DH-Falcon: A language for large-scale graph processing on Distributed Heterogeneous systems

Unnikrishnan C
Department of CSA,
Indian Institute of Science, Bangalore

Rupesh Nasre
Department of CSE,
Indian Institute of Technology, Madras

Y N Srikant
Department of CSA,
Indian Institute of Science, Bangalore

***Abstract*—Graph models of social information systems typically contain trillions of edges. Such big graphs cannot be processed on a single machine. The graph object must be partitioned and distributed among machines and processed in parallel on a computer cluster. Programming such systems is very challenging. In this work, we present `DH-Falcon`, a graph DSL (domain-specific language) which can be used to implement parallel algorithms for large-scale graphs, targeting Distributed Heterogeneous (CPU and GPU) clusters. `DH-Falcon` compiler is built on top of the Falcon compiler, which targets single node devices with CPU and multiple GPUs. An important facility provided by DH-Falcon is that it supports mutation of graph objects, which allows programmer to write dynamic graph algorithms. Experimental evaluation shows that `DH-Falcon` matches or outperforms state-of-the-art frameworks and gains a speedup of up to 13× for different benchmarks.**

I. Introduction

A graph models the relationship between two entities as edges between two points. In domains such as social information systems, the number of edges can be in billions or trillions. Such large graphs are processed on distributed computer systems (or clusters). Google's Pregel [1], which uses the Bulk Synchronous Parallel (BSP) model of execution and PowerGraph [2] which follows the Gather-Apply-Scatter (GAS) model of execution are examples of popular frameworks for large-scale graph processing on CPU clusters. Distributed processing is efficient only if a graph object is partitioned and distributed among machines uniformly so that there is work balance and less communication overhead across nodes.

To the best of our knowledge, currently there is no framework or domain-specific language (DSL) that can be used to implement large-scale graph processing algorithms targeting heterogeneous distributed systems. We propose `DH-Falcon` that adapts the same constructs of Falcon [3, 4]. The user need not specify *any* device-specific constructs in the code (such as <*GPU*> tag in Falcon). The DSL codes are explicitly parallel and *the same DSL code can be converted to an executable for CPU, GPU, multi-GPU machine, CPU cluster, GPU cluster and CPU+GPU cluster*. The `DH-Falcon` compiler allows mutation of graph objects, and hence supports programming dynamic graph algorithms. In comparison to other graph frameworks such as PowerGraph, `DH-Falcon` constructs are at a higher level of abstraction, improving readability, programmer productivity, as well as opportunities for efficient code generation. The `DH-Falcon` programmer need not use MPI, OpenMP, CUDA, etc., to make the programs efficient.

Our contributions in this work are:

- Design of the `DH-Falcon` compiler targeting heterogeneous distributed systems which include CPU and GPU clusters and multi-GPU machines.
- Support for mutation of graph objects which enables writing dynamic graph algorithms in the `DH-Falcon` programming language. A salient feature of `DH-Falcon` is that it also supports non-vertex centric graph algorithms.
- Code generation schemes for heterogeneous and distributed systems, program analysis to reduce communication overhead between distributed machines, allocation of variables and efficient storage of graph objects on target systems and a distributed locking mechanism.
- Performance evaluation of the `DH-Falcon` compiler. Results show that the `DH-Falcon` DSL codes have a speedup of up to 13× over PowerGraph on benchmarks when executed on a 16 node CPU cluster for public large-scale graph inputs. The `DH-Falcon` codes when run on a multi-GPU machine with 8 GPUs show a performance that is comparable to that of the Totem framework. The `DH-Falcon` codes yield good speedup when run on an 8-node GPU cluster and heterogeneous clusters with each node having i) CPU or GPU ii) CPU and GPU.

II. Related Work

PowerGraph [2] is a framework for distributed graph, and Machine Learning and Data Mining (MLDM) algorithms, targeting CPU clusters with the Gather-Apply-Scatter(GAS) model of execution. The Pregel [1] framework uses the Bulk Synchronous Parallel (BSP) model of execution and computation is done in a sequence of supersteps. The Distributed-Graphlab [5] framework follows an asynchronous execution model and can be used to implement graph and MLDM algorithms. Large-scale real world graphs have a power-law degree distribution and are difficult to partition [6]. We compare and contrast `DH-Falcon` with PowerGraph, GraphLab and Pregel in Section III-A. A summary of the major differences between these is provided in Table I.

GPS (Graph Processing System) [7] is an open source framework and follows the execution model of Pregel. The Green-Marl [8] compiler was extended for CPU-clusters [9]

Item	DH-Falcon	PowerGraph	GraphLab	Pregel
multi-GPU device	√	x	x	x
CPU cluster	√	√	√	√
GPU cluster	√	x	x	x
GPU+GPU cluster	√	x	x	x
Synchronous execution	√	√	x	√
ASynchronous execution	x	√	√	x
Dynamic algorithms	√	√	√	√
Graph Partitioning	edge-cut	vertex-cut	edge-cut	edge-cut
Message Volume	low	high	high	low

TABLE I. Comparison of various distributed frameworks

and it generates GPS-based Pregel-like code. Mizan [10] uses dynamic monitoring algorithm execution and does vertex migration at run time to balance computation and communication. Hadoop [11] follows the *MapReduce()* programming model and uses the hadoop distributed file system (HDFS) for storing data. Giraph [12] is an open source framework for large-scale graph processing and uses hadoop. PowerLyra [13], implemented as a separate computation engine of PowerGraph, combines hybrid partition which uses edge-cut and vertex-cut. Haloop [14] is a framework which follows the *MapReduce()* pattern with support for iterative computation, and with good caching and scheduling methods. Twister [15] is also a framework which follows the *MapReduce()* model of execution. Pregel like systems can outperform *MapReduce()* systems in graph analytic applications. The Trinity [16] distributed graph engine provides a high level specification language for graph management and computing. Parallel Boost Graph Library (PBGL) [17] extends BGL for distributed systems. All the works mentioned above are frameworks for multicore-CPU clusters.

The GraphChi [18] framework processes large-scale graphs using a single machine, with the graph being split into different parts (called shards). Shards are loaded one by one into RAM and then processed. Such a framework is useful in the absence of distributed clusters. The Ligra [19] framework implements several graph traversal algorithms for large-scale graphs on shared memory systems. X-Stream [20] is an edge-centric graph processing framework for in-core as well as out-of-core processing on a single shared memory system. Graphine [21] uses an agent-graph model to partition graphs, uses scatter-agent and combine-agent to reduce communication overhead. The Totem [22] framework supports heterogeneous execution of graph algorithms on a single machine with multiple GPUs and a multi-core CPU. LightHouse [23] converts Gree-Marl DSL code to CUDA Code. GraphIn [24] supports incremental dynamic graph analytics using the incremental GAS programming model.

III. Motivation

A. Requirements of Large-Scale Graph Processing and Demerits of current frameworks

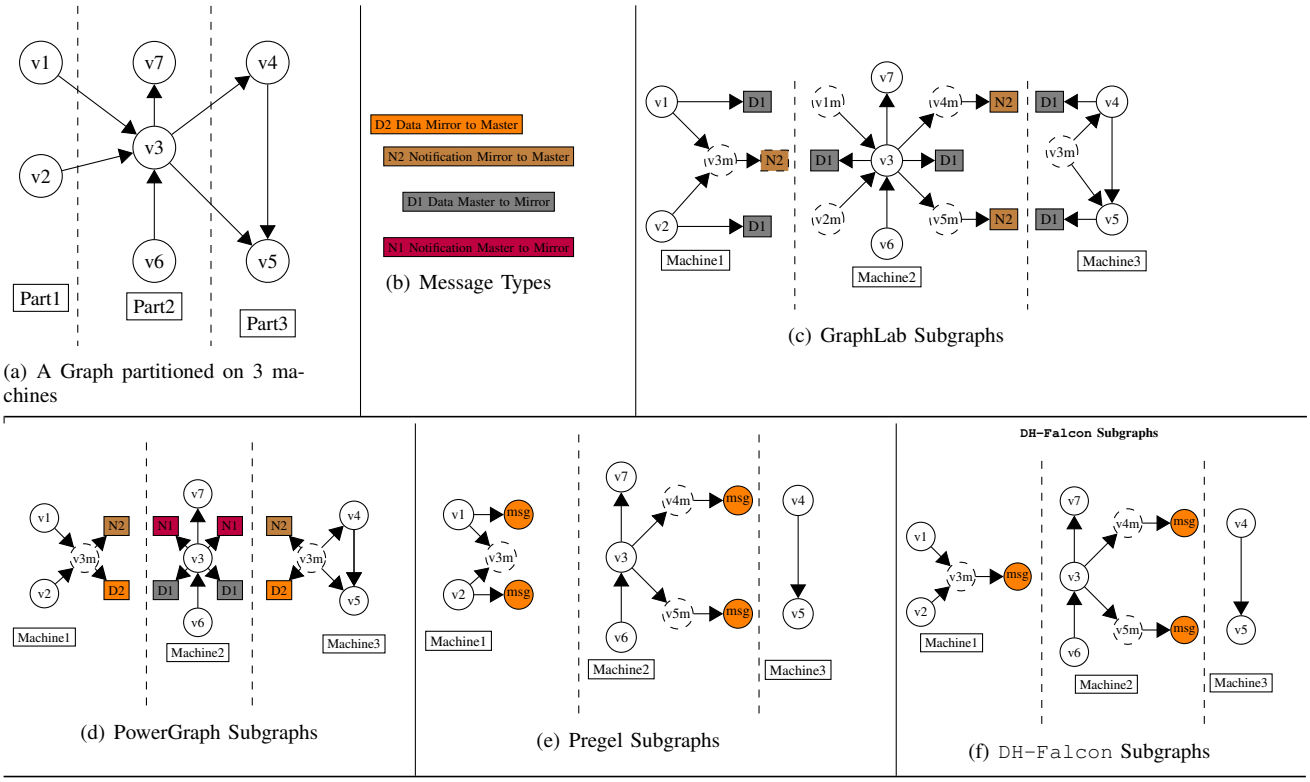
Distributed graph processing follows a common pattern: (i) A vertex gathers values from its neighboring vertices on remote machines, and updates its own value. (ii) It then modifies property values of its neighboring vertices and edges. iii) It broadcasts the modified values to the remote machines. Figure 1 shows a comparison of GraphLab, PowerGraph, Pregel and DH-Falcon, related to graph storage and communication patterns on vertex $v3$ in the directed graph on Figure 1(a).

1) PowerGraph: PowerGraph uses balanced p-way vertex cut to partition graph objects. This can produce work balance but can result in more communication compared to random edge-cut partitioning. When a graph object is partitioned using vertex cut, two edges with the same source vertex may reside on different machines. So, if n machines are used for computation and if there are x edges with source vertex v and $x > 1$, then these edges may be distributed on p machines where $1 \leq p \leq min(x, n)$. PowerGraph takes one of the machines as the master-node for vertex v and the other machines as mirrors. As shown in Figure 1(d), edges with $v3$ as source vertex are stored on Machine2 ($(v3, v7)$) and Machine3 ($(v3, v4)$), and Machine2 is taken as the master-node.

Computation follows the Gather-Apply-Scatter (GAS) model and needs communication before and after a parallel computation (Apply). PowerGraph supports both synchronous and asynchronous executions. Mirror vertices ($v3m$) on Machine1 and Machine3 send their new values and notification messages to the master-node $v3$ on Machine2 and activate vertex $v3$. Vertex $v3$ then reads the values received from the mirrors and $v6$ (Gather), updates its own value and performs the computation (Apply). Thereafter, $v3$ sends its new data and notification message to mirror $v3m$ on Machine1 and Machine3 (Scatter).

2) GraphLab: The GraphLab framework uses random edge cut to partition graph objects and follows the asynchronous execution model. Due to asynchronous execution it has more storage overhead as each edge with one remote-vertex is stored twice (e.g., $v1 \rightarrow v3m$ on Machine1 and $v1m \rightarrow v3$ on Machine2). It also has to send multiple messages to these duplicate copies which results in more communication volume. When edge cut is used for partitioning, all the edges with a source vertex v will reside on the same machine, as shown in Figure 1(c). Here, before vertex $v3$ starts the computation, remote vertices $(v1, v2)$ send their new values to their mirrors in Machine2 and activate vertex $v3m$. $v3m$ on Machine1 then sends a notification message to $v3$. Now, vertex $v3$ reads values from $v1m, v2m$ and $v6$, updates its own value and performs the computation. Thereafter, it sends its new data to the mirrors in Machine1 and Machine3. Vertices $v4m$ and $v5m$ send a notification message to activate $v4$ and $v5$ in Machine3.

3) Pregel: The Pregel framework uses random edge-cut to partition the graph object (Figure 1(e)). Pregel follows the Bulk Synchronous Parallel (BSP) Model [25] of execution and there is synchronization after each step, with execution being carried out in a series of supersteps. Communication happens with each vertex sending a single message to the master-node of the destination vertex of the edge. Pregel sends two messages from Machine1, $(v1 \rightarrow v3)$ and $(v2 \rightarrow v3)$. By default, it does not aggregate the two messages to $v3$ to a



(a) A Graph partitioned on 3 machines

(b) Message Types

(c) GraphLab Subgraphs

(d) PowerGraph Subgraphs

(e) Pregel Subgraphs

(f) DH-Falcon Subgraphs

Figure 1: Comparison of DH-Falcon and Other Distributed graph frameworks

single message. This needs to be done by the programmer by overriding the $Combine()$ method of the $Combiner$ class [1] and the $Combine()$ method should be commutative and associative. Pregel in a superstep S_i reads (Gather) values sent in the superstep S_{i-1}, performs the computation (Apply) and sends the updated values to remote machines (Scatter) which will be read in superstep S_{i+1}.

4) DH-Falcon: DH-Falcon follows the BSP model of execution and uses random edge cut to partition graph objects (Figure 1(f)). The execution is carried out as a series of supersteps similar to Pregel. DH-Falcon combines messages to $v3$ as a single message and the amount of data communicated is less than that of all the three frameworks mentioned above. The DH-Falcon compiler also requires that operations which modify mutable graph properties be commutative and associative.

Pregel and DH-Falcon have barrier as an overhead after each step, but this helps in reducing communication volume. PowerGraph and GraphLab have more communication volume due to vertex-cut partitioning and asynchronous execution respectively. Table I compares the frameworks mentioned above.

B. Falcon

Falcon is a Graph DSL for writing graph algorithms targeting a single machine with heterogeneous devices, Nvidia-GPUs and multi-core CPUs [3, 4]. It extends the C programming language with additional data types for graph processing. The Falcon DSL codes for GPU and CPU are different. All the declaration statements for variables on GPU should be preceded by the <GPU> tag. Partitioned execution of graph algorithms on multiple devices on the same machine is possible in Falcon with the programmer specifying explicitly how the graph object should be partitioned and updated in the DSL code.

Falcon has several built-in data types: Graph, Point, Edge, Set and Collection. It provides foreach and parallel sections statements for specifying parallelism, and single statement for synchronization. *Atomic* library functions such as MIN, MAX, etc., which are abstractions over the ones available in C++ and CUDA are also available in Falcon. Falcon compiler generated codes outperform or match handwritten CUDA/C++ codes of the state-of-the-art frameworks [3].

IV. DH-Falcon Overview

A. Introduction

The DH-Falcon is a graph DSL built on top of Falcon [3, 4] and follows the BSP [25] model of execution. The programmer writes a single program in Falcon and with proper command line arguments, it is converted to different high-level language codes (C++, CUDA) with the required library calls (OpenMP, MPI/OpenMPI) for the target system by the DH-Falcon compiler (see Figure 2). These codes

are then compiled with the native compilers (g++, nvcc) and libraries to create the executables. For distributed targets, the DH-Falcon compiler performs static analysis to identify the data that needs to be communicated between devices at various points in the program (See Sections V-C and V-E).

B. Data Types and Their Representation in DH-Falcon

DH-Falcon and Falcon have the same data types and their representation is the same as that of Falcon for a single device (GPU or CPU) system. For a distributed system the representation of data types differs.

1) Point and Edge: In a distributed setup, a Point object has a global-vertex-id, as well as a local-vertex-id or a remote-vertex-id in a *localgraph* object. The programmer can only view and operate on a Point based on global-vertex-id. The local-vertex-id is used for storing and processing *localgraph* objects on each machine/device and remote-vertex-id is used for communication between machines/devices in each super-step of the BSP model of execution. Each edge is stored in a single *localgraph* with modified values for source and destination vertex-id.

2) Graph: A Graph stores its points and edges in the vectors points[] and edges[]. The methods *addEdgeProperty()* and *addPointProperty()* are used to add properties to the edges and points (respectively) of the graph object. The *addProperty()* method is used to add a new property to the whole Graph object (not to each Point or Edge). This feature is used in Delaunay Mesh Refinement (DMR) [26] algorithm, where the graph object is a collection of triangles and not just points and edges.

3) Distributed Graph Storage in DH-Falcon: When an algorithm is run on n nodes for a Graph G, the graph object is partitioned into n subgraphs (*localgraphs*) $G_0, G_1, .., G_{(n-1)}$ and node/device i stores the *localgraph* G_i. Each localgraph G_i will be processed by a process P_i with rank i, $0 \leq i < n$, among the n processes created during program execution. Each edge and its properties in the Graph G is stored in exactly one subgraph $G_k, 0 \leq k < n$ and every vertex (point) is assigned a master-node (m-$node$).

A master-node k stores all the edges $e(u, v)$, with vertex u (v) of the edge having $m\text{-}node(u) = k$ ($m\text{-}node(v) = k$) when G_k is stored in edge-list (reverse-edge-list) format. In that case, the destination vertices may have a different master-node and such a vertex becomes a *remote-vertex(rv)* in G_k. In a *localgraph* object G_k, the global-vertex-id of each vertex p is converted to local-vertex-id ($m\text{-}node(p) = k$) or remote-vertex-id($m\text{-}node(p) \neq k$).

The DH-Falcon compiler assigns a local-vertex-id for each master-vertex in the subgraph. There is an ordering among local-vertex-id and remote-vertex-id in the *localgraph*. For any local-vertex x and any remote-vertex y in any subgraph G_k, we set $id(x) < id(y)$. If two remote-vertices x and y in a subgraph G_k belong to different master-nodes i and j respectively, and if $i < j$, we set $id(x) < id(y)$. This gives a total ordering between *localpoints* and *remotepoints* in a *localgraph* G_k of G. It helps in sending updated remote-vertex property values of each *remotepoint* with master-node p in a *localgraph* G_k to the localgraph G_p ($p \neq k, 0 \leq p < n$) on node p, as the boundaries for remote-vertices of each node are well defined. The communication happens after a parallel computation. The local-vertex properties need to be communicated in algorithms which modify the local-vertex properties, like the pull-based computation (See Algorithm 4) instead of push-based computation. The DH-Falcon performs static analysis to determine this, and generates efficient code with minimal communication overhead (See Section V-E).

Algorithm 1: Distributed-Union in DH-Falcon

```
1 if( rank(node)!=0 ){
      add each union request Union(u, v) to the buffer
      Send the buffer to node with rank==0
      receive parent value from node zero
      update local set
2 }
3 if( rank(node)==0 ){
      receive Union(u, v) request from remotenodes
      perform union; update parent of each element
      send parent value to each remote node
4 }
```

4) Set: DH-Falcon implements distributed Union-Find on top of the Union-Find of Falcon [3]. In a distributed setup, the first process ($rank = 0$) is responsible for collecting union requests from all other nodes. This node performs the union and sends the updated parent value to all other nodes involved in the computation as given in Algorithm 1.

Algorithm 2: Collection Synchronization in DH-Falcon

```
foreach(item in Collection)
   if (item.master-node!=rank(node))
    add item to buffer[item.master-node] and delete item from
Collection
foreach (i ∈ remote-node) send buffer to remote-node(i)
foreach (i ∈ remote-node)receive buffer from remote-node(i)
foreach (i ∈ remote-node){
  foreach (j ∈ buffer[i]){
    update property values using buffer[i].elem[j]
    addtocollection(buffer[i].elem[j])
} }
```

5) Collection: A Collection can have duplicate elements. The add() function of Collection is overloaded and also supports adding elements to a Collection object where duplicate elements are not added. This avoids sending the same data of remote nodes to the corresponding master-nodes multiple times. It is up to the programmer to use the appropriate function. The global Collection object is synchronized by sending remote elements in a Collection object to the appropriate master-mode. Collection object is synchronized as shown in Algorithm 2.

C. Examples: Shortest Path Computation and Pagerank

The single-source shortest path (SSSP) computation finds the shortest distance from the source point to all other points

Algorithm 3: Single Source Shortest Path in DH-Falcon

```
1  int changed = 0;
2  relaxgraph (Edge e, Graph graph) {
3      Point (graph) p=e.src;
4      Point (graph) t=e.dst;
5      MIN(t.dist,p.dist+graph.getWeight(p,t),changed);
6  }
7  main(int argc,char *argv[]) {
8      ......
9      foreach (t In graph.points) t.dist=1234567890;
10     graph.points[src].dist=0;
11     while( 1 ){
12         changed = 0; //keep relaxing
13         foreach(t In graph.edges) relaxgraph(t,graph);
14         if(changed == 0)break;
15     }
16     .....
17 }
```

in a graph object. Algorithm 3 shows the main parts of the DSL code for SSSP computation in DH-Falcon for multiple platforms or target devices. Note that, unlike Falcon, DH-Falcon does not have any hardware-centric constructs. Before the parallel foreach call in Line 9, the graph object is read from the disk, and its extra property *dist* is allocated (added using $addPointProperty()$ function). The $read()$ function gets converted to different versions of $read()$ based on the command line argument given for target system to the DH-Falcon compiler. For example, if the target system is a GPU cluster, this converts to a $read()$ function which reads Graph object to CPU memory and then partitions the Graph object and copies the localgraph object on each node from its CPU memory to GPU device memory. The SSSP computation happens in the while loop (Lines 11–15), by repeatedly calling the *relaxgraph()* function. The *dist* property value is reduced atomically using *MIN()* function (Line 5) and *changed* variable will be set to one if *dist* value is reduced. The computation finishes when a fixed-point reached, which is checked in Line 14.

Main parts of the pagerank DSL code in DH-Falcon is shown in Algorithm 4. This algorithm follows a pull-based computation by iterating over innbrs of each Point (Line 3, Algorithm 4). The *ADD()* function used in the algorithm (Line 4) is not atomic as Point *p* modifies its own value.

Algorithm 4: Pagerank in DH-Falcon

```
1  pagerank(Point p, Graph graph) {
2      double val=0.0;
3      foreach (t in p.innbrs) val += t.PR / t.outDegree();
4      p.PR = ADD(val * d, (1 - d) / graph.npoints);
5  }
6  main(int argc, char *argv[]) {
       ......
       foreach (t in graph.points) t.PR = 1 / graph.npoints;
       int cnt = 0;
7      while( ITERATIONS < cnt ){
           foreach (t in graph.points) pagerank(t, graph);
           ++cnt;
8      }
       .....
9  }
```

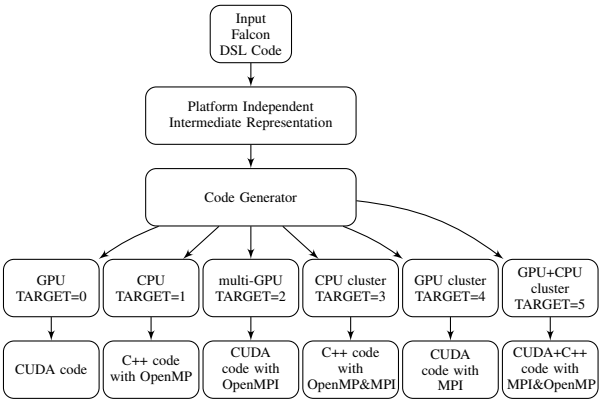


Figure 2: DH-Falcon Code Generation overview

D. Parallelization and Synchronization constructs

1) Foreach statement: A foreach statement in a distributed setup is executed on the localgraph of each machine. A foreach statement gets converted to a CUDA kernel call or an OpenMP pragma based on the target device. There is no nested parallelism and the inner loops of a nested foreach statement are converted to simple for loops. The DH-Falcon compiler generated C++/CUDA code has extra code before and after the parallel kernel call to reach a global consistent state across a distributed system, which may involve data communication. A global barrier is imposed after this step.

To iterate over all the edges of a *localgraph*, either *points* or *edges* iterator can be used. If *points* iterator is used, then a foreach statement using *outnbrs* or *innbrs* iterator (nested under *points* iterator) on each *point* will be needed and this second foreach statement gets converted to a simple for loop. This can create thread divergence on GPUs for graphs that have power-law degree distribution. If iterated over *edges*, each thread receives the same number of edges to operate on, minimizing thread divergence and improving GPU performance. For example, when the SSSP computation is performed on twitter [27] input on a single machine with 8 GPUs, it showed 10× speedup while iterating over *edges* compared to iterating over *points*. In twitter input, half of the edges are covered by 1% of the vertices and the out-degree varies from 0 to 2,997,469.

2) Parallel Sections statement: This statement is used with multi-GPU machines, when there are enough devices and the programmer wants to run a different algorithm on each device, with the graph being loaded from the disk only once for all the algorithms [3].

3) Single Statement: single statement is the synchronization construct of DH-Falcon. It can be be used to lock a single element or a Collection of elements in a distributed system. The DH-Falcon compiler implements distributed locking based on the rank of the process on both CPU and GPU. The details of the implementation can be found in Section V-D and it is used in our implementation of Boruvka's-MST algorithm [28].

V. Code Generation

A. Overview

The code generation methodology of the DH-Falcon compiler is as shown in Figure 2. The DH-Falcon compiler can take a single DSL code and convert it to high-level language code for different target systems. The generated high-level language code is then compiled with a native compiler (nvcc/g++) and libraries (OpenMPI/MPI, OpenMP). Code generation depends on what value is given for the command line argument *TARGET* by the programmer during compilation of the DSL code. The target systems supported by DH-Falcon are (i) single machine with multi-core CPU (ii) single machine with multi-core CPU and one or more GPUs iii) distributed systems with each machine of type (i) or (ii).

The generated code for distributed systems will contain *MPI_Isend()* (non-blocking send) and *MPI_Recv()* calls for communication of updated mutable graph object properties among *localgraph* object on each machine or device. Code generated for a multi-GPU system, supports communication with *cuda-aware-mpi* support of OpenMPI. For other distributed systems with GPUs, the DH-Falcon compiler disables the *cuda-aware-mpi* feature and code with explicit copy of data between CPU and GPU memory is generated along with *MPI_Isend()* and *MPI_Recv()* operations.

There is no distributed locking support across multiple GPU devices in MPI or OpenMPI. The DH-Falcon compiler implements a distributed locking across multiple GPUs and CPUs. This is required for the synchronization statement (single statement) of DH-Falcon.

B. Distributed Graph Storage

The first part the compiler should handle is support for partitioning the input Graph object into N pieces, where N is the number of tasks created to execute on the devices of distributed system. DH-Falcon uses random edge-cut partitioning for graph objects.

DH-Falcon uses C++ classes *DHGraph* and *DGGraph* for storing graph object on CPU and GPU respectively. The *DHGraph* class has functions which read partition-ids of each point and then assign edges to *localgraphs* with *localpoints* and *remotepoints*. For communication between remote-nodes each *remotepoint* is mapped to its master-node and local-vertex-id in the master-node using a hash table. These hash tables are stored in the CPU and/or GPU based on the target system. A *localpoint* value is mapped to its global-vertex-id and vice versa. This is needed when the *localpoint* mutable property value needs to be scattered to remote nodes.

C. Allocation and Synchronization of Global variables

Line 1 of Algorithm 3 declares the global variable *changed*. This variable is accessed inside the function *relaxgraph()*, which is called inside the *foreach* statement from Line 13. The allocation of the variable *changed* depends on the target system. Its declaration gets converted to one of the three different code fragments given in Algorithm 5 depending on the value of *TARGET*. For a target system with CPU and GPU devices, the variable *changed* will be duplicated to two copies, one each on CPU and GPU (Line 3, Algorithm 5). The DH-Falcon compiler generates CUDA and C++ version of the *relaxgraph()* function for the above target system. The CPU and GPU copy of the variable *changed* will be used in C++ and CUDA code (respectively). The analysis carried out by the DH-Falcon compiler for global variable allocation is as shown in Algorithm 6.

Algorithm 5: Generated code for global variable *changed*

```
int changed; //for CPU and CPU cluster
__device__ int changed; //GPU, Multi-GPU and GPU cluster
__device__ int changed;int FCPUchanged; // CPU+GPU cluster
```

Algorithm 6: Global variable allocation in DH-Falcon

```
1 foreach( parallel_region p in program ){
2     foreach( var in globalvars ){
          if (def(var,p) or use(var,p))
              allocate var on target device/devices of parallel code
3     }
4 }
```

Line 14 of Algorithm 3, reads the global variable *changed* to check the exit condition. Here the value of the variable *changed* should be synchronized across all nodes before the read access. The code generated by DH-Falcon for this access, for heterogeneous systems follows the code pattern in Algorithm 7. Algorithm 8 shows the way global variables are synchronized based on the commutative and associative function using which the global variable was modified before a read access.

Algorithm 7: Pseudo code for synchronizing variable *changed*

```
for(each remote node i) sendtoremotenode(i, changed);
int tempchanged = 0;
1 for each remote node i {
      receivefromremotenode(i, tempchanged);
      changed = changed + tempchanged;
2 }
```

Algorithm 8: Pseudo code for synchronizing global variable var

```
for (each remote node i) sendtoremotenode(i, var);
Type tempvar = 0;// Type = Data type of var
1 for each remote node i {
      receive from remote node(i, tempvar);
      update var using tempvar
      based on function used to modify var.(MIN,MAX,ADD etc).
2 }
```

D. Distributed locking using single statement

The usage of single statement in a function *fun()* is shown in Algorithm 9. In a distributed execution, the point p may be present as local or remote-vertex in multiple nodes, as edges $(u, p), (v, p)$ and (x, p) on nodes $N1$, $N2$ and $N3$. The single statement converts to a Compare and Swap (CAS) operation in the generated code for each node, and exactly one

Algorithm 9: single statement in `DH-Falcon`

```
1  fun(Point t, Graph graph) {
2      foreach( p In t.outnbrs ){
3          if( single(p.lock) ){
               stmt_block{}
4          }
5      }
6  }
7  main() {
8      ....
9      foreach (Point p In graph) fun(p, graph);
       ......
10 }
```

Algorithm 10: Code generation for single statement `DH-Falcon`

Input: Function fun() with `single` statement
Output: Functions fun1() and fun2(), synchronization code

(I) Reset lock.
  forall (Point t in Subgraph G_i of G) t.lock ← MAX_INT
(II) Generate code for *fun1()* from *fun()*
  (a) In *fun1()* remove statements inside `single` statement.
  (b) Convert `single`(t.lock) to `CAS`(t.lock,MAX_INT,rank).
(III) Synchronize *lock* value.
  (a) Send successful *lock* values to process with rank zero.
  (b) At rank zero process
    Make *lock* value to *MIN* of all values.
    Send *lock* value to all remote-nodes.
  (c) On nodes with rank > zero
    Receive *lock* value from rank zero process.
    Update *lock* value.
(IV) Generate code for *fun2()* from *fun()* .
  (a):- Convert `single` to CAS(t.lock,rank,MAX_INT-1).
  (b):- Generate code for *fun2()* from *fun()* including all statement.
(V) At call site of *fun()*, generate code with parallel call to *fun1()* and *fun2()* inorder.

thread will succeed in getting a *lock* on p in each of the nodes $N1$, $N2$ and $N3$. However, *only one thread across all nodes* should succeed in getting the *lock* on p as per the semantics of the `single` statement. The `DH-Falcon` compiler generated code ensures that a function with a `single` statement is executed in two phases and the semantics is preserved. In the first phase, the function is executed only up to and including the `single` statement which tries to get the *lock*. Then, all the processes send to process with *rank* zero (P_0), all the successful `CAS` operations using *MPI_Isend()*. Thereafter, P_0 process collects the messages from remote-nodes (*MPI_Recv()*) and sets *lock* value for all the points to the least process *rank* among all the processes which succeeded in getting the *lock*. For the `Point` p mentioned above, if the nodes $N1$, $N2$ and $N3$ have the *ranks* 1, 2, and 3, respectively, the *lock* value will be set to 1 by process P_0. After this, process P_0 sends the modified *lock* value back to each remote-node, and they update the *lock* value. In second phase, the `single` statement will be executed with `CAS` operation checking for each `Point` p, whether the current *lock* value equals the *rank* of the process, and if so, $stmt\_block\{\}$ will be executed. A successful *single* statement on a `Point` p will have value (MAX_INT-1) for the property lock, after second `CAS` operation. Otherwise value could be MAX_INT or a value less than number of processes (*rank*) used in the program execution.

The pseudo code for distributed locking code generation is shown in Algorithm 9. Function *fun()* is duplicated to two versions, *fun1()* and *fun2()*. *fun1()* simply tries to get the *lock*. Code for combining *lock* value of each element to the minimum *rank* value by the process P_0 follows. *fun2()* executes the $stmt\_block\{\}$, as now *lock* is given to process with least *rank* and only one thread across all nodes will succeed in getting the *lock* for a `Point` p. Such an implementation is used in the Boruvka-MST implementation.

E. Optimized communication of Mutable Graph Properties

Algorithm 11: Code generation for synchronization of graph properties in `DH-Falcon`

(I) store use and def.
  forall *Functions* fun *in the program* **do**
    store mutable graph properties read by fun in vector $fun.use[]$.
    store mutable graph properties modified by fun in vector $fun.def[]$.
  end
(II) create call-string(CS) of the parallel functions in program.
(III) findout data to be communicated.
  forall *Functions* fun *in the* CS **do**
    forall *properties* p *in* $fun.def[]$ **do**
      forall *successor succ of* fun *in* CS **do**
        if ($succ.use[]$ *contains* p, *before* p *is modified again*)
          add p to $fun.comm[]$
        }
      end
    end
  end
(IV) prefix and suffix code for communication.
  (a):-prefix code.
  forall *Functions* fun *in the* CS **do**
    forall *properties ppty in* $fun.comm[]$ **do**
      copy $pppty[i]$ to $temppppty[i]$ for all elements i.
    end
  end
  4(b):-**forall** *Functions* fun *in the* CS **do**
    forall *properties ppty in* $fun.comm[]$ **do**
      forall *remote-nodes* rn_x *of Subgraph* G_k **do**
        if($temppppty[i] \neq ppty[i]$)
        add $ppty[i]$ to $buffer_x$
      end
    end
    forall(remote-node rn_x)send $buffer_k$ to node rn_k.
  end

In the SSSP example of Algorithm 3, when the remote vertex property value *dist* is modified (Line 5, Algorithm 3) it should be synchronized with the master-node of the remote-vertex. The `DH-Falcon` compiler generates code for such synchronization. The analysis is as shown in Algorithm 11. Each function which is the target of a `foreach` statement, stores used and modified mutable graph properties in the arrays $use[]$ and $def[]$ (Step 1). Each function checks whether the values modified by the elements in $def[]$ are used by any successor of the function, before they are modified again

Input	Type	|V|	|E|
ljournal [31]	social	5,363,260	77,991,514
arabic [32]	web	22,744,080	631,153,669
uk2005 [33]	web	39,460,000	921,345,078
uk2007 [33]	web	105,896,555	3,738,733,602
twitter [27]	social	41,652,230	1,468,364,884
frontier [34]	social	65,608,366	1,806,067,135

TABLE II. Input graphs and their properties

(Step 3). If so, code to synchronize all such properties is generated (Step 4). The property values to be communicated can be remote-points (push-based update) or local-points (pull-based update). DH-Falcon DSL code for SSSP and auto-generated code targeting multi-GPU devices can be found here [29].

F. Other features

The DH-Falcon compiler generates code to store edge weights only if they are accessed in the program using *getWeight()* functions. The pagerank, K-CORE, and BFS computation do not use edge weights. The receive buffer (to get updated values) is allocated only for one node, as updation is done synchronously. But the send buffer is allocated for all remote-nodes as the send operation is asynchronous.

VI. Experimental Evaluation

We have used large-scale graphs available in the public domain for result analysis and they are listed in Table II. For scalability checking we have generated RMAT graphs of bigger size with GT-Graph [30] tool with parameter values a=0.45, b=0.25, c=0.15 and d=0.15.

A. Distributed Machines

To evaluate the generated distributed code, we used three different systems.

1) CPU cluster: We used a sixteen node CRAY XC40 cluster. Each node of the cluster consists of two CPU sockets with 12 Intel Haswell 2.5 GHz CPU cores each, 128 GB RAM and connected using Cray aries interconnect.

2) GPU cluster: We used an eight node CRAY cluster. Each node in the GPU cluster has Intel IvyBridge 2.4 GHz based single CPU socket with 12 cores and 64 GB RAM, and single Nvidia-Tesla K40 GPU card with $2,880$ cores and 12 GB device memory and connected using Cray aries high-speed interconnect. This cluster is also used for heterogeneous execution with i) first four nodes using only CPU and other four nodes using GPU, and ii) four nodes using both CPU and GPU.

3) Multi-GPU machine: A *single* machine with *eight* Nvidia-Tesla K40 GPU cards, each GPU with $2,880$ cores and 12 GB memory, Intel(R) Xeon(R) CPU multicore CPU with 32 cores and 100 GB memory.

B. CPU Cluster Execution

1) Public Inputs: Figure 3 shows the speedup of DH-Falcon over PowerGraph on sixteen node CPU cluster for public inputs in Table II. The benchmarks used are Single

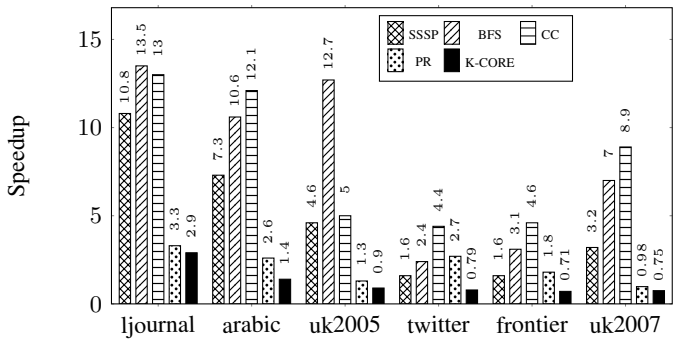
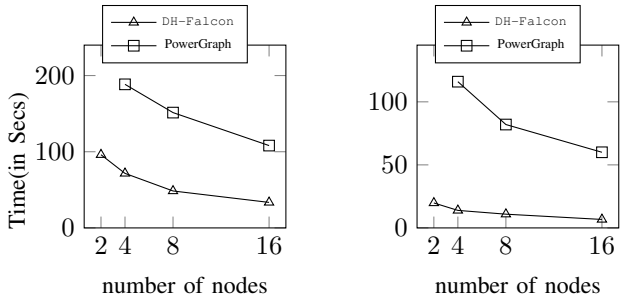


Figure 3: Speedup over PowerGraph on 16 node CPU cluster



(a) SSSP (b) Connected Components

Figure 4: DH-Falcon Vs PowerGraph- on UK-2007

Source Shortest Path (SSSP), Breadth First Search (BFS), Pagerank (PR), Connected Components (CC) and K-CORE. The PowerGraph running time is taken as the best of the coordinated and the oblivious ingress methods. It is found that the amount of data communicated by PowerGraph is high and is up to 5× to 30× more compared to DH-Falcon. DH-Falcon is able to outperform PowerGraph for most of the (benchmark, input) pair. The major reason being amount of data communicated by DH-Falcon code is less compared to PowerGraph as DH-Falcon uses edge-cut partitioning and optimized communication. The pagerank and k-core implementations of DH-Falcon send more amount of data compared to BFS, SSSP and CC. The pagerank algorithm showed less speedup, as the algorithm modifies value of each point in the subgraph and this has to be scattered to all the remote-nodes. That is, more data is communicated in pagerank algorithm compared to other algorithms. The k-core running time is calculated as the average time of running algorithm for 11 iterations for ($kmin = 10$) to ($kmax = 20$). The DH-Falcon implementation of k-core also communicates more volume of data. Figure 4 shows the running time for SSSP and CC on *uk-2007* input for number of nodes ranging from two to sixteen. PowerGraph failed to run on two nodes.

2) Scalability Test: The DH-Falcon and PowerGraph codes were run on four big RMAT inputs generated using GT-Graph. The inputs have 300, 600, 900, 1200 million vertices and number of edges being ten times the number of vertices. The scalability is compared for benchmarks SSSP,

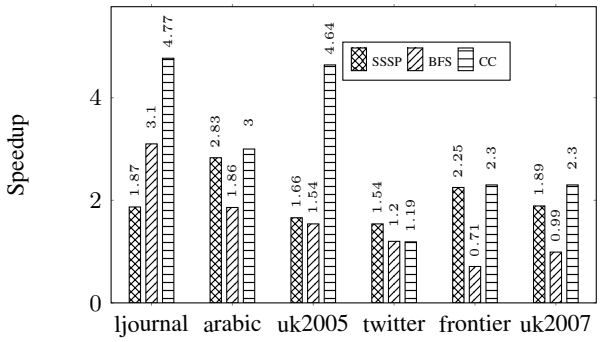
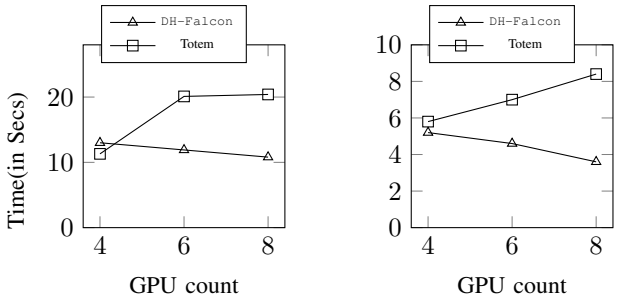


Figure 5: Speedup over Totem on 8 GPU multi-GPU machine



(a) SSSP-Uk2007 (b) CC-Frontier

Figure 6: Running Time- Public inputs on 8 GPU machine

BFS and CC and the results are shown in Table III. The PowerGraph framework failed to run on the rmat1200 input. Here also DH-Falcon is able to outperform the PowerGraph framework.

Algo	Framework	rmat 300	rmat 600	rmat 900	rmat 1200
SSSP	PowerGraph	158.2	358.9	442.4	segfault
	DH-Falcon	112	238.9	384.7	478.7
CC	PowerGraph	107	305	324	segfault
	DH-Falcon	34.9	72.9	92.4	188.8
BFS	PowerGraph	24.8	49.7	93.2	segfault
	DH-Falcon	15.8	33.2	42.9	75.1

TABLE III. Running Time (in Secs) of rmat graph on fixed 16 node CPU cluster.

C. GPU execution

For GPU execution of DH-Falcon we used two different device configurations, multi-GPU machine and GPU cluster.

1) Multi-GPU machine: DH-Falcon codes were executed for all the public inputs and the DH-Falcon performance is compared with Totem [22]. The results are shown in Figure 5 when all the benchmarks were run using all the eight GPUs. Out of the eight GPUs, two sets with four GPUs (devices (0 to 3) and (4 to 7)) each were having *peer-access* capability. Totem showed a sharp increase in running time when number of GPUs is changed from four to five and thereby showing non-linear scalability. The DH-Falcon compiler is not using *peer-access* capability and showed linear capability. So if inputs fit within four GPUs Totem was able to achieve better performance for some inputs and benchmarks compared to DH-Falcon. The DH-Falcon compiler uses OpenMPI with cuda_aware_mpi feature for communication between GPUs. DH-Falcon allows iterating over edges in a localgraph object using *edges* iterator, which provides work-balance across threads in each GPU. The special behaviour of Totem when increasing number of GPUs from 4 to 6 is shown in Figure 6 for SSSP on uk-2007 input and for CC on frontier input.

2) GPU Cluster: Figure 7 shows relative speedup of DH-Falcon on 8 nodes for GPU over 8 node CPU cluster, on public inputs. BFS algorithm shows less speedup on GPU

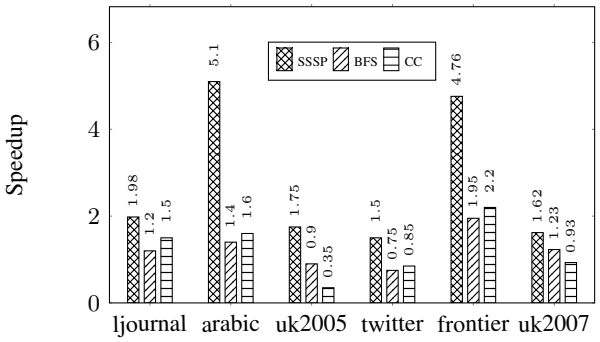


Figure 7: Relative speedup on 8 node GPU cluster over 8 node CPU cluster

cluster as BFS is not a compute-bound kernel and communication between GPUs on different nodes has to go through the CPU. In BFS nearly 90% time is spent on communication on GPU cluster. The CPU cluster codes on an average spend 35% time for communication. But in the GPU cluster the computation finishes fast and more than 60% of the time is spent for communication. This is also due to the fact that communication between GPUs of two nodes has to go through the CPU and so GPU cluster communication will take more time compared to CPU cluster communication for the same volume of data.

D. Scalability Test for multi-GPU machine, GPU cluster and CPU +GPU Cluster

For scalability analysis on distributed systems with GPUs three rmat graphs with 100, 200 and 250 million vertices were created with each graph having edges ten times of number of vertices. Table IV shows the running time for different systems on benchmarks BFS, CC and SSSP. First two columns show the running time of DH-Falcon and Totem on multi-GPU machine. Other two columns show running time on distributed systems with each machine having one GPU (column III) and one GPU or CPU (column IV). Multi-GPU system has better running time as there is very little communication overhead between GPUs on a single machine. The GPU cluster running time is high as the communication time from GPU to GPU on two nodes is very high. The CPU +CPU cluster has worst performance as there is mismatch in computation time for

parallel code in GPU and CPU devices.

Input	Algo	multi GPU	Totem	GPU cluster	GPU+CPU cluster
rmat100	BFS	1.2	1.2	3.72	8.4
	CC	1.1	2.98	2.74	8.3
	SSSP	4.99	6.18	9.3	31.6
rmat200	BFS	1.8	1.5	6.89	16.7
	CC	2.4	5.0	5.7	17.4
	SSSP	10.9	10.36	18.1	66.9
rmat250	BFS	2.9	2.1	8.65	20.1
	CC	3.46	5.7	7.21	20.9
	SSSP	12.30	13.1	21.9	39.6

TABLE IV. Running Time (in Secs) of rmat graph on fixed 8 devices (8 GPUs or four GPU+ four CPU).

E. Boruvka MST

The Boruvka MST algorithm uses the Union-Find Set data type of DH-Falcon. This algorithm also uses the single statement of DH-Falcon. The single statement is used to add only one edge connecting two disconnected components among the many possible edges with same weight. Running time of the algorithm for public inputs is shown below in Table V. The outermost foreach statements were called using points iterator and the kernel with single statement was similar to the one given Algorithm 10. The twitter input has similar running time on multi-GPU machine and GPU cluster as iterator points was used and it created thread divergence. A code with edges iterator can be written like the SSSP example in Algorithm 3, which will improve running time for twitter input. The memory available on 8 GPUs was not sufficient to run MST on uk-2007 input.

System	ljournal	arabic	uk2005	twitter	frontier
Multi-GPU machine	9.05	27.98	62.3	279.1	112.7
GPU cluster	19.4	49.6	105.7	287.3	150.9
CPU cluster	44	141	278	709	1275

TABLE V. Running Time of MST (in Seconds) on Different Distributed Systems.

F. Dynamic Graph Algorithms

The DH-Falcon compiler allows mutation of graph objects and hence supports programming dynamic graph algorithms. DH-Falcon compiler looks at algorithms which add edges and points to the graph object and allocate more space to store edges for each vertex. The programmer can specify as command line arguments minimum (*min*) and maximum (*max*) space to be allocated per vertex. The *read()* function allocates extra space which is equal to second highest in the 3-tuple (*min*, *max*, *outdegree*) for each vertex. The deletion of edges and points is done using marking.

1) Dynamic-SSSP: The incremental dynamic-SSSP gives speedup of around 4.5× on GPU cluster, 11× on multi-GPU machine and 7.5× on CPU cluster. The rmat graphs of Table IV and Table III were used for GPU and CPU systems respectively. After the initial SSSP computation up to 5% edges were added during experimentation to the rmat-graphs and SSSP is computed incrementally from previous computation. Other vertex-centric incremental dynamic algorithms can be programmed in DH-Falcon in a similar fashion.

2) Delaunay Mesh Refinement (DMR): The DH-Falcon implements the distributed DMR based on PCDM algorithm [35]. The DMR algorithm has graph with mesh of triangles. This algorithm is totally different from other algorithms discussed above, where graph is collection of edges. So in the distributed implementation of DMR in DH-Falcon, each triangle in the *localgraph*, which contains a constrained edge is added to a Collection object $coll1$. An edge e is a constrained edge if it present in two *subgraphs* G_i and G_j, $i \neq j$. Then triangles with constrained edges which are refined in a superstep S_i, are added to another Collection object $coll2$. The $coll2$ object will be synchronized by DH-Falcon using $coll1$. Then triangles in $coll2$ will be refined in remote-node which contains the same constrained edge. The refinement algorithm is same as that of PCDM algorithm. The triangular mesh is partitioned using ParMetis [36] Tool, which provides good partitioning with very few constrained edges. When the mesh size increases, there is an increase in the percentage of constrained edges produced by ParMetis. Table VI show running time on different systems for DMR algorithm with 8 devices for meshes with 5, 10 and 15 million triangles.

System	r5M	r10M	r15M
Multi-GPU machine	1.2	1.7	4.3
GPU cluster	3.1	4.1	9.5
CPU cluster	20.1	30.2	51.8

TABLE VI. Running Time of DMR (in Seconds) on Different Distributed Systems.

VII. Conclusion and Future Work

We presented DH-Falcon, a domain-specific language for expressing graph algorithms targeting heterogeneous distributed systems. It supports writing explicitly parallel programs and makes programming easier. we illustrated its simplicity where a single DSL program is converted to codes of different targets. Experimental evaluation shows that the efficiency of compiler generated codes is close to that of the frameworks for distributed systems. In future, we aim to extend DH-Falcon to support more input formats such as meshes as graph objects.

References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference*

on Management of Data, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[2] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 17–30. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387883

[3] U. Cheramangalath, R. Nasre, and Y. N. Srikant, "Falcon: A graph manipulation language for heterogeneous systems," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 54:1–54:27, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2842618

[4] U.Cheramangalath, R.Nasre, and Y. N. Srikant, "Falcon-graphdsl compiler," Jan 2017. [Online]. Available: https://github.com/falcon-graphdsl/

[5] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012. [Online]. Available: http://dx.doi.org/10.14778/2212351.2212354

[6] K. Lang, "Finding good nearly balanced cuts in power law graphs," Tech. Rep., 2004.

[7] S. Salihoglu and J. Widom, "Gps: A graph processing system," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, ser. SSDBM. New York, NY, USA: ACM, 2013, pp. 22:1–22:12. [Online]. Available: http://doi.acm.org/10.1145/2484838.2484843

[8] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, "Green-Marl: A DSL for Easy and Efficient Graph Analysis," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 349–362. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151013

[9] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun, "Simplifying Scalable Graph Processing with a Domain-Specific Language," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 208:208–208:218. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544162

[10] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, "Mizan: A system for dynamic load balancing in large-scale graph processing," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 169–182. [Online]. Available: http://doi.acm.org/10.1145/2465351.2465369

[11] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[12] R. Shaposhnik, C. Martella, and D. Logothetis, *Practical Graph Analytics with Apache Giraph*, 1st ed. Berkely, CA, USA: Apress, 2015.

[13] R. Chen, J. Shi, Y. Chen, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 1:1–1:15. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741970

[14] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 285–296, Sep. 2010. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920881

[15] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 810–818. [Online]. Available: http://doi.acm.org/10.1145/1851476.1851593

[16] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 505–516. [Online]. Available: http://doi.acm.org/10.1145/2463676.2467799

[17] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," in *In Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[18] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 31–46. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387884

[19] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *SIGPLAN Not.*, vol. 48, no. 8, pp. 135–146, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2517327.2442530

[20] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 472–488. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522740

[21] J. Yan, G. Tan, Z. Mo, and N. Sun, "Graphine: Programming graph-parallel computation of large natural graphs for multicore clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 6, pp. 1647–1659, June 2016.

[22] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and

M. Ripeanu, "A Yoke of Oxen and a Thousand Chickens for Heavy Lifting Graph Processing," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 345–354. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370866

[23] G. Shashidhar and R. Nasre, *LightHouse: An Automatic Code Generator for Graph Algorithms on GPUs*. Cham: Springer International Publishing, 2017, pp. 235–249. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-52709-3_18

[24] D. Sengupta, N. Sundaram, X. Zhu, T. L. Willke, J. Young, M. Wolf, and K. Schwan, *GraphIn: An Online High Performance Incremental Graph Processing Framework*. Cham: Springer International Publishing, 2016, pp. 319–333.

[25] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181

[26] L. P. Chew, "Guaranteed-quality Mesh Generation for Curved Surfaces," in *Proceedings of the Ninth Annual Symposium on Computational Geometry*, ser. SCG '93. New York, NY, USA: ACM, 1993, pp. 274–280. [Online]. Available: http://doi.acm.org/10.1145/160985.161150

[27] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 591–600. [Online]. Available: http://doi.acm.org/10.1145/1772690.1772751

[28] S. Chung and A. Condon, "Parallel implementation of Bouvka's minimum spanning tree algorithm," pp. 302–308, 1996.

[29] U.Cheramangalath, R.Nasre, and Y. N. Srikant, "Dh-falcon sssp dsl and auto-generated code," may 2017. [Online]. Available: https://github.com/falcon-graphdsl/Falcon-A-Graph-Manipulation-Language-for-Heterogeneous-Systems/blob/master/GPU/sssp.pdf

[30] D. A. Bader and K. Madduri, "Gtgraph: A synthetic graph generator suite," 2006.

[31] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '09. New York, NY, USA: ACM, 2009, pp. 219–228. [Online]. Available: http://doi.acm.org/10.1145/1557019.1557049

[32] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Softw. Pract. Exper.*, vol. 34, no. 8, pp. 711–726, Jul. 2004. [Online]. Available: http://dx.doi.org/10.1002/spe.587

[33] P. Boldi, M. Santini, and S. Vigna, "A large time-aware web graph," *SIGIR Forum*, vol. 42, no. 2, pp. 33–38, Nov. 2008. [Online]. Available: http://doi.acm.org/10.1145/1480506.1480511

[34] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *2012 IEEE 12th International Conference on Data Mining*, Dec 2012, pp. 745–754.

[35] L. P. Chew, N. Chrisochoides, and F. Sukup, "Parallel constrained delaunay meshing," 1997.

[36] "Parmetis - parallel graph partitioning and fill-reducing matrix ordering," 2013. [Online]. Available: http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview