CSR. $NEI[v]$ can be replaced by $STR[v+1] - STR[v]$ if we extend one element in STR array and use the meaning of the row index array in CSR. In our DI data structure, the meaning of $STR[v]$ is not

ambiguous. It is the starting position of vertex v in the edge index array or $STR[v] = -1$ if v has no edge. However, in CSR, if v has no edges, the value of $STR[v]$ is not -1, and it is the starting position of the next nonzero element after v or the total number of nonzero elements. So, for any case, when we need to make sure $STR[v]$ is really the starting position of v, we must execute an additional check because the value of $STR[v]$ itself cannot provide such information. When we use the DI data structure to express a much smaller graph sketch, the number of the total vertices N is much smaller than before. So, in DI, we use an additional N size array NEI (much smaller than the original graph) to make the parallel operations on STR have clear semantics and easy to understand.

21.4 MULTILOCALE BREADTH-FIRST SEARCH AND TRIANGLE COUNTING ALGORITHMS

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. Distributed memory architectures are considered as multiple locales. Multilocale algorithms can take advantage of both the distributed resources and parallel resources of single shared memory computing node.

21.4.1 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, they 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. We have developed two versions of our parallel BFS algorithm in Arkouda. The first is the high-level multilocale version and the second is the corresponding low-level version. We will give the details in the following subsections.

21.4.1.1 High-Level Multilocale 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 multilocale 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 multilocale 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 Algorithm 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

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 distributed 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 the 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
```

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 25, we will continue the standard loop if the current search frontier is not empty. From line 8 to 21, we use the *coforall* parallel construct to execute the search on each locale in parallel. From line 10 to 20, we will execute a 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 neighbors *SetNeighbor* of the current vertex *i*. Since some neighbors 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 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),

and 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. 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 locality and avoid expanding the same edges on multiple locales.

21.4.1.2 Low-level Multilocale BFS Algorithm

In the high-level BFS algorithm, we use two *DistBag* classes to hold 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 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 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 Algorithm 2.

From line 1 to 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 Algorithm 1. From line 8 to 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 a parallel search on all the vertices in 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 Algorithm 1 is that we put the expanded elements into different sets. If the elements are local, we put them into the *SetNextFLocal* set in parallel (from line 16 to 18). If they are not owned by the current locale, we will put them into *SetNextFRemote* (from line 19 to 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 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 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 line 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 and this optimization cannot beat the high-level data structure implementation for our Delaunay benchmark test (see Section 21.7.2.2). This comparison shows the advantage of Chapel's high-level data structure for easy programming and high performance.

21.4.2 PARALLEL TRIANGLE COUNTING ALGORITHM

For graph stream analysis, we can directly employ existing exact graph algorithms onto our sketch because the sketch is also expressed as a graph. Here, we will use a typical graph analysis algorithm, triangle counting, to show how we can develop optimized exact algorithms based on our *DI* data structure.

To improve the performance of a distributed triangle counting algorithm, two important problems are maintaining load balancing and avoiding remote access or communication as much as possible.

In this work, we will develop a multilocale exact triangle counting algorithm for distributed memory clusters.

For power law graphs [48, 1, 20], a small number of vertices may have high degrees. So, if we divide the data based on number of vertices, it is easy to cause an unbalanced load. Our method divides the data based on the number of edges. At the same time, our *DI* data structure will keep the edges connected with the same vertex together. So, the edge partition method will result in good load balancing and data access locality.

However, if each locale just directly employs the existing edge iterator [2] on its edges, the reverse edges of undirected graphs are often not in the same locale. This will cause new load balancing problem. So, we will first generate the vertices based on the edges distributed to different locales. Then, each locale will employ the vertex iterator to calculate the number of triangles. So, the combined edge-vertex iterator method is the major innovation for our triangle counting method on distributed systems.

When we employ the high-level parallel language Chapel to design the parallel exact triangle counting algorithm, there are two additional advantages. (1) Our *DI* data structure can work together with the *coforall* or *forall* parallel construct of Chapel to exploit the parallelism. (2) We can take advantage of the high-level module *Set* provided by Chapel to implement parallel set operation easily and efficiently.

At a high level, our proposed algorithm takes advantage of the multilocale 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 compute node can also process their own data in parallel. The following steps are needed to implement the multilocale exact triangle counting algorithm. (1) The *DI* graph edge data should be distributed evenly onto the distributed memory to balance the load. (2) Each distributed node only counts the triangle including the vertices assigned to the current node. (3) All the triangles calculated by different nodes should be summed together to get the exact number of triangles.

Our multilocale exact triangle counting algorithm is described in Algorithm 3. For a given graph sketch partition $G_{sk} = \langle E_{sk}, V_{sk} \rangle$, we will use an array *subTriSum* to keep each locale's result (line 2). Here in line 3, we use *coforall* instead of *forall* to allow each *loc* in *Locales* to execute the following code in parallel so we can fully exploit the distributed computing resources. The code between line 3 and line 17 will be executed in parallel on each locale. Each locale will use a local variable *triCount* to store the number of triangles (line 5). Lines 6 and 7 are important to implement load balancing. Assume edges from e_{start} to e_{end} are assigned to current locale,

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 the 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
```

we can get the corresponding vertex ID $StartVer = SRC[e_{start}]$ and $EndVer = SRC[e_{end}]$ as the vertices interval the current locale will handle. Since different locales may have different edges with the same starting vertex, the interval $[StartVer..EndVer]$ of different locales may overlap. At the same time, some starting vertex in *SRCr* index array may not appear in *SRC*, so we should also make sure there is no "hole" in the complete interval $[0..|V_{sk}|-1]$. In line 7, we will make sure all the intervals will cover all the vertices without overlapping.

Algorithm 3: Edge-Vertex Iterator triangle counting algorithm

```
1  TC(G_sk = ⟨E_sk, V_sk⟩)
   /* G_sk is the given graph sketch partition, E_sk, V_sk are edge and
      vertex sets.                                                    */
2  var subTriSum=0: [0..numLocales-1] int;//store each locale's number of triangles
3  coforall (loc in Locales) do
4      if (current loc is my locale) then
5          var triCount=0:int;//initialize number of local triangles
6          Assign StartVer and EndVer based on edge index array
7          Adjust StartVer and EndVer to cover all vertices and avoid overlapping
8          forall (u in StartVer..EndVer with (+ reduce triCount)) do
9              u_adj = {x|⟨u,x⟩ ∈ E_sk ∧ (x > u)} //build the u adjacency vertex set in parallel
10             forall v ∈ u_adj do
11                 v_adj = {x|⟨v,x⟩ ∈ E_sk ∧ (x > v)} //build the v adjacency vertex set in
                     parallel
12             end
13             TriCount+ = |u_adj ∩ v_adj|;
14         end
15         subTriSum[here.id] = triCount;
16     end
17 end
18 return sum(subTriSum)
```

Our method includes the following steps: (1) If the current locale's *StartVer* is the same as the previous locale's *EndVer*, this means that one vertex's edges have been partitioned into two different locales. We will set $StartVer = StartVer + 1$ to avoid two locales executing the counting on the same vertex. (2) If current locale's *StartVer* is different from the previous locale's *EndVer*, and the difference is larger than 1, this means that there is a "hole" between the last locale's vertices and the current locale's vertices. So, we will let the current locale's *StartVer* = the last locale's $EndVer + 1$. (3) If the current locale is the last locale, we will let its *EndVer* = the last vertex ID. If the current locale is the first locale, we will let $StartVer = 0$.

From line 8 to 14, we will count all the triangles starting from the vertices assigned to the current locale in parallel. In line 9, we will generate all the adjacent vertices u_{adj} of the current vertex u and its vertex ID is larger than u. From line 10 to 12, for any vertex $v \in u_{adj}$, we will generate all the adjacent vertices v_{adj} of current vertex v and its vertex ID is larger than v. So, the number of vertices in $u_{adj} \cap v_{adj}$ is the number of triangles having edge $\langle u, v \rangle$. Since we only calculate the triangles whose vertices meet $u < v < w$, we will not count the duplicate triangles. In this way, we can avoid the unnecessary calculation. In line 15, each locale will assign its total number of triangles to the corresponding position of array *subTrisum*. At the end, in line 18, when we sum all the number of triangles of different locales, we will get the total number of triangles.

21.5 GRAPH ANALYTICS WORKFLOW

The major steps to execute graph analysis in Arkouda are as follows: (1) acquiring graph data from a data generator or a graph file; (2) building the double index graph expression; and (3) executing queries on memory graph to generate exact or approximate solutions.

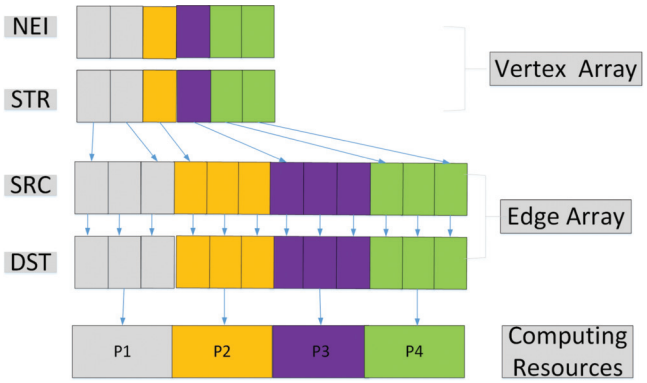


Figure 21.2 Edge based sparse graph partition.

The specific components in steps (2) and (3) are different for a static graph and a graph stream. For a static graph, we will build the complete graph expression in memory in step (2) and generate the exact query result on the graph in step (3). For a dynamic graph stream, we will build the graph sketch of the graph stream in step (2) and generate the approximate query result on the graph stream in step (3). In the following parts, we will introduce how we build a complete graph expression in distributed memory, how we build a graph sketch based on a sliding window stream, and how we develop regression models to generate approximate query solutions.

21.5.1 EDGE ORIENTED SPARSE GRAPH PARTITIONING AND BUILDING

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 a 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 computing nodes instead.

Figure 21.2 shows the basic idea of our sparse graph partition method. The edge arrays *SRC* and *DST* will be distributed using *BLOCK* method onto different computing nodes to make sure most of the nodes will have the same load. When we assign an edge's vertex entry in index array *NEI* and *STR* to the same processors, this approach 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 number of elements assigned to different processors may be very different. In our current implementation, we just partition *NEI* and *STR* arrays evenly as the edge arrays.

21.5.2 SLIDING WINDOW STREAM BASED SKETCH BUILDING

Our stream model has the following features: (1) Only one pass is needed to build the graph sketch. (2) A parameter, Shrinking Factor, is introduced to allow users to directly control the size of a sketch. So, the users' requirement can be used to build a sketch with much less space. (3) The graph stream sketch is divided into three different partitions (we also refer to the three independent small graphs as sub-sketches), and this method can help us avoid global heterogeneity and skewness but take advantage of the local similarity. (4) Our sketch is expressed as a graph so the exact graph analysis method on a complete graph can also be used on our different sub-sketches.

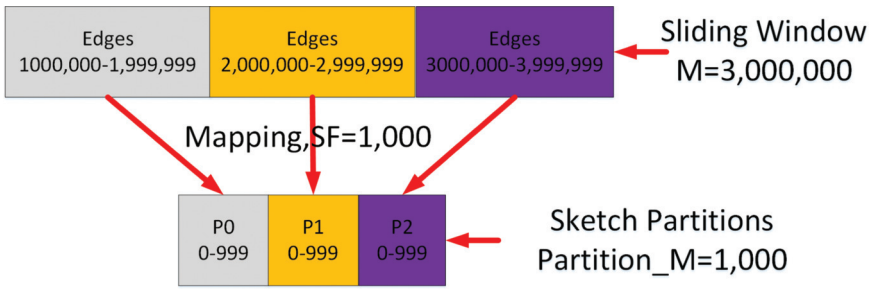


Figure 21.3 Mapping a graph stream into a multi-partition sketch.

(5) A general multivariate linear regression model is used to generate the approximate solution based on the results from different sub-sketches. The regression method can be employed to support general graph analysis methods.

We first describe our method when the stream comes from a sliding window, which means that only the last M edges falling into the given time window will be considered.

We define the shrinking factor SF (a positive integer) to constrain the space allocation of a stream sketch. This means that the size of the edges and vertices of the graph sketch will be $\frac{1}{SF}$ of the original graph. The existing research on sketch building [49] shows that multiple pairwise independent hash function [17] methods can often generate better estimation results because they can reduce the probability of hash collisions. In our solution, we sample on different parts of the given sliding window instead of the complete stream and then map the sampled edges into multiple (here, three) independent partitions to completely avoid collisions. We name the three partitions as Head, Mid, and Tail partitions or sub-sketches, respectively. Each partition will be assigned with $\frac{1}{3SF}$ space of the original graph.

Since we divide the sketch into three independent partitions, we selected a very simple hash function. For a given sliding window, we let the total number of edges and vertices in this window be M and N. Then, the size of each partition will have $Partition_M = \frac{M}{3SF}$ edges and $Partition_N = \frac{N}{3SF}$ vertices. For any edge $e = \langle i, j \rangle$ from the original graph, we will map the edge ID e to $mod(e, Partition_M)$. At the same time, its two vertices i and j will be mapped to $mod(i, Partition_N)$ and $mod(j, Partition_N)$. If $e < \frac{M}{3}$, then we will map $e = \langle i, j \rangle$ to the Head partition. If $e \geq \frac{2M}{3}$, we will map $e = \langle i, j \rangle$ to the Tail partition. Otherwise, we will map $e = \langle i, j \rangle$ to the mid partition.

Figure 21.3 is an example to show how we map totally 3,000,000 edges in a given sliding window into a sketch with three partitions. The edges from 1,000,000 to 1,999,999 are mapped to the head partition p0 that has 1000 edges. The edges from 2,000,000 to 2,999,999 are mapped to the mid partition p1 and the edges from 3,000,000 to 3,999,999 are mapped to the tail partition p2. Each partition is expressed as a graph.

After the three smaller partitions are built, we sort the edges and create the DI data structure to store the partition graphs.

We map different parts of a stream into corresponding sub-sketches and this is very different from existing sketch building methods where they map the same stream into different sketches with independent pairwise hash functions so each sketch is an independent summarization of the stream. However, in our method, one sub-sketch can only stand for a part of the stream and we use different sub-sketches to capture the different local features in a stream. Our regression model (see Subsection 21.5.3) will be responsible for generating the final approximate solution.

More partitions will help to capture more local features of a stream. However, we aim for a regression model that is as simple as possible. Too many partitions will make our regression model have more variables and become complicated. We select three as the default number of partitions because this can achieve sufficient accuracy based on our experimental results. Of course, for some special case, we may choose more partitions but use the same proposed framework.

21.5.3 REAL-WORLD GRAPH DISTRIBUTIONS BASED REGRESSION MODEL

Instead of developing different specific methods for different graph analysis problems, we propose a general regression method to generate the approximate solution based on the results of different sub-sketches.

One essential objective of a stream model is generating an accurate estimation based on the query results from its sketch. The basic idea of our method is exploiting the real-world graph distribution information to build an accurate regression model. Then we can use the regression model to generate approximate solutions for different queries.

Specifically, for the triangle counting problem, when we know the exact number of triangles in each sub-sketch (see Table 21.6), the key problem is how we can estimate the total number of triangles.

To achieve user-required accuracy for different real-world applications, our method exploits the features of real-world graph distributions. Many sparse networks, social networks in particular, have an important property - their degree distributions follow the power law distribution. Normal distributions are often used in the natural and social sciences to represent real-valued random variables whose distributions are not known. So, we develop two regression models for the two different graph degree distributions.

We let E_H, E_M, E_T be the exact number of triangles in the Head, Mid and Tail sub-sketches. The exact triangle counting algorithm is given in Subsection 21.4.2.

For normal degree distribution graphs, we assume the total number of triangles has a linear relationship with the number of triangles in each sub-sketches. We take E_H, E_M, E_T as three randomly sampling results of the original stream graph to build our multivariate linear regression model. TC_{Normal} is the estimated number of triangles and the regression model is given in Eq. 21.1.

$$TC_{Normal} = a \times E_H + b \times E_M + c \times E_T \tag{21.1}$$

For power law graphs, the sampling results of E_H, E_M, E_T can be significantly different because of the skewness in degree distribution. A power law distribution has the property that its log-log plot is a straight line. So, we assume the logarithmic value of the total number of triangles in a stream graph should have a linear relationship with the logarithmic values of the exact number of triangles in different degree sub-sketches. Then, we have two ways to build the regression model. One is unordered and another is ordered. The unordered model can be expressed as in Eq. 21.2. In this model, the relative order information of sampling results cannot be captured.

$$TC_{powerlaw,log} = a \times E_{H,log} + b \times E_{M,log} + c \times E_{T,log} \tag{21.2}$$

where $TC_{powerlaw,log} = log(TC_{powerlaw})$ is the logarithmic value of the estimated value $TC_{powerlaw}$; $E_{H,log} = log(E_H)$, $E_{M,log} = log(E_M)$, and $E_{T,log} = log(E_T)$; $log(E_H)$, $log(E_M)$, and $log(E_T)$ are the logarithmic values of E_H, E_M, and E_T respectively.

Then, we refine the regression model for power law distribution as follows. We get E_{min}, E_{median} and E_{max} by sorting E_H, E_M, and E_T. They mean the minimum, median, and maximum sampling values of the number of triangles in different sub-sketches. They will be used

to stand for the sampling results on a power law distribution at the right (low possibility), middle (a little bit high possibility) and left part (very high possibility). We let $E_{min,log} = log(E_{min})$, $E_{median,log} = log(E_{median})$, and $E_{max,log} = log(E_{max})$. Our ordered multivariate linear regression model for power law graphs is given in Eq. 21.3.

$$TC_{powerlaw,log} = a \times E_{min,log} + b \times E_{median,log} + c \times E_{max,log} \tag{21.3}$$

The accuracy of our multivariate linear regression model is given in Subsection 21.7.4.1 and we can see that the ordered regression model is better than the unordered regression model. Both of our normal and ordered power law regression models achieve very high accuracy.

21.6 INTEGRATION WITH ARKOUDA

Based on the current Arkouda framework, the communication mechanism can be directly reused between Python and Chapel. To support large graph analysis in Arkouda, the major work is implementing the graph data structure and corresponding analysis functions in both Python and Chapel package. In this section, we will introduce how we define the graph classes to implement our *DI* data structure and develop a corresponding Python benchmark to drive the BFS and triangle counting methods.

21.6.1 GRAPH CLASSES DEFINITION IN PYTHON AND CHAPEL

Our *DI* data structure can also support graph weight. If each vertex has a different weight, we use an array V_WEI to express the weight values. If each edge has a weight, we use an array E_WEI to stand for different weights. Based on our *DI* data structure, four classes: directed graph (GraphD), directed and weighted graph (GraphDW), undirected graph (GraphUD), undirected and weighted graph (GraphUDW) are defined in Python. Four corresponding classes SegGraphD, SegGrapDW, SegGraphUD, SegGraphUDW are also defined in Chapel to present different kind of graphs. In our class definition, a directed graph is the base class. Then, undirected graph and directed and weighted graph are the two child classes. Finally, undirected and weighted graph will inherit from undirected graph.

All the graph classes including their major attributes are given in Table 21.1. The left columns are the Python classes and their descriptions. The right columns are the corresponding Chapel classes and their descriptions. Based on the graph class definition, we can develop different graph algorithms.

21.6.2 BFS AND TRIANGLE COUNTING BENCHMARK

For BFS graph analysis, we develop two Python functions to get the graph data. The first is "rmat_gen" function that can use the R-MAT method [14] to generate graph based on different parameters. The second is "graph_file_read" function that can directly read a graph from a file. When the graph is built in memory, we will call the "graph_bfs" function (it will call the BFS algorithm Algorithm 1) to execute BFS analysis.

For triangle counting analysis, to compare the performance of exact method and our approximate method, we implement two triangle counting-related functions. For exact method, we reuse the "graph_file_read" Python function and develop the "graph_triangle" Python function to read a complete graph file and call our triangle counting kernel algorithm (Algorithm 3) to calculate the number of triangles.

Table 21.1
Double-Index Sparse Graph Class Definition in Python and Chapel

Python		Chapel	
Class	**Attribute and Description**	**Class**	**Attribute and Description**
GraphD	n_vertices : number of vertices *n_edges*: number of edges *directed*: directed graph or not *weighted*: weighted graph or not *src* : the source of every edge in the graph *dst* : the destination of every edge in the graph *start_i* : starting index of all the vertices in *src* *neighbor*: number of neighbors all the vertices	SegGraphD	*n_vertices* : number of vertices *n_edges*: number of edges *directed*: directed graph or not *weighted*: weighted graph or not *srcName/src* : the name and data of source array *dstName/dst*: the name and data instance of the destination array *startName/start_i*: the name and data instance of the starting index array *neighborName/neighbor*: the name and data instance of the neighbor array
GraphDW(GraphD)	*v_weight* : weight of vertex *e_weight*: weight of edge	SegGraphDW:SegGraphD	*v_weightName/v_weight* : name and data instance of the vertex weight array *e_weightName/e_weight*:name and data instance of the edge array
GraphUD(GraphD)	*srcR*: the source of every reverse edge in the graph *dstR* : the destination of every reverse edge in the graph *start_iR* : starting index of all the vertices in reverse *src* *neighborR*: number of neighbors all the vertices in *srcR*	SegGraphUD:SegGraphD	*srcNameR/srcR* : name and data instance of the reverse source array *dstNameR/dstR*: the name and data instance of the reverse destination array *startNameR/start$_i$R*: the name and data of the reverse starting index array *neighborNameR*: the name and data of the reverse neighbor array
GraphUDW(GraphUD)	*v_weight* : weight of vertex *e_weight*: weight of edge	SegGraphUDW:SegGraphUD	*v_weightName/v_weight* : the name and data of the vertex weight array *e_weightName*:the name and data of the edge array

For our approximate method, we implement the "stream_file_read" function to simulate the sliding window stream by reading a graph file's edges one by one. The graph sketch introduced in Section 21.5.2 will be built automatically when we read the edges.

After the sketch of a sliding window stream is built in the memory, we will call the "stream_tri_cnt" function to first calculate the exact number of triangles in each sub-sketch (it will also call our kernel triangle counting Algorithm 3) and then use the regression model to estimate the total number of triangles in the given stream.

For the power law distribution regression model, in the benchmark, we also implement single variable regression models using the maximum, minimum, and median result of the three sub-sketches respectively. Our testing results show that the proposed regression method in Subsection 21.5.3 can achieve better accuracy.

21.7 EXPERIMENTS

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. Due to Arkodua being designed primarily for data analysis in a HPC setting, an architecture setup that aptly fits an HPC scenario was chosen for testing.

21.7.1 EXPERIMENTAL SETUP FOR BFS ANALYSIS

To evaluate the results of the proposed integrated solution, we used two kinds of graphs. The first is the R-MAT method [14] 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 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 [4]. 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 21.2 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.

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

Table 21.2
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

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 multilocale algorithm, we will show how a simple replacement in the data structure and parallel construct can affect the performance significantly.

21.7.2 EXPERIMENTAL RESULTS OF BFS ALGORITHM

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.

21.7.2.1 Graph Building

The experimental results from Figures 21.4–21.11 show the graph building time and the building efficiency of different graphs in Arkouda. We can see that for Figures 21.8 and 21.10, 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 Figures 21.9 and 21.11 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.

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 Figures 21.4 and 21.6 can help us see what happened for different graphs. We can see in Figure 21.4, 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 Figure 21.6, 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 Figures 21.5 and 21.7. 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.

21.7.2.2 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 multilocale 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. Figure 21.12 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 Algorithm 2. *BagL* is the result of high level multilocale Algorithm 1. *BagG* means that we will

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

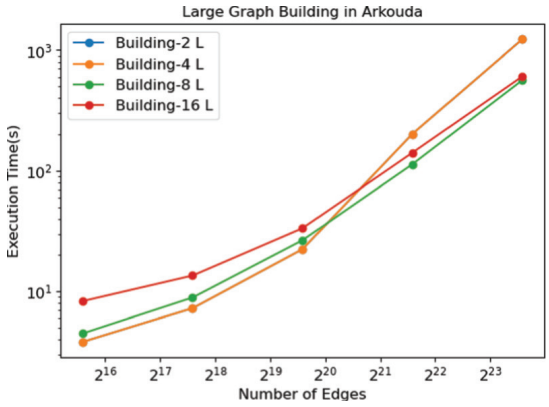


Figure 21.4 Graph building time (R-MAT).

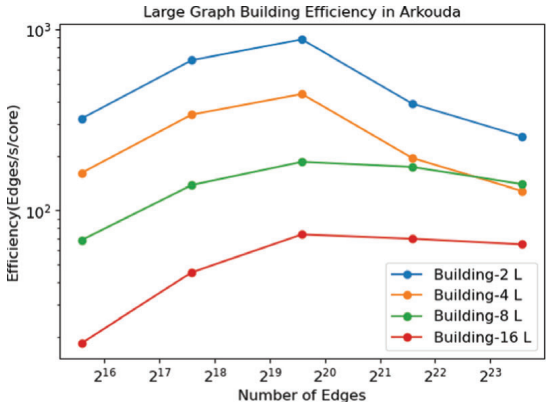


Figure 21.5 Graph building efficiency (R-MAT).

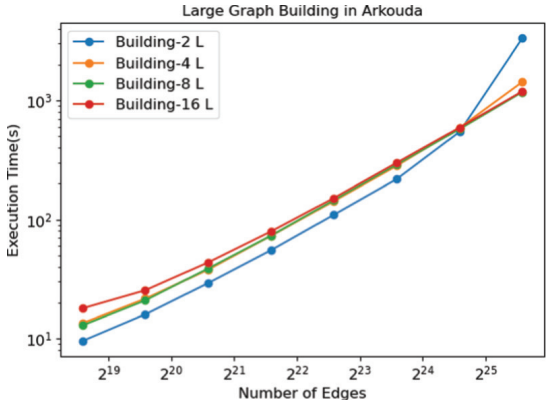


Figure 21.6 Graph building time (Delaunay).

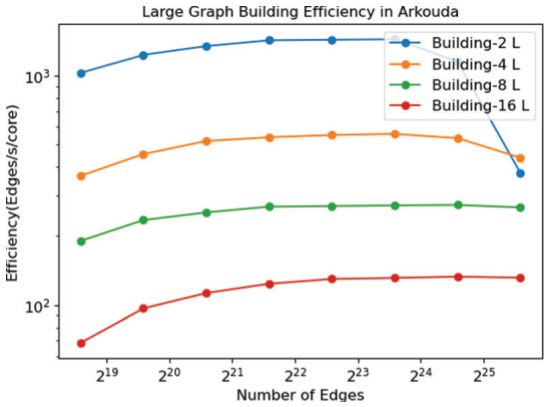


Figure 21.7 Graph building efficiency (Delaunay).

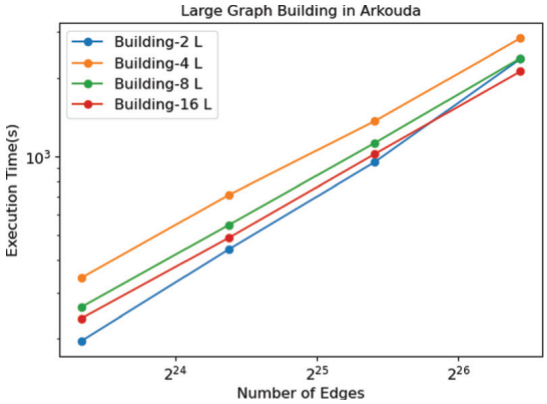


Figure 21.8 Graph building time (KRON).

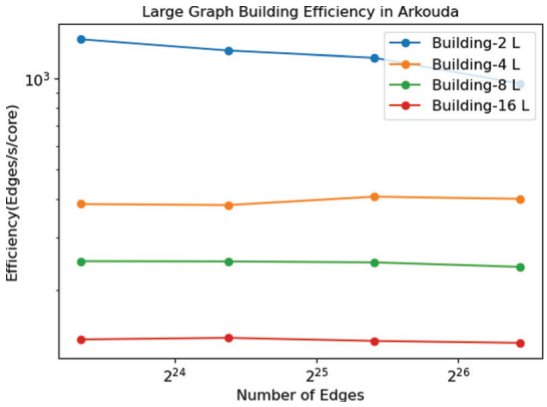


Figure 21.9 Graph building efficiency(KRON).

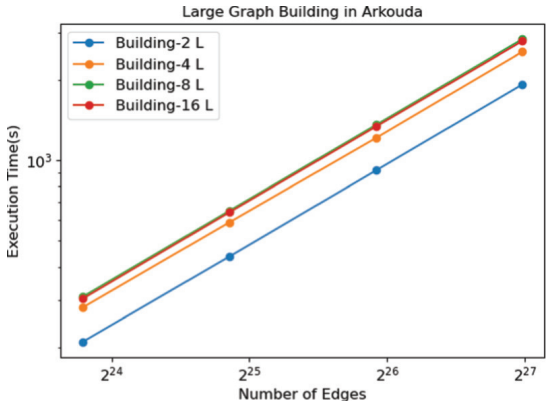


Figure 21.10 Graph building time (RGG).

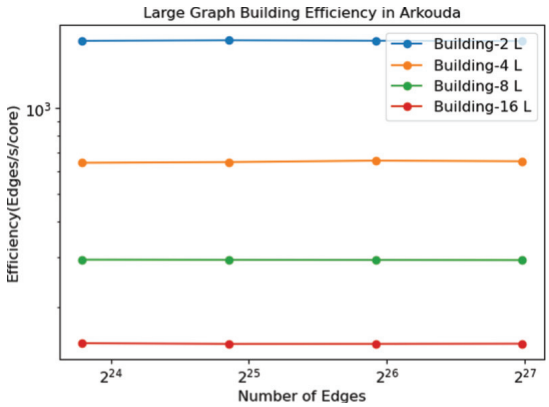


Figure 21.11 Graph building efficiency (RGG).

remove line 11 of Algorithm 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 Algorithm 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 multilocale BFS algorithm framework, *DistBag*, *Set* and *Domain* 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 multilocale BFS methods in Figures 21.12–21.15, the legend *ForAll* means we will use $forall$ parallel construct to expand the vertex at line 10 in Algorithm 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 Algorithm 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 Figures 21.12–21.15, we have the following observations: (1) For all the data structures *DistBag*, *Set* and *Domain*, the performance of distributed parallel

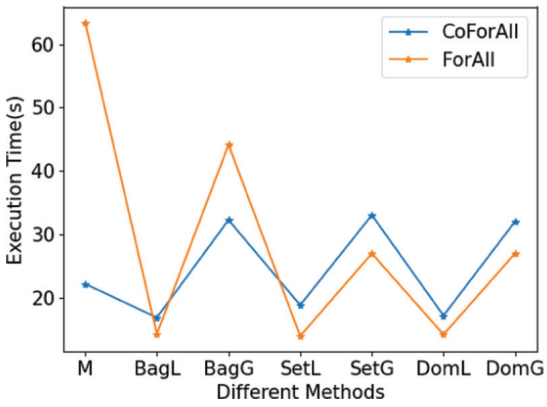


Figure 21.12 BFS time (*delaunay_n*17).

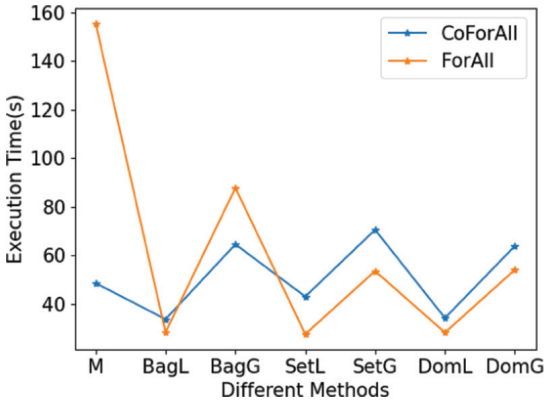


Figure 21.13 BFS time (*delaunay_n*18).

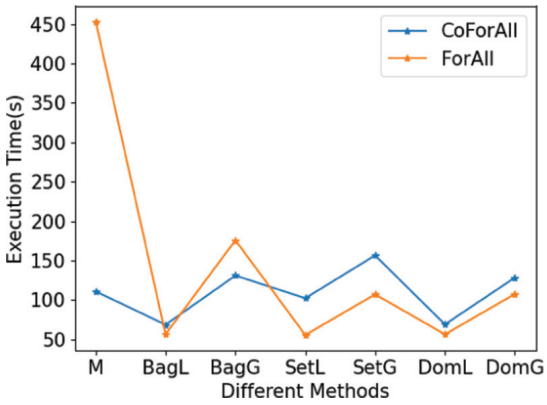


Figure 21.14 BFS time (*delaunay_n*19).

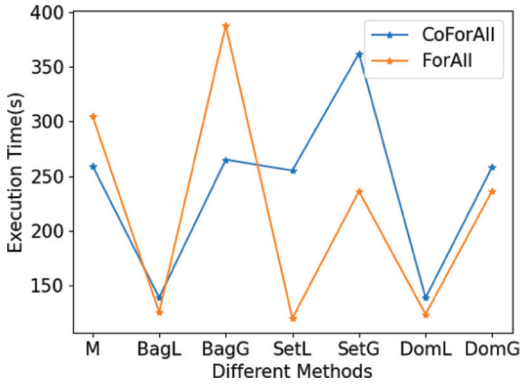


Figure 21.15 BFS time (*delaunay_n*20).

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 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 Figure 21.15). 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 $\frac{1}{numLocales}$ 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 adding 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. We need to send the remote elements to their owners. We will incur additional cost for such operations.

The reverse Cuthill–McKee algorithm (RCM) [18] 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 21.3 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.

Table 21.3
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

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 four 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.

21.7.3 EXPERIMENTAL SETUP FOR TRIANGLE COUNTING

We will use graph files and read the edges line by line to simulate the sliding window of a graph stream. Selected graphs for testing were chosen from various publicly-available data sets for the benchmarking of graph methods. These include data sets from the 10th DIMACS Implementation Challenge [6], as well as some real-life graph data sets from Stanford Network Analysis Project (SNAP).[2] Pertinent information about these data sets for the triangle counting method, are displayed in Table 21.4.

[2] http://snap.stanford.edu/data/index.html.

Table 21.4
Graph benchmarks utilized for the triangle counting algorithm.

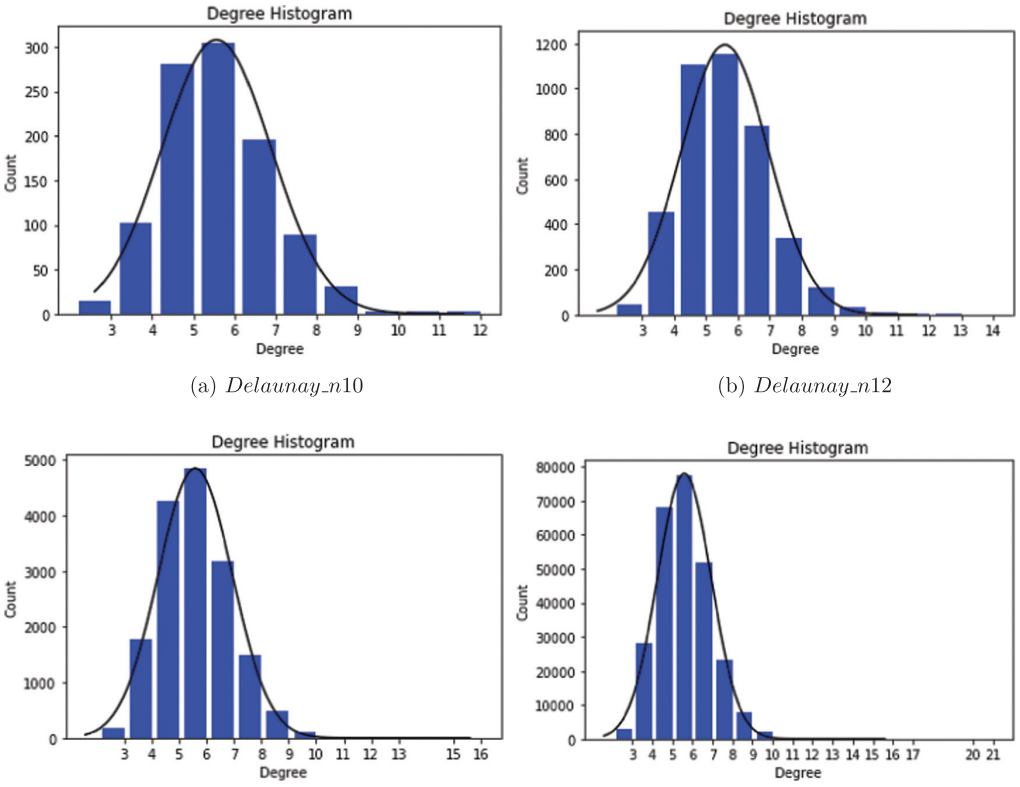
Degree Distribution	Graph Name	Edges	Vertices	Mean	Standard Deviation
Normal	delaunay_n10	3056	1024	5.96875	3.10021
	delaunay_n11	6127	2048	5.98340	3.76799
	delaunay_n12	12264	4096	5.98828	3.49865
	delaunay_n13	24547	8192	5.99292	3.25055
	delaunay_n14	49122	16384	5.99634	3.30372
	delaunay_n15	98274	32768	5.99817	3.37514
	delaunay_n16	196575	65536	5.99899	3.35309
	delaunay_n17	393176	131072	5.99938	3.26819
	delaunay_n18	786396	262144	5.99972	3.2349
Degree Distribution	**Graph Name**	**Edges**	**Vertices**	**a**	**k**
power law	amazon	925872	334863	1920877	2.81
	dblp	1049866	317080	3299616.65	2.71
	youtube	2987624	1134890	701179.83	1.58
	lj	34681189	3997962	50942065.49	2.40
	orkut	117185083	3072441	40351890.91	2.12
	ca-HepTh	51971	9877	23538.9	2.29
	ca-CondMat	186936	23133	68923.6	2.23
	ca-AstroPh	396160	18772	46698.3	1.84
	email-Enron	367662	36692	10150.2	1.54
	ca-GrQc	28980	5242	6610.35	2.04
	ca-HepPh	237010	12008	4601.72	1.44
	loc-brightkite_edges	214078	58228	44973.1	1.88

Based on the graphs' degree distribution, we divide them into two classes, normal distribution and power law distribution. Prior research has shown that some real-world graphs follow power law distributions. Normal distributions are also found ubiquitously in the natural and social sciences to represent real-valued random variables whose distributions are not known. For normal distributions, their typical parameters, the mean and standard deviation fitted from the given graphs, are listed in Table 21.4 too. Figure 21.16 shows the comparison between the fitting results and the original graph data. We can see that the fitting results can match with the original data very well. In other words, the Delaunay graphs follow the normal distribution.

For graphs that follow the power law distribution $y(x) = a \times x^{-k}$, we also give their fitting parameter a and k in Table 21.4. Figure 21.17 gives some examples to show such graphs. We can see that in general, their signatures are close to power laws. However, we can also see that some data does not fit a power law distribution very well.

21.7.4 EXPERIMENTAL RESULTS OF TRIANGLE COUNTING

In this section, we will evaluate the proposed triangle counting algorithm from two aspects: (1) The accuracy of the proposed approximate solution. (2) The memory and execution time savings based on the proposed graph sketch.



(a) *Delaunay_n10* (b) *Delaunay_n12*

(c) *Delaunay n14* (d) *Delaunay n18*

Figure 21.16 Normal distribution graphs. (a) *Delaunay_n10*. (b) *Delaunay_n12*. (c) *Delaunay_n14*. (d) *Delaunay_n18*.

21.7.4.1 Accuracy

In order to evaluate the accuracy of our method, we conducted experiments on 9 normal distribution and 12 power law distribution benchmark graphs with shrink factors 4, 8, 16, 32, and 64. So in total we have 105 test cases. For each test, we will have three triangle numbers for the Head, Mid and Tail sketch partitions. We will use the exact number of triangles in the three sub-sketches to give the approximate number of triangles in the given graph stream. The testing results are given in Table 21.6.

For the normal distribution, based on our regression model expressed as Eq. 21.1, we can get the parameters and express the multivariate linear regression equation as

$$TC_{Normal} = 0.7264 \times E_H + 1.4157 \times E_M + 1.7529 \times E_T$$

When we compute the absolute value of all the percent error, the mean error is 4%, the max error is 25%, R-squared value is 1 and this means that the sketch results are absolutely related with the final number of triangles.

We also evaluate our regression model to see how the accuracy changes with the shrinking factor. For normal distribution, we built five regression models by selecting the shrinking factor as 4, 8, 16, 32, and 64, respectively. The result is shown in Figure 21.18. We can see that both the mean and max error will increase when we doubly increase the value of shrinking factor step by step. However, the mean is increasing very slow. The mean error changes from 1% to 5.97%. The

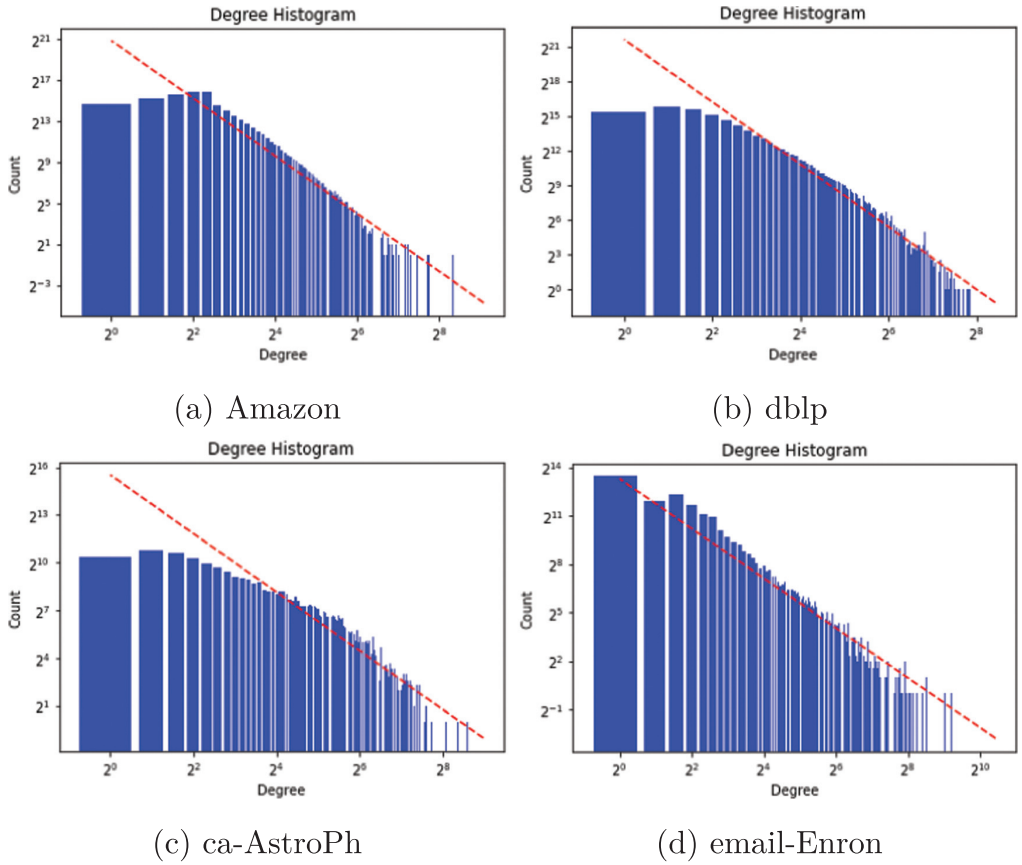


Figure 21.17 Power law distribution graphs. (a) Amazon. (b) dblp. (c) ca-AstroPh. (d) email-Enron.

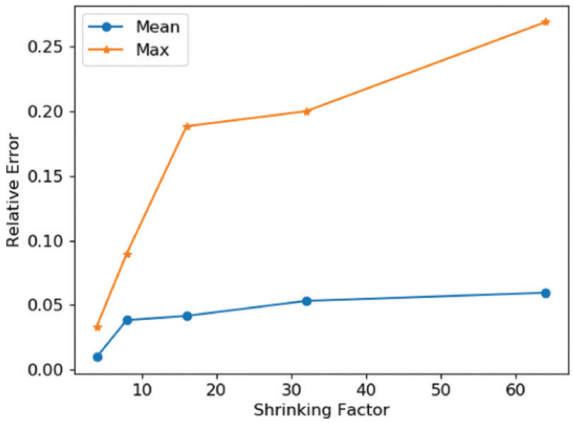


Figure 21.18 How accuracy changes with shrinking factor for normal distribution graphs.

Table 21.5
Comparison between Ordered and Unordered Regression Models for Power Law Distribution.

Metrics		AbsMaxError		AbsMeanError		R-squared	
Regression Methods		Ordered	Unordered	Ordered	Unordered	Ordered	Unordered
Shrink Factor	4	6.76%	8.26%	2.84%	3.75%	91.42%	85.10 %
	8	6.08%	8.89%	3.05%	4.13%	91.59%	82.90%
	16	4.16%	8.66%	1.97%	3.64%	96.47%	87.85%
	32	6.67%	10.80%	3.02%	4.02%	90.85%	82.73%
	64	4.66%	6.68%	2.86%	3.02%	94.12%	92.04%

max error will increase from 3.36% to 26.87%. This result shows that our regression model for normal distribution graphs is stable and very accurate. When the graph size shrinks into the half of the previous size, there is a very small effect on the accuracy when the number of triangles are not less than some threshold value. This is important because we can use much smaller sketches to achieve very accurate results.

For the power law distribution, based on our ordered regression model Eq. (21.3) that is refined from the unordered regression model, the log value multivariate linear regression equation can be expressed as

$$TC_{powerlaw,log} = -0.4464 \times E_{min,log} + 0.1181 \times E_{median,log} + 1.4236 \times E_{max,log}$$

The mean error is 3.2%. The max value is 7.2%. The R-squared value is 0.91. This means that the model can describe the data very well. If we use the unordered regression model Eq. (21.2), the mean error is 4.5%. The max value is 12.5%. R-squared is 0.81. All the results are worse than the results of the ordered regression model Eq. (21.3).

We also performed experiments to show the accuracy of ordered and unordered regression models when we build five different regression models using different shrinking factors 4, 8, 16, 32, and 64. The experimental results are given in Table 21.5 and we can have two conclusions from the results. (1) All the results of our ordered regression model are better than the results of the unordered regression model. This means that when we exploit more degree distribution information and integrate it into our model, we can achieve better accuracy. (2) When we double the value of the shrinking factor, the change in accuracy is much smaller than the change in normal regression model. It also means that in the power law model, the accuracy is much less sensitive to the shrinking factor because the power law graph is highly skewed. This conclusion also means that we can use a relatively smaller shrinking factor to achieve almost the same accuracy.

The accuracy evaluation results show that the proposed regression model for normal distribution graphs and power law distribution graphs can achieve very accurate results. Both mean relative error is less than 4%. However, if we employ the normal regression model Eq. (21.1) on power-law distribution data, the mean relative error can be as high as 26%, the max relative error is 188%. This means that employing different regression models for different degree distribution graphs may significantly improve the accuracy.

Table 21.6
Exact Number of Triangles in Different Sub-Sketches.

Graph Name	Shrinking Factor	Head	Mid	Tail	Exact
delaunay_n10	4	98	138	150	2047
delaunay_n10	8	63	72	80	2046
delaunay_n10	16	35	41	41	2044
delaunay_n10	32	20	12	16	2043
delaunay_n10	64	3	3	10	2042
delaunay_n11	4	195	302	294	4104
delaunay_n11	8	95	151	144	4103
delaunay_n11	16	62	71	63	4101
delaunay_n11	32	26	32	29	4100
delaunay_n11	64	12	24	14	4009
delaunay_n12	4	258	607	575	8215
delaunay_n12	8	163	306	293	8214
delaunay_n12	16	90	139	147	8212
delaunay_n12	32	52	78	76	8211
delaunay_n12	64	51	42	32	8210
delaunay_n13	4	646	1160	1197	16442
delaunay_n13	8	335	598	588	16441
delaunay_n13	16	187	297	304	16439
delaunay_n13	32	108	144	153	16438
delaunay_n13	64	68	62	81	16437
delaunay_n14	4	1099	2356	2299	32921
delaunay_n14	8	507	1205	1184	32920
delaunay_n14	16	269	568	579	32918
delaunay_n14	32	135	294	303	32917
delaunay_n14	64	93	144	137	32916
delaunay_n15	4	2085	4571	4585	65872
delaunay_n15	8	943	2335	2320	65871
delaunay_n15	16	486	1140	1159	65869
delaunay_n15	32	265	568	580	65868
delaunay_n15	64	137	282	299	65867
delaunay_n16	4	4168	9648	9510	131842
delaunay_n16	8	2170	4666	4601	131841
delaunay_n16	16	1063	2387	2348	131839
delaunay_n16	32	569	1174	1169	131838
delaunay_n16	64	326	576	580	131837
delaunay_n17	4	8154	18873	18938	263620
delaunay_n17	8	4012	9543	9501	263619
delaunay_n17	16	2132	4595	4773	263617
delaunay_n17	32	1042	2288	2347	263616
delaunay_n17	64	534	1156	1180	263615
delaunay_n18	4	16904	37774	38061	527234

(*Continued*)

Table 21.6 (*Continued*)
Exact Number of Triangles in Different Sub-Sketches.

Graph Name	Shrinking Factor	Head	Mid	Tail	Exact
delaunay_n18	8	8155	19005	19107	527233
delaunay_n18	16	4176	9454	9562	527231
delaunay_n18	32	2097	4674	4686	527230
delaunay_n18	64	1004	2369	2348	527229
amazon	4	4944	6383	18381	667259
amazon	8	1427	1752	7351	667258
amazon	16	621	617	3614	667256
amazon	32	331	290	2250	667255
amazon	64	177	153	1546	667254
dblp	4	20269	41390	168996	2225882
dblp	8	6876	17043	67782	2225881
dblp	16	2684	4846	20310	2225879
dblp	32	1097	2726	11456	2225878
dblp	64	545	2325	2678	2225877
ca-HepTh.txt	4	1267	1072	3853	28677
ca-HepTh.txt	8	642	591	1741	28676
ca-HepTh.txt	16	456	390	480	28674
ca-HepTh.txt	32	391	267	219	28673
ca-HepTh.txt	64	260	217	157	28672
ca-CondMat.txt	4	7960	10280	12051	174578
ca-CondMat.txt	8	4264	4970	5543	174577
ca-CondMat.txt	16	2862	2847	2655	174575
ca-CondMat.txt	32	1888	1693	1403	174574
ca-CondMat.txt	64	1179	1292	845	174573
ca-AstroPh.txt	4	75594	78234	78855	1374119
ca-AstroPh.txt	8	41331	47387	40395	1374118
ca-AstroPh.txt	16	24828	28244	20468	1374116
ca-AstroPh.txt	32	14984	16692	11128	1374115
ca-AstroPh.txt	64	6450	6507	6322	1374114
email-Enron.txt	4	70077	28783	25933	727044
email-Enron.txt	8	22331	13593	15197	727043
email-Enron.txt	16	11640	8381	5002	727041
email-Enron.txt	32	6681	5212	2392	727040
email-Enron.txt	64	3561	2912	1056	727039
ca-GrQc.txt	4	1193	3374	5300	48265
ca-GrQc.txt	8	671	2131	1096	48264
ca-GrQc.txt	16	346	880	733	48262
ca-GrQc.txt	32	276	694	385	48261
ca-GrQc.txt	64	64	221	150	48260
ca-HepPh.txt	4	118540	74598	187003	3345241

(*Continued*)

Table 21.6 (*Continued*)
Exact Number of Triangles in Different Sub-Sketches.

Graph Name	Shrinking Factor	Head	Mid	Tail	Exact
ca-HepPh.txt	8	44937	33404	64281	3345240
ca-HepPh.txt	16	16775	16426	35817	3345238
ca-HepPh.txt	32	3708	6643	8724	3345237
ca-HepPh.txt	64	705	3707	2899	3345236
loc-brightKite_edges.txt	4	12232	4435	2489	301812
loc-brightKite_edges.txt	8	5266	3217	1652	301811
loc-brightKite_edges.txt	16	2579	1673	927	301809
loc-brightKite_edges.txt	32	1061	1335	380	301808
loc-brightKite_edges.txt	64	552	856	280	301807

21.7.4.2 Performance

Existing exact triangle counting algorithms show that for very large graphs, the communication between different compute nodes will consume the major part of the time. So, how to reduce the total communication or how to improve data access locality during triangle counting is the key factor to improve the final performance.

Since our triangle counting method can maintain load balancing and locality, it can achieve high performance for large graph streams. Our experiment results show that our streaming triangle algorithm displays high locality. These results are given in Table 21.7. We calculate all the local data access times and all the remote data access times during the triangle counting procedure. We define the locality access ratio $LAR = \frac{NL}{NL+NR}$ where NL is the total number of local accesses and NR is the total number of remote accesses. For the three sub-sketches, Head, Mid and Tail, we calculate their LAR values, respectively. From the table we can see for all cases, the LAR value is larger than 50% and the average LAR value is 74%. This means that during our triangle counting procedure, most of the data comes from local memory. This is the major reason our method can achieve high performance.

To show the speedup of the streaming process, we compare the execution time of the original complete graph triangle counting and the execution time of our sketch method. The experimental results are given in Table 21.8. From the table we can see that our streaming method uses much less space and runs much faster to estimate the number of triangles in large graph stream with only one pass. Since the experimental time is very long for large graphs, here we just give three different shrinking factors to show the trend. We can see that for very large graphs, we can use a very large shrinking factor to significantly reduce the processing space and the absolute speedup is also very high. However, the relative speedup compared with the shrinking factor is a little bit lower. For relatively small graphs, our shrinking factor is also relatively small, so the absolute speedup is also small but the relative speedup is high. Our experimental results show that the speedup will increase almost linearly with the shrinking factor.

We also use an even larger graph benchmark com-Friendster that has 1,806,067,135 edges and 65,608,366 vertices[3] and run it with 2 locales. When the shrinking factor is 32,768, it will take 44,719.4 seconds to build the sketch with a maximum of 235,332KB memory footprint.

[3] https://snap.stanford.edu/data/.

Table 21.7
Local Access Ratio of different stream graphs.

Filename	Head Local Access Ratio	Mid Local Access Ratio	Tail Local Access Ratio
delaunay_n10	0.736364	0.75	0.642045
delaunay_n11	0.578635	0.789894	0.707775
delaunay_n12	0.737226	0.906832	0.878136
delaunay_n13	0.710651	0.902913	0.91839
delaunay_n14	0.661084	0.914058	0.941802
delaunay_n15	0.699193	0.963969	0.947161
delaunay_n16	0.685931	0.968672	0.953034
delaunay_n17	0.692844	0.972879	0.963517
delaunay_n18	0.693935	0.983028	0.975103
ca-HepTh.txt	0.687343	0.645873	0.650115
ca-CondMat.txt	0.630154	0.663701	0.6764
ca-AstroPh.txt	0.61735	0.615687	0.615749
email-Enron.txt	0.376773	0.632443	0.785357
ca-GrQc.txt	0.721421	0.671429	0.687907
ca-HepPh.txt	0.643376	0.648718	0.593803
loc-brightkite_edges.txt	0.71338	0.603389	0.721972
amazon	0.731695	0.723097	0.738042
dblp	0.605114	0.711748	0.830386
youtube	0.669736	0.671671	0.9342
lj	0.570621	0.728383	0.993926
orkut	0.556868	0.569822	0.861902

Table 21.8
Speedup of Stream graph processing.

Graph Name	Factor	Stream Time (s)	Exact Time (s)	Speedup
delaunay_n12	4	2.04689	2.72048	1.33
delaunay_n12	8	1.8969	2.72048	1.43
delaunay_n12	16	1.77604	2.72048	1.53
delaunay_n13	4	2.67814	4.70596	1.76
delaunay_n13	8	2.27257	4.70596	2.07
delaunay_n13	16	2.0258	4.70596	2.32
delaunay_n14	4	3.94324	8.91793	2.26
delaunay_n14	8	3.04275	8.91793	2.93
delaunay_n14	16	2.60059	8.91793	3.43
delaunay_n15	4	6.34795	17.4302	2.75
delaunay_n15	8	4.44783	17.4302	3.92

(*Continued*)

Table 21.8 (*Continued*)
Speedup of Stream graph processing.

Graph Name	Factor	Stream Time (s)	Exact Time (s)	Speedup
delaunay_n15	16	3.6619	17.4302	4.76
delaunay_n16	4	11.2135	34.2433	3.05
delaunay_n16	8	7.29339	34.2433	4.70
delaunay_n16	16	5.65829	34.2433	6.05
delaunay_n17	4	21.0084	68.4882	3.26
delaunay_n17	8	13.0833	68.4882	5.23
delaunay_n17	16	9.71683	68.4882	7.05
delaunay_n18	4	40.4338	135.191	3.34
delaunay_n18	8	24.8456	135.191	5.44
delaunay_n18	16	17.8642	135.191	7.57
delaunay_n18	32	14.1647	135.191	9.54
delaunay_n18	64	12.0415	135.191	11.23
delaunay_n19	32	26.6216	270.491	10.16
delaunay_n19	64	22.5967	270.491	11.97
delaunay_n19	128	20.9203	270.491	12.93
delaunay_n20	32	51.7499	536.331	10.36
delaunay_n20	64	44.3933	536.331	12.08
delaunay_n20	128	40.2489	536.331	13.33
delaunay_n21	32	104.236	1076.77	10.33
delaunay_n21	64	87.0196	1076.77	12.37
delaunay_n21	128	80.7149	1076.77	13.34
delaunay_n22	32	208.478	2155.85	10.34
delaunay_n22	64	177.637	2155.85	12.14
delaunay_n22	128	158.13	2155.85	13.63
delaunay_n23	64	352.531	4356.982	12.36
delaunay_n23	128	317.053	4356.982	13.74
delaunay_n23	256	301.295	4356.982	14.46
delaunay_n24	128	645.002	8708.3756	13.50
delaunay_n24	256	604.981	8708.3756	14.39
delaunay_n24	512	598.49	8708.3756	14.55
amazon	16	20.4202	147.926	7.24
amazon	32	16.2884	147.926	9.08
amazon	64	14.0285	147.926	10.54
ca-AstroPh	4	16.1657	33.1578	2.05
ca-AstroPh	8	11.0193	33.1578	3.01
ca-AstroPh	16	8.38061	33.1578	3.96
ca-CondMat	4	8.70114	16.5758	1.91
ca-CondMat	8	6.20076	16.5758	2.67
ca-CondMat	16	4.86599	16.5758	3.41
ca-GrQc	4	2.65749	3.23613	1.22
ca-GrQc	8	2.20744	3.23613	1.47

(*Continued*)

Table 21.8 (*Continued*)
Speedup of Stream graph processing.

Graph Name	Factor	Stream Time (s)	Exact Time (s)	Speedup
ca-GrQc	16	2.02119	3.23613	1.60
ca-HepPh	4	10.263	21.3077	2.08
ca-HepPh	8	6.98969	21.3077	3.05
ca-HepPh	16	5.45666	21.3077	3.90
ca-HepPh	32	4.83524	21.3077	4.41
ca-HepPh	64	4.3246	21.3077	4.93
ca-HepTh	4	3.78554	5.37471	1.42
ca-HepTh	8	2.92663	5.37471	1.84
ca-HepTh	16	2.46299	5.37471	2.18
dblp	16	22.0434	163.092	7.40
dblp	32	17.8394	163.092	9.14
dblp	64	15.4684	163.092	10.54
email-Enron	4	16.133	63.1888	3.92
email-Enron	8	10.859	63.1888	5.82
email-Enron	16	8.04875	63.1888	7.85
lj	512	409.672	6937.4	16.93
lj	1024	393.518	6937.4	17.63
lj	2048	397.583	6937.4	17.45
loc-brightkite_edges	4	14.1111	30.1892	2.14
loc-brightkite_edges	8	10.1597	30.1892	2.97
loc-brightkite_edges	16	8.09854	30.1892	3.73
youtube	512	36.2397	951.094	26.24
youtube	1024	34.9246	951.094	27.23
youtube	2048	34.6978	951.094	27.41

The triangle counting time is 22,368.2 seconds. When the shrinking factor is 65,536, it will take 44,593.0 seconds to build the sketch with a maximum of 235,288KB memory footprint. The triangle counting time is 22,296.0 seconds. If we use the exact method also with 2 locales, the program will fail because it runs out of memory. This example really shows that our method can handle very large graphs in limited memory footprint.

21.8 RELATED WORK

In this section, we introduce existing BFS and triangle counting algorithms. We also introduce the exact method and approximate method to handle graph streams.

21.8.1 BFS ALGORITHM

Since BFS is a very basic graph algorithm, it has been investigated from different aspects. For shared-memory BFS algorithms, Leiserson et al. [31] proposed a data structure "bag" to replace shared 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. [5] 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. [40] implemented the BFS algorithm on the 128-processor Cray XMT system.

There are also many BFS algorithms on GPU. For examples, Luo et al. [32] 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. [37] 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. [8] 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. [3] 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. [29] used both Reverse Cuthill-Mckee algorithm and SIMD executions to improve their BFS algorithm's performance. Fan et al. [21] 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.

21.8.2 TRIANGLE COUNTING ALGORITHM

Triangle counting is a key building block for important graph algorithms such as transitivity and K-truss. There are two kinds of triangle counting algorithms, exact triangle counting for static graphs and approximate triangle counting for graphs built from dynamic streams [36]. Since the proposed *DI* sketch is a graphical model, all the static triangle counting algorithms can also be used on the small *DI* sketch.

Graph Challenge[4] is an important competition on large graph algorithms (including triangle counting). Starting from 2017, many excellent static triangle counting algorithms have been developed. They target three hardware platforms, shared memory, distributed memory, and GPU.

The shared memory methods take advantage of some fast parallel packages, such as Kokkos Kernels [51] or Cilk [52], to improve their performance. However, the GPU methods [10, 42, 11] use massively parallel fine-grain hardware threads of a GPU to improve the performance. Distributed-memory triangle counting focuses on very large graphs that cannot fit in a single node's memory. Some heuristics [44], optimized communication library [45], and graph structures [22] are used to improve the performance. We leverage the ideas in such methods to develop our Chapel-based multilocale triangle counting algorithm.

[4] https://graphchallenge.mit.edu/challenges.

The basic idea of graph stream analysis is estimating the exact query result of a graph stream based on the sampling results. Colorful triangle counting [41] is an example. However, it needs to know the number of triangles and the maximum number of triangles of an edge to set the possibility value. This is not feasible in practical applications. Reduction-based [7] triangle counting is a typical method which can design a theoretical algorithm based on user-specified values (ε, δ). Such a method often cannot directly be used satisfactorily in practical applications because hidden constant values often impact the performance. Neighborhood sampling [43] is another method for triangle counting with significant space and time complexity improvements. Specifically, Braverman *et al.* [12] discuss the difficulty of the triangle counting algorithm in a streaming model. Other sampling methods, such as [13], have space usage that depends on the ratio of the number of triangles and the number of triples or the algorithm will require the edge stream to meet a specific order. Jha *et al.* [28] apply the birthday paradox theory on sampling data to estimate the number of triangles in a graph stream.

21.8.3 GRAPH STREAM SKETCH

A much smaller sketch allows for many queries over the large graph stream to get approximate results efficiently. Therefore, how to build a graph stream's sketch is of fundamental importance for graph stream analytics. There are several methods that build the sketch by treating each stream element independently without keeping the relationships among those elements. For example, CountMin [17] allows fundamental queries in data stream summarization such as point, range, and inner product queries. It can also be used to find quantiles and frequent items in data stream. Bottom K sketch [16] places both priority sampling and weighted sampling without replacement within a unified framework. It can generate tighter bounds when the total weight is available. gSketch [53] introduces the sketch partitioning technique to estimate and optimize the responses to basic queries on graph streams. By exploiting both data and query workload samples, gSketch can achieve better query estimation accuracy than that achieved with only the data sample. We borrow the sketch partitioning idea in our implementation of Double-Index (DI) sketch. However, these existing sketches focus on ad-hoc problems (they can only solve the proposed specific problems instead of general problems), so they cannot support general data analytics over graph streams.

TCM sketch [49] uses a graphical model to build the sketch of a graph stream. This means that TCM is a general sketch to allow complicated queries. However, TCM's focus is setting up a general sketch theoretically instead of optimizing the practical performance for real-world data sets. Our Double-Index (DI) sketch is specially designed for real-world sparse graph streams and it can achieve high practical performance.

21.8.4 COMPLETE GRAPH STREAM PROCESSING METHOD

Several dynamic graph management and analytics solutions are as follows. Aspen [19] takes advantage of purely-functional trees data structure, C-trees, to implement quick graph updates and queries. LLAMA's streaming graph data structure [33] is motivated by the CSR format. However, like Aspen, LLAMA is designed for batch-processing in the single-writer multi-reader setting and does not provide graph mutation support. GraphOne [30] can run queries on the most recent version of the graph while processing updates concurrently by using a combination of an adjacency list and an edge list.

Systems like LLAMA, Aspen and GraphOne focus on designing efficient dynamic graph data structures, and their processing units do not support incremental computation. KickStarter [50] maintains the dependency information in the form of dependency trees, and performs an

incremental trimming process that adjusts the values based on monotonic relationships. GraphBolt [35] incrementally processes streaming graphs and minimizes redundant computations upon graph mutation while still guaranteeing synchronous processing semantics. DZiG [34] is a high-performance streaming graph processing system that retains efficiency in the presence of sparse computations while still guaranteeing BSP semantics. However, all such methods have the challenging requirement that the memory must be large enough to hold all the streams. For very large streams, this is often feasible.

21.9 CONCLUSION

Interactive graph analysis is a challenging problem. In this chapter, we present our solutions to solve the problem in an open source framework Arkouda. The advantage of Arkouda lies in two aspects: high productivity and high performance. High productivity means that the end users can use a popular data science language such as Python to explore different graph streams. 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 in an interactive way.

We design a double index data structure to support quick search from a given vertex to edges or from given edge to vertices. This greatly reduces the total memory footprint for large sparse graphs or graph streams. Based on the proposed double index data structure, we can build the graph or graph stream in memory using very limited space. Our graph sketch can support general stream query problems because our sketch uses a small graph to summarize a large graph. We define a shrinking factor to let users control the size of final sketch accurately. We exploit the ubiquitous normal and power law degree distributions of given graph streams to propose two regression models to estimate the results of the graph streams from their sketch partitions.

Based on the double index data structure, we develop a multilocale BFS algorithm and a triangle counting algorithm that can maintain load balancing and high local data access ratio to improve the graph processing performance. All our methods have been integrated into the Arkouda framework and users can use a Jupyter Notebook to drive the interactive graph analytics pipeline easily.

Experimental results on 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.

Experimental results on two kinds of large sparse graph streams, normal distribution and power law distribution, show that the proposed method can achieve very high approximate results with the mean error no larger than 4%. The average local access ratio is 74% and the graph stream processing speedup increases almost linearly with the shrinking factor. This work shows that Arkouda is a promising framework to support large-scale interactive graph analytics.

ACKNOWLEDGMENTS

We appreciate the help from Michael Merrill and William Reus as the cofounders of Arkouda, 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, A.L. Barabási, R. Albert, H. Jeong, and G. Bianconi. Power-law distribution of the world wide web. *Science*, 287(5461):2115–2115, 2000.
2. Mohammad Al Hasan and Vachik S. Dave. Triangle counting in large networks: A review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(2):e1226, 2018.
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. David A. Bader and Kamesh Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *2006 International Conference on Parallel Processing (ICPP'06)*, pages 523–530. IEEE, 2006.
6. David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Graph Partitioning and Graph Clustering, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, volume 588 of *Contemporary Mathematics*. American Mathematical Society, 2013.
7. Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *SODA*, volume 2, pages 623–632, 2002.
8. Scott Beamer, Aydin Buluc, Krste Asanovic, and David Patterson. Distributed memory breadth-first search revisited: Enabling bottom-up search. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, pages 1618–1627. IEEE, 2013.
9. John T. Behrens. Principles and procedures of exploratory data analysis. *Psychological Methods*, 2(2):131, 1997.
10. Mauro Bisson and Massimiliano Fatica. Static graph challenge on GPU. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2017.
11. Mark Blanco, Tze Meng Low, and Kyungjoo Kim. Exploration of fine-grained parallelism for load balancing eager k-truss on GPU and CPU. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.
12. Vladimir Braverman, Rafail Ostrovsky, and Dan Vilenchik. How hard is counting triangles in the streaming model? In *International Colloquium on Automata, Languages, and Programming*, pages 244–254. Springer, 2013.
13. Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 253–262, 2006.
14. 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.
15. 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.
16. Edith Cohen and Haim Kaplan. Tighter estimation using bottom k sketches. *Proceedings of the VLDB Endowment*, 1(1):213–224, 2008.
17. Graham Cormode and Shan Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
18. Elizabeth Cuthill and James McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, pages 157–172, 1969.
19. Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 918–934, 2019.

20. 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.
21. Dongrui Fan, Huawei Cao, Guobo Wang, Na Nie, Xiaochun Ye, and Ninghui Sun. Scalable and efficient graph traversal on high-throughput cluster. *CCF Transactions on High Performance Computing*, pages 1–13, 2020.
22. Sayan Ghosh and Mahantesh Halappanavar. Tric: Distributed-memory triangle counting by exploiting the graph structure. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2020.
23. Irving J. Good. The philosophy of exploratory data analysis. *Philosophy of Science*, 50(2):283–295, 1983.
24. Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., Newton, MA, 2013.
25. Parry Husbands and Charles Isbell. The parallel problems server: A client-server model for interactive large scale scientific computation. *Proceedings of VECPAR'98*, June 1998.
26. Parry Husbands, Charles L. Isbell, and Alan Edelman. Interactive Supercomputing with MITMatlab. August 2001.
27. 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.
28. Madhav Jha, Comandur Seshadhri, and Ali Pinar. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 589–597, 2013.
29. Zite Jiang, Tao Liu, Shuai Zhang, Zhen Guan, Mengting Yuan, and Haihang You. Fast and efficient parallel breadth-first search with power-law graph transformation. *arXiv preprint arXiv:2012.10026*, 2020.
30. Pradeep Kumar and H. Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. In *17th USENIX Conference on File and Storage Technologies (FAST19)*, pages 249–263, 2019.
31. Charles E. Leiserson and Tao B. Schardl. 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*, pages 303–314, 2010.
32. Lijuan Luo, Martin Wong, and Wen-mei Hwu. An effective gpu implementation of breadth-first search. In *Design Automation Conference*, pages 52–55. IEEE, 2010.
33. Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*, pages 363–374. IEEE, 2015.
34. Mugilan Mariappan, Joanna Che, and Keval Vora. Dzig: sparsity-aware incremental processing of streaming graphs. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 83–98, 2021.
35. Mugilan Mariappan and Keval Vora. Graphbolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.
36. Andrew McGregor. Graph stream algorithms: A survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.
37. Duane Merrill, Michael Garland, and Andrew Grimshaw. High-performance and scalable gpu graph traversal. *ACM Transactions on Parallel Computing (TOPC)*, 1(2):1–30, 2015.
38. 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.
39. Michael Merrill, William Reus, and Timothy Neumann. Arkouda: Numpy-like arrays at massive scale backed by chapel. In *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*. IEEE, 2019.
40. David Mizell and Kristyn Maschhoff. Early experiences with large-scale xmt systems. In *Proceedings of Workshop on Multithreaded Architectures and Applications (MTAAP'09)*, 2009.
41. Rasmus Pagh and Charalampos E. Tsourakakis. Colorful triangle counting and a MapReduce implementation. *Information Processing Letters*, 112(7):277–281, 2012.

42. Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. H-index: Hash-indexing for parallel triangle counting on GPUs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2019.
43. A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Counting and sampling triangles from a graph stream. *Proc. VLDB Endow.*, 6(14):1870–1881, September 2013.
44. Roger Pearce. Triangle counting for scale-free graphs at scale in distributed memory. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–4. IEEE, 2017.
45. Roger Pearce, Trevor Steil, Benjamin W. Priest, and Geoffrey Sanders. One quadrillion triangles queried on one million processors. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–5. IEEE, 2019.
46. 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.
47. Guido Rossum. Python reference manual. 1995.
48. Andrew T. Stephen and Olivier Toubia. Explaining the power-law degree distribution in a social commerce network. *Social Networks*, 31(4):262–270, 2009.
49. Nan Tang, Qing Chen, and Prasenjit Mitra. Graph stream summarization: From big bang to big crunch. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1481–1496, 2016.
50. Keval Vora, Rajiv Gupta, and Guoqing Xu. Kickstarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*, pages 237–251, 2017.
51. Michael M. Wolf, Mehmet Deveci, Jonathan W. Berry, Simon D. Hammond, and Sivasankaran Rajamanickam. Fast linear algebra-based triangle counting with KokkosKernels. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.
52. Abdurrahman Yaşar, Sivasankaran Rajamanickam, Michael Wolf, Jonathan Berry, and Ümit V Çatalyürek. Fast triangle counting using Cilk. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.
53. Peixiang Zhao, Charu C. Aggarwal, and Min Wang. gSketch: On query estimation in graph streams. *arXiv preprint arXiv:1111.7167*, 2011.