Falcon:- A Graph Manipulation Language for Distributed Heterogeneous Systems

A THESIS

SUBMITTED FOR THE DEGREE OF

Doctor of Philosophy

IN THE

Faculty of Engineering

BY

Unnikrishnan Cheramangalath

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

July, 2017

Declaration of Originality

I, **Unnikrishnan Cheramangalath**, with SR No. **04-04-00-10-12-11-1-08721** hereby declare that the material presented in the thesis titled

Falcon:- A Graph Manipulation Language for Distributed Heterogeneous Systems

represents original work carried out by me in the **Deparment of Computer Science and Automation** at **Indian Institute of Science** during the years **2011-2017**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discusions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name:Professor Y N Srikant Advisor Signature

DEDICATED TO

To My Parents
Narayanan Cheramangalath and Ammu Panikkath Narayanan

Acknowledgements

I would like to thank Prof. Y. N. Srikant for providing an opportunity to work with him and guiding me throughout my research with immense pleasure. I thank him for giving valuable research inputs and encouragement. Whenever I required a help, I always found Prof. Y.N. Srikant. I am deeply obliged to him for his support in academics, research collaborations, career building and logistics to name few.

I am intensely indebted to Prof. Rupesh Nasre, IIT Madras as a friend starting from my first year of PhD and later as a mentor in my research work. His valuable inputs had a major role in my work being published in reputed journal and conference. I am also thankful to him for the hospitality I got at IIT Madras, whenever I went for research discussions.

I express my gratitude to Prof. Govindrajan and Prof. Lakshmi for their support in running experiments on CRAY Cluster of SERC. I thank Prof. Uday and Prof. Mathew Jacob for the research inputs.

I am grateful to administrative staffs Ms.Suguna, Ms.Padmavathi, Ms.Lalitha and technical staffs Mr. Jadgish and Mr. Pushparaj for their support throughout my research.

I am very much thankful to Prof. Sebastian Hack, University of Saarland, Saarbrücken for providing an opportunity to do an internship under him. I also thank Roland Leißa for his inputs and suggestion during the internship.

I express gratitude to my labmates Rajesh, Ashish, Kartik, Namrata, Nitesh and Parita for their friendship which gave memorable and enjoyable moments. Special acknowledgement to Nitesh and Parita for extending my work which lead to a publication.

I thank my friends from IISc - Rijesh, Sarath, Nimesh, Rohit for their support and help. I thank my relatives in IISc, Rajit and Yadunath for their support and giving memorable days with my family.

I cannot thank enough my mother and brothers for their support. Last but not the least, I have no words to express my gratitude to Anagha my daughter, and Gayathri my wife, for being calm, for their countless sacrifices and motivation throughout this journey. I would like to dedicate this thesis to my lovely family.

Abstract

Graphs model relationships across real-world entities in web graphs, social network graphs, and road network graphs. Graph algorithms analyze and transform a graph to discover graph properties or to apply a computation. For instance, a pagerank algorithm computes a rank for each page in a webgraph, and a community detection algorithm discovers likely communities in a social network, while a shortest path algorithm computes the quickest way to reach a place from another, in a road network. 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.

Graph algorithms can be executed on multi-core CPUs, GPUs with thousands of cores, multi-GPU devices, and CPU+GPU clusters, depending on the size of the graph object. While programming such algorithms on heterogeneous targets, a programmer is required to deal with parallelism and and also manage explicit data communication between distributed devices. This implies that a programmer is required to learn CUDA, OpenMP, MPI, etc., and also the details of the hardware architecture. Such codes are error prone and difficult to debug. A Domain Specific Language (DSL) which hides all the hardware details and lets the programmer concentrate only the algorithmic logic will be very useful.

With this as the research goal, Falcon, graph DSL and its compiler have been developed. Falcon programs are explicitly parallel and Falcon hides all the hardware details from the programmer. Large graphs that do not fit into the memory of a single device are automatically partitioned by the Falcon compiler. Another feature of Falcon is that it supports mutation of graph objects and thus enables programming dynamic graph algorithms. The Falcon compiler converts a single DSL code to heterogeneous targets such as multi-core CPUs, GPUs, multi-GPU devices, and CPU+GPU clusters. Compiled codes of Falcon match or outperform state-of-the-art graph frameworks for different target platforms and benchmarks.

Publications based on this Thesis

- Unnikrishnan Cheramangalath, Rupesh Nasre, and Y. N. Srikant. Falcon: A graph manipulation language for heterogeneous systems. Transactions on Architecture and Code Optimizations (TACO), 12(4):54, 2016. http://dl.acm.org/citation.cfm?doid=2836331.2842618.

- Unnikrishnan Cheramangalath, Rupesh Nasre, and Y. N. Srikant. DH-Falcon: A Language for large-scale graph processing on Distributed Heterogeneous systems. IEEE CLUSTER, 2017 **(Accepted)**.

- Nitesh Upadhyay, Parita Patel, Unnikrishnan Cheramangalath and Y. N. Srikant. Large Scale Graph Processing in a Distributed Environment. HeteroPar (2017) **(Accepted)**.

- Unnikrishnan Cheramangalath, Rupesh Nasre, and Y. N. Srikant. Falcon Graph DSL Compiler Version 1, Jan 2017. [Online]. Available at https://github.com/falcon-graphdsl/ .

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Graphs model relationships across real world entities in web graphs, social network graphs, and road network graphs. Graph algorithms analyze and transform a graph to discover graph properties or to apply a computation. For instance, a pagerank algorithm computes a rank for each page in the webgraph, a community detection algorithm discovers likely communities in a social network, while a shortest path algorithm computes the quickest way to reach from one place to another in a road network.

An algorithm is irregular if its data access pattern or control-flow pattern is unpredictable at compile time. Static analysis techniques prove inadequate to deal with the analysis and parallelization of irregular algorithms, and we require dynamic techniques to deal with such situations. Traditionally, graph algorithms have been perceived to be difficult to analyze as well as parallelize because they are irregular.

GPUs further complicate graph algorithm implementations: managing separate memory spaces of CPU and GPU, SIMD (single instruction multiple data) execution, exposed thread hierarchy, asynchronous CPU/GPU execution, etc. Hand-written and efficient implementations are not only difficult to code and debug, but are also error prone.

Depending on the domain, the number of edges in a graph may vary and 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. Programming such systems requires the Message Passing Interface(MPI), which provides APIs for sending and receiving data across machines, distributed shared memory, etc. Graph algorithms can be executed on multi-core CPU clusters, massively parallel GPU clusters and multi-GPU devices with each GPU having thousands of cores. There are many frameworks for large scale graph processing

on CPU clusters. Google's Pregel [65], which uses the Bulk Synchronous Parallel (BSP) model of execution over vertices and PowerGraph [46] which follows Gather-Apply-Scatter (GAS) model of execution over edges are examples of such frameworks. Distributed processing will be efficient only if a graph is partitioned and distributed across machines properly, so that there is work balance on all the nodes and communication between the nodes is minimized. Such large graphs are partitioned using random partitioning, and graph partitioning tools have failed to partition such graphs properly with their heuristics.

It would be really helpful if a programmer can specify a graph algorithm in a hardware independent manner and focus solely on the algorithmic logic. Unfortunately, such an approach which essentially requires auto-parallelization of sequential code provides limited performance in general when compared to a manually parallelized hardware-centric code (by an expert). Our goal in this work is to bridge this performance gap between auto-generated code and a manually crafted implementation. We wish to let the programmer write the algorithm at a higher level (much higher than CUDA and OpenCL), without any hardware-centric constructs. For a distributed system, the programmer need not specify how the graph should be partitioned across distributed machines, and how communication and synchronization between machines should happen. To achieve a performance close to that of a hand crafted code, we make two compromises: (i) we allow only graph algorithms to be specified (i.e., we do not provide special constructs for other types of algorithms), and (ii) we require the code to be explicitly parallel. The first compromise trades generality for speed, while the second one allows our code generator to emit hardware-specific code.

With this in focus `Falcon`, a domain specific language (DSL) for graph processing has been developed and implemented. This makes programming graph algorithms easier. The DSL codes are explicitly parallel and the same DSL code can be converted to executables for CPU, GPU, multi-GPU machine, CPU cluster, GPU cluster and CPU+GPU cluster. The target system for which the executable is to be created can be specified as a command line argument during the compilation of the DSL code. The `Falcon` compiler allows mutation of graph objects and hence dynamic graph algorithms can be programmed in `Falcon`. A `Falcon` programmer need not learn MPI, OpenMP, CUDA etc., and the details of the hardware architecture. `Falcon` programs are at a higher level of abstraction compared to CUDA/C++ code for GPU/CPU, and distributed framework (like PowerGraph) codes. It follows the BSP [95] model of execution. Experimental evaluation shows that our implementations outperform or match with the state-of-the-art frameworks which work for single GPU, multi-core CPU, CPU cluster and multi-GPU devices. The important feature of the `Falcon` is that the a single DSL code can be converted to all the targets mentioned above.

1.2 Contributions of our work

We have designed and developed a domain specific language (DSL), `Falcon`, for graph manipulation. `Falcon` supports a wide range of target systems which include

- A single machine with multi-core CPU. The machine can also have one or more GPUs.
- CPU clusters, with each machine having a multi-core CPU and private memory.
- GPU clusters, with each machine having a GPU. Such a machine will also have a multi-core CPU on which the operating system runs.
- CPU+GPU clusters, where both CPU and GPU of each machine in the cluster is used for computation.

A single DSL code written by a programmer can be converted to all the target systems mentioned above by giving proper command line arguments to the `Falcon` compiler. This makes programming easier, as a programmer is required to concentrate only on the algorithmic logic and need not learn different libraries, languages and the details of device architectures, which are required for efficient manual coding. For example, if programmer has no framework available, he/she will be forced to learn CUDA, OpenMP, Message Passing Interface (MPI) etc., and device architectures such as SIMT execution model of GPU, distributed memory of clusters, etc. Programming using these APIs and libraries is error prone and programs are difficult to debug. The `Falcon` compiler generates high level language codes from the DSL code with these libraries. The generated code is then compiled with the appropriate device-specific compilers and libraries to create the executables. This approach enables better productivity.

`Falcon` extends the C language with additional data types relevant to graph processing. It has parallel and synchronization constructs for specifying explicitly parallel graph algorithms. Synchronization constructs are important in graph algorithms as most of the algorithms are irregular. `Falcon` also support atomic and reduction operations. The `Falcon` compiler creates an abstract syntax tree (AST) from the input DSL program. Traversals on this AST emit code for appropriate targets.

Unlike previous reported DSLs for graph algorithms [53, 79], `Falcon` supports mutation of graph objects, where the structure of the graph changes dynamically. This feature is important in current social networks. For example, if WebGraph is taken as a graph where a webpage is a vertex and a link to a wepage y from a webpage x is an edge $x \rightarrow y$ of the graph. WebGraph changes over time due to addition and deletion of webpages and links. This requires updating the pagerank value of each webpage and dynamic algorithms allow new values to be calculated

from current values. This saves a lot of time compared to computation of values from scratch. Another example is the computation of the shortest distance on road networks, which also change over time.

`Falcon` supports devices with more than one GPU (multi-GPU devices). It allows running different algorithms on the same input graph in parallel on multi-GPU devices (assuming that the graph fits within the GPU memory). To the best of our knowledge, there is no graph framework or DSL which supports this.

The `Falcon` compiler generates code for distributed systems, and thereby enabling computations on large scale graphs. In such a system, the `Falcon` compiler does following things automatically.

- Partitioning the input graph and storing it as subgraphs on each device participating in the computation.
- Communication of graph properties across devices using the BSP execution model.
- Generation of optimized communication code by sending only modified values.

The CUDA framework does not have any support for a barrier for the entire kernel. Similarly MPI library does not provide a distributed locking mechanism. The `Falcon` compiler has implemented support for both of these in software and provides them to the programmer as constructs in `Falcon`.

Meshes can be viewed as graphs. For example, a mesh of triangles can be viewed as a graph with points and edges, with the properties that an edge will be part of one triangle (boundary edge) or two triangles. If a triangle is taken as a graph property, it can have up to three neighbouring triangles, three angles, etc. `Falcon` allows adding extra properties to points and vertices in the graph as found in previous graph DSLs. The novelty of `Falcon` is that it allows properties to be added to a graph, such as triangle, and process the graph as a mesh of triangles. This feature of `Falcon` allows processing dynamic meshes where the structure of the mesh changes during program execution. Algorithms such as Delaunay Mesh Refinement (DMR) [26] have been implemented in `Falcon` for all the targets mentioned before. Mesh algorithm are used in computational geometry.

`Falcon` allows writing different implementations for the same algorithm using its constructors and data types. For example, the single source shortest path (SSSP) algorithm can be implemented in different ways in `Falcon` and each implementation is meant for a particular target system and a set of input graph properties. This is discussed in detail in Chapter 2.

In brief, *to the best of our knowledge,* `Falcon` *is the first framework which supports such a wide variety of architectures for implementing static and dynamic graph algorithms.*

1.3 Organization of the thesis

Rest of the thesis is organized as described below. Chapter 2 discusses basic graph related definitions, well known graph algorithms, different ways of implementing graph algorithms, hardware architectures, and different execution models. Chapter 3 discusses past research on graph frameworks for heterogeneous distributed systems. Chapter 4 explains the design of `Falcon` DSL. Chapter 5 deals with code generation and experimental evaluation on a single machine. Chapter 6 contains the details of code generation for distributed execution of graph algorithms. We conclude in Chapter 7 and provide directions for future work.

Chapter 2

Preliminaries and Background

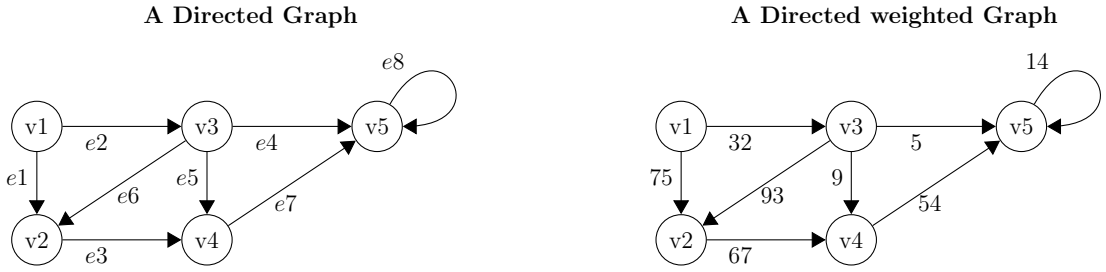
2.1 Graph definitions [16]

A *graph* is an abstract structure that represents a structural relationship between entities. A graph G(V,E) consists of a set of vertices V and a set of edges E. An *edge* in E is a pair of vertices from V. A graph can have a set of optional *mutable properties* D associated with V or E. A graph is called a *weighted graph* if it has a weight or a number associated with each edge. Some algorithms require the weight to be a nonnegative integer number or real number or positive number etc. A graph is *undirected* if all its edges are bidirectional. A graph is *directed* if all its edges are directed from one vertex to another. For a directed edge $e(u \rightarrow v)$, u is called the source and v is called the sink of the edge e, u is called a predecessor of v and v is a successor of u. An example of an unweighted directed graph is shown in Figure 2.1(a) with

V = {v1, v2, v3, v4, v5} and
E= {e1, e2, e3, e4, e5, e6, e7, e8 } where
e1={v1, v2}, e2={v1, v3}, e3={v2, v4}, e4={v3, v5}, e5={v3, v4},
e6={v3, v2}, e7={v4, v5}, e8={v5, v5}

Figure 2.1(b) shows a weighted directed graph with the same set of vertices and edges of Figure 2.1(a).

An edge with identical ends is called a *loop*. Edge *e8* in Figure 2.1(a) is a loop. An edge with distinct end points is called a *link*. A graph is *simple* if it has no self loops and no two edges join the same pair of vertices (e.g $u \rightarrow v \rightarrow u$). Two or more vertices are called *adjacent vertices* if they are connected by a common edge. Two or more edges are called *adjacent edges* if they are incident with a common vertex. We denote the number elements in V and E by



(a) A directed graph

(b) A weighted directed graph

Figure 2.1: Examples for directed unweighted and weighted graph

$|V|$ and $|E|$ respectively. The *out-degree* of a vertex is defined as the number of outgoing edges from the vertex. The *in-degree* of a vertex is defined as the number of incoming edges to the vertex. A simple graph in which each pair of vertices is joined using separate edges is called a *complete graph*. The *diameter* of a graph is defined as the greatest distance between any pair of vertices.

A graph $G(V, E)$ is a *bipartite graph* if the vertex set V can be partitioned into two disjoint subsets X and Y, such that each edge has one end in X and the other end in Y. The partition (X,Y) is called a *bipartition* of the graph G. An example of a bipartite graph is a movie-actor graph where the set X represents movies and the set Y represents actors and the edges are between an element $m \in X$ to an element $a \in Y$, denoting "a is acted in movie m". A graph is *k-partite*, if the vertex set can be partitioned into k disjoint subsets, such that no edge has both end points in the same subset.

A graph G_s is a *subgraph* of G if $V(G_s) \subseteq V(G)$ and $G(E_s) \subseteq E(G)$ and $D(G_s)$ is a restriction of $D(G)$. When $G_s \subseteq G$ and $G_s \neq G$, G_s is a *proper subgraph* of G and G is a proper supergraph of G_s. Let f:E$\rightarrow$ F be a function from a set E to a set F, so that the domain of f *dom* is given by *dom*(f)$\subseteq$E. If a set A is a subset of E (A$\subset$ E), then the restriction of f to A is given by the function f^A: A$\rightarrow$ F.

A *path* in a graph G(V,E) from vertex x_1 to x_n is a sequence of vertices $\{x_1,x_2,...,x_n\}$ such that (x_1,x_2), (x_2,x_3), ..., $(x_{(n-1)},x_n) \in$ E and each x_i are distinct. A *cycle* of a graph G(V,E) is a set $E' \subseteq E$ that forms a *path* such that the first and last vertex of the *path* are the same. A graph is called *cyclic* graph if it has at least one *path* which is a *cycle*. A graph is called *acyclic* if has no *path* which forms a *cycle*. A *tree* is a undirected, connected, acyclic graph. For two vertices u,v$\in$ V of a graph G(V,E), v is *reachable* from u if there is a *path* from u to v, otherwise v is *unreachable* from u.

A graph $G(V, E)$ is *connected* when there is a path between every pair of vertices in V. In

a *connected graph*, there are no unreachable vertices. A graph that is not connected is called a *disconnected graph*. A *spanning tree* T of an undirected, connected graph G is a subgraph that is a tree which includes all the vertices of G and number of edges in T is $|V| - 1$.

2.2 Well-known graph algorithms [31]

In this section we look at well known graph algorithms, which have applications in different areas of science [31] and forms a part of some of our benchmarks.

Breadth First Search (BFS) is one of the simplest algorithms for traversing a graph. Given a directed or undirected graph $G(V, E)$ and a distinguished source vertex s, BFS explores the edges of G to find every vertex that is reachable from s. It computes the *distance* (smallest number of edges) from s to each reachable vertex. For any vertex v reachable from s, a simple path from s to v corresponds to a shortest path from s to v in G, that is, a path containing the smallest number of edges. BFS algorithms are used in many applications such as the *Ford−Fulkerson* algorithm for computing the maximum flow in a network.

Depth First Search (DFS) is another algorithm for traversing the vertices in a graph, where it searches *deeper* in the graph whenever possible. DFS traverses the unexplored outgoing edges of the most recently discovered vertex v. Once all the outgoing edges of v are traversed, the search starts from the unexplored edges of the parent of the vertex v. As an example of its applications, DFS is used to find connected components of a graph .

Single Source Shortest Path (SSSP) algorithm [105] is used to find a path from a source vertex to every vertex in V, with minimum weight in a weighted, directed graph G(V,E). One popular SSSP algorithm is the *Bellman-Ford* Algorithm [3], which also finds negative cycles in a graph with negative weights. *Dijkstra's* algorithm [55] solves the SSSP problem on a weighted, directed graph G(V,E) for the case in which all the edge weights are nonnegative. SSSP algorithms are applied automatically to find the distance between two locations in systems like Google Maps and GPS (Global Positioning System).

Minimum Spanning Tree (MST) is a subset of the edges of a connected, weighted undirected graph that connects all the vertices together, without any cycles and has minimum possible total edge weight. Any undirected graph (not necessarily connected) has a *minimum spanning forest*, which is the union of the minimum spanning trees for its connected components. Well know algorithms for MST computation are *Prim's* [2] and *Kruskal's* [1] algorithms. *Boruvka's* MST algorithm [90] uses the *Union-Find* data structure to find the MST. MST has direct applications in design of networks including telecommunication networks and transport networks.

PageRank algorithm was developed by Larry Page and Sergey Brin in 1996 as part of a research on search engines in 1996 [61]. It is used to rank a Webgraph and is used to give

priority to each *url* or *web-page* in the World Wide Web (WWW). The pagerank value is used by search engines to rank web pages. The algorithm uses a *dampling factor* denoted by d, which is usually set to 0.85. Then pagerank pr of a vertex v (*url* in Webgraph) in a graph G(V,E) is defined by the equation given below.
$pr(v) = (1-d) \div N + d \times Sum(pr(u)/out\text{-}degree(u)), (\forall u, v \in V \bigwedge e(u \to v) \in E) \bigwedge N == |V|$.
Survey Propagation is an algorithm for finding an approximate solution to the Boolean Satisfiability Problem (SAT) [18] that takes a k-SAT formula as input, constructs a bipartite factor graph over its literals and constraints, propagates probabilities along its edges, and deletes a vertex when its associated probability is close enough to 0 or 1.
K-core of a graph is the largest subgraph with all the vertices having minimum degree K [13]. The K-core algorithm is used to study the clustering structure of social network graphs. This algorithm has applications in areas such as network analysis and computational biology.
Triangle Counting algorithm counts the number of triangles in a graph [94]. The triangle counting algorithm has applications in social network analysis.
Connected Components- Two vertices of an undirected graph are in the same connected component if and only if there is a path between them. A directed graph is *weakly connected* if it is connected without considering the direction of edges. A directed graph is *strongly connected* if there is a directed path between every pair of vertices. A **weakly connected component**(**WCC** or **CC**) of G(V,E) [90] is a set of vertices $V' \subseteq V$ such that there is an undirected path between every pair of vertices in V'. A **strongly connected component** (**SCC**) [91] of a directed graph G(V,E) is a set of vertices $V' \subseteq V$ such that there is a directed path between every pair of vertices in V'. **SCC** algorithms can detect cyclic dependencies in programs and communities in social network graphs.

There are many graph algorithms which have been proved to be *NP-Complete*. Usually, these algorithms are solved with heuristics. Some of the well know NP-Complete graph problems are discussed below.
Graph Coloring is a way of coloring the vertices and edges of a graph [43]. A coloring of graph such that no two adjacent vertices share the same color is called a *vertex coloring* of graph. Similarly, an *edge coloring* assigns a color to each edge so that no two adjacent edges share the same color. A coloring using at most k colors is called *k-coloring*. Graph coloring has applications in process scheduling, register allocation phase of a compiler and also in pattern matching.
A **Vertex Cover** of an undirected graph G(V,E) is a subset $V' \subseteq V$ satisfying the condition: if *e(u,v)* is an edge of G, then either $u \in V'$ or $v \in V'$ (or both) [23]. The size of a vertex cover is the number of vertices in it. The vertex cover problem is to find a vertex cover of minimum

size in a given undirected graph. The vertex cover problem has applications in hypergraphs.
Travelling Salesman Problem- In the Travelling Salesman problem, we are given a complete, weighted, undirected graph G (V, E) with positive weights for edges and we are required to find a tour of G with minimum cost, where every vertex is visited once [52]. This algorithm has applications in microchip manufacturing, DNA sequencing etc.
A **Clique** in an undirected graph G(V,E) is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E [20]. A clique is a complete subgraph of G. The size of a clique is the number of vertices it contains. The clique problem is defined as finding a clique of maximum size in the graph G. The clique problem has applications in social networks, bioinformatics and computational chemistry.

2.3 Graph storage formats

Figures 2.2(a) and 2.2(b) shows two graph storage schemes namely Adjacency Matrix format and Compress Sparse Row (CSR) Format respectively for the input graph given in Figure 2.3. The adjacenary matrix format has a storage overhead of O($|V|^2$). If the input graph is sparse, most of the entries in the input graph will be invalid (∞) and this results in suboptimal storage utilization.

	s	u	v	t	w
s	0	5	100	∞	∞
u	∞	0	10	80	115
v	∞	∞	0	40	∞
t	∞	∞	∞	0	18
w	∞	∞	∞	∞	0

(a) ADJACENCY MATRIX

	s	u	v	t	w		
index	0	2	5	6	7	7	
vertices	u	v	v	t	w	t	w
weight	5	100	10	80	110	40	18

(b) CSR FORMAT

Figure 2.2: Representation of Graph in Figure 2.3

In the CSR format, the graph storage use three one dimensional arrays. The edges of the graph object are stored using two arrays *vertices* and *weight*. The edges with a source vertex v are stored in adjacent locations starting from location *index[0]*. The *index*[x] ($0 \leq x < |V|$) stores the starting index in the *vertices* and *weight* arrays for edges with source vertex x. For example

- (s,u) and (s,v): two entries of vertices and weight arrays starting from *index[0]*(=0).

vertex (dist,pred)	s	u	v	t	w
initial	(0,-)	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$
itr1	(0,-)	(5,s)	(100,s)	$(\infty, -)$	$(\infty, -)$
itr2	(0,-)	(5,s)	(15,u)	(85,u)	(120,u)
itr3	(0,-)	(5,s)	(15,u)	(55,v)	(103,t)
itr4	(0,-)	(5,s)	(15,u)	(55,v)	(73,t)
final	(0,-)	(5,s)	(15,u)	(55,v)	(73,t)

Table 2.1. SSSP computation using Algorithm 1

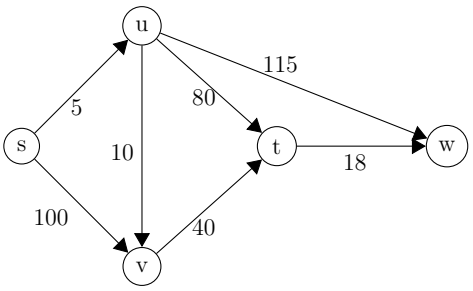


Figure 2.3. An input graph for SSSP algorithm

- (u,v),(u,t) and (u,w) : three entries of vertices and weight arrays starting from *index[1]*(=2).

This representation has a storage overhead of $|V|+1$ for the *index* array and $|E|$ for the *vertices* and the *weight* arrays. So, the total overhead is $|V| + 1 + 2 \times |E|$ or O$(V + E)$. This saves a lot of space for sparse graphs and most natural graphs are sparse.

Coordinate List (COO) format is another popular graph storage format. It stores a graph object as list of (src-vertex, dst-vertex, weight) tuples. The tuples are sorted by the ascending order of (src-vertex, dst-vertex) pair to have improved locality of access. This format also saves space and storage complexity is $3 \times |E|$ or $O(E)$ and is suitable for sparse graphs. The COO format of the graph in Figure 2.3 will have one entry for each edge as given below.
(s, u, 5), (s, v, 100), (u, v, 10), (u, t, 80), (u, w, 1150), (v, t, 40), (t, w, 18).

Algorithm 1: Bellman-Ford SSSP algorithm

```
1  for( i=0 to |V|-1 ){
2      distance[i]= ∞;
3      predecessor[i] = null;
4  }
5  distance[s] = 0;
6  for(  i = 1 to |V|-1 ){
7      for( each edge (u, v) with weight w ){
8          if( distance[u] + w <distance[v] ){
9              distance[v] = distance[u] + w;
10             predecessor[v] = u;
11         }
12     }
13 }
14 for( each edge (u, v) with weight w in edges ){
15     if( distance[u] + w <distance[v] ){
16         error "negative-weight cycle in Graph"
17         exit;
18     }
19 }
20 return distance[], predecessor[];
```

Here we discuss different ways of computing SSSP (using different algorithms) for a directed graph G(V,E). It shows that there are many possible implementations for a given algorithm with different storage and computational complexities.

2.3.1 Bellman-Ford algorithm

Figure 2.3 shows a sample input graph and Table 2.1 shows a possible SSSP computation for the graph using Algorithm 1. Assume the algorithm takes the edges in the order *(t→w), (u→w), (v→t), (u→t), (u→v), (s→u), (s→v)*. The table shows the vertices whose distances are reduced in blue color. It shows the predecessor in the shortest path along with the shortest distance. The vertex s is taken as the source vertex. The computational complexity of *Bellman-Ford* algorithm is $O(V \times E)$.

The above algorithm is purely sequential with one edge processed at a time. But the algorithm can be modified to a parallel version where multiple edges are processed at the same

time by different processors in a device.

2.3.2 Worklist based SSSP computation

iteration	active vertices	$s(dist, pred)$	$u(dist, pred)$	$v(dist, pred)$	$t(dist, pred)$	$w(dist, pred)$
initial	NIL	(0,-)	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$	$(\infty, -)$
itr1	s	(0,-)	(5,s)	(100,s)	$(\infty, -)$	$(\infty, -)$
itr2	v,u	(0,-)	(5,s)	(15,u)	(85,u)	(120,u)
itr3	w,t,v	(0,-)	(5,s)	(15,u)	(55,v)	(103,t)
itr4	w,t	(0,-)	(5,s)	(15,u)	(55,v)	(73,t)
itr5	NIL	(0,-)	(5,s)	(15,u)	(55,v)	(73,t)

Table 2.2. SSSP computation using Algorithm 2

Algorithm 2 shows the pseudo-code for a worklist based SSSP computation. The above algorithm requires that there is no negative cycle in the graph. The *distance* and *predecessor* of all vertices are initialized as in Algorithm 1 and *distance* of source vertex s is then made zero. Then computation proceeds using two worklists *current* and *next*, which can store a subset of vertices in the graph. The source vertex s is added to the worklist *current* (Line 7). The computation happens in the while loop (Line 8–20), which exits when the shortest distance to all the *reachable* vertices are computed. In the while loop, each vertex u in the worklist *current* is taken (Line 9) and its outgoing edges are processed in the inner loop (Lines 10–16).

For each edge $e(u \rightarrow v)$ taken for processing, the distance reduction is done as in Algorithm 1. If the distance of the vertex v is reduced, it is added to the worklist *next* (Line 14), which contains the vertices to be processed in the next iteration of the while loop. Once all the elements in *current* are processed, worklists *current* and *next* swapped (Line 18), making *size* of *next* zero. After the swap of *next* and *current* worklist, size of *current* will be number of elements added to *next* during the last execution of the loop in Lines 9–17. If no elements are added to *next*, size of *current* after swap operation will be zero, this is the fix-point for computation and the algorithm terminates. The above algorithm has a computation complexity of $O(V + E)$ which is much better than the *Bellman-Ford* algorithm complexity of $O(V \times E)$. But the algorithm has the the restriction that there should be no negative cycle in the graph, otherwise it will go into an infinite loop. Table 2.2 shows the iterations of Algorithm 2 for the input graph in Figure 2.3. The number of iterations of algorithm is the same as that of the *Bellman-Ford* algorithm($|V| - 1$), but the *Bellman-Ford* algorithm in each iteration traverses all the edges, while the worklist based algorithm traverses each edge only once over all the iterations, thus

Algorithm 2: Worklist based SSSP algorithm

```
1  for( each vertex v in V ){
2  |   distance[v]= ∞;
3  |   predecessor[v] = null;
4  }
5  distance[s] = 0;
6  Worklist<vertex >current,next;
7  current.add(s);
8  while( 1 ){
9  |   for(  each vertex u In current ){
10 |   |   for( each edge (u, v) with weight w ){
11 |   |   |   if( distance[u] + w <distance[v] ){
12 |   |   |   |   distance[v] = distance[u] + w;
13 |   |   |   |   predecessor[v] = u;
14 |   |   |   |   next.add(v);
15 |   |   |   }
16 |   |   }
17 |   }
18 |   swap(current,next);
19 |   if(current.size==0)break;
20 }
21 return distance[], predecessor[];
```

reducing the running time.

2.3.3 SSSP algorithm with CSR format

The SSSP algorithm for graph stored in CSR format is shown in Algorithm 3. The graph should not have negative cycles for correct execution of the algorithm. At the beginning of the while loop the *changed* variable is set to zero. When *distance* value of any vertex is reduced in the if statement block 13–17, variable *changed* is set to one (Line 16). The algorithm reaches a fix point when *distance* value of no vertex is reduced after the termination of for loop (Lines 8–19), which ensures Line 16 not executed. Hence variable *changed* is zero (the value set in Line 7) after processing all the edges and the program exits.

This is a naive implementation and the worst case running time is $O(V \times E)$ as the number of iterations in worst case can be $|V|-1$. Even though the *theoretical* running time is $O(V \times E)$, for all natural sparse graphs, running time is much lesser than this value and the algorithm terminates in very few iterations. The running time of each iteration of the `while` loop can be reduced by adding more vertex properties to the graph and processing only a subset of the

Algorithm 3: SSSP algorithm on CSR format Graph

```
1  for( i=0 to |V| − 1 ){
2  |  distance[i]= ∞;
3  |  predecessor[i] = null;
4  }
5  distance[s] = 0;
6  while( 1 ){
7  |  changed=0;
8  |  for( i=0 to |V| − 1 ){
9  |  |  start=index[i];
10 |  |  end=index[i+1]-1;
11 |  |  for( j=start to end ){
12 |  |  |  dst=vertices[j];
13 |  |  |  if( distance[i] + weight[dst] <distance[dst] ){
14 |  |  |  |  distance[dst] = distance[i] + weight[j];
15 |  |  |  |  predecessor[dst] = i;
16 |  |  |  |  changed=1;
17 |  |  |  }
18 |  |  }
19 |  }
20 |  if(changed==0)break;
21 }
22 return distance[], predecessor[];
```

edges. Such an implementation can be seen in Algorithm 20 in Section 3.1.1.

2.3.4 Δ-Stepping SSSP algorithm

In the Δ-Stepping SSSP Algorithm [69], vertices are ordered using a set of *worklists* called *buckets* representing priority ranges of Δ, where Δ is a positive value. The *bucket* $B[i]$ will have vertices whose current *distance* value is given by $(i-1) \times \Delta \leq distance < i \times \Delta$. The *buckets* are processed in an increasing order of index value i and a *bucket* $B[i]$ is processed only after bucket $B[i-1]$ is processed. Algorithm 4 shows the Δ-Stepping SSSP algorithm.

The function *Relax* takes as argument a vertex v and an `int` value x. If the current *distance* of v is greater than x, the vertex v is removed from current bucket $B[$ *distance(v)*$\div\Delta]$ (Line 4) and it is added to the bucket $B[x \div \Delta]$ (Line 5). Then the *distance* of vertex v is reduced to x (Line 6).

The *SSSP* algorithm works in the following way. Initially for each vertex v in the graph, two sets *heavy* and *light* are computed, where

Algorithm 4: Δ-Stepping SSSP algorithm

```
1  Bucket B;
2   Relax( vertex v, int x) {
3  if(  (x <distance(v) ) ){
4  |   B[ distance(v)÷ Δ]=B[distance(v)÷ Δ] \ v;
5  |   B[ x÷ Δ]=B[x÷ Δ] ∪ v;
6  |   distance(v)=x;
7  }
8  }
9  SSSP () {
10 for(  each v in V ){
11 |   Set heavy( v ) := { ( v, w ) in E : weight( v, w ) >Δ }
12 |   Set light ( v ) := { ( v, w ) in E : weight( v, w ) <= Δ }
13 |   distance ( v ) := INF // Unreached
14 }
15 relax( s , 0); // bucket zero will have source s.
16 i := 0
17 // Source vertex at distance 0
18 while(  NOT isEmpty(B) ){
19 |   Bucket S := φ;
20 |   while(  B [ i ] ≠ φ ){
21 |   |   Set Req := { ( w, distance( v ) + weight ( v,w )) : v in B [ i ] ⋀ ( v, w ) in light(
   |   |   v ) }//add light weight edges for relaxation
22 |   |   S := S ∪ B [ i ];//store all elements in B[i] to bucket S for Line 27
23 |   |   B [ i ] := φ;
24 |   |   foreach ( ( v,x ) in Req) Relax( v , x ) //relax. may add elements to B[i] again
25 |   }
26 |   //done with B[i].add heavy weight edge for relaxation
27 |   Req := { ( w,distance( v ) + weight ( v,w )) : v in S ⋀ ( v, w ) in heavy( v ) }
   |   foreach( ( v, x ) in Req) relax( v , x );//relax heavy weights.
28 |   i := i + 1
29 }
30  }
```

heavy(v)= (∀ (v,w) ∈ E) ⋀ (*weight (v,w)* >Δ)
light(v)= (∀ (v,w) ∈ E) ⋀ (*weight (v,w)* ≤ Δ)
Then the *distance* of all the vertices is made ∞ (Lines 10–14).

The algorithm starts by relaxing the *distance* value of source vertex *s* in Line 15 with a *distance* value of zero. This will add the source vertex to *bucket* zero (Line 5). Then the algorithm enters the while loop in Lines 18 to 29, processing *buckets* in an increasing order of index value *i*, starting from zero.

An important feature of the algorithm is that, once the processing of *bucket B[i]* is over, no more elements will be added to the bucket $B[i]$, when the buckets are processed with increasing values of index *i*. A *bucket* $B[i]$ is processed in the while loop (Lines 20 to 25). Algorithm terminates when all the *buckets* $B[i]$, i≥0 are empty. The performance of the algorithm depends on the input graph and the value of the parameter Δ, which is a positive value. For a Graph G(V,E) with random edge weights, maximum node degree d (0<d≤1) , the sequential Δ-stepping algorithm has a time complexity of $O(|V| + |E| + d \times P)$, where P is the maximum SSSP distance of the graph. So, this algorithm has running time which is linear in $|V|$ and $|E|$.

We have seen different ways of implementing SSSP algorithms. This is true for many graph algorithms. The complexity of the algorithms have also been discussed. The Δ- stepping algorithm has been proved to be the best for SSSP computation on single core and multi-core CPUs which have Multiple Instruction Multiple Data (MIMD) architecture. But this algorithm is not the best for machines which follow Single Instruction Multiple Data (SIMD) architecture, where all the threads execute the same instructions in synchronism but work on different data. For such architectures, the optimized Bellman-Ford variant is faster. But if the graph object has a high diameter (e.g., road network), a worklist based algorithm is faster on an SIMT (Single Instruction Multiple Thread, e.g., GPU) machine.

2.4 Parallel graph algorithms

Current generation computing devices have multiple cores inside them, and parallel algorithms running on many cores benefit from them. Most graph algorithms can be made to run in parallel. For example, the *Bellman-Ford* SSSP algorithm in Algorithm 1 can be made parallel by processing the edges in parallel, using separate threads. Results of a parallel execution should preserve *sequential consistency* which can be defined as: "the result of a parallel execution is the same as that of the operations performed by all the threads on all the devices being executed in some sequential order". Graph algorithms are *irregular*, where multiple threads may try to update the same vertex or edge properties. The irregularity of the graph algorithms depends on the run time parameters such as the graph structure and can not be handled using any

compile time analysis. In such cases the updation of the properties should be done using *atomic operations*, so as to preserve the serial consistency. When Algorithm 1 in Section 2.3.1 is made parallel by processing all the edges in parallel, code in Lines 8–11 should have atomic operations such as *atomicMIN* to reduce the distance of the vertex. Due to the irregular nature of the graph algorithm, the speedup obtained by parallel algorithms will not be linear as in regular algorithms, such as matrix multiplication.

Algorithm 5: Parallel Bellman-Ford SSSP algorithm

```
1  parallel for( each vertex v in V ){
2  |   distance[v]= ∞;
3  |   predecessor[v] = null;
4  }
5  distance[s] = 0;
6  parallel for( i = 1 to |V|-1 ){
7  |   parallel for( each edge (u, v) with weight w ){
8  |   |   atomic if( distance[u] + w <distance[v] ){
9  |   |   |   distance[v] = distance[u] + w;
10 |   |   |   predecessor[v] = u;
11 |   |   }
12 |   }
13 }
14 parallel for( each edge (u, v) with weight w in edges ){
15 |   if( distance[u] + w <distance[v] ){
16 |   |   error "negative-weight cycle in Graph"
17 |   |   exit;
18 |   }
19 }
20 return distance[], predecessor[];
```

Algorithm 5 shows the parallel version (pseudo code) of the Bellman-Ford SSSP algorithm. The code has `parallel for` where all the elements are processed in parallel. Lines 1-4 initialize *distance* and *predecessor* of each vertex in parallel. The `parallel for` loops in Lines 6-13 and Lines 7-12 process all the elements in parallel. Due to the irregular nature of the algorithm, the code enclosed in the `if` statement needs to be executed atomically (which is shown as `atomic if` operation in the pseudo code(Lines 8-11)). This happens as two threads may try to update the distance of a vertex v using edges p→v and u→v at the same time, and this needs to be serialized for correct output. In the implementation of the algorithm in a high level language, a programmer must use the atomic operations provided by the language. Speedup that can be achieved depends on the number of conflicting accesses between the threads in the parallel

Algorithm 6: Parallel SSSP algorithm on CSR format Graph

```
1  parallel for( i=0 to |V| − 1 ){
2  |  distance[i]= ∞;
3  |  predecessor[i] = null;
4  }
5  distance[s] = 0;
6  while( 1 ){
7  |  changed=0;
8  |  parallel for(  i=0 to |V| − 1 ){
9  |  |  start=index[i];
10 |  |  end=index[i+1]-1;
11 |  |  parallel for( j=start to end ){
12 |  |  |  dst=vertices[j];
13 |  |  |  atomic if( distance[i] + weight[dst] <distance[dst] ){
14 |  |  |  |  distance[dst] = distance[i] + weight[j];
15 |  |  |  |  predecessor[dst] = i;
16 |  |  |  |  changed=1;
17 |  |  |  }
18 |  |  }
19 |  }
20 |  if(changed==0)break;
21 }
22 return distance[], predecessor[];
```

region.

Similarly Algorithm 3 in Section 2.3.3 which computes SSSP on an input graph object stored in the CSR format can also be made parallel as shown in Algorithm 6. A parallel version of the worklist based SSSP computation is shown in Algorithm 7. Both these algorithms follow the same code pattern with `parallel for` and `atomic if` operations.

Parallel algorithms can be implemented using the `OpenMP` library in a multi-core CPU and by using the CUDA library in Nvidia-GPU.

2.5 Graph partitioning

When a graph algorithm is executed on a distributed system which requires *message passing* between machines, the graph object should be partitioned and distributed across all the devices involved in the computation. Execution of the graph algorithm may involve communication of graph object properties (like *distance* in SSSP) before and after a parallel computation step. So, the partitioning algorithm must ensure that there is less communication overhead and

Algorithm 7: Parallel Worklist based SSSP algorithm

```
1  parallel for( each vertex v in V ){
2  |  distance[v]= ∞;
3  |  predecessor[v] = null;
4  }
5  distance[s] = 0;
6  Worklist<vertex >current,next;
7  current.add(s);
8  while( 1 ){
9  |  parallel for( each vertex u In current ){
10 |  |  parallel for( each edge (u, v) with weight w ){
11 |  |  |  atomic if( distance[u] + w <distance[v] ){
12 |  |  |  |  distance[v] = distance[u] + w;
13 |  |  |  |  predecessor[v] = u;
14 |  |  |  |  next.add(v);
15 |  |  |  }
16 |  |  }
17 |  }
18 |  swap(current,next);
19 |  if(current.size==0)break;
20 }
21 return distance[], predecessor[];
```

proper work balance across nodes. Large-scale graphs are very sparse and follow the power-law distribution. Such graphs are difficult to partition [8, 4] and popular frameworks have used random or hashed partitioning strategy. There are two type of graph partitioning, *vertex-cut* and *edge-cut*.

In a *vertex-cut* partitioning every edge is assigned one of the machines involved in the computation. The number of edges in each machine is almost the same.In this partitioning model, two or more edges with the same source vertex (e.g $u \rightarrow v$ and $u \rightarrow w$) may reside on multiple machines. So, one of the machines is taken as the *master* node of the vertex. The *master* node holds the most recent values of the vertex. This partitioning strategy gives work balance but can result in more communication volume as edges with the same source vertex reside on two or more devices.

In the *edge-cut* partitioning strategy, a vertex v is assigned a machine along with the edges with v as the source vertex. This partitioning strategy will have less communication overhead, but the ideal work balance achieved by *vertex-cut* partitioning will not be achieved.

2.6 Dynamic graph algorithms

Some algorithms make changes in the graph topology at run time by addition and/or deletion of vertices and/or edges. Such algorithms are called *dynamic graph algorithms*. A dynamic graph algorithm is said to be fully dynamic if it has both insertion and deletion of edges or vertices. It is said to be partially dynamic if it has only one type of update, either insertion or deletion. A dynamic graph algorithm is called incremental if only insertions are allowed, and it is called a decremental algorithm if only deletions are allowed [39].

In an incremental SSSP algorithm [81] where edges are added with nonnegative weights, the shortest distance from the source vertex can possibly only decrease with the addition of edges. So, instead of computing SSSP from scratch, it can be computed starting from the current shortest distance value. The same argument applies to incremental variants of BFS and Connected Components also.

Algorithm 8: Incremental SSSP algorithm

```
1  SSSP(graph,distance,predecessor) {
2  |  //compute SSSP
3  }
4  AddEdges(graph) {
5  |  read edges;
6  |  add edges to the graph;
7  }
8  Incremental-SSSP(graph,distance,predecessor) {
9  |  parallel for( each vertex v in V ){
10 |  |  distance[v]= ∞;
11 |  |  predecessor[v] = null;
12 |  }
13 |  SSSP(graph,distance,predecessor);
14 |  AddEdges(graph);
15 |  SSSP(graph,distance,predecessor);
16 }
```

Algorithm 8 shows the pseudo code for incremental SSSP computation. The SSSP function (Lines 1-3) computes SSSP using any one of the algorithm mentioned before (e.g., Lines 5-20, Algorithm 7). The initialization of distance and predecessor is done using a `parallel for` in Lines 9-12. Then SSSP is computed (Line 13). The *AddEdges()* function (Lines 4-7) add new edges to the graph. After the initial SSSP computation, *AddEdges()* is called (Line 15). Then, SSSP is computed from the current distance values by calling SSSP() again (Line 15), without resetting distance and predecessor of each vertex and without computing SSSP from scratch.

Dynamic graph algorithms are important because the topology of real life graphs changes over time, and only some properties need to be recomputed (e.g., rank of a webpage, shortest path in road networks, etc.).

2.7 Mesh algorithms

Mesh generation algorithms are used in areas such as computational geometry. Meshes can be mesh of triangles, quadrilaterals etc. Mesh generation is also called grid generation. In computer simulations, an algorithm may begin with a set of points in a d-dimensional space ($d \geq 2$) and generate a mesh, which satisfies some constraints. Meshes can be considered as special types of graphs where there is a relationship between edges and vertices. If it is a mesh of triangles, the relationship between edges and vertices is that, an edge will be a part of one triangle (boundary edge in a mesh) or two triangles (edge not belonging to the boundary of mesh). It is possible to view meshes as graphs with such constraints and write graph algorithms to create and process such meshes. Two popular mesh algorithms are described below.

Algorithm 9: Delaunay Triangulation pseudo code

```
1  DT ( ) {
2      Mesh mesh;
3      worklist wl;
4      initialize mesh;
5      add all points to worklist wl;
6      for( each point p1 in wl ){
7          Worklist cav;
8          Point p2;
9          Triangle tr1,tr2;
10         p2=the closest point of p1 in mesh;
11         tr1= triangle with p2 as one of its point;
12         tr2= triangle which contains p1 in its circumcircle;
13         cav= all neighboring triangle of tr2 whose circumcircle contains p1;
14         retriangulate cav;
15     }
16 }
```

2.7.1 Delaunay Triangulation

Delaunay triangulation (DT) produces a mesh of triangles by triangulation of a set of 2-Dimensional points such that the circumcircle of any triangle in the mesh does not contain any other points. The algorithm takes as input a set of 2-Dimensional points contained inside a big surrounding triangle and builds the delaunay mesh by inserting a new point and retrian-

gulating the affected portions of the mesh. The output is a mesh which satisfies the delaunay triangulation condition and the set of vertices of the mesh is the set of input point.

One possible implementation of DT [98] is given in Algorithm 9.

For the above algorithm, points can be taken in any order and all orders will lead to to a valid mesh, where the circumcircle of all triangles contains no other points.

2.7.2 Delaunay Mesh Refinement(DMR)

Algorithm 10: DMR algorithm pseudo code

```
1  DMR ( ) {
2      Mesh mesh;
3      worklist bad;
4      initialize mesh;
5      for( each triangle t in mesh ){
6          if(t is a bad triangle) add t to bad;
7      }
8      for( eah triangle t in bad ){
9          if( t is not deleted ){
10             worklist cav,newtria;
11             cav= cavity(t);
12             delete triangles in cav from mesh;
13             retriangulate cav;
14             add new triangles to mesh and newtria;
15             for( each p in newtria ){
16                 if(p is bad triangle) add p to bad;
17             }
18             delete t from mesh;
19         }
20     }
21 }
```

A DMR algorithm [26] takes a delaunay triangulated mesh and refines it such that no triangle has an angle less than 30 degrees and the circumcircle of each triangle contains no other points. The algorithm takes an input Delaunay mesh and produces a refined mesh by retriangulating the portions of the mesh where there are triangles with angle less than 30 degrees (called *bad triangles*). Pseudo code of the DMR is shown in Algorithm 10.

In the DMR algorithm, an initial *worklist* (*bad*) which contains all the triangles which have one or more angles with degree less than 30. In a DMR implementation based on worklist, in each step/iteration a bad triangle t is taken from the worklist. The *cavity* of the triangle t, is a set of triangles affected by the bad triangle t. The cavity (*cav*) is retriangulated. In

the retriangulation, all the triangles in cavity are deleted. Then a new point is inserted at the circumcenter of t or at the middle of a boundary edge, if the cavity contains a triangle at the boundary of the mesh. New triangles are created by adding edges from each point in the cavity to the new inserted point. The newly created triangles are checked and they are added to the worklist if found to be bad. The DMR algorithm is used to model objects and terrains.

2.7.3 Morph algorithms

An algorithm is called a *morph algorithm* [77] if it modifies its neighborhood by adding or deleting vertices and edges. It may also updating values associated with the vertices and edges. An algorithm is a *cautious morph algorithm*, if a thread or process gets a lock on all the elements which it is going to modify, before modifying them. Morph algorithms are also dynamic graph algorithms, where it changes the structure of the graph object. The DMR algorithm has a *cautious morph* implementation.

2.8 Graph classification based on its properties

Graphs have properties such as diameter, outdegree, indegree, size etc. Graphs can be classified into different categories based on these properties. There are public graphs like road networks which store the map of roads, and social network graphs which show the connectivity relationship between people etc. Graph classes have different values for properties mentioned above. For example, road networks have high diameters, and social graphs have low diameters, etc. We look at different graph classes and their properties. It is important to look at these graph classes as the performance of an algorithm on a device may also depend on these graph properties (e.g., low and high diameter of social and road networks respectively).

2.8.1 Road networks

A road network can be represented as a graph where a vertex represents the junction of two or more roads and the edges represent the roads connecting the junctions. The diameter of a graph is defined as the greatest distance between any pair of vertices. Road network graphs have very high diameter. Further, vertices in a road network (junctions) have small *out-degree*. Road networks are used by GPS and Google Maps for different applications such as shortest path, optimal path (considering current traffic) computations. The difference between the smallest and the highest degree in road a network graph is small.

2.8.2 Random graphs

A random graph is created using a set of isolated vertices and adding edges between them at random. Different random graph models give graphs with different probability distributions. The ErdősRényi model [35] assigns equal probability to all graphs with exactly E edges V vertices. For example, for G(3,2) (which has graphs with 3 vertices and 2 edges), where V = $\{V_0, V_1, V_2\}$, possible edges are (V_0-V_1), (V_0-V_2), (V_1-V_2). Number of subgraphs possible for G(3,2) is three and all these graphs have an equal probability of 1/3 when a graph is generated using the ErdősRényi model. Random graphs have a small *diameter*. The maximum degree of a vertex is higher than that of a road network. Random graphs need not be fully connected. The difference between the smallest and the highest degree of vertices in a random graph is small.

2.8.3 Real World graphs

Real world graphs of social networks such as Facebook, Twitter etc., can be weighted (the number of messages between two people (vertices)) and there could be multiple edges between the same pair of vertices. Such graphs could be unipartite like people in a closed community group, bipartite like a *movie-actor* database and also possibly multipartite. In a multipartite graph there are multiple classes of vertices and edges are drawn between vertices of different classes. Such real world graphs have a heavy-tailed distribution with very few vertices having a very large degree (outdegree or indegree) and others having very low degrees. As an example, in the Twitter network, when celebrities are followed by a large number of people, while others are followed only by their close friends. One famous heavy-tailed distribution is the power-law distribution, and social graphs follow this distribution. Two variables x and y are related by power-law if
$y(x) = Ax^{-\gamma}$, where
A and γ are positive constants and γ is called the power-law exponent.

A random variable is x distributed according to power-law if its probability density function is given by
$p(x) = Ax^{-\gamma}, \gamma > 1$
The degree distribution of social network graphs follow power-law as given below. Social network graphs have a small diameter which is also called as the *small-world phenomenon* [12]. Real world graphs have a community structure with vertices forming groups and groups forming within groups.

2.8.4 Recursive Matrix(R-MAT) model graphs

Social graphs or real world graphs follow the power-law distribution. R-MAT graphs are graphs which follow the power-law distribution and can be created manually [48]. The basic algorithm used by a R-MAT graph generator is to recursively subdivide the adjacency matrix of the graph into four equal-sized partitions, and distribute edges within these partitions with unequal probabilities. The adjacency matrix will be initially empty. Then edges are inserted in to the matrix one by one. Each edge chooses one of the four partitions with probabilities a, b, c, d respectively (a + b + c + d = 1). The chosen partition is again subdivided into four smaller partitions, and the procedure is repeated until we reach a simple cell (i, j) of the matrix where $0 \leq i, j < N$. This is the cell of the adjacency matrix occupied by the edge. There can be duplicate edges (ie., edges which fall into the same cell in the adjacency matrix).

2.8.5 Large-Scale graphs

Large-scale graphs are graphs of very big size. These large-scale graphs cannot be processed on a single machine, and so processing is done on a distributed system or computer cluster. R-MAT graphs can imitate the large-scale graphs and can be used to create large-scale graphs. A lagre-scale graphs can have trillions of edges.

2.8.6 Hypergraphs

A hypergraph [19] is a generalization of a graph in which an edge can join any number of vertices, not necessarily two. A hypergraph $G(V, E)$ is a graph where `V` is the set of vertices, and `E` is a set of non-empty subsets of `V` called hyper-edges or edges. A k-uniform hypergraph is a hypergraph such that all its hyper edges have size k. A 2-uniform hypergraph is a graph. A hypergraph has applications in combinatorial optimization, game theory, and in several fields of computer science such as machine learning, databases and data mining.

2.8.7 Webgraphs

The webgraph shows links between the pages of the World Wide Web (WWW). Webgraph is a directed graph, where vertices correspond to the pages in the WWW, and there is an edge $e(u \rightarrow v)$ if there is a hyperlink to page v in page u.

2.9 Parallel computing devices

2.9.1 Multi-core CPU

A multi-core CPU is a single computing device with two or more independent processing units or *cores*, with a shared volatile memory. `OpenMP` library [32] is the most popular tool to run parallel codes on multi-core CPU. Multi-core CPUs follow Multiple Instruction Multiple Data(MIMD) model of execution, with shared memory between cores. Algorithm 11 shows the *C++* code for adding two matrices *d* and *e*, and storing the result in the matrix *f*. The `for` loop in Lines 14 to 16 does the matrix addition. The for loop is made parallel by the `OpenMP parallel for` *pragma* on Line 13, which creates 24 threads.

Algorithm 11: Parallel Matrix Addition using `OpenMP` on multi-core CPU

```
1  #include < stdio.h >
2  #include < stdlib.h >
3  #include < omp.h >
4  void readMatrix(int *arr, int n, int m){
5      for(int i=0;i<n;i++)
6          for(int j=0;j<m;j++)scanf("%d",&arr[i * m + j]);
7  }
8  void main(int argc, char *argv[]){
9  int tid,i,j,rows=256,cols=256;
10 int d[rows][cols],e[rows][cols],f[rows][cols];
11 readMatrix(d,rows,cols);//read first matrix
12 readMatrix(e,rows,cols);//read second matrix
13 #pragma omp parallel for num_threads(24)
14 for( int i=0;i<row*col;i++ ){
15 |   f[i]=d[i]+e[i];
16 }
17 printf("Values of Resultant Matrix C are as follows:");
18 for(i=0;i<rows;i++)
19     for(j=0;j<cols;j++) printf("Value of C[%d][%d]=%d",i,j,c[i][j]);
20 }
```

2.9.2 Nvidia-GPU

Nvidia is a commercial company that developes GPUs for gaming and general purpose computing, and also System on Chip units(SOCs) for mobile computing and automative units. It launched their first GPU in 1999 named *GeForce 256 SDR*.

The Nvidia GPU architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). Each SM consist of many Streaming Processors(SPs) (See Figure 2.4). As an example Nvidia-K40c GPU consist of 2880 Streaming Processors (SPs) which are divided to 15 SMs with each SM having 192 cores. It has 12 GB global memory. Each SM has a shared memory and access latency to shared memory is 100× slower than that of the global memory. It also has a constant, texture memory and thousands of registers to be used among threads running on the SM. Size of texture, constant and shared memory is of the order of KBs.

In GPU programming, CPU is called the *host* and GPU is called the *device*. Using CUDA library of Nvidia [30], programmers can write GPU programs called as *kernels*, which can have thousands of threads and it is invoked from the *host*. Any function which is called from a *kernel* code is called a *device* function. *Kernel* and *device* function definition starts with the keyword `__global__` and `__device__` respectively in CUDA. CUDA extends *C++* with additional keywords and functions specific to GPU. When a CUDA *kernel* is invoked from the *host* (CPU), blocks of the kernel are distributed to streaming multiprocessors (SMs). A global variable which is allocated on the GPU is also preceded by `__device__` keyword in the declaration statement (e.g., __device__ int changed;). The threads of a thread block execute concurrently on one SM, and multiple thread blocks can execute concurrently on an SM. As thread blocks terminate, new blocks are launched on the vacated SM. A thread block cannot migrate from one SM to another SM.

A multiprocessor (SM) can run hundreds of threads concurrently. The multiprocessor follows the Single Instruction Multiple Thread (SIMT) architecture. The threads are issued in order and there is no branch prediction and speculative execution.

2.9.2.1 SIMT architecture of Nvidia-GPU

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. When a multiprocessor is given one or more thread blocks to execute, it partitions them into *warps* and each *warp* gets scheduled by a warp scheduler for execution. Each *warp* contains threads of consecutive, increasing thread IDs with the first *warp* containing thread 0. A *warp* executes one common instruction at a time, and full efficiency is realized when all 32 threads of a *warp* follow same execution path. If the threads of a *warp* diverge due to conditional statements in the code, the *warp* serially executes each branch path taken and disables threads that are not on that path. When all the paths are complete, the threads come back to the same execution path. Branch divergence occurs only within a *warp* and each *warp* executes independently. The SIMT architecture is similar to SIMD (Single Instruction, Multiple Data) vector organizations in that a single instruction controls multiple processing

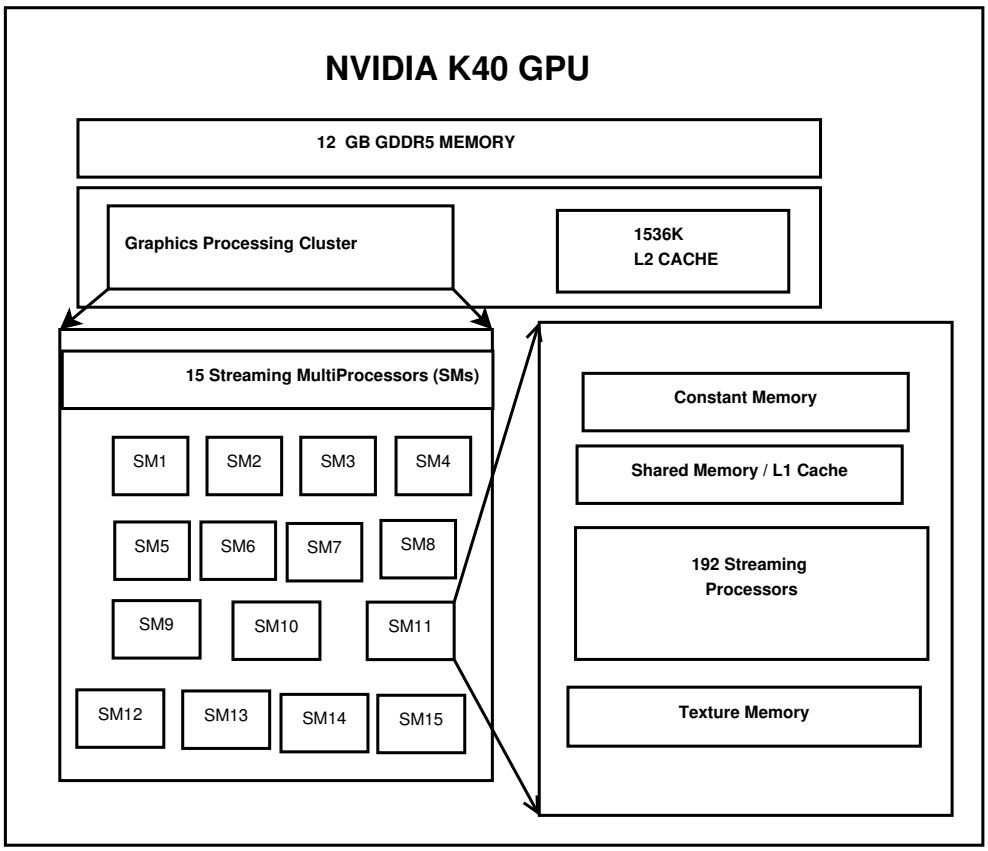


Figure 2.4: Nvidia-GPU architecture

elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For program correctness, SIMT architecture of GPU can be ignored. Execution time improvements can be achieved by taking care of the warp divergence. The *nvcc* compiler of CUDA is used to compile GPU codes.

2.9.2.2 Example-matrix addition on GPU

The Algorithm 12 shows the *CUDA* code for matrix addition on Nvidia-GPUs. GPU and CPU have separate memory space called *device* memory and *host* memory respectively. So, space for matrices is allocated on the CPU (Lines 15–17) in variables a_h,b_h and c_h using the *malloc()* function. The GPU matrices are allocated in the variables a_d,b_d and c_d (Lines 19–21) using the *cudaMalloc()* function of the *nvcc* library. Input matrices are then read to arrays on the *host* memory a_h and b_h and then copied to *device* memory arrays a_d and b_d respectively

Algorithm 12: Matrix addition using CUDA on Nvidia-GPU

```
#include < iostream >
#include < cuda.h >
__global__ void MatrixAdd(int *A,int *B,int *C){
int i = blockIdx.x*blockDim.x + threadIdx.x;
C[i]=A[i]+B[i];
}
void readMatrix(int *arr, int n, int m){
    for(int i=0;i<n;i++)
        for(int j=0;j<m;j++) scanf("%d",&arr[i * m + j]);
}
int main(){
int rows=256,cols=256, i, j, index;
int N=rows*cols;
// allocate arrays on host(CPU)
a_h = (int *)malloc(sizeof(int)*N);
b_h = (int *)malloc(sizeof(int)*N);
c_h = (int *)malloc(sizeof(int)*N);
// allocate arrays on device(GPU)
cudaMalloc((void **)&a_d,N*sizeof(int));
cudaMalloc((void **)&b_d,N*sizeof(int));
cudaMalloc((void **)&c_d,N*sizeof(int));
readMatrix(a_h,rows,cols);//read first Matirx
readMatrix(b_h,rows,cols);//read second Matrix
cudaMemcpy(a_d,a_h,N*sizeof(int),cudaMemcpyHostToDevice);//copy a_h to device
cudaMemcpy(b_d,b_h,N*sizeof(int),cudaMemcpyHostToDevice);//copy b_h to device
MatrixAdd<<< 256,256>>>(a_d,b_d,c_d);//compute on device
cudaDeviceSynchronize();
cudaMemcpy(c_h,c_d,N*sizeof(int),cudaMemcpyDeviceToHost);//copy result to host
for( j=0;j<rows;j++ ){
    for( i=0;i<cols;i++ ){
        index = j*rows+i;
        printf("A + B = C: %d %d %d + %d = %d",i,j,a_h[index],b_h[index],c_h[index]);
    }
}
}
```

using the *cudaMemcpy()* function (Lines 22–25).

Then the *CUDA* kernel *MatrixAdd()* is called, which does the matrix addition on GPU. The number of *thread_blocks* and *threads_per_block* are specified before the argument list of the *MatrixAdd()* function in CUDA syntax. These variables can have three dimensional values in x, y, z. The *MatrixAdd()* uses just one dimension (x, with both values set to 256). Then the kernel will have 256×256 threads (256 *thread_blocks* and 256 *threads_per_block*) and each thread computes one element in the resultant matrix *c_d*. This matrix is then copied to *host* (CPU) memory matrix *c_h* and the result is then printed. The value of the variables used in Line 4 are $0 \leq blockIdx.x < 256$, $blockDim.x == 256$ and $0 \leq threadIdx.x < 256$.

2.10 Computer clusters or distributed systems

A computer cluster consists of a set of connected computers that work together so that they can be viewed as a single system. Machines in a cluster are connected to each other through fast Ethernet networks, with each machine running its own instance of an operating system. In most cases all the machines in a cluster will have the same hardware. Communication between machines are done using software libraries such as MPI or OpenMPI. Computer clusters are mandatory for large-scale graph processing where a graph object cannot be stored on a single machine. Large-scale graphs are partitioned and distributed across machines in the cluster.

A CPU cluster consists of a set of machines, with each machine having one or more multi-core CPU. A GPU cluster consists of a set of machines connected using network switches with each machine having a CPU that runs the operating system and one or more GPU device. Each node in the GPU cluster has a 12 core CPU and a GPU. We used this for heterogeneous execution where one node uses both i) CPU and GPU or ii) Both CPU and GPU. An heterogeneous cluster with CPU and GPU is a distributed system with each node having i) CPU and GPU or ii) CPU.

2.11 MPI sample program

The MPI programming model [38] assumes a distributed memory model with each device having its on private memory. If there are N processes executing on P machines ($P \leq N$), each machine will have its own private memory. Communication between processes is performed through *message passing*. The basic primitives in MPI for message passing is *MPI_Send()* for sending data to a remote machine and *MPI_Recv()* for receiving data from a remote machine. These functions takes as argument data, type of data and size of data to be send, *message-id* and *process-id* to identify the process on the remote node. *MPI_Isend()* is a non-blocking version of *MPI_Send()* and takes similar arguments. *MPI_Recv()* function is for receiving data

Algorithm 13: MPI sample program

```
1  #include<mpi.h>
2  int main(int argc,char *argv[]){
3  int rank,size,number;
4  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5  MPI_Comm_size(MPI_COMM_WORLD, &size);
6  if( rank == 0 ){
7  |  number = 10;
8  |  MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
9  }
10 if( rank == 1 ){
11 |  MPI_Recv(&number, 1, MPI_INT, 0, 0,
   |  MPI_COMM_WORLD,MPI_STATUS_IGNORE);
12 |  printf(Process 1 received number %d from process 0, number);
13 }
14 }
```

from remote nodes and have arguments similar to *MPI_Send()* and this function call blocks until data is received. The status of *send* and *recv* operations can be found using variables of type *MPI_Status* specified as argument to the function.

Algorithm 13 shows the code for a distributed system with two machines, with one process running on each machine. The process *ranks* will be 0 and 1. The process with *rank* 0 sends an integer value 10 stored in the variable *number* to the process with *rank* 1, using a *message-id* 0. The process with *rank* 1 waits for a message from process with *rank* 0 with *message-id* 0. After the receive operation, the value is printed by process 1.

The above code is for CPU machines. When it is a distributed system with GPUs, there is a requirement for communication between GPUs on two remote nodes. The MPI library does not allow GPU locations as arguments to any of the library calls. The send operation is performed by copying data from the *device* memory to the *host* memory and then initiating an *MPI_Send()*. The receive operation, receives the data in the *host* memory, and then copies the data from the *host* memory to the *device* memory. The OpenMPI library [41] has a *cuda-aware-support* and it accepts *device* locations as arguments for *send* and *receive* operations, which eliminates code for copying data between *device* and *host*. OpenMPI handles this copy operation automatically.

2.12 Multi-GPU machine

Nvidia is a commercial company that developes GPUs for gaming and general purpose computing, and also System on Chip units(SOCs) for mobile computing and automative units. A multi-GPU machine is a single machine with more than one GPU and a single mutlticore-CPU. Multi-GPU machines can also be used for large-scale graph processing by partitioning the graph and storing each subgraph on a separate GPU device on the machine. Programs can be written in two ways :-

- One process, separate thread for each GPU device and this requires `OpenMP` library or Pthreads library.
- Separate process for each GPU and this requires MPI or OpenMPI library for communication between processes.

When a graph algorithm is run on multiple GPUs, the input graph is read into the CPU memory, and then it is partitioned to subgraphs and distributed across multiple GPUs devices. When only one process is used, expert coding is needed to make this efficient. In the case of multiple processes, the first process can read the graph into the CPU memory shared by all the processes [89], and after this, distribution and processing of each subgraph is performed by other processes.

2.13 Execution models for distributed systems

When an algorithm is executed on a distributed system with the graph partitioned into subgraphs and stored across multiple devices (Section 2.5) and the output should preserve *sequential consistency* as discussed in Section 2.4. To achieve this, there should be communication across devices which store subgraphs. Some of the popular and standard programming models for distributed execution algorithms are described below.

2.13.1 Bulk Synchronous Parallel (BSP) model

In the BSP [95] model program execution is divided into three parts.

- *Computation* which happens concurrently on all the devices participating in program execution. In our case this will be computation on subgraphs on each node.
- *Communication*, where the processes participating in execution exchange data with each other. For a graph algorithm the data communicated will be graph properties, like *distance* of vertices in SSSP computation.

Algorithm 14: Distributed SSSP algorithm in BSP model

```
1  parallel for( i=0 to |V_local+remote| − 1 ){
2      distance[i]= ∞;
3      predecessor[i] = null;
4  }
5  distance[s] = 0;
6  while( 1 ){
7      changed=0;
8      parallel for( i=0 to |V_local| − 1 ){
9          start=index[i];
10         end=index[i+1]-1;
11         parallel for( j=start to end ){
12             dst=vertices[j];
13             atomic if( distance[i] + weight[dst] <distance[dst] ){
14                 distance[dst] = distance[i] + weight[j];
15                 predecessor[dst] = i;
16                 changed=1;
17             }
18         }
19     }
20     barrier();
21     synchronize distance and predecessor value of remote-node with master-node;
22     barrier();
23     synchronize value of changed across all nodes;
24     barrier();
25     if(changed==0)break;
26 }
27 return distance[], predecessor[];
```

- *Synchronization*, where all the processes participating in the computation join at the *synchronization* point, before proceeding to the next *computation/communication*.

Algorithm 14 shows the pseudo code for an SSSP in BSP model, for a graph stored in CSR format. This is the modified version of Algorithm 3 and the algorithm assumes an edge-cut partitioning. First, distance and predecessor of each local and remote vertex of the graph is initialized (Lines 1-4). Then, the distances of vertices are processed by reducing all the outgoing edges of local vertices in the subgraph, in parallel on all the machines (Lines 8-19). After the distance is reduced, modified distance value of remote vertices are synchronized with the master node by taking the minimum value across all the devices. The program exits when *changed* variable is zero across all the devices after parallel computation.

2.13.2 Asynchronous execution model

An asynchronous execution model also has the steps *computation* which happens concurrently on all the devices and *communication* or *message passing*. But there will be no *synchronization* point in the program code. The processes send the data which needs to be communicated to the devices as and when they arrive and at the receiving side, data is processed as and when it arrives.

Algorithm 15: Distributed SSSP computation in asynchronous model

```
1  Update(v,dist,pred) {
2  |   distance[v]=dist; predecessor[v]=pred;
3  }
4  SSSP() {
5  |   parallel for( all v neighbours of s ){
6  |   |   predecessor[v]=s;
7  |   |   sendmsg Update(v,0,s);
8  |   }
9  |   while( Message Update(v,dist,pred ){
10 |   |   if( distance[v] >dst ){
11 |   |   |   distance[v]=dist;
12 |   |   |   predecessor[v]=pred;
13 |   |   |   parallel for( all u neighbours of v ){
14 |   |   |   |   sendmsg Update(u,distance[v],predecessor[v]);
15 |   |   |   }
16 |   |   }
17 |   }
18 }
```

Algorithm 15 shows a distributed asynchronous SSSP computation. The algorithm first initializes distance and predecessor values, followed by distance of source vertex being made zero. Then an update message is sent to all the neighbours of the source vertex (Lines 5-8). The Update function with arguments v,*dist* and *pred*, reduces the distance of the vertex v to *dist* and sets predecessor to *pred* (Lines 1-3). Following this, the computation happens in the while loop as long as each node receives an Update message (Lines 9-17). When an Update message is received, the received values are checked. If the distance of the vertex v is greater than the received value *dist*, distance and predecessor of v are updated and Update messages are sent to all the neighbours of vertex v (Lines 10-16). Computation finishes when no more messages are sent.

In the *BSP* model of execution as there is a *synchronization* step, multiple *messages* to the same vertex on a particular machine can be *aggregated*, which reduces the communication volume. But if there is any imbalance in *computation* time across the devices, the processes which finish first will be idle till all other process complete their computation. As there is no *synchronization* in the *asynchronous execution* model, the processes will not be idle as long as there is data to be processed, but the *aggregation* of messages may not be possible as data is sent to the devices as and when it arrives, and this results in more communication volume.

2.13.3 Gather-Apply-Scatter (GAS) model

The GAS model consist of three steps:

- *Gather* phase, where data on adjacent vertices and edges is *gathered* using a commutative and associative function.
- *Apply* phase, where an *apply* function does computation on active vertices or edges. and
- *Scatter* phase, where new values of updated vertices and edges produced by the *apply* function are scattered to the remote nodes.

Algorithm 16: SSSP in Gather-Apply-Scatter Model (PowerGraph)

```
1  gather(D_u, D_(u,v) , D_v ):
2      return D_v + D_(v,u)
3  sum(a, b):
4      return min(a, b)
5  apply(D_u , new_dist):
6      D_u = new_dist
7  scatter(D_u ,D_(u,v) ,D_v ):
8      if( decreased(D_u ) ){
9      |  Activate(v)
10     |  return D_u + D_(u, v)
11 }
```

Algorithm 16 shows SSSP computation in the GAS model. In the *gather* phase, the gather function is run in parallel for all the neighbours of the vertex D_u. The *gather()* function uses the weight of edge ($D_{u,v}$) and the distance of the adjacent vertex D_v, and modifies the value of the vertex D_u using the commutative and associative function *sum* which is *min* for the SSSP algorithm (the function name is `sum` but its usage is '+', according to the syntax adopted by

PowerGraph). The *apply* function modifies the value of D_u. The *dist* value of the vertex is D_u is already modified by the *gather* function which uses *min* as the *sum* operator, and hence the *apply* function is just an assignment. In the *scatter* phase, if the value of D_u has been modified, the vertices D_v are activated for all the edges with D_u as the source vertex ($D_{(u,v)}$). The new values are scattered through the edges $D_{(u,v)}$.

In the asynchronous execution model, communication volume will be high, but data gets processed as and when it arrives. In the BSP model, communication volume is less, but computation imbalance between processes can lead to long running times due to the *barrier()* (synchronization) called on all the processes. The GAS model can be adopted to both BSP and asynchronous models, depending on the necessity of a barrier after the apply phase of the computation and on how the data should be communicated. For example, the PowerGraph framework supports both synchronous and asynchronous computations over its GAS model.

Chapter 3

Related Works

3.1 Frameworks for multi-core CPU machines

3.1.1 Green-Marl

Green-Marl [53] is a graph DSL for implementing parallel graph algorithms on multi-core CPUs. The Green-Marl compiler generates `OpenMP` based parallel code from the DSL code for multi-core CPU with shared memory.

It has five primitive data types `Int`, `Long`, `Bool`, `Float` and `Double`. Green-Marl has two graph data types `DGraph` (for directed graph) `UGraph` (for undirected graph) along with two data types `Node` and `Edge` to represent vertices and edges in the graph (respectively). A progammer can use `Graph` instead of `DGraph` as `Graph` is an alias for `DGraph`. Node and edge properties can be attached to vertices and edges of a graph object and specified using `Node_Prop` and `Edge_Prop` respectively. Green-Marl has three types of worklists data types namely `Set`, `Order` and `Sequence`. These data types may contain a set of vertices in a graph or neighbours of a given vertex etc. Elements in a `Set` are unique but not ordered. Elements in an `Order` are unique and ordered. Elements in a `Sequence` are ordered but not unique. Green-Marl has the usual operators, such as `While`, `Do-While`, `for`, `If`, `If-Else` etc., for defining sequential execution of the program. It has operators to describe parallel computation. The `Foreach` statement of Green-Marl allows parallel and independent processing of elements given as arguments to `Foreach`. The syntax of `Foreach` statement is given below.

`Foreach` (iterator : source (-). range) (filter) body_statement

An example of a `Foreach` statement is given below.

`Foreach` (v : G.Nodes) (cond) { ... }

Here all the vertices v in the graph G which satisfy the condition *cond* execute the body of the `ForEach` statement. The argument '`-`' is used to process elements in the reverse order. Arguments '`-`' and `filter` are optional. *Range* used in above example is *Nodes* and iterator is v.

Green-Marl has support for reductions in the body of the `Foreach` loop using special operators: `+=`, `*=`, `Max=`, `min=`, `&&=` and `||=`, denoting *Sum*, *Product*, *Max*, *Min*, *All* and *Any* respectively.

Algorithm 17: Reduction

```
1 Graph G;
2 Node_Prop <Int>dist;
3 int sum = 0;
4 ForEach (t : G.Nodes) {
5     sum += t.dist;
6 }
```

A simple example of reduction in Green-Marl is given in Algorithm 17. Here the vertices in the Graph object G have an integer property *dist* and *sum* stores the sum of *dist* values of all the vertices in G using the reduction operator `+=`.

Algorithm 18: Group Assignment

```
1 Graph G;
2 Node_Prop <Int>dist;
3 G.dist = +INF;
```

Algorithm 19: Inplace Reduction

```
1 Graph G;
2 Node_Prop <Int>dist;
3 Int total = Sum(s:G.Nodes){s.dist;}
```

Green-Marl also has *group assignments*. An example is shown in Algorithm 18. Here the value of the vertex property *dist* is made *+INF* for all the vertices in the graph object G in parallel in the generated code. The *group assignment* statement is an example of *implicit parallelism* in Green-Marl. Another example of *implicit parallelism* in Green-Marl is *in-place reduction*. An example for in-place reduction is shown in Algorithm 19, which stores the sum of *dist* property values of all the vertices in the graph G in the variable *total* in parallel. Green-Marl supports built-in iterators - which implement parallel traversals of DFS and BFS and these can simplify the development of some graph algorithms.

Algorithm 20 is an implementation of the SSSP algorithm in Green-Marl. The program consists of the SSSP function with four parameters. The parameters are i) G (type `Graph`) ii)

Algorithm 20: SSSP algorithm in Green-Marl

```
1  Procedure SSSP(G:Graph, dist: Node_Prop <int >, len:Edge_Prop <int >, root: Node) {
2      Node_Prop <Bool >updated;
3      Node_Prop <Bool >updated_nxt;
4      Node_Prop <Bool >dist_nxt;
5      Bool fin= True;
6      G.dist=(G==root) ? 0: +INF;
7      G.updated=(G==root) ? True:False;
8      G.dist_nxt=G.dist;
9      G.updated_nxt=G.updated;
10     while( fin ){
11         fin=False;
12         ForEach(n:G.nodes) (n.updated) {
13             ForEach(s: n.Nbrs) {
14                 Edge E=s.ToEdge();
15                 <s.dist_nxt ; s.updated_nxt >min= <n.dist+e.len; True >;
16             }
17         }
18         G.dist=G.dist_nxt;
19         G.updated=G.updated_nxt;
20         G.updated_nxt=False;
21         fin= Exist (n: G.nodes) {n.updated};
22     }
23 }
```

vertex property *dist* (type `Node_Prop`) iii) edge property *len* (type `Edge_Prop`), which stores the weight of the edges in G and iv) *root* (type `Node`) which specifies the source vertex for SSSP computation.

Lines 2–4 add three vertex properties *updated*, *updated_nxt* and *dist_nxt* to the graph object G. The *dist* value of the *root* vertex is made zero and for all other vertices *dist* value is made infinity (*+INF*), using *group assignment* in Line 6. Similarly the vertex property *updated* is made *True* for the source (*root*) vertex and *False* for all other vertices in the *group assignment* in Line 7.

The computation happens in the while loop (Lines 10–22) until a fix point is reached and the shortest distance to all the vertices reachable from the source vertex are computed. The `Foreach` statement in Lines 12–17 executes in parallel mode. It takes all the outgoing edges of a vertex n, whose *updated* property is *True* and tries to reduce the distance of the destination vertex s of the edge using the *min* reduction function (Lines 13–16). The weight of an edge e is found out using `Edge_prop` *len*. The *min* reduction operation can take more than one argument and the one used in Line 15 takes two arguments. If there is a reduction in $s.dist\_nxt$

(ie *s.dist_nxt=n.dist+e.len*), then *s.updated_nxt* is set *True*. After the end of the outermost `ForEach` loop, the updated distance stored in *dist_nxt* is copied to *dist* for all the vertices. The *updated_nxt* property is copied to *updated* and then *updated_nxt* property is set to *False* for all the vertices. The variable *fin* which is made *False* at the beginning of while loop, will be *True* if *updated* property is *True* for atleast one vertex. If this condition is not satisfied, the program exits the while loop and the algorithm terminates.

Green-Marl does not support mutation of the graph object (i.e., adding and removing vertices and edges to/from the graph object) and hence dynamic graph algorithms cannot be written in Green-Marl. Green-Marl supports only multi-core CPUs, and graph algorithms targeting GPU devices cannot be programmed in Green-Marl. Green-Marl does not have support for *worklist* based implementations like Δ-stepping SSSP computation.

3.1.2 Galois

Galois [77] is a framework for implementing graph algorithms on multi-core CPUs. Galois supports mutation of graph objects via *cautious* speculative execution. Galois uses a data-centric formulation of algorithms called *operator formulation*. Galois defines

- *Active Elements*: which are the vertices or edges where computation needs to be performed at a particular instance of program execution.
- *Neighborhood*: the vertices or edges which are read or written by active elements in a particular instance of execution.
- *Ordering* of the active elements present at a particular instance of program execution.

In *unordered* algorithms, where any active element can be taken for processing (e.g DMR), but *ordered* algorithms have an order of processing elements(e.g, Δ-Stepping SSSP).

Galois uses a *worklist* based execution model, where all the *active elements* will be in a *worklist* and they are processed in *ordered* or *unordered* fashion. During the processing of *active elements*, new *active elements* are created, which will be processed in the following rounds of computation. Galois has a `foreach` operator to process active elements in parallel. The `foreach` operator takes as argument an *ordered* or *unordered* worklist. If an argument is an *ordered worklist*, the elements in the *worklist* are processed based on the order specified in the program.

Galois classifies graph algorithms based on *operator*, *active nodes* and *topology*. An *operator* is classified as

- *morph*: if it modifies the structure of the input graph by addition or deletion of edges or vertices.
- *local computation*: if it updates vertex and edge properties without modifying the graph connectivity.
- *reader*: if it does not modify the graph in anyway. Operations can be traversal of graph and reading property values of edges and vertices.

An algorithm classified based on *active-nodes* defines how vertices become active and also the order of processing active elements.

- *Topology-driven* and *data-driven*: These two describe how vertices become active. In some algorithms active elements are determined by the graph structure and they are called *topology- driven* algorithms. Examples are algorithms which iterate over all the edges, like the Bellman-Ford SSSP. But in data-driven algorithms, nodes become active in a data-dependent manner, like the worklist based SSSP.
- *Ordering*: Some algorithms require ordering among active elements (e.g, Δ-Stepping SSSP) while some algorithms do not need any ordering (e.g, DMR).

Galois classifies graphs based on *topology* as

- *Unstructured*: an example is a typical social network graph.
- *Semi-structured*: an example is a tree.
- *structured*: an example is a rectangular grid.

Algorithm 21 shows the pseudocode for SSSP in Galois. Galois uses an *order by interger metric (OBIM)* bucket for Δ-stepping implementation of SSSP as declared in Line 28. The operator in the *InitialProcess* struct reduces the distance of the neighbours of the source vertex and adds to the OBIM buckets (Lines 20-23). This function is called from SSSP class in Line 31. Then the parallel *for_each_local* iterator of Galois calls the operator of Process structure in Line 32. This calls the operator of Process defined in Lines 9-12. The parallel iterator finishes once all buckets are free and the SSSP distance of all the vertices are computed. The *relaxNode()* and*relaxEdge()* functions are not shown in the algorithm. They are used to reduce the distance values of vertices as done in other SSSP algorithms.

Galois also supports mutation of graph objects using *cautious morph* implementations and also algorithms based on mesh networks. Galois implements Delaunay Mesh Refinement (DMR)

Algorithm 21: SSSP in Galois C++ framework

```
1  struct UpdateRequest {
2      Vertex n;
3      Dist w;
4  };
5  typedef Galois::InsertBag<UpdateRequest>Bag;
6  struct Process {
7      AsyncAlgo* self;
8      Graph& graph;
9   void operator()(UpdateRequest& req, Galois::UserContext<UpdateRequest>& ctx) {
10 |        self->relaxNode(graph, req, ctx);
11 |
12 }
13 };
14 struct SSSP{
15     struct InitialProcess {
16         AsyncAlgo* self;
17         Graph& graph;
18          Bag& bag;
19         Node& sdata;
20          void operator()(typename Graph::edge_iterator ii) {
21 |           self->relaxEdge(graph, sdata, ii, bag);
22 |
23 }
24     };
25    void operator()(Graph& graph, GNode source) {
26 |           using namespace Galois::WorkList;
27 |           typedef dChunkedFIFO<64> Chunk;
28 |           typedef OrderedByIntegerMetric<UpdateRequestIndexer<UpdateRequest>, Chunk,
       10> OBIM;
29 |           Bag initial;
30 |           graph.getData(source).dist = 0;
31 |           Galois::do_all( graph.out_edges(source, Galois::MethodFlag::NONE).begin(),
       graph.out_edges(source, Galois::MethodFlag::NONE).end(), InitialProcess(this, graph,
       initial, graph.getData(source)));
32 |           Galois::for_each_local(initial, Process(this, graph), Galois::wl<OBIM>());
33 |
34 }
35 };
```

and Delaunay Triangulation (DT) as *cautious morph* algorithms. Galois does not support multiple graph objects. Programming a new benchmark in Galois requires much effort, as understanding the C++ library and parallel iterators are more difficult compared to a DSL based approach. Galois neither supports GPU devices, nor distributed computing.

3.1.3 Elixir

Elixir [79] is a graph DSL to develop and implement parallel graph algorithms for analyzing static (ie., non-mutable) graphs and it targets multi-core CPUs. Elixir uses both declarative and imperative constructs for determining computations over a graph. Elixir does not support structural transformation of the graph such as addition and deletion of vertices are not supported. Elixir has its own attribute grammar and the compiler converts the program in Elixir to parallel C++ code with calls to the *Galois* framework routines. The main feature of Elixir is the classification of operations over the graph.

Operations in Elixir depend on *active-elements*, *operator*, and *ordering*. *Active-elements* are locations in the graph where computation needs to be performed (subgraphs) . An *operator* is the computation that should be done on an *active-element*. An *operator* reads and writes graph elements in the region containing the *active-elements*.

To specify how the operators (*op*) are applied to the graph, Elixir has the following expressions

- *foreach op*: applies the operator to all the matched elements or subgraphs.
- *for i = low..high op*: applies the operator for each value of i between *low* and *high*.
- *iterate op*: *iterate op* applies the redex operator *op* until there is atleast one valid element to be processed.

To specify the order in which the operations need to be performed on subgraphs, schedulers are used. Elixir supports static and dynamic scheduling policies:

- Metric e (approx Metric e): determines the strict (approximate, allowing violation) order of processing of the subgraphs in accordance with a given metric in the form of e (smaller the value of the metric, higher the priority).
- Group V: specifies that the vertices in a group V, must be processed together. This optimization improve the spatial and temporal locality of the vertices of the graph.
- Unroll k: comparisons that form chains of length k, successively one after another in the same way as unfolding of cycles in imperative programming languages.

Algorithm 22: SSSP algorithm in Elixir

```
1  Graphs [ nodes (node:Node, dist:int), edges(src:Node,dst:Node, wt:int) ]
2  source:Node
3  initdist= [ nodes ( node a, dist d)] →
4  d = if (a==source) 0 else +INF
5  relaxEdge= [ nodes ( node a ,dist ad )
6  nodes (node b, dist bd)
7  edges (node a, node b, wt w)
8  ad+w <bd ] →
     [ bd = ad+w ]
9  init = foreach initdist
10 sssp = iterate relaxedge >>sched
11 main init; sssp
```

- (Op1 or op2) >>fuse: transformation of the mapped subgraphs. The template for *op1* and *op2* are executed, *op1* followed by *op2*. Fusing improves locality and amortizes the cost of acquiring and releasing locks necessary to guarantee atomic operator execution.

Algorithm 22 shows several possible SSSP implementations in Elixir (see the explanation in the next paragraph). Line 1 defines the graph. Each vertex (Node) has a property *dist* of type `int`. Each edge has a source (*src*) and destination (*dst*) vertex, and an integer weight (*wt*). The source vertex for sssp computation is defined in Line 2. Line 11 defines the *SSSP* algorithm which consists of calls to two functions, *init* followed by *sssp*. The *init* function (Line 9) calls *initdist* using *foreach* and initializes the distance of source vertex to zero and the distance of all other vertices to +INF (Lines 3–4). Then control comes to the *sssp* function (Line 10), which calls the *relaxedge* function, that specifies the way distance has to be reduced. This reduction is done with an `iterate` statement and *sched* specifies the scheduling mechanism. The *relaxedge* function will be called many times until a fixpoint is reached. The relaxEdge statement (Line 5–8) specifies a template, the structural part of which is defined as an edge, and the conditional part of which reduces *dist* value of the vertex: If the sum of the *dist* (*ad*) value of source vertex (*a*) and *weight* of the edge $a \rightarrow b$ (*w*) is less than the *dist* value (*bd*) of destination vertex (*d*), a new path with a smaller cost has been found and *dist* attribute of destination vertex is updated.

The *sched* agrument of the *iterate* statement of *sssp* function (Line 10, Algorithm 22) defines how the *sssp* function should be executed. Different values for *sched* will yield different SSSP algorithm implementations in Elixir.

- Dijkstra's [55] algorithm
  sched = *metric ad* >>*group b*

- Δ-stepping algorithm
  DELTA : `unsigned int`
  sched = metric (ad + w) / DELTA.

Elixir does not support mutation of graph objects, distributed computing and GPU devices.

3.1.4 Other works

X-Stream [82] uses edge-centric processing for graph applications rather than using vertex centric processing for algorithms such as SSSP and Strongly Connected Component (SCC). It supports both in-memory and out-of-core graph processing on a single shared-memory machine using scatter-gather execution model. The Stanford Network Analysis Platform (SNAP) [63] provides high-level operations for large network analysis including social networks and target multi-core CPUs. Ligra [88] is a framework for writing graph traversal algorithms for multi-core shared memory systems which uses two different routines, one for mapping vertices and the other for mapping edges. Polymer [103] is a NUMA aware graph framework for multi-core CPUs and it is built with a hierarchical barrier to get more parallelism and locality. The CoRD [92] framework proposes methods for speculative execution on a multi-core CPU. It supports rollback and morph algorithms which need not be cautious. A speculative execution where the execution is restarted from previous consistent state up to which speculation was correct is proposed in [93]. This has less overhead compared to the cost execution from scratch on miss-speculation.

The frameworks mentioned in this section lacs completeness in terms of support for heterogeneous targets, dynamic algorithms, etc.

3.2 Frameworks for Machines with a multi-core CPU and multiple GPUs

The GPU devices have a massively parallel architecture and they follow the SIMT model of Execution. For example, the *Nvidia K-40* GPU has 2,880 cores, 12 GB device memory and a base clock rate of 745 MHz. Nowadays GPUs are being used for General Purpose computing (GPGPU) also. Graph algorithms are *irregular*, require atomic operations, and can result in thread divergence when executed on a Streaming Multiprocessor (SM). Writing an efficient GPU program requires a deep knowledge of the GPU architecture, so that the algorithm can be implemented with less thread divergence, fewer atomic operations, coalesced access etc.Past research has shown that graph algorithms perform well on GPUs and much better than multi-core CPU codes even though they have the limitations mentioned above. We look at some of

the past works which deal with graph algorithms on GPUs.

Graph algorithm implementation on GPUs started with handwritten codes. Efficient implementations of local computation algorithms such as Breadth First Search (BFS) and Single Source Shortest Path (SSSP) on GPU have been reported several years ago [49, 50]. The BFS implementation from Merril [68] is novel and efficient. There have also been successful implementations of other local computation algorithms such as n-body simulation [22], betweenness centrality [85] and data flow analysis [66, 78] on GPU. Different ways of writing SSSP programs on GPU along with their merits and demerits have been explored in [9] and it concludes that worklist-based implementation will not benefit much on GPU compared to that on a CPU.

In the recent past, many graph processing frameworks have been developed which come with structured APIs and optimizations enabling writing efficient graph algorithms on GPU. We look at some of these.

3.2.1 LonestarGPU

The LonestarGPU [73] framework supports mutation of graph objects and implementation of cautious morph algorithms. It has cautious morph implementations of algorithms like Delaunay Mesh Refinement, Survey Propagation, Boruvka's-MST and Points-to-Analysis. Boruvka's-MST algorithm has a local computation implementation using the `Union-Find` data structure and the current version of LonestarGPU has modified the MST algorithm to a more efficient local computation implementation. LonestarGPU also has implementations of algorithms like SSSP, BFS, Connected Components etc, with and without using *worklists*. Since it is a framework, a programmer who wants to write a new algorithms must learn CUDA, GPU architecture and LonestarGPU framework data types. LonestarGPU does not provide any API based programming style for GPUs and it does not support execution of an algorithm on multiple GPUs by graph partitioning or running different algrorithms at the same time.

3.2.2 Medusa

Medusa [104] is a programming framework for graph algorithms on GPUs and multi-GPU devices. It provides a set of APIs and a run time system to program graph algorithms targeting GPU devices. The programmer is required to write only sequential *C++* code with these APIs. Medusa provides a programming model called the *Edge-Message-Vertex* or *EMV* model. Medusa provides APIs for processing vertices, edges or messages on GPUs. A programmer can implement an algorithm using these APIs. APIs provided by Medusa are shown in Table 3.1. APIs on vertices and edges can also send messages to neighbouring vertices.

Medusa programs require user-defined data structures and implementation of Medusa APIs

APIType	Parameter	Variant	Description
ELIST	Vertex V,Edgelist el	Collective	Apply to edge-list el of each vertex v
EDGE	Edge e	Individual	Apply to each edge e
MLIST	Vertex v, Message-list ml	Collective	Apply to message-list ml of each vertex v
MESSAGE	Message m	Individual	Apply to each message m
VERTEX	Vertex v	Individual	Apply to each vertex v
Combiner	Associative Operation o	Collective	Apply an associative operation to all edge-lists or message-lists

Table 3.1. Medusa API

for an algorithm. The Medusa framework automatically converts the Medusa API code into CUDA code. The APIs of Medusa hide most of the CUDA specific details. The generated CUDA code is then compiled and linked with the Medusa libraries. Medusa runtime system is responsible for running programmer written codes (with Medusa APIs) in parallel on GPUs.

Algorithm 23: Pagerank Psuedo code

```
1  compute( p, graph) {
2      double val=0.0;
3      for (each innbr t of p) val += t.PR / t.outdegree;
4      p.PR = val * 0.85+ 0.15
5  }
6  pagerank(graph) {
       for (each t in V ) t.PR = 1 / |V|;
       int i = 0;
7      while( i <100 ){
8          for ( each t in V ) compute(t, graph);
9          ++i;
10     }
11 }
```

Algorithm 23 presents the sequential version of the pagerank algorithm for the reader's quick reference. Algorithm 24 shows the pagerank algorithm implementation using Medusa APIs. The pagerank algorithm is defined in Lines 26 to 31. It consist of three user defined APIs, *SendRank* (Lines 2–7) which operates on *EdgeList*, a vertex API *UpdateVertex* (Lines 9–13) which operates over the vertices and a *Combiner()* function. The *Combiner()* function is for combining message values received from the *Edgelist* operator, which sends the message using the *sendMsg* function (Line 6). The *Combiner()* operation type is defined as addition (Line 36)

Algorithm 24: Medusa Pagerank Algorithm

```
1  //Device code APIs:
2  struct SendRank{// ELIST API
3  __device__ void operator() (EdgeList el, Vertex v) {
4      int edge_count = v.edge_count;
5      float msg = v.rank/edge_count;
6      for(int i = 0; i <edge_count; i ++) el[i].sendMsg(msg);
7  }
8  }
9  struct UpdateVertex{ // VERTEX API
10 __device__ void operator() (Vertex v, int super_step) {
11     float msg_sum = v.combined_msg();
12     vertex.rank = 0.15 + msg_sum*0.85;
13 }
14 }
15 struct vertex{ //Data structure definitions:
16 float pg_value;
17 int vertex_id;
18 }
19 struct edge{
20 int head_vertex_id, tail_vertex_id;
21 }
22 struct message{
23 float pg_value;
24 }
25 Iteration definition:
26 void PageRank() {
27     InitMessageBuffer(0); /* Initiate message buffer to 0 */
28     EMV<ELIST>::Run(SendRank);/* Invoke the ELIST API */
29     Combiner(); /* Invoke the message combiner */
30     EMV<VERTEX>::Run(UpdateRank);/* Invoke the VERTEX API */
31 }
32 int main(int argc, char **argv) {
33     .....
34     Graph my_graph;
35     //load the input graph.
36     conf.combinerOpType = MEDUSA_SUM;
37     conf.combinerDataType = MEDUSA_FLOAT;
38     conf.gpuCount = 1;
39     conf.maxIteration = 30;
40     Init_Device_DS(my_graph);/*Setup device data structure.*/
41     Medusa::Run(PageRank);
42     Dump_Result(my_graph);/* Retrieve results to my_graph. */
43     .....
44     return 0;
45 }
```

and message type as `float` (Line 37) in the *main()* function. The *main()* function also defines the number of iterations for *pagerank()* function as 30 (Line 39) and then the *pagerank()* function is called using *Medusa::Run()* (Line 41). The *main()* function in Medusa code initializes the algorithm specific parameters like *msgtype*, *aggregator* function,number of GPUs, number of iterations etc. Then loads the graph on to the GPU/GPUs and calls the *Medusa::Run* function which consists of the main kernel. After the kernel finishes its execution, the result is copied using the *Dump_Result* function (Line 42).

The *SendRank EdgeList* API takes an *EdgeList el* and a *vertex v* as arguments and computes a new value for *v.rank* and this value is sent to all the neighbours of the vertex *v* stored in *Edgelist el*. The value sent using the *sendMsg* function is then aggregated using the *Combiner()* function (Line 29) which is defined as the *sum* of the values received. The *UpdateVertex Vertex* API then updates the pagerank using the standard equation to compute the pagerank of a vertex (Line 12).

Medusa supports the execution of graph algorithms on multiple GPUs of the same machine, by partitioning large input graph and storing them on multiple GPUs. Medusa uses the EMV model, which is an extension of the Bulk Synchronous Parallel (BSP) Model. Medusa does not support running different algorithms on different devices at the same, when a graph object fits within a single GPU. Also it does not support distributed execution on GPU clusters.

3.2.3 Gunrock

The Gunrock [97] framework provides a data-centric abstraction for graph operations at a higher level which makes programming graph algorithms easy. Gunrock has a set of APIs to express a wide range of graph processing primitives. Gunrock also has some GPU-specific optimizations. It defines *frontiers* as a subset of edges and vertices of the graph which are actively involved in the computation. Gunrock defines *advance*, *filter*, and *compute* primitives which operate on *frontiers* in different ways.

- An *advance* operation creates a new *frontier* from the current *frontier* by visiting the neighbors of the current *frontier*. This operation can be used for algorithms such as SSSP and BFS which activate subsets of neighbouring vertices.
- The *filter* primitive produces a new *frontier* from the current *frontier*, which will be a subset of the current *frontier*. An example algorithm which uses such a primitive is the Δ-Stepping SSSP.
- The *compute* step processes all the elements in the current frontier using a programmer defined computation function and generates new frontier.

The SSSP algorithm in Gunrock is shown in Algorithm 25. The *SSSP* algorithm starts with a call to *SET_PROBLEM_DATA()* (Lines 1–6) which initializes the distance *dist* to ∞ and predecessor *preds* to NULL for all the vertices. This is followed by *dist* of *root* node being made to 0. Then the *root* node is inserted to the worklist *frontier*. The computation happens in the `while` loop (Lines 20–24) with consecutive calls to the functions *ADVANCE* (Line 21), *FILTER* (Line 22) and *PRIORITYQUEUE* (Line 23). The *ADVANCE* function with the call to *UPDATEDIST* (Lines 7–10), reduces the distance of the destination vertex *d_id* of the edge *e_id* using the value *dist[s_id]+weight[e_id]* where *s_id* is the source vertex of the edge. All updated *vertices* are added to the *frontier* for processing in the coming iterations. Then the *ADVANCE* function calls *SETPRED* (Lines 11–14) which sets the predecessor in the shortest path of vertices from root node. The *FILTER* function removes redundant vertices from the frontier using a call to *REMOVEREDUNDANT*, this reduces the size of the worklist *frontier* which will be processed in the next iteration of the `while` loop. Computation stops when *frontier.size* becomes zero.

In Gunrock, programs can be specified as a series of bulk-synchronous steps. Gunrock also looks at GPU specific optimizations such as kernel fusion. Gunrock provides load balance on irregular graphs where the *degree* of the vertices in the *frontier* can vary a lot. This variance is very high in graphs which follow power-law distribution. Instead of assigning one thread to each vertex, Gurnock loads the neighbor list offsets into the shared memory, and then uses a Cooperative Thread Array (CTA) to process operations on the neighbor list edges. Gunrock also provides *vertex-cut* partitioning, so that neighbours of a vertex can be processed by multiple threads. Gunrock uses a priority queue based execution model for SSSP implementation. Gunrock was able to get good performance using the execution model and optimizations mentioned above on a single GPU device. Gunrock does not support mutation of graph objects and mesh based cautious speculative algorithms. It does not support multi-GPU devices.

3.2.4 Totem

Totem [44, 45] is a heterogeneous framework for graph processing on a single machine. It supports using a multi-core CPU and multiple GPUs on a single machine. When multiple devices are used for computation, the graph is partitioned and stored in the devices used for computation. Totem follows the Bulk Synchronous Parallel (BSP) model of execution. Computation happens in a series of supersteps called *computation*, *communication* and *synchronization*. Totem stores graphs in the Compressed Sparse Row (CSR) format. It partitions graphs is a way similar to *edge-cut* partitioning. It supports large-scale graph processing on a single machine.

Totem uses two buffers on each device for communication called as *outbox* and *inbox* buffers.

Algorithm 25: SSSP algorithm in Gunrock

```
1  procedure SET_PROBLEM_DATA (G, P, root)
2      P.dist[1..G.verts] ← ∞
3      P.preds[1..G.verts] ← NULL
4      P.dist[root] ← 0
5      P.frontier.Insert(root)
6  end procedure
7  procedure UPDATELDIST (s_id, d_id, e_id, P )
8      new_dist ← P.dist[s_id] + P.weights[e_id]
9      return new_dist <atomicMin(P.dist[d_id], new_dist)
10 end procedure
11 procedure SETPRED (s_id, d_id, P )
12     P.preds[d_id] ← s_id
13     P.output_queue_ids[d_id] ← output_queue_id
14 end procedure
15 procedure REMOVEREDUNDANT (node_id, P )
16     return P.output queue_id[node_id] == output_queue_id
17 end procedure
18 procedure SSSP(G, P, root)
19     SET_PROBLEM_DATA (G, P, root)
20     while P.frontier.Size() >0 do
21         ADVANCE (G, P, UPDATEDIST, SETPRED)
22         FILTER (G, P, REMOVEREDUNDANT)
23         PRIORITYQUEUE (G, P )
24     end while
25 end procedure
```

The *outbox* buffer is allocated with space for each remote vertex, while the *inbox* buffer has an entry for each local vertex that is a remote vertex in another subgraph on a different device. The communication buffer will have two fields, one for the remote vertex id and the other for messages for the remote vertex. Totem partitions a graph onto multiple devices, with less storage overhead. It aggregates boundary edges (edges whose vertices belong to different master devices) to reduce communication overhead. It sorts the vertex ids in the *inbox* buffer to have better cache locality. Totem does not have a feature to run multiple algorithms on the same input graph using different devices on a machine. Such a feature is useful when we want to compute some properties of an input graphs such as number of Connected Component, maximum *degree*, pagerank etc., of by running different algorithms on the same input graph using multiple devices.

Totem has inbuilt benchmarks which the user can specify as a numerical value. A User can also specify how a benchmark should be executed:how many GPUs to use, the percentage of

Algorithm 26: Totem Engine structure

```
1  totem_config config={
2      graph,
3      partitioning_algo,
4      init_func,
5      kernel_fun,
6      msg_reduce_func,
7      finalize_func,
8  };
9  totem_config(&config);
10 totem_execute();
```

th graph that should go to GPU etc. Heterogeneous computing is useful as some algorithms does not perform well on GPUs. Examples are algorithms such as SSSP, BFS etc on GPU for road-network. This happens as road-networks have a large *diameter* and less parallelism is possible. So the user can decide where the algorithm should be executed (CPU or GPU). Writing a new benchmark in Totem is very hard, as the programmer must understand the low-level implementations of Totem framework in C++ and CUDA. The basic structure of the Totem framework is shown in Algorithm 26, where a Totem benchmark defines parameters in the *totem_cofig* class. Then the benchmark runs with a call to *totem_config*, followed by *totem_execute()*.

3.2.5 IrGL

IrGL [76] implements three optimizations namely, *iteration outlining*, *cooperative conversion* and parallel execution of nested loops. IrGL is an intermediate code reperesentation, on which these optimizations are applied and CUDA code is generated from it. *Iteration outlining* moves the iterative loop from the *host* code to the *device* code and this eliminates the performance bottleneck associated with multiple kernel calls in an iterative loop. *Cooperative conversion* reduces the total number of atomic operations by aggregating functions over thread, warp and thread-block level.

IrGL provides the constructs, `ForAll`, `Iterate`, `Pipe`, `Invoke`, `Respawn` etc. IrGL provides nested parallelism for its parallel `ForAll` construct and provides a *retry* worklist to the kernel which is hidden from the programmer. The `Pipe` statement can be invoked with an optional argument *Once* and in that case the kernels inside the `Pipe` statement block will be excuted once. IrgL has a `Respawn` statement which adds an element to the *retry* worklist.

Algorithm 27 shows a Δ-stepping SSSP implementation in IRGL for GPU [9]. The INIT function which is called using the `Invoke` statement in Line 11 initializesthe SSSP computation

by making the distance of all the vertices ∞, and then making the distance of the source vertex zero, and adding it to the worklist using the *push* operation. The INIT function will be called only once as it is enclosed inside the `Pipe` *Once*. Then the SSSP function (Lines 1-9) is called using the `Invoke` statement (Line 13). SSSP function is nested inside `Pipe` (without *once*) and SSSP has a `Respawn` statement in Line 4, which adds elements to the *retry* worklist. If the distance value is greater than the current delta value, it is added to the worklist (different from the *retry* worklist) using the *push* operation (Line 6). The `Respawn` operation makes the SSSP call loop until the the *retry* worklist (bucket with current Δ value) becomes empty. After that, duplicate elements are removed from other buckets and Δ is incremented. The `Pipe` loop (Lines 12-16) exits when the worklist (all buckets) becomes empty.

IrGL provides worklist based implementation of algorithms where the *retry* worklist is transparent to the programmer and constructs like `Respawn`, `Pipe` and `Iterate` are used to process the elements. The `ForAll` statement iterates over all the elements of an object in parallel (which is given as its argument). IrGL does not provide any support for clusters of GPUs.

Algorithm 27: SSSP using `Pipe` construct in IrGL

```
1  Kernel SSSP(graph, delta) {
2      .....
3      if ( dst . distance ≤ delta ){
4          Respawn ( dst )
5      else
6          WL.push ( dst )
7      .....
8      }
9  }
10 Pipe Once {
11     Invoke INIT( graph , src )
12     Pipe {
13         Invoke SSSP( graph , curdelta ) ;
14         Invoke remove_dups( ) ;
15         curdelta += DELTA ;
16     }
17 }
```

3.2.6 Other works

High Performance Vertex-Centric Graph Analytics on GPUs [36] presents Warp Segmentation to improve GPU utilization by dynamically assigning appropriate number of threads to process a vertex. This work supports large-scale graph processing on multiple GPUs with optimized

communication where only the updated boundary vertices are communicated. Performance efficiency is achieved by processing only active vertices in each iteration. For multi-GPU graph computation, this work provides dynamic load balancing across GPUs. This work presents Collaborative Context Collection (CCC) and Collaborative Task Engagement (CTE) techniques for efficient implementation of other irregular algorithms. CCC is a compiler technique to enhance the SIMD efficiency in loops that have thread divergence. The CTE library does load balancing across threads in an SIMD group. GasCL (Gather-Apply-Scatter with OpenCL) [6] is a graph processing framework built on top of OpenCL which works on several accelerators and supports parallel work distribution and message passing.

The MapGraph [40] framework provides high-level APIs, making it easy to write graph programs and obtain good speedups on GPUs. MapGraph dynamically chooses scheduling strategies depending on the size of the worklist and the size of the adjacency lists for the vertices in the frontier. Halide [80] is a programming model for image processing on CPUs and GPUs. There has been works on speculative parallelization of loops with cross-iteration dependences on GPUs [37]. The iGPU [67] architecture proposes a method for breaking a GPU function execution into many idempotent regions so that in between two continuous regions, there is very little live state, and this fact can be used for speculative execution.

Paragon [84] uses a GPU for speculative execution and on misspeculation, that part of the code is executed on CPU. An online profiling based method [56] partitions work and distributes it across CPU and GPU. CuSha [58] proposes two new ways of storing graphs on GPU called G-Shards and Concatenated Windows, that have improved regular memory access patterns. OpenMP to GPGPU [62] is a framework for automatic code generation for GPU from OpenMP CPU code. There is no support from the CUDA compiler to have a barrier for the all threads in a *kernel* blocks. Such a feature is needed in some *cautious* morph algorithm (e.g, DMR). A barrier for all the threads in a *kernel* can be implemented in software, by launching the *kernel* with less number of threads and with the help of *atomic* operations provided by CUDA, and each thread processes a set of elements. Such an implementation can be found in [100].

Frameworks mentioned in this section lack completeness in terms of support for morph algorithms, multi-GPU executions, and distributed execution on GPU clusters.

3.3 Frameworks for distributed systems

Natural graphs have very big sizes. Such large-scale graphs are sparse and follow the power-law degree distribution. Such graphs are processed on a computer cluster. Programming for a computer cluster requires learning the MPI library and explicit communication code has to be inserted in the program, with proper synchronizations to preserve sequential consistency.

To achieve good performance there should be work balance across machines in the cluster and communication overhead should be minimum. Also the graph should be partitioned across machines with less storage overhead. This is a very hard problem and there are many frameworks which make programming on a computer cluster easy. Popular frameworks are Pregel, GraphLab and PowerGraph. We look at features of these framework in brief.

3.3.1 GraphLab

GraphLab [64] is an asynchronous distributed shared memory abstraction in which vertex programs have shared access to a distributed graph with data stored on every vertex and edge. Each vertex program may directly access information on the current vertex, adjacent edges, and adjacent vertices irrespective of the edge direction. Vertex programs can schedule neighboring vertex-programs to be executed in the future. GraphLab ensures serializability by preventing neighboring program instances from running simultaneously. By eliminating messages, GraphLab isolates user defined algorithms from the movement of data, allowing the system to choose when and how to move the program state. GraphLab uses *edge-cut* partitioning of graphs and for a vertex v all its outgoing edges will be stored in the same node.

Algorithm 28: GraphLab Execution Model

```
1 Input: Data Graph G = (V, E, D)
2 Input: Initial vertex worklist T = {v1 , v2 , ...}
3 Output: Modified Data Graph G = (V, E, D')
4 while( (T ≠ φ) ){
5 |   v ← GetNext(T )
6 |   (T', Sv ) ← update(v, Sv )
7 |   T ← T ∪ T'
8 }
```

The Execution model of GraphLab is shown in Algorithm 28. The *data graph* G(V,E,D) (Line 3) of GraphLab stores the program state. A Programmer can add data with each vertex and edge based on the requirement of the algorithm requirement. The *update* function (Line 6) of Graphlab takes as input a vertex v and its *scope* S_v (data stored in v, its adjacent vertices and edges). The *update* function returns modified scope S_v and a set of vertices T' which require further processing. The set T' is added to the set T (Line 7), so that it will be processed in upcoming iteration. Algorithm terminates when T becomes empty (Line 4).

GraphLab does not supports GPU devices. Programming new algorithms in GraphLab is harder compared to a DSL based approach. The execution results in more data communication

due to the asynchronous execution model.

3.3.2 PowerGraph

PowerGraph [46] gives a shared-memory view of computation and thereby programmer need not to program communication between machines in a cluster. Graph properties should be updated using commutative and associative functions. PowerGraph supports the BSP model of execution and also the asynchronous model of execution. The graphs can have a user defined vertex data D_v for a vertex v and edge data $D_{u,v}$ for an edge $u \rightarrow v$. PowerGraph follows Gather-Apply-Scatter (GAS) model of execution as a state-less vertex-program which implements the *GAS-VertexProgram* interface as shown in Algorithm 29.

Algorithm 29: Gather-Apply-Scatter-VertexProgram Interface of PowerGraph

```
1 interface GASVertexProgram(u) {
2 // Run on gather_nbrs(u)
3 gather(D_u , D (u,v) , D_v ) → Accum
4 sum(Accum left, Accum right) → Accum
5 apply(D_u ,Accum) → D_u^new
6 // Run on scatter_nbrs(u)
7 scatter(D_u^new ,D(u,v) ,D_v ) → (D_(u,v)^new, Accum)
8 }
```

The program is composed of functions *gather*, *sum*, *apply* and *scatter*. Each function is invoked in stages by the PowerGraph engine following the semantics in Algorithm 30. The *gather* function is invoked on all the adjacent vertices of a vertex u. The gather function takes as argument the data on an adjacent vertex and edge, and returns an accumulator specific to the algorithm. The result is combined using the commutative and associative *sum* operation. The final gathered result a_u is passed to the *apply* phase of the GAS model. The *scatter* function is invoked in parallel on the edges adjacent to a vertex u producing new edge values $D_{(u,v)}$. The scatter function returns an optional value Δa which is used to update the accumulator a_v for the *scatter_nbrs* v of the vertex u. The *nbrs* in the *scatter* and *gather* phase can be *none*, *innbrs*, *outnbrs*, or *allnbrs*.

If PowerGraph is run using the BSP model, the gather, apply, and scatter phases are executed in order. Each *minor-step* is run synchronously on the active vertices with a barrier at the end. A *super-step* consists of a single sequence of *gather*, *apply* and *scatter minor-steps*. Changes made to the graph properties are committed at the end of each minor-step. Vertices activated in each *super-step* are executed in the subsequent *super-step*. If PowerGraph is run using the asynchronous engine, the engine processes active vertices as processor and network

Algorithm 30: PowerGraph Program Semantics

```
1  Input: Center vertex u
2  if( cached accumulator a_u is empty ){
3  |  foreach( neighbor v in gather_nbrs(u) ){
4  |  |  a_u ← sum(a_u , gather(D_u , D (u,v) , D_v ))
5  |  }
6  }
7  D_u ← apply(D_u , a_u )
8  foreach( neighbor v scatter_nbrs(u) ){
9  |  (D (u,v) , Δa) ← scatter(D_u , D_(u,v) , D_v )
10 |  if( a_v && Δa are not Empty ){
11 |  |  a_v ← sum(a_v , Δa)
12 |  }
13 |  else{
14 |  |  a_v ← Empty
15 |  }
16 }
```

resources become available. Changes made to the graph properties during the apply and scatter functions are immediately committed to the graph and visible to subsequent computations.

PowerGraph uses balanced vertex cut where edges of the graph object are assigned evenly to all the to processes when program is run on p machines. 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 a 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.

Vertex cut partitioing can result in computation balance, but this gives rise to a huge increase in communication volume as a vertex will be present in many nodes, and update of graph property values requires scatter and gather.

3.3.3 Pregel

The Pregel [65] framework uses random edge-cut to partition graphs. and follows the Bulk Synchronous Parallel (BSP) Model [95] of execution, with the with execution being carried out in a series of *supersteps*. The input graph G(V,E,D) can have mutable properties associated with vertices and edges. In each *superstep*, vertices carry out the computation in parallel. A

vertex can modify property values of its neighbouring vertices and edges, send messages to vertices, receive messages from vertices, and if required change the topology of the graph. All active vertices perform computation and all the vertices are as set active initially. A deactives itself by calling the *VoteToHalt()* function and it gets reactivated when a message is received from another vertex. Once all vertices call the *VoteToHalt()* function and no message is sent across vertices, the algorithm terminates.

Algorithm 31: Vertex API of Pregel

```
1  template <typename VertexValue, typename EdgeValue, typename MessageValue>
2  class Vertex {
3  public:
4     virtual void Compute(MessageIterator* msgs) = 0;
5     const string& vertex_id() const;
6     int64 superstep() const;
7     const VertexValue& GetValue();
8     VertexValue* MutableValue();
9     OutEdgeIterator GetOutEdgeIterator();
10    void SendMessageTo(const string& dest_vertex,const MessageValue& message);
11    void VoteToHalt();
12 }
```

Algorithm 31 shows the vertex class API in Pregel. The message, vertex and edge data types are specified as templates in Line 1, and these types will be different for different algorithms. A programmer needs to override the `virtual` *Compute()* function, which will be run on all the active vertices in each superstep. The value associated with a vertex can be read using the *GetValue()* function and values can be modified by the function *MutableValue()*. Values associated with out-edges can be read and modified using the functions given by the out-edge iterator.

Vertices communicate by sending messages. Typically a message contains the destination vertex which should receive the message and the message data. A message sent to a vertex v will be available before the *Compute()* operation of next super-step.

3.3.4 Giraph

Giraph [29, 87] is an open source framework written in *Java* which is based on the Pregel model and runs on the Hadoop infrastructure. Giraph has extended the basic Pregel model with additional functionalities such as master computation, sharded aggregators, out-of-core computation, composable computation etc. Giraph can be used to build machine learning and data mining (MLDM) applications along with large scale processing [87].

3.3.5 Other works

GPS (Graph Processing System) [83] is an open source framework and follows the execution of model of Pregel. Green-Marl compiler was extended to CPU-clusters [54] and it generates GPS based Pregel like code. Mizan [57] uses dynamic monitoring of algorithm execution, irrespective of graph input and does vertex migration at run time to balance computation and communication. Hadoop [99] follows the MapReduce() processing of graphs and uses the Hadoop distributed file system (HDFS) for storing data. HaLoop [21] is a framework which follows MapReduce() pattern with support for iterative computation and with better caching and scheduling methods. Twister [34] is also a framework which follows the MapReduce() model of execution. Pregel like systems can outperform MapReduce() systems in graph analytic applications. The GraphChi [60] framework processes large graphs using a single machine, with the graph being split into parts (called shard) and loading shards one by one into RAM and then processing each shard. Such a framework is useful in the absence of distributed clusters. Graphine [101] uses the agent-graph model to partition graphs, uses scatter agent and combine agent and it reduces communication overhead compared to that in PowerGraph. GraphIn [86] supports incremental dynamic graph analytics using incremental-GAS programming model. The Parallel BGL is a distributed version of the Boost Graph Library (BGL). GRACE [96] E provides a synchronous iterative graph programming model for programmers. It has a parallel execution engine for both synchronous and user-specified built-in asynchronous execution policies. Table 3.2 divides the major related works we disussed into different groups based on target systems supported, kind of work (framework, DSL etc.) and support for speculation.

References	A	B	C	D	E	F	G	H
Green-Marl [53],Elixir [79], [54]	√	x	x	√	x	x	x	
LonestarGPU [72]	x	√	x	x	√	√	√	
Medusa[104], [62]	x	√	x	x	√	x	√	
Totem [45][44],	x	√	x	√	√	x	√	
Galois[77]	x	√	x	√	x	√	√	
[22], [71], [9], [58], [66], [78],[49][50]	x	x	x	x	√	x	√	
[93] [92]	x	x	x	√	x	√	√	
[10]	x	x	√	√	x	x	√	
GraphLab[64],Pregel [65], Giraph [87], PowerGraph[46], [47], [83], [101]	x	x	√	√	x	x	√	√

Table 3.2. **Related work comparision -** **A**=DSL, **B**=Framework, **C**=Libray, **D**= CPU, **E**=GPU, **F**= Speculation, **G**=handwritten code, **H**=Distributed (multi-node) Computation

Chapter 4

Overview of Falcon

4.1 Introduction

Falcon is a graph DSL targeting distributed heterogeneous systems including CPU cluster, GPU cluster, CPU+GPU cluster, multi-GPU machine in addition to machine with single multi-core CPU and GPU. 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 Falcon compiler (see Figure 4.1). These codes are then compiled with the native compilers (g++, nvcc) and libraries to create executables. For distributed targets, the Falcon compiler performs static analysis to identify the data that needs to be communicated between devices at various points in the program (See Sections 6.6.5 and 6.6.9). Falcon extends the C programming language. In addition to the full generality of C (including pointers, structs and scope rules), Falcon provides the following types relevant to graph algorithms: Point, Edge, Graph, Set and Collection. It also supports constructs such as foreach and parallel sections for parallel execution, single for synchronization, and reduction operations.

Initial version of Falcon [24] required an optional $<GPU>$ tag in the declaration statement, which if present tells the compiler to allocate the variable on the GPU. But this requirement has been removed in the new version Falcon [25], with a simple program analysis. The programmer can specify the target system for which code needs to be generated as a compile time argument, and the compiler allocates variables appropriately and code for the specified target is generated. The generated code is then compiled with the appropriate compiler and libraries to create the executable.

We begin with an explanation of DSL code in Falcon for SSSP computation. The special data types, constructs and their informal semantics are discussed later. A brief summary of

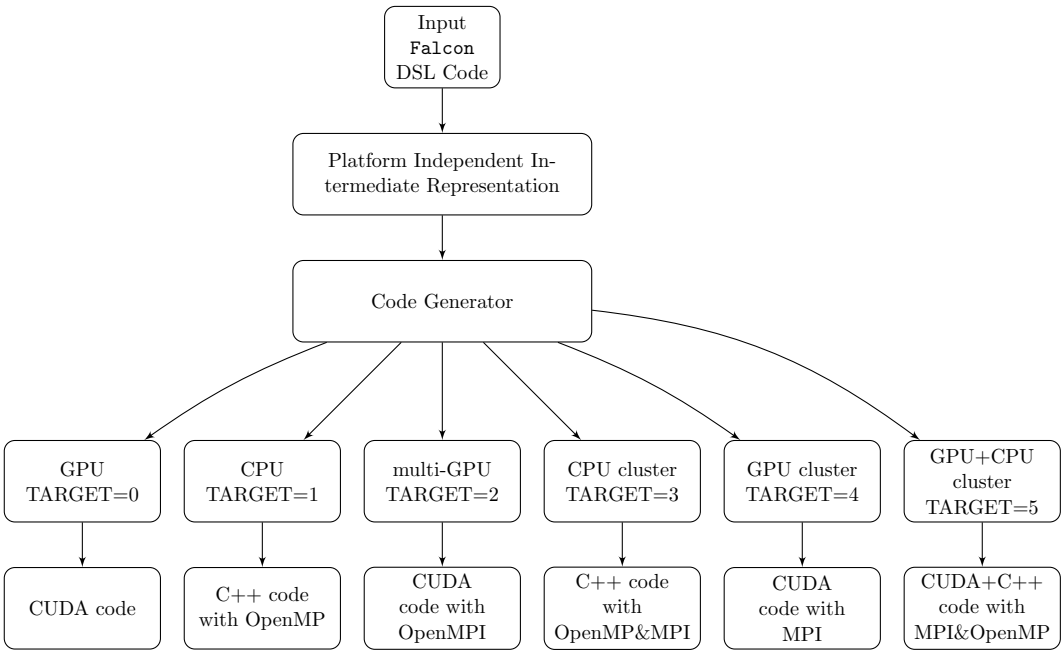


Figure 4.1: `Falcon` DSL overview

the special data types and constructs are provided in table 4.1.

4.2 Example: Shortest Path Computation

Single source shortest path (SSSP) computation is a fundamental operation in graph algorithms. Given a graph G(V,E) with a designated source vertex s and nonnegative edge weights, it computes the shortest distance from the source vertex s to every other vertex $v \in V$. Algorithm 32 shows the code for SSSP computation in `Falcon`.

Lines 17–20 add four properties *dist*, *uptd*, *olddist*, *pred* respectively to each `Point` (vertex) in the `Graph` object, *hgraph*. The algorithm first initializes *dist*, *olddist* and *pred* values of all the vertices to a large value (Line 22). Also *uptd* property value is made false for all vertices.

The *dist* variable of the source vertex is then made zero (Line 23), followed by *uptd* values of source vertex made *true* (Line 24). It then progressively *relaxes vertices* to determine whether there is any shorter path to a vertex via some other incoming edge (Line 27). This is done by checking the condition ($\forall(u,v) \in E$) $(dist[v]) > (dist[u] + weight(u,v))$. If this condition is satisfied, then the distance of the destination vertex v is changed to the smaller value via u (Line 5), using an atomic operation (more on this later). An invariant is that a vetex's distance never increases (it monotonically reduces). This procedure is repeated until we reach a fix point (lines 27-32).

The *relaxgraph()* function is called repeatedly (Line 27) and it keeps on reducing *dist* value

Data type	Description
Point	Can be up to three dimensions and stores a `float` or `int` values in each dimension.
Edge	Edge consist of source and destination `Points`, with optional nonnegative `int` weight.
Graph	Entire Graph. Consist of Points and Edges. new properties can be added to `Graph`, which can be used to view graph as mesh of triangles or rectangles.
Set	A static collection. Implemented as a `Union-Find` data structure.
Collection	A dynamic collection. Elements can be added to and deleted from `Collection`.
foreach	A construct to process all elements in an object in parallel.
parallel sections	A construct to execute codes concurrently on multiple devices.
single	A synchronization construct to lock an element or a `Collection` of elements.

Table 4.1. Data Types, parallel and synchronization constructs in `Falcon`

of each `Point` (Line 5). The `foreach` for *relaxgraph()* is with a condition (t.uptd) that makes sure that only points which satisfy the condition will execute the code inside the *relaxgraph()* function. In the first invocation of *relaxgraph()*, only the source vertex will perform the computation. Since multiple threads may update the distance of the same vertex (e.g., when relaxing edges (u_1, v) and (u_2, v)), some synchronization is required across the threads. This is achieved by providing atomic variants for commonly used operations. The *MIN()* function used by *relaxgraph()* is an atomic function that reduces *dist* atomically (if necessary) and if it does change, the third argument value will be set to 1 (Line 5).

So, whenever there is a reduction in the value of *dist* for even one `Point`, the variable *changed* is set to 1. When the *relaxgraph()* function finishes the computation the *uptd* property value of all the vertices is `false` as Line 3 resets *uptd* property value to `false`. After each call to *relaxgraph()*, the *reset1()* function makes *uptd* `true` only for points whose distance from the source vertex was reduced in the last invocation of the *relaxgraph()* function (Line 29).

The variable *changed* is reset to zero before *relaxgraph()* is called in each iteration (Line 26). Its value is checked after the call and if it is zero, indicating a fixed-point, the control leaves the `while` loop (Line 28). At this stage, the computation is over.

The predecessor of each vertex on the shortest path from the source vertex is stored in the property *pred*, using the `for` loop in Lines 31- 36, which iterates over edges P1→P2, with weight W (Lines 32-34). $pred[P2]$ is updated to $P1$, if the the two conditions $dist[P2] = dist[P1] + W$

Algorithm 32: Optimized SSSP code in `Falcon`

```
1  int changed = 0;
2  relaxgraph(Point p, Graph graph) {
3  |   p.uptd=false;
4  |   foreach( t In p.outnbrs ){
5  |   |   MIN(t.dist, p.dist + graph.getweight(p, t), changed);
6  |   }
7  }
8  reset( Point t, Graph graph) {
9  |   t.dist=t.olddist=1234567890; t.uptd=false; t.pred=1234567890;
10 }
11 reset1( Point t, Graph graph) {
12 |   if( t.dist<t.olddist)t.uptd=true;
13 |   t.olddist=t.dist;
14 }
15 main(int argc, char *argv[]) {
16 |   Graph hgraph; // graph object
17 |   hgraph.addPointProperty(dist, int);
18 |   hgraph.addPointProperty(uptd, bool);
19 |   hgraph.addPointProperty(olddist, int);
20 |   hgraph.addPointProperty(pred, int);
21 |   hgraph.read(argv[1]);
22 |   foreach (t In hgraph.points)reset(t,hgraph);
23 |   hgraph.points[0].dist = 0; // source has dist 0
24 |   hgraph.points[0].uptd=true;
25 |   while( 1 ){
26 |   |   changed = 0; //keep relaxing on
27 |   |   foreach(t In hgraph.points) (t.uptd) relaxgraph(t,hgraph);
28 |   |   if(changed == 0)break;//reached fix point
29 |   |   foreach(t In hgraph.points)reset1(t,hgraph)
30 |   }
31 |   for( (int i=0;i<hgraph.nedges;i++) ){
32 |   |   Point(hgraph) P1=hgraph.edges[i].src;
33 |   |   Point (hgraph) P2= hgraph.edges[i].dist;
34 |   |   int W= hgraph.getWeight(P1,P2);
35 |   |   if(P2.dist== (P1.dist+W) && P2.pred!=1234567890)P2.pred=P1;
36 |   }
37 |   for(int i = 0; i <hgraph.npoints; ++i)
38 |       printf("i=%d dist=%d\n", i, hgraph.points[i].dist);
39 }
```

and ($pred[P2] == 1234567890$) are satisfied (Line 35).

4.3 Benefits of Falcon

`Falcon` DSL code for SSSP computation is shown in Algorithm 32. The program has no target specific information. But during compilation of the DSL code, appropriate arguments can be given to the compiler to generate code for heterogeneous targets: multi-core CPUs, GPUs, multi-GPU machines, CPU clusters, GPU clusters and CPU+GPU clusters. This improves programmer productivity who now writes a single program in `Falcon`. In the absence of such a DSL, programmer will be forced to write separate codes in different languages (e.g., C++, CUDA) and with different libraries (e.g., OpenMP, MPI/OpenMPI). This requires a lot of programming effort and such codes are difficult to debug and also error prone.

Some features which are not available in CUDA and MPI are supported in software by the `Falcon` compiler. Novelties of `Falcon` are mentioned below.

- It supports a Barrier for the GPU kernel. (not supported by CUDA)
- It supports Distributed locking across CPU and GPU clusters. (not supported by MPI)
- A single DSL code converted to different targets by the `Falcon` compiler.
- `Falcon` supports usage of a multi-GPU machine to run different benchmarks for a single input graph on different GPUs. To the best of our knowledge this facility is not provided by any other framework.
- The `Falcon` compiler generates efficient code, making DSL codes match or outperform state-of-the-art frameworks for heterogeneous targets.
- Support for dynamic algorithms and GPU devices is another feature, which is absent in recent powerful graph DSLs like GreenMarl [53] and Elixir [79].
- A programmer need not be concerned with the details of device architectures, thread and memory management etc., making `Falcon` novel, attractive, and easy to program.

4.4 Data Types in `Falcon`

Table 4.1 shows a list of special data types in `Falcon` with a short description.

field	type	description
x,y,z	var	stores `Point` coordinates in each dimension.
isdel	var	returns `true` if `Point` object is already deleted.
getOutDegree	function	returns number of outgoing edges of a `Point`
getInDegree	function	returns number of incoming edges of a `Point`
del	function	delete a `Point`

Table 4.2. Fields of `Point` data type in `Falcon`

4.4.1 Point

A `Point` data type can have up to three dimensions. A `Point` can store either `int` or `float` values in their fields. The Delaunay Mesh Refinement(DMR) [26] algorithm has two dimensional points, with floating point values. The `Point` data type needs multiple dimensions for such mesh based algorithms. Algorithms like SSSP, BFS, MST etc., need only one dimensional points with nonnegative integer `Point` identifier. The `Falcon` compiler does not have separate data types for points with different dimensions. It is decided by command line arguments and input. The number of *outgoing* (*incoming*) edges of a vertex can be found using *getOutDegree()* (*getInDegree()*) function of the `Point` data type. A vertex can be deleted from the graph object using *del()* function and *isdel* field can be used to check whether a vertex is already deleted. The major fields of `Point` data type and their description is provided in Table 4.2.

4.4.2 Edge

field	type	description
src	var	source vertex of an `Edge`
dst	var	destination vertex of an `Edge`
weight	var	weight of an `Edge`
isdel	var	returns `true` if the `Edge` is already deleted.
del	function	delete an `Edge`

Table 4.3. Fields of `Edge` data type in `Falcon`

An edge in `Falcon` connects two `Points` in the `Graph` objects. Edges can have optional nonnegative weight associated with it and edges can be directed or undirected. The *src* and *dst*

field of an edge returns the source and destination vertex of an edge. The *weight* field returns weight of the edge. The *isdel* field is set `true` if the edge is deleted and *del()* function is used to delete an edge. The major fields of `Edge` data type and their description is provided in Table 4.3.

4.4.3 Graph

field	type	description
npoints	var	number points in the `Graph` object ($\|V\|$).
nedges	var	number of edges in the `Graph` object ($\|E\|$).
read	function	read a `Graph` object.
getType	compile time function	Used to create a new `Graph` object with similar extra properties from a `Graph` object.
addPointProperty	function	add a new property to each vertex of the `Graph` object.
addEdgeProperty	function	add a new property to each edge of the `Graph` object.
addProperty	function	add a new property to the `Graph` object.
getWeight	function	get weight of an edge in the `Graph` object.
addPoint	function	add a new vertex to the `Graph` object.
addEdge	function	add a new edge to the `Graph` object.
delPoint	function	delete a vertex from the `Graph` object.
delEdge	function	delete an edge from the `Graph` object.

Table 4.4. Fields of `Graph` data type in `Falcon`

The major fields of `Graph` data type and their description is provided in Table 4.4. A `Graph` stores its points and edges in vectors *points[]* and *edges[]*. The method *addEdgePropery()* is used to add a property to each edge in a `Graph` object with the same syntax as that of *addPointProperty()* used in Line 17 of Algorithm 32.

The `addProperty()` method is used to add a new property to the whole `Graph` object (not to each `Point` or `Edge`). Such a facility allows a programmer to maintain additional data structures with the graph which are not necessarily direct functions of points and edges. For instance, such a function is used in DMR [26] code as the graph consists of a collection of *triangle*s, each *triangle* with three `Point`s, three `Edge`s along with a few extra properties. The statement shown below illustrates the way DMR code uses this function for a `Graph` object, *hgraph*.

hgraph.addProperty(*triangle*, `struct` *node*);
The structure *node* has all the fields which are required for the *triangle* property for the DMR implementation. This will add to *hgraph*, a new iterator `triangle` and a field `ntriangle` which stores the number of triangles.

Some other statements with a `Graph` object *hgraph* and their description is given in the table 4.5 with a short description.

statement	description
hgraph.read(fname)	read the `Graph` object with name of file stored in `char` array *fname*.
hgraph.addEdgeProperty(cost,int)	add an `int` property *cost* to each `Edge` of `Graph` object.
hgraph.getWeight(src,dst)	get weight of the `Edge` $src \rightarrow dst$ of the `Graph` object.
hgraph.addEdge(src,dst)	add an `Edge` $src \rightarrow dst$ to the `Graph` object.
hgraph.addPoint(P)	add a `Point` P to the `Graph` object.
hgraph.delEdge(src,dst)	delete an `Edge` $src \rightarrow dst$ of the `Graph` object.
hgraph.delPoint(P)	delete a `Point` P of the `Graph` object.
hgraph.getType() graph	creates a new `Graph` object graph, which inherits properties of the hgraph object.

Table 4.5. `Falcon` Statements with `Graph` fields

4.4.4 Set

A `Set` is a an aggregate of unique elements (e.g., a set of threads, a set of nodes, etc.). A `Set` has a maximum size and it cannot grow beyond that size. Two important operations on a `Set` data type that are used in graph algorithms are to *find* an element in a set and perform a *union* with another disjoint set (other set operations such as intersection and complement may be implemented in future versions of `Falcon`). Such a set is naturally implemented as a union-find data structure and we have also implemented it as suggested in [70], with our own optimizations. `Falcon` requires that *union()* and *find()* operations should not be called in the same method, because this may give rise to race conditions. The compiler gives a warning to the programmer in the presence of such codes. However we could not detect race condition. The *parent* field of a `Set` stores the representative key of each element in a `Set`. A `Set` data type can be used to implement, as an example, Boruvka's MST algorithm [90].

The way a `Set` data type is declared in MST code is shown in Algorithm 33. Line 2 declare

an object of `Set` data type. The `Set` object *hset* contains set of all the points in the `Graph` object *hgraph*. As edges get added to the MST, the two end points of the edge are union-ed into a single `Set`. The algorithm terminates when the `Set` has a single representative (assuming that the graph is connected) or when no edges get added to the MST in an iteration (for a disconnected graph, giving an MST forest). We mark all the edges added to the MST by using the `Edge` property, *mark* of the `Graph` object. This makes the algorithm a local computation, as the structure of the `Graph` does not change.

Algorithm 33: `Set` declaration in `Falcon`

```
1 Graph hgraph;
2 Set hset[Point(hgraph)];
```

Algorithm 34 shows how minimum weight edges are marked in the MST computation. Function *MinEdge()* takes three parameters: a `Point` to operate on, the underlying `Graph` object , and a `Set` of points. The `Point` which is the representative for the `Set` of p is stored in *t1* using the *find()* function in Line 11. Line 12 takes each outgoing neighbor of the `Point` p and finds the representative for the `Set` of outgoing neighbour t and stores it in *t2* (Line 13). Then algorithm checks whether those neighbors and p belong to different sets ($t1 \neq t2$). If so (Line 15), the code checks whether the edge ($p \rightarrow t$) has the minimum weight connecting the two sets *t1* and *t2* (Line 16). If it is indeed of minimum weight, the code tries to lock the `Point` *t1* using the `single` construct (See Section 4.6.1) in Line 17. If the locking is successful, this edge is added to the MST. After *MinEdge()* completes, each end-point of the edge which was newly added to the MST is put into the same `Set` using the *union* operation (performed in the caller).

4.4.5 Collection

A `Collection` refers to a multiset. Thus, it allows duplicate elements to be added to it and its size can vary (no maximum limit like `Set`). The extent of a collection object defines its implementation. If its scope is confined to a single function, then we use an implementation based on dynamic arrays. On the other hand, if a collection spans multiple function/kernel invocations, then we rely on the implementation provided by the Thrust library [74] for `GPU` and Galois worklist and its run time for multi-core `CPU`. Usage of Galois worklist for multi-core `CPU` made it possible to write many efficient worklist based algorithms in `Falcon`. Implementation of operations on `Collection` such as `reduction` and `union` will be carried out in the near future.

Delaunay Mesh Refinement [26] needs local `Collection` objects to store a cavity of bad

Algorithm 34: Finding the minimum weight edge in MST computation

```
1  minset(Point P,Graph graph, Set set[Point(graph)]) {
2  |  //finds an Edge with minimum weight from the Set to which Point P belongs to a
   |  different Set
3  }
4  mstunion(Point P,Graph graph, Set set[Point(graph)]) {
5  |  //union the Set of Point P with the Set of Point P' such that// Set(P)!=Set(P') and
   |  Edge(P,P') is the minimum //weight edge of P,going to different Set Performed only
   |  for the Point P that satisfies this condition.
6  }
7  MinEdge ( Point p, Graph graph, Set set[Point(graph)]) {
8  |  Point (graph) t1,(graph)t2;
9  |  int t3;
10 |  Edge (graph) e;
11 |  t1 = set.find(p);
12 |  foreach( t In p.outnbrs ){
13 |  |  t2 = set.find(t);
14 |  |  t3 = graph.getweight(p, t);
15 |  |  if (t1 != t2) {
16 |  |  if (t3 == t1.minppty.weight) {
17 |  |  single (t1.minppty.lock) {
18 |  |  |  e = graph.getedge(p, t);
19 |  |  |  e.mark = true;
20 |  |  } } }
21 |  }
22 }
```

triangles and to store newly added triangles. Hence, it can be implemented using dynamic arrays. Our implementation creates an initial array with a default size. When it gets full, it dynamically allocates another array of larger size, copies all the elements from the old array to the new array, and deallocates the old array. In general, repeated copying of elements is expensive. However, we significantly reduce this cost by repeated doubling of the array size. A `Collection` can be declared in the same way as a `Set`. A programmer can use *add()* and `del()` functions to operate on it and the current length of a `Collection` can be found using the *size* field of the data type. Algorithm 35 shows how *Collection* objects are used in DMR code. Lines 7 declares a `Collection` object with the name *pred* which contains elements of type `struct` *node*. `struct` *node* has fields to store values required for processing triangles in DMR.

Algorithm 35: `Collection` declaration in `Falcon`

```
1 struct node { //structure for triangle
2     Point nodes[3], neighedgestart[3];
3     struct_rec node neighbors[3];
4     int isbad,isdel,obtuse,owner,dims,index;
5 };
6 Graph hgraph;
7 Collection pred[struct node (hgraph)];
```

4.5 Variable declaration

Variable declarations in `Falcon` can occur in two forms as shown with `Point` variables `P0` and `P1` below (`Edge` declarations are similar). Given a `Graph` object g, we say that g is the `parent` of the points and edges in g.

`Point` P1, (*graph*)*P0*; //parent `Graph` of P0 is *graph*

When a point or edge variable has a parent `Graph` object, it can be assigned values from that parent only and whatever modifications we make to that object will be reflected in the parent `Graph` object. In the above example, P0 can be assigned values that are `Point` objects of *graph* only (see also line 8 of Algorithm 34). But If a variable is declared without a parent and a value is assigned to it, it will be copied to a new location and any modification made to that object will not be reflected anywhere else (e.g., P1 in the above example).

`Falcon` has a new keyword named `struct_rec`, that is used to declare recursive data structures. In C, a recursive data structure can be implemented using pointers and the *malloc()* library function. With `struct_rec`, a programmer can support a recursive data structure without explicitly using pointers, (like in `Java`). Line 3 of Algorithm 35 shows the usage of `struct_rec` field which declares a field type *node*, same as that of parent `struct` in which it is enclosed.

4.6 Parallelization and synchronization constructs

In `Falcon` we provide the `single` statement, `foreach` statement, `parallel sections` statement and reduction operations.

single(t1){ stmt block1 } **else** {stmt block2}	The thread that gets a lock on item t1 executes stmt block1 and other threads execute stmt block2.
single(coll){ stmt block1} **else** {stmt block2}	The thread that gets a lock on all elements in the collection executes stmt block1 and others execute stmt block2.

Table 4.6. Single statement in `Falcon`

4.6.1 `single` statement

This statement is used for *synchronization* across threads. It ensures mutual exclusion for the participating threads. In graph algorithms, we use a `single` statement to lock a set of graph elements, as discussed later in this section.

When compared to other synchronization constructs such as `synchronized` construct of Java or `lock` primitives in the pthreads library the `single` construct differs in two aspects: (i) it has a non-blocking entry, and (ii) only one thread executes the code following it.

`Falcon` supports two variants of `single`, as given in Table 4.6: with one item and with a `Collection` of items. In both the variants, the `else` block is optional (Algorithm 34, Line 17). The first variant tries locking one item. As it is a non-blocking entry function, if multiple threads try to get a lock on the same object, only one will be successful, others will fail. In the second variant, a thread tries to get a lock on a `Collection` of items given as an argument. This allows a programmer to implement cautious forms of algorithms wherein all the shared data (e.g., a set of neighboring nodes) are locked before proceeding with the computation. A thread succeeds if all the elements in the `Collection` object are locked by that thread. As an example, a thread in DMR code tries to get a lock on a cavity, which is a `Collection` of triangles. In both the variants, the thread that succeeds in acquiring a lock executes the code following it and if the optional `else` block is present, all the threads that do not acquire the lock execute the code inside the `else` block. If two or more threads try to get a lock on same element (present in `Collection` object of those threads), `Falcon` makes sure that the thread with the lowest *thread-id* always succeeds, by taking *minimum* with *thread-id* on the locking element. This avoids *live-lock* and ensures progress.

4.6.2 `foreach` statement

This statement is one of the *parallelizing* constructs in `Falcon`. It processes a set of elements in parallel. This statement has two variants as shown in Table 4.7. The `condition` and `advance_expression` are optional for both the variants. If the `condition` is present, the elements in the *object* which satisfy the `condition` will execute the *stmt_block* and others

foreach(item (advance_expression) **In** object.**iterator**) (condition) { stmt_block }	Used for **Point**, **Edge** and **Graph** objects
foreach(item (advance_expression) **In** object) (condition) { stmt_block}	Used for **Collection** and **Set** object

Table 4.7. `foreach` statement in `Falcon`

DataType	Iterator	Description
Graph	points	iterate over all points in graph
Graph	edges	iterate over all edges in graph
Graph	pptyname	iterate over all elements in new ppty.
Point	nbrs	iterate over all neighboring points
Point	innbrs	iterate over src point incoming edges (Directed `Graph`)
Point	outnbrs	iterate over dst point of outgoing edges (Directed `Graph`)
Edge	nbrs	iterate over neighbor edges
Edge	nbr1	iterate over neighbor edges of Point P1 in Edge(P1,P2) (Directed `Graph`)
and Edge	nbr2	iterate over neighbor edges of Point P2 in Edge(P1,P2) (Directed `Graph`)

Table 4.8. Iterators for `foreach` statement in `Falcon`

will not do any operation. Use of a `condition` was explained in Algorithm 32, Section 4.2. An `advance_expression` is used to iterate from a given position instead of the starting or ending positions. A `+ advance_expression` (`- advance_expression`, respectively) makes the iterations go in the forward (backward, respectively) direction, starting from the position given by the value of `advance_expression`. `advance_expression` is optional and its default value is taken as 0. If we want to iterate from the *end* position to the *beginning* position and from an *offset* before the *end*, `(- offset)` and if we want to iterate from the *beginning* to the *end* from an *offset* after the *begin*, we use `(+ offset)` as `advance_expression`. The *object* used by `foreach` can be also be dereferencing of a pointer to an object. The Boruvka's MST implementation uses `advance_expression` and dereferencing of a pointer to an object in `foreach` statements. A `foreach` statement gets converted to a CUDA kernel call or an `OpenMP pragma` or *Galois::worlist* call based on the object on which it is called and the target system. Iterators used in `foreach` statement for different `Falcon` data types are shown in Table 4.8.

In a `Graph`, we can process all the points and edges in parallel using *points* and *edges* iterator respectively. An iterator called `pptyname` is generated automatically when a new property is added to a `Graph` object using *addProperty()* function. This is used in the morph algorithms. When a property named *triangle* is added to a `Graph` object using `addProperty()`, it generates an iterator called *triangle*. Similarly, the `Point` data type has iterators *outnbrs*, which processes all outgoing neighbors in parallel. Iterators *nbrs* and *innbrs* process all the neighbors and incoming neighbors respectively in parallel. The `Edge` data type has iterator which processes all neighboring edges in parallel. *There is no nested parallelism in our language.* A nested `foreach` statement is converted to simple nested `for` loops in the generated code, except for the outermost `foreach` that is executed in parallel. The outermost `foreach` statement (executed in parallel) has an implicit global barrier after it (in the generated code).

4.6.3 `parallel sections` statement

Algorithm 36: `parallel sections` syntax in `Falcon`

```
1 parallel sections  {
2     section {
3          statement_block
4     }
5      one or more section statements //(Lines 2-4) above
6 }
```

The Syntax of this statement is shown in Algorithm 36. Each `section` inside the `parallel sections` statement runs as a separate parallel region. With this facility, `Falcon` can support multi-`GPU` systems and concurrent execution of CUDA kernels and parallel execution of `CPU` and `GPU` code is possible.

Algorithm 37: `parallel sections` example code in `Falcon`

```
1 parallel sections  {
2     section {
3          SSSP(graph);
4     }
5 section {
6           BFS(graph);
7     }
8 }
```

The code in Algorithm 37 shows an example of `parallel sections`. In the code there are

two parallel regions enclosed in two `section` statements (Lines 2-4 and Lines 5-7) with calls to the functions *SSSP* and *BFS* on the same input graph object *graph*. If there is more than one device (say 2 GPUs) on a machine, each section will be scheduled on a different device (separate GPU) concurrently and running time will be maximum of the running time of SSSP and BFS.

4.6.4 Reduction operations

Reduction operators such as `ReduxSum`, which sums a set of items and `ReduxMul` which multiplies a set of items are provided by `Falcon`. The syntax of the reduction operation is given below.

`if`(cond) res `ReduxOp`=obj.`ppty`;

Here, all the elements in the object *obj* are taken and for the elements which satisfy the condition *cond* specified in the `if` statement, the `ReduxOp` is assigned on the object property `ppty`. The condition *cond* is optional and if it is not present, `ppty` value of all elements in the object *obj* are used for the `ReduxOp`. The result is stored in variable *res*. An example `Falcon` DSL code is given below.

`if`(graph.edges.mark==1)mst `ReduxSum`=graph.edges.weight;

The above DSL code is from Boruvka's MST algorithm. The above code accumulates MST cost, after all the edges of the MST are computed. The `Graph` variable *graph* has an `Edge` property *mark*, which is set to 1 for an edge if the edge is a part of the MST. So, the above code accumulates the MST cost in the variable *mst* by adding the *weight* of the edges whose *mark* property value is set to 1 using the `ReduxSum` operation. Reduction makes finding properties on `Graph`s easy with a single DSL statement. We leave the support for arbitrary associative functions as reduction operations for future work.

4.7 Library functions

We provide atomic library functions MIN,MAX,SUB,AND, etc., which are abstractions over similar ones in CUDA [75] and GCC [5] . MIN atomic function was used in Algorithm 32, Section 4.2. We also provide a *barrier()* function which acts as a barrier for the entire group of threads in a CUDA kernel and `OpenMP` parallel region. A *genericbarrier()* which supports barriers for a group of related threads is also available. A *barrier()* is required for the entire kernel in algorithms like DMR. This is done automatically using the `single` construct of `Falcon`.

But in some graph algorithms, a *barrier()* may be required for the entire parallel region (outermost `foreach` loop), or for a set of threads (*genericbarrier()*), for example, after processing out-neighbours of a point.

Chapter 5

Code Generation for Single Node Machine

5.1 Overview

This chapter explains how the `Falcon` compiler generates code for a single node machine with multi-core CPU and one or more GPUs ($0 \leq \text{TARGET} \leq 2$, See Figure 4.1, Chapter 4). Distributed execution of graph algorithms using multiple GPUs on large-scale graphs are discussed in next chapter. This chapter discusses how `Falcon` can be used to run different algorithms on different GPUs at the same time, where the graph object fits within a single GPU. The `Falcon` compiler generates CUDA or C++ code from the input DSL code based on the target device given as an argument during compiling the DSL code.

Currently, `Falcon` supports two types of graph representation: (i) Compressed Sparse Row (CSR) format, and (ii) Coordinate List (COO) or Edge List format. Graphs are stored as C++ classes in `Falcon` generated code. The GGraph and HGraph C++ classes are used to store graph objects on the GPU and the CPU respectively, and both inherit from a parent Graph class. The Graph class has a field named *extra* (of type `void *`) which stores all the properties added to a `Graph` object using *addPointProperty()*, *addEdgeProperty()*, and *addProperty()* methods of the `Graph` class. The `Point` and `Edge` data types can have either integer (default) or floating point values and are stored in a `union` type with the fields *ipe* and *fpe* respectively. The generated code is compiled with `nvcc` or `g++` compiler. The `Falcon` compiler names for all data types and functions specific to CPU and GPU start with H(Host) and G(Gpu) respectively in the generated code.

Figure 5.1 gives an overview of how parallelization and synchronization constructs are converted to CUDA (C++) high-level language code with appropriate libraries for GPU (CPU).

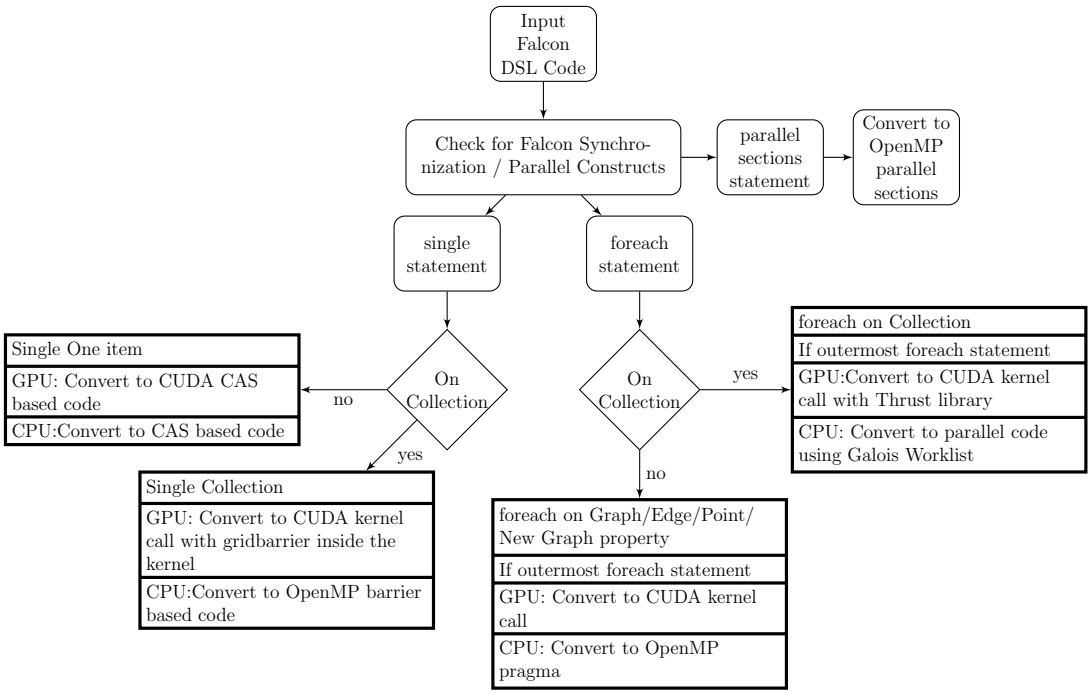


Figure 5.1: Falcon Code Generation overview for Parallelization and Synchronization Constructs for Single Node Device

As shown in the figure only outer the most `foreach` loop is parallelized. If the `foreach` loop is over a `Collection` object, `Falcon` generates a code which uses *Galois* library for the multi-core CPU and CUDA code with *thrust* library for the GPU. Galois has efficient parallel iterators over worklists and it has worklist based implementations many graph and mesh refinement algorithms. For `foreach` statements over other data types, `OpenMP` based code is generated for CPU and CUDA code is generated for GPU. If the `single` statement is on one element compare and swap (CAS) based code is generated for GPU and GPU. For `single` on collection parallel code with barrier for all the threads is generated.

`Falcon` is strongly typed. The compiler checks for undeclared variables, type mismatch involved in an assignment, invalid iterator usage, invalid field-access, invalid property, and usage of the supported data-types (such as `Collection`). `Falcon` just gives a warning when an undefined function is called in the DSL code. It does not treat this as an error, as a programmer is free to call any C library function. It is mandatory for the programmer to write the *main()* function in `Falcon` DSL code. If it is missing, it is treated as an error. Graph objects are passed by reference to all the functions in the generated code.

5.2 Code generation for data structures

5.2.1 Point and Edge

Each `Edge` in `Falcon` stores the destination `Point` and the *weight* (if required) of the `Edge` in *edges* array of the Graph class. When a program uses the `innbrs` and the `outnbrs` iterators, the *inedges* array of the `Graph` class stores two fields: source `Point` of the incoming `Edge` and an index into the *edges* array, that is used to get the weight of the incoming `Edge` which is stored in *edges* array. A `Point` can have upto three dimensions and have fields x, y and z to store the value of each dimension. In the DMR algorithm, `Point` data type is two dimensional with x and y fields.

Algorithm 38: Extra properties added to `Graph` object *graph*

```
1 graph.addPointProperty(dist,int);
2 graph.addPointProperty(olddist,int);
3 graph.addPointProperty(uptd,int);
```

5.2.2 Allocation of extra-properties

`Point` and `Edge` of a `Graph` object are converted to integer ids. All the extra-properties of a `Graph` object are stored in the *extra* field, and can be type cast to any structure. By default, extra-properties of a `Graph` object are stored in a structure with the name *struct_objectname* and are assigned to the *extra* field of a `Graph` object. If a `Graph` object is created by the `getType()`, a compile time function, its extra-properties are assigned to a structure with the name *struct_parentobjectname*, which will have fields for extra-properties of the parent object and all the objects created by the `getType()` compile time function.

By default extra-properties are stored in a structure with the name struct_objectname and assigned to the *extra* field of the graph object. If two objects have the same extra-properties, both use the same structure. In the SSSP example, Graphs on the GPU and the CPU have the same extra-properties and are allocated in a structure with the same name. Algorithm 39 shows how extra-properties of a graph object on the GPU and the CPU are allocated for the statements in Algorithm 38. Same properties were used in SSSP example of previous chapter (Algorithm 32, Section 4.2). The extra-properties of the CPU (GPU) graph object are allocated using the *malloc()* (*cudaMalloc()*) function and are assigned to the *extra* field of the CPU (GPU) graph object.

When a property is added to a graph object using *addProperty()* method of the `Graph` data type, a function is generated by the `Falcon` compiler which gets the size of the property added

Algorithm 39: Allocating extra-property for `Graph` object on `GPU` and `CPU`

```
1  #define ep (struct struct_hgraph )
2  struct struct_hgraph {
3    int *dist, *olddist;
4    bool *uptd;
5  };
6  struct struct_hgraph tmp;
7  alloc_extra_graph(GGraph &graph) {
8  |   cudaMalloc((void **) &(graph.extra), sizeof (ep ));
9  |   cudaMemcpy(&tmp, (ep *)(graph.extra), sizeof (ep),cudaMemcpyDeviceToHost);
10 |   cudaMalloc((void **) &(tmp.dist), sizeof (int)* graph.npoints);
11 |   cudaMalloc((void **) &(tmp.olddist), sizeof (int)* graph.npoints);
12 |   cudaMalloc((void **) &(tmp.uptd), sizeof (bool)* graph.npoints);
13 |   cudaMemcpy(graph.extra, &tmp, sizeof(ep), cudaMemcpyHostToDevice);
14 }
15 alloc_extra_graphcpu(HGraph &graph) {
16 |   graph.extra= ( ep *) malloc(sizeof(ep));
17 |   ((ep *)(graph.extra))→dist=(int *) malloc(sizeof(int)*graph.npoints);
18 |   ((ep *)(graph.extra))→olddist=(int *) malloc(sizeof(int)*graph.npoints);
19 |   ((ep *)(graph.extra))→uptd=(bool *) malloc(sizeof(bool)*graph.npoints);
20 }
```

from command line argument stored in $argv[]$ array. An example for the generated code is given in Algorithm 40 for the statement shown in the first two lines of the same Algorithm, which adds an `int` property *changed* to the `Graph` object *hgraph* and creates a new `Graph` object *graph* using *getType()* method.

The *hgraph* and *graph* objects are stored on the CPU and the GPU respectively. The read_hgraph_pptysize() (Line 9) reads the size (number of elements) of the `Graph` property *changed* using the command line arguments given by the user ($argv[]$). This value is read into *nchanged* field of the structure used to store extra-properties of *hgraph* (Line 10). Then in the *alloc_extra_hgraph()* (Line 12) function the extra-property *changed* is allocated using the value of *nchanged* (Line 14). In the *main()* function the *extra* field of *hgraph* object is allocated (Line 25) first, followed by calls to *read_hgraph_pptysize()* (Line 27) and *alloc_extra_hgraph()* (Line 28) respectively. Then the *extra* field of the GPU variable *graph* is allocated (Line 30), the value of *nchanged* is copied from *hgraph* to *graph* (Line 33). Then the *graph* object extra properties are allocated using a call to *alloc_extra_graph* (Line 35).

Algorithm 40: Property allocation for `addProperty()` method of `Graph` class

```
hgraph.addproperty(changed,int);
hgraph.getType() graph;
---------------------------------------------------------------
#define ep (struct struct_hgraph )
struct struct_hgraph {
int nchanged;
int *changed ;
};
void read_hgraph_pptysize(HGraph &hgraph, char *argv[],int pos) {
    (((ep *)(hgraph.extra))→nchanged)=atoi(argv[pos]);
}
void alloc_extra_hgraph(HGraph &hgraph,int flag) {
    if(flag==0)hgraph.extra= (ep *) malloc(sizeof(ep)) ;
    ((ep *)(hgraph.extra))→changed=(int *)malloc(sizeof(int ) * ((ep
    *)(hgraph.extra))→nchanged) ;
}
void alloc_extra_graph(GGraph &graph,int flag) {
    struct struct_hgraph temp;
    if(flag==0)cudaMalloc((void **)&(graph.extra),sizeof(ep));
    cudaMemcpy(&temp,((ep *) (graph.extra)),sizeof(ep),cudaMemcpyDeviceToHost);
    cudaMalloc((void **)&( temp.changed),sizeof(int )* temp.nchanged);
    cudaMemcpy(graph.extra,&temp,sizeof(ep),cudaMemcpyHostToDevice);
}
 main(int argc, char *argv[]) {
    .....
    hgraph.extra= ( ep *) malloc(sizeof(ep));
    .....
    read_hgraph_pptysize(hgraph,argv,pos);
    alloc_extra_hgraph(hgraph,1);
    .......
    cudaMalloc((void **)(&graph.extra),sizeof(ep ));
    struct struct_hgraph ftemp1;
    cudaMemcpy(&ftemp1,graph.extra,sizeof(ep ),cudaMemcpyDeviceToHost);
    ftemp1.nchanged=((ep *)(hgraph.extra))→nchanged;
    cudaMemcpy(graph.extra,&ftemp1,sizeof(ep ),cudaMemcpyHostToDevice);
    alloc_extra_graph(graph,1);
}
```

5.2.3 Collection

A `Collection` that spans across multiple functions is implemented using the *Thrust Library* for GPU, and the *Galois worklist* along with its runtime code for CPU. This made it possible to have worklist based implementation of Boruvka's MST and Δ-Stepping SSSP algorithms in `Falcon` DSL for multi-core CPU.

Usage of a `Collection` in `Falcon` is shown in Algorithm 41 and the code generated by `Falcon` that uses the Thrust library (for GPU) is shown in Algorithm 42.

Algorithm 41: `Collection` declaration in `Falcon`

```
1 Graph graph;
2 Collection coll1[Point(graph)];
```

Algorithm 42: Code generated for `Collection` on `GPU` by `Falcon`

```
1 int *coll1;
2 thrust::device_vector<int>collfalctemp(graph.npoints);
3 coll1= thrust::raw_pointer_cast(&collfalctemp[0]);
```

Algorithm 43: `Falcon` DSL code for Collection on Multi-core `CPU`

```
1 Graph hgraph;
2 struct node{
3       Point(hgraph) n1;
4      int w;
5 };
6 Collection pred[struct node];
7 pred.OrderByIntValue(w,10);
8 foreach(t In pred)relaxNode1(t,hgraph,pred);
```

Algorithm 43 shows how the `Collection` data type is used in the Δ-stepping SSSP implementation and the code generated by the `Falcon` compiler is shown in Algorithm 44. *Galois InsertBag* is a worklist with OBIM (Order By Integer Matrix) (Line 16, Algorithm 44) and it is a set of buckets. This is specified in `Falcon` DSL code using *OrderByIntValue()* function (Line 7, Algorithm 43) of `Collection` data type. The value 10 given as an argument to *OrderByIntValue()* is converted to 2^{10}=1024 in the generated code (Line 3, Algorithm 44), making the value of Δ=1024. The function *relaxnode1()* gets called using *Galois:for_each_local()* (Line 18, Algorithm 44) iterator which calls the *operator()* function of *struct Process0* using OBIM scheduling policy. This is a parallel call and *Process0()* will call *relaxNode1()*, which is the argument to `foreach` call on `Collection` object *pred* (Line 8, Algorithm 43) in the DSL

code.

Algorithm 44: Code generated for Collection on Multi-core `CPU`

```
1  struct nodeIndexer:public std::unary_function<struct node,unsigned int> {
2      unsigned int operator()(const struct node &val)const {
3          unsigned int t=val.w/1024;
4          return t;
5      }
6  }
7  template< typename Pusher0>
8  struct Process0 {
9      Process0(){}
10     void operator()(struct node &req,Pusher0 &pred ){
11         relaxNode1(req,hgraph,pred);
12     }
13 };
14 using namespace Galois::WorkList;
15 typedef dChunkedFIFO<64> Chunk;
16 typedef OrderedByIntegerMetric<struct nodeIndexer,Chunk,10> OBIM;
17 Galois::InsertBag<struct node> pred ;
18 Galois::for_each_local(pred,Process0(),Galois::wl<OBIM>());
```

5.2.3.1 Support for collection without duplicates

Algorithm 45: `Falcon` code for `Collection` without duplicates

```
1  Collection coll[Point(graph)];
2  //add function, make sure the no duplicates in Collection
3  add(Graph graph, Collection coll[Point(graph)], Point p) {
4  |   single(p1.lock)coll. add(p1);
5  }
6  //delete Point
7  del( Graph graph, Collection coll[(graph)], Point p) {
8  |   Point (graph) P1=coll.del();
9  |   p1.lock=0;
10 }
```

An example `Falcon` code is shown in Algorithm 45. There is a property by name *lock* associated with `Collection`, which is automatically added by the `Falcon` compiler. Add and del are user defined functions collection on top of the add and del function of the `Collection` class. The add function first tries to get the lock on adding Point's `lock` property, using he single statement. If successful the basic add function of the `Collection` is called. This ensures that

each `Point` is added only once. Similarly del function first deletes the point from `Collection` and resets the lock property to zero after processing the element, so that it can be added later.

Algorithm 46: Pseudo Code for Set

```
1  Linkset(int t1,int t2) {
2      if( t1<t2 ){
3          x1=CAS(&(parent[t1]),t1,t2);
4          if(x1==t1&& parent[t1]==t2)return t2;
5          return 0;
6      }
7      if( t2≤ t1 ){
8          x1=CAS(&(parent[t2]),t2,t1);
9          if(x1==t2&&parent[t2]==t1)return t1;
10         return 0;
11     }
12 }
13 FindSet(int n1) {
14     if(parent[n1]!=n1)
15       parent[n1]=Findset(parent[n1]);
16     return parent[n1];
17 }
18 void allocate(int n) {
19     size=n;
20     parent=(int *)malloc(sizeof(int)*(n));
21     for( (int i=0;i<n;i++) ){
22         (parent[i])=i;
23     }
24 }
```

Algorithm 47: Pseudo Code for Set

```
1  Findcompress(int n1) {
2      if(parent[n1]==n1)return n1;
3      int rep=parent[n1];
4      int prev=-1;
5      while( parent[rep]≠rep ){
6          int next=parent[rep];
7          if(prev≥0 && par-
           ent[prev]==rep)parent[prev]=next;
8          prev=rep;
9          rep=next;
10     }
11     return rep;
12 }
13 Union(int n1,int n2) {
14     int t1,t2;
15     t1=n1;t2=n2;
16     do {
17         do {
18             t1=Findcompress(t1);
19             t2=Findcompress(t2);
20             if(t1==t2) return ;
21         }while(t1≠parent[t1]||
       t2≠parent[t2]);
22     }while(!(LinkSet(t1,t2)));
23 }
```

5.2.4 Set

The `Falcon` compiler has two C++ classes HSet and GSet which implement the CPU and GPU `Set` data types (resp.). Each of these classes has the same functions named, `union()` to union two sets and `find()` to find the representative key of an element, called *parent*. By default, the *parent* for a set will be an integer number, which denotes the maximum value of an element in

that set. For example, for a set of first ten `Points` (0-9) in a graph object, all the elements will have the *parent* as 9. When the union of two sets with *parents* V1 and V2 is performed, a single set with *parent* being max(V1,V2) is formed. The code for updating the *parent* of individual elements to max(V1,V2) is automatically generated by the `Falcon` compiler.

The `Set` data type is implemented as a Union-Find [42] data structure. The CPU and GPU versions are stored in HSet and GSet C++ classes. Both these classes have the functions *Allocate()*, *LinkSet()*, *FindSet()* and *Union()* as given in psuedo codes of Algorithms 46 and 47. The *FindSet()* and *Union()* correspond to *find* and *union* operations of Union-Find data structure. The *FindSet()* function is used to find the *parent* of an element in the set. It is also used to update the *parent* value of elements after a *Union()* operation of two or more sets. The *LinkSet()* function is used to update the parents of two sets in a *Union()* operation to the maximum of the *parent* values of the two sets. The *allocate()* function allocates the *parent* field of the set whose size is argument to the function. Then it sets for each i, 0<size, parent[i]=i.

5.3 Translation of statements

5.3.1 `Foreach` statement code generation

Code generation for a `foreach` statement depends on the object on which it is called and where (GPU/CPU) the object is allocated. Nested parallelism using `foreach` is not supported. We convert inner `foreach` statements of nested `foreach` statements to simple `for` loop statements during code generation. The `foreach` statement inside the *relaxgraph()* function (Algorithm 32, Section 4.2) processes all the neighbors of a `Point` serially, using a simple `for` loop. The outermost loop is converted to a CUDA kernel call / `OpenMP` pragma (except for a `Collection` on CPU) in the generated code. `Falcon` stores the beginning index of neighbors of a `Point` p in the *index* field of the `Graph` class and the *total* neighbors of the `Point` p is calculated by taking the difference of *index[p+1]* and *index[p]*.

We explain the code generation of `foreach` using an example. We present the `Falcon` DSL code for Breadth First Search (BFS), with no atomic operations and the BFS distance is calculated on a *level* base (Algorithm 48). The `Graph` object *graph* is added with a `Point` property *dist* (Line 13) and then the input graph object is *read* (Line 14). The BFS distance of all the vertices is made ∞ (a maximum value), by the parallel `foreach` call on all points of the graph object (Line 15).

Algorithm begins with initialization of the BFS distance of the source-vertex to zero (Line 16). Then the algorithm computes the BFS distance of all the vertices in the `while` loop (Lines 17-22). For the computation of the BFS distance, an integer variable *lev* is used, which is ini-

Algorithm 48: Level Based BFS code in `Falcon`

```
1  int changed = 0;
2  relaxgraph(Point p, Graph graph,int lev) {
3      foreach( t In p.outnbrs ){
4          if( t.dist>(lev+1) ){
5              t.dist=lev+1;
6              changed=1;
7          }
8      }
9  }
10 main(int argc, char *argv[]) {
11     Graph graph; // graph object
12     int lev=0;
13     graph.addPointProperty(dist, int);
14     graph.read(argv[1]);
15     foreach (t In graph.points)t.dist=1234567890;
16     graph.points[0].dist = 0; // source has dist 0
17     while( 1 ){
18         changed = 0; //keep relaxing on
19         foreach(t In graph.points) (t.dist==lev) relaxgraph(t,graph,lev);
20         if(changed == 0)break;//reached fix point
21         lev++;
22     }
23     for(int i = 0; i <graph.npoints; ++i)printf("i=%d dist=%d\n", i, graph.points[i].dist);
24 }
```

tialized to zero and incremented by 1 at the end of each iteration. All the vertices whose distance is equal to *lev* are processed in each iteration of the `while` loop. This is done by the `foreach` statement which contains a call to the *relaxgraph()* function (Line 19) with the condition ($t.dist == lev$). In the *relaxgraph()* function a `Point` p which has *dist* value *lev* takes all its neighbouring (*outnbrs*) and reduces their *dist* value to *(lev+1)*, if it is currently greater than *(lev+1)* implying that the neighbour is still unexplored (Line 5). At the beginning of the `while` loop, variable *changed* is made zero (Line 18). The variable *changed* is made 1, if *dist* value of any one vertex is reduced (Line 6). The loop iterates until BFS distance of all the reachable vertices from the source vertex are computed. Once the BFS distance of all the reachable vertices is computed, there will be no vertex whose distance gets reduced in next iteration of loop, and the value of the variable *changed* is not modified and loop exits.

In this algorithm, there will be two or more threads which may write to the same location. This can happen in an iteration when the *lev* value is x, and there are two vertices u and v with value of dist x and both vertices have a common outneighbour *outnbr* w which is currently

not visited (*dist* value is ∞). Then the threads for u and v will modify the *dist* value of the vertex w simultaneously using the edges u→w and v→w to *(lev+1)*. Here an atomic operation is not required as all the threads write the same value for all the vertices whose *dist* value is reduced. This concept originates from Concurrent Read Concurrent Write (CRCW) of PRAM model [90], which says: *as far as all threads are writing same value, the atomic operation can be removed*. BFS can be computed in a similar manner as given in Algorithm 32, Section 4.2 with the MIN atomic operation and weight of edge p→t replaced by 1 in the *relaxgraph()* kernel.

Algorithm 49: Code generated for `GPU` BFS relaxgraph() and its call.

```
1  #define t (((struct struct_hgraph *)(graph.extra)))
2  __global__ void relaxgraph(GGraph graph,int lev, int x) {
3      int id = blockIdx.x * blockDim.x + threadIdx.x + x;
4      if( id <graph.npoints && t→dist[id] == lev ){
5          int falcft0 = graph.index[id];
6          int falcft1 = graph.index[id+1]-graph.index[id];
7          for( (int falcft2 = 0; falcft2 <falcft1; falcft2++) ){
8              int ut0 = (falcft0 + falcft2); //edge index
9              int ut1 = graph.edges[ut0].ipe; //dest point
10             if( t→dist[ut1]>(lev+1) ){
11                 t→dist[ut1]=lev+1;
12                 changed=1;
13             }
14         }
15     }
16 }
17 int flcBlocks=(graph.npoints/TPB+1)>MAXBLKS?MAXBLCKS:(graph.npoints/TPB+1);
18 for(int kk=0;kk<graph.npoints;kk+=TPB*flcBlocks)
19       relaxgraph <<<flcBlocks,TPB >>>(graph,lev, kk);
```

Algorithm 49 shows the code generated for the *relaxgraph()* function and its `foreach` statement in the *main()* function in Algorithm 48, with the target being a GPU. Since `foreach` statement inside *relaxgraph()* is nested inside another `foreach` statement from *main()*, the `foreach` statement in *relaxgraph()* is converted to a simple for loop. The *index* field of the graph object stores, for a vertex v, an index into the *edges* array, whose value is the position of the first outgoing edge with source vertex v. The *edges* array stores the edges sorted by source-vertex-id of the edges in the graph. It stores destination the vertex and weight (if required) of the edge in adjacent locations. So the size of (*edges*) will be $2 \times |E|$ (with weight) or $|E|$ (without weight). In algorithms such as PageRank and Connected Components, the weight of the edges are not required. Also in Algorithm 48 for BFS, edge weight is not needed and in the generated code shown in Algorithm 49, weight is not stored in the *edges* array.

Algorithm 50: Code generated for `CPU` BFS relaxgraph() and its call

```
1  #define t (((struct struct_hgraph *)(graph.extra)))
2  void relaxgraph(int &p ,HGraph &graph) {
3      if( id <graph.npoints &&t→dist[id] == lev ){
4          int falcft0 = graph.index[id];
5          int falcft1 = graph.index[id+1]-graph.index[id];
6          for( (int falcft2 = 0; falcft2 <falcft1; falcft2++) ){
7              int ut0 = (falcft0 + falcft2); //edge index
8              int ut1 = graph.edges[ut0].ipe; //dest point
9              if( t→dist[ut1]>(lev+1) ){
10                 t→dist[ut1]=lev+1;
11                 changed=1;
12             }
13         }
14     }
15 }
16 #pragma omp parallel for  num_threads(TOT_CPU)
17 for (int i = 0; i <graph.npoints; i++)relaxgraph(i, graph);
```

The starting index of the edges of the vertex *id* is found from the *index* array and stored in the variable *falcft0* (Line 5). Total number of outgoing edges for the vertex *id* is obtained by taking the difference of *index[(id+1)]* and *index[id]* and is stored in the variable *falcft1* (Line 6). The `for` loop (Line 7) processes all the outgoing edges of the vertex *id* stored in the index *falcft0* to *(falcft0+falcft1-1)* of the *edges* array. The edge index is first copied to the variable *ut0* (Line 8), and then the destination vertex is stored in the variable *ut1* (Line 9). Then the distance of the destination vertex is reduced (Line 11), if it is currently greater than *(lev+1)*.

The `foreach` statement which calls the *relaxgraph()* (Line 19, Algorithm 48) gets converted to the CUDA code shown in (Lines 17-19, Algorithm 49), which calls the *relaxgraph()* function in a `for` loop. The variable TPB (Threads Per Block) corresponds to
(MaxThreadsPerBlock - MaxThreadsPerBlock % CoresPerSM)
for the GPU device on which the CUDA kernel is being called. For example, the Nvidia Kepler GPU with MaxThreadsPerBlock=1024 and CoresPerSM=192, value of TPB=(1024-1024%192)=960. We also make sure that a kernel executes by splitting a kernel call into multiple calls, if the number of thread-blocks for the kernel call is above the allowed value for the device, which is stored in *MAXBLKS* variable. `Falcon` uses the *cudaGetDeviceProperty()* function of CUDA to read the device parameters. These are needed in deciding the maximum threads within a single block in a CUDA kernel call and the maximum number of threads in a CUDA kernel call which require a *barrier()* for the entire kernel.

Algorithm 50 shows the code generated for the *relaxgraph()* function and its parallel call (Line 19, Algorithm 48) when the target is a multi-core CPU. The variable TOT_CPU (Line 16) stores the number of CPU cores available. The parallel call gets converted to an `OpenMP` *pragma* for a multi-core CPU (Line 16). `Graph` type is converted to HGraph or GGraph based on where it is allocated. This convention is used throughout `Falcon`.

Algorithm 51: Function generated by `Falcon` for `foreach` call in Line 15, Algorithm 48

```
1 __global__ void falctfun0(GGraph graph, int x) {
2     int id = blockIdx.x * blockDim.x + threadIdx.x + x;
3     if( id <graph.npoints ){
4         (((struct struct_hgraph *)(graph.extra)))→dist[id]=1234567890;
5     }
6 }
7 int flcBlocks=(graph.npoints/TPB+1)>MAXBLKS?MAXBLCKS:(graph.npoints/TPB+1);
8 for(int kk=0;kk<graph.npoints;kk+=TPB*flcBlocks)
9       falctfun0 <<<flcBlocks,TPB >>>(graph, kk);
```

Line 15 of Algorithm 48 initializes the *dist* property of all the vertices of the graph object to ∞ using a `foreach` statement, which is a structure access *t.dist*, that is not enclosed inside a function. For GPU devices, such a code should be converted to a kernel call and a kernel should be a function. The `Falcon` compiler creates a new function named *falctfun0* which takes as argument a graph object. Inside the function the *dist* property is initialized to ∞ and this function is called from the host (CPU). The generated code and the kernel call are shown in Algorithm 51. Similar code is generated for other iterators.

We have experimented with warp-based code generation as well. However, we found that performance improvement is not always positive across benchmarks.

5.3.2 Inter-device communication

Copying data between the CPU and the GPU is translated to *cudaMemcpy* operation which has different forms for the various assignment statements in `Falcon`. When an entire property of a `Graph` object , say `Point` or `Edge` property is copied from GPU or to GPU, a *cudaMemcpy* operation is called to transfer a block of data. `Falcon` allows direct usage of GPU variables of basic types such as `int`, `bool` etc. inside CPU code. These statements will be converted to *cudaMemcpyFromSymbol* (Line 20,Algorithm 48) and *cudaMemcpyToSymbol* (Line 18, Algorithm 48) for data transfer from GPU and to GPU respectively, using compiler generated temporary variables.

In the BFS() example, *dist* property of all the points are printed after BFS computation (Line 23, Algorithm 48). So if the computation happens on GPU device, the *dist* value of all the points of the graph object are copied to temporary array *FALCtempist* as shown in Algorithm 52. Then the printf prints value in *FALCtempdist* array. The *FALCtempdist* array is allocated on the CPU using the *malloc()* function.

Algorithm 52: Prefix code generated for Line 23 in Algorithm 48

```
1 #define ep (struct struct_hgraph)
2 struct struct_hgraph temp3;
3 cudaMemcpy(temp3, (ep *)(graph.extra), sizeof(ep), cudaMemcpyDeviceToHost);
4 cudaMemcpy(FALCtempdist, temp3.dist, sizeof(int) * graph.npoints,
  cudaMemcpyDeviceToHost);
```

The above statement needs two cudaMemcpy operations because *graph.extra* is a *device* (GPU) location and we cannot access *graph.extra.dist* in cudaMemcpy, as this implies dereferencing a *device* location (something that cannot be done from the host).

Algorithm 53: Code generated for Line 20 in Algorithm 48

```
1 int falctemp4;
2 cudaMemcpyFromSymbol((void *)&falctemp4,changed,
  sizeof(int),0,cudaMemcpyDeviceToHost);
3 if( falctemp4==0 )break;
```

The statement
`if`(changed==0)`break` //(Line 20, Algorithm 48)
will be preceded by a *cudaMemcpyFromSymbol* operation which copies value of *changed* variable on *device* to a temporary variable *falctemp4* on *host* (CPU) and the `if` statement uses this temporary variable in condition instead of *changed* as shown in Algorithm 53.

Algorithm 54: Code generated for Line 18 in Algorithm 48

```
1 int falcvt3=0;
2 cudaMemcpyToSymbol(changed,&(falcvt3), sizeof(int),0,cudaMemcpyHostToDevice);
```

Similarly the statement
changed=0; // (Line 18, Algorithm 48)
assigns the GPU variable *changed* a value 0. In the generated code a temporary variable *falcvt3* initialized to zero is copied to the GPU variable *changed* as shown in Algorithm 54.

Recent advances in GPU computing allow access to a unified memory across CPU and GPU (e.g., in CUDA 6.0 and Shared Virtual Memory in OpenCL 2.0 and AMD's HSA architecture). Such a facility clearly improves programmability and considerably eases code generation. However, concluding about the performance effects of a unified memory would require detailed experimentation. For instance, CUDA's unified memory uses pinning pages on the host. For large graph sizes, pinning several pages would interfere with the host's virtual memory processing, leading to reduced performance. We defer the issue of unified memory in Falcon to a future work.

5.3.3 `parallel sections`, multiple GPUs and Graphs

`Falcon` supports concurrent kernel execution using `parallel sections`. `Falcon` also supports multiple GPUs and multiple Graphs. When multiple GPUs are available and multiple GPU `Graph` objects exist in the input program, each `Graph` object will be assigned a GPU number in a round robin fashion by the `Falcon` compiler. A GPU is assigned more than one `Graph` object if the number of GPU `Graph` objects exceeds the total number of GPUs available. `Falcon` *assumes that a* `Graph` *object fits completely within a single* GPU and proceeds with code generation. If there is more than one GPU `Graph` object, object allocation and kernel calls will be preceded by a call to *cudaSetDevice()* function, with the GPU number assigned to the object as its argument. It is possible to execute either the same algorithm or different algorithms on the `Graph` objects in the various GPUs.

For parallel kernel execution on different GPUs, each `foreach` statement should be placed inside a different `section` of the `parallel sections` statement. The `parallel sections` statement gets converted to an `OpenMP` parallel region pragma, which makes it possible for the code segments in different sections inside the `parallel sections` to run in parallel. The method that we use for assigning Graphs to different GPUs is not optimal and the search for a better one is part of future work. The code fragment in Algorithm 55 shows how SSSP and BFS are computed at the same time on different GPUs using a `parallel sections` statement of `Falcon`. An Important point to be noted here relates to how the variable *changed* is used in the code. if we declare *changed* as shown in Line 1 of Algorithm 55 , it will be allocated in GPU device 0. So, to ensure that *changed* appears in each device, it is added as a graph property (Line 5). The allocation of *changed* extra-property on CPU and GPU follow the code pattern given in Algorithm 40, Section 5.2.2. The device on which each graph object needs to be allocated can be specified as a command line argument during `Falcon` code compilation.

Algorithm 55: Multi-GPU BFS and SSSP in Falcon.

```
1  int changed;
2  SSSPBFS(char *name) { //begin SSSPBFS
3  Graph graph;//Graph object on CPU
4  graph.addPointProperty(dist,int);
5  graph.addProperty(changed,int);
6  graph.getType() graph0;//Graph on GPU0
7  graph.getType() graph1;//Graph on GPU1
8  graph.addPointProperty(dist1,int);
9  graph.read(name);//read Graph from file to CPU
10 graph0=graph;//copy entire Graph to GPU0
11 graph1=graph;//copy entire Graph to GPU1
12 foreach(t In graph0.points)t.dist=1234567890;
13 foreach(t In graph1.points)t.dist=1234567890;
14 graph0.points[0].dist=0;
15 graph1.points[0].dist=0;
16 parallel sections { //do in parallel
17     section {//compute BFS on GPU1
18         while(1){
19             graph1.changed[0]=0;
20             foreach(t In graph1.points)BFS(t,graph1);
21             if(graph1.changed[0]==0) break;
22         }
23     }
24     section {//compute SSSP on GPU0
25         while(1){
26             graph0.changed[0]=0;
27             foreach(t In graph0.points)SSSP(t,graph0);
28             if(graph0.changed[0]==0) break;
29          }
30     }
31 }
31 }//end SSSPBFS
```

Algorithm 56: Usage of single statement in DMR(Pseudo code)

```
1  refine(Graph graph,triangle t) {
2      Collection triangle[pred];
3      if( t is a bad triangle and not deleted ){
4          find the cavity of t(set of surrounding triangles)
5          add all triangles in cavity to pred
6      }
7      single(pred){
8      //statements to update cavity
9      }else
10     {
11     //abort
12     }
13 }
```

5.3.4 Synchronization statement

The `single` statement is used for synchronization in `Falcon`. The second variant of the `single` statement (Secton 4.6.1, Chapter 4) is needed in functions which make structural modifications to graphs (cautious morph algorithms) and it requires a barrier for the entire function to be inserted automatically during code generation. The total number of threads inside a CUDA kernel with a grid barrier cannot exceed a value specific to the GPU device and so these functions run in such a way that one thread processes more than one element. Cautious functions need `single` to be called on a collection object, which can contain a set of points or edges of the graph object. `single` should be called before any modification to the graph object elements (points, edges etc.), properties stored in the collection object, and no new elements can be added to the collection object after the `single` statement. The `Falcon` compiler performs this check and if this condition is violated the user is warned about possible incorrect results.

There is no support for grid barrier in CUDA and we have implemented it as given in [100]. The CPU code uses the barrier provided by `OpenMP`, which acts as a barrier for all the worker threads. The way a `single` statement is used in DMR is shown in Algorithm 56. Here *pred* is a `Collection` object which stores the set of all triangles in the cavity. If a *lock* is obtained on all the triangles in *pred* by a thread, then it updates the cavity else it aborts.

Pseudo Code in Lines 7-12 of Algorithm 56 get converted to the CUDA code shown in Algorithm 57. Both GPU and CPU versions follow the above code pattern, with appropriate GPU and CPU functions. We lock the triangles based on the thread-id and if two or more cavities overlap, only the thread with the lowest thread-id will succeed in locking the cavity and others abort. The global barrier makes sure that the operations of all the threads are

Algorithm 57: Generated CUDA code

```
1  #define t ((struct struct_graph *)(graph.extra))
2  for(int i=0;i<pred.size;i++) t→owner[pred.D_Vec[i]]=id;
3  gpu_barrier(++goal,arrayin,arrayout);//global barrier
4  for( (int i=0;i<pred.size;i++) ){
5  |  if((t→owner[pred.D_Vec[i]]<id) break;//locked by lower thread,exit
6  |  else if(t→owner[pred.D_Vec[i]]>id) t→owner[cav1]=id;//update lock with lower id
7  }
8  gpu_barrier(++goal,arrayin,arrayout);//global barrier
9  int barrflag=0;
10 for( (int i=0;i<pred.size;i++) ){
11 |  if( t→owner[pred.D_Vec[i]]≠id ){
12 |  |  barrflag=1;break;
13 |  }
14 }
15 if(barrflag==0 ){ //update cavity }
16 else { //abort }
```

complete up to the barrier before any thread can proceed. This generated code is similar to that used in LonestarGPU [73].

Algorithm 58: semantics of CAS operation

```
1    CAS(T *loc, T old, T new);
2  ------------------------------
3  CAS(T *loc, T old, T new) {
4  |  if( *loc==old ){
5  |  |  *loc=new;
6  |  |  return old;
7  |  }
8  |  return *loc;
9  }
```

Before going to the code generation of first variant of `single` statement, we define the semantics of compare_and_swap() (CAS) operation available in different devices. Syntax and semantics of CAS is given in Algorithm 58. T is the data type given as an argument to CAS and CAS is an atomic operation, which makes sure that only one thread will update the value of the memory location *loc*.

Algorithm 59: `single` on one element- `Falcon` DSL code

```
1 Point(graph) P;
2 single(P.lock){
3     //block of code
4 }
```

Algorithm 60: `GPU` code for Algorithm 59

```
1 #define hcas __sync__val__compare__and__swap
2 #define ep (struct struct_graph)
3 #define gx graph.extra
4 int P;
5 if( hcas(&((ep *)(gx))→lock[P],0,1)==0 ){
6 |   //block of code
7 }
```

Algorithm 61: `CPU` code for Algorithm 59

```
1 #define dcas atomicCAS
2 #define ep (struct struct_graph)
3 #define gx graph.extra
4 int P;
5 if( dcas(&((ep *)(gx))→lock[P],0,1)==0 ){
6 |   //single GPU code //block of code
7 }
```

The first variant of `single` statement that locks a single object does not need a barrier. It uses the compare_and_swap (CAS) variant of CUDA [75] and GCC [5] for GPU and CPU respectively. Algorithm 59 shows the DSL code for `single` on one element. This converts to CAS operations with CUDA function being atomicCAS (dcas) (Algorithm 60) and GCC function being __sync__val__compare__and__swap (hcas) (Algorithm 61). The *lock* value is reset to zero, before *CAS* is called on the lock element. The effect of an operation
CAS(lock,0,1)
is as follows : if multiple threads try to lock a `Point` object P at the same time, only one thread will succeed and it will execute the cod inside the `single` block and others just do nothing. This type of `single` statement is normally used in local computation algorithms such as Boruvka's MST computation. In order for the `single` to work properly, the property value must be reset to zero before entering the function in which `single` is executed.

5.3.5 Reduction functions

Reduction operations have been implemented in `Falcon`. Translation of reduction functions to CUDA functions is straightforward [51]. An example reduction operation `ReduxSum` in `Falcon` is shown below.
`if`(graph.edges.mark==1) mstcost `ReduxSum`=graph.edges.weight;

Here, for each `Edge` object of the GPU `Graph` object *graph*, if the value of the boolean extra-property *mark* is true, its weight is added to *mstcost*.

Algorithm 62: MST reduction Operation CUDA code

```
1  #define DH cudaMemcpyDeviceToHost
2  #define HD cudaMemcpyHostToDevice
3  __device__ unsigned int dreduxsum0;
4  __global__ void RSUM0(GGraph graph,int FALCX) {
5      int id = blockIdx.x * blockDim.x + threadIdx.x+FALCX;
6      __shared__ volatile unsigned int Reduxarr[1024];
7      if( id <graph.nedges ){
8          if(((struct struct_graph *)(graph.extra))→mark[id]==true)
9              Reduxarr[threadIdx.x]=graph.edges[2*id+1];
10         else
11             Reduxarr[threadIdx.x]=0;
12         __syncthreads();
13         for( (int i=2;i<=TPB;i=i*2) ){
14             if(threadIdx.x==0) Reduxarr[threadIdx.x]+=Reduxarr[threadIdx.x+i/2];
15             __syncthreads();
16         }
17         if(threadIdx.x==0)atomicAdd(&dreduxsum0,Reduxarr[0]);
18     }
19 }
20 //host (CPU) code
21 main(int argc, char *argv[]) {
22     ....
23     unsigned int hreduxsum0=0;
24     cudaMemcpyToSymbol(dreduxsum0,&hreduxsum0,sizeof(unsigned int),0,HD);
25     for( (int kk=0;kk<graph.nedges;kk+=flcBlocks*TPB) ){
26         RSUM0<<<flcBlocks,TPB>>>(graph,kk);
27     }
28     cudaDeviceSynchronize();
29     cudaMemcpyFromSymbol(&hreduxsum0,dreduxsum0,sizeof(unsigned int ),0,DH);
30     mstcost=hreduxsum0; ....
31 }
```

Algorithm 62 shows the generated CUDA code for above statement. Variables hreduxsum0 and dreduxsum0 are CPU and GPU variables automatically generated by `Falcon`. The kernel block size is 1024 and if an edge e_i in a thread block is a part of mst (*mark[i]==true*), where 0≤i<1024, the edge weight is stored Reduxarr[i] (Line 9). Each block of the CUDA kernel

stores the sum of the weights of the edges processed by that block and present in MST, in Reduxarr[0] (Lines 13-16). Then this value is added to the MST cost of the graph by adding Reduxarr[0] to dreduxsum0 atomically (Line 17). The value of the dreduxsum0 variable, which has the MST cost of the graph object is then copied to the mstcost variable on the CPU (host) after the *RSUM0* kernel finishes its execution.

5.4 Modifying graph structure

Deletion of a graph element is by marking its status. Each point and edge has a boolean flag that marks its deletion status. We provide an interface that enables a programmer to check if an object has been deleted by another thread.

Addition of `Point` and `Edge` to a graph object is performed using atomic operations. For a `Graph` object with the name say *graph*, we add global variables *falcgraphpoint*, *falcgraphedge* which will be initialized to the number of points and edges in the *graph*(resp.). When we call *graph*.`addPoint` in a `Falcon` program, that code will be replaced by a call to an automatically generated function *falcaddgraphpointfun*(). This function atomically increments *falcgraphpoint* by one. Analogous functions exist for `Edge` and properties added using the `addProperty` function. Currently, none of the properties (attributes) associated with graph elements are automatically deleted (including the one added using addProperty); their deletion must be explicitly coded by the programmer. DMR implementation deletes *triangle*s by storing a boolean flag in the property *triangle* and making that flag value `true` for deleted triangles.

Automatic management of size is also needed for morph algorithms. For example in DMR, the `Graph` size increases and the pre-allocated memory may not be sufficient. A call to the compiler generated *realloc()* function is inserted automatically after the code that modifies the `Graph` size. This *realloc()* function considers current size, the change in size and the available extra memory allocated and performs `Graph` reallocation, if necessary.

While it is true that graph algorithms exhibit irregularity, overall, the following aspects help us achieve better coalescing and locality

- CSR representation enables accessing the nodes array in a coalesced fashion. It also helps achieve better locality as the edges of a node are stored contiguously.
- Shared memory accesses for warp-based execution and reductions help improve memory latency.
- Optimized algorithms. Note that a high-level DSL allows us to tune an algorithm easily, such as the SSSP optimization discussed in Section 4.

5.5 Experimental evaluation

To execute the CUDA codes, we have used an Nvidia multi-GPU system with Four GPUs (One Kepler K20c GPU with 2496 cores running at 706 MHz and 6 GB memory, two Tesla C2075 GPUs each with 448 cores running at 1.15 GHz and 6 GB memory, one Tesla C2050 GPU with 448 cores running at 1.15 GHz and 6 GB memory). Multi-core codes were run on Intel(R) Xeon(R) CPU, with two hex-core processors (total 12 cores) running at 2.4 GHz with 24 GB memory. All the `GPU` codes were by default run on Kepler K20c (device 0). The `CPU` results are shown as speedup of 12-threaded codes against single-threaded Galois code. We used Ubuntu 14.04 server with g++-4.8 and CUDA-7.0 for compilation.

We compared the performance of the `Falcon`-generated CUDA code against LonestarGPU-2.0 and Totem [45][44], and the multi-core code against that of Galois-2.2.1 [77], Totem and GreenMarl [53]. LonestarGPU does not run on multi-core CPU and Galois has no implementation on GPU. While Totem supports implementation of an algorithm on multiple `GPU`s using graph partitioning, which is useful for extremely large graphs that do not fit on a single `GPU`. We have shown results with Totem executing only on a single `GPU` so as to make fair comparison.

Input	Graph Type	Total Points	Total Edges	BFS distance	Max Nbrs	Min Nbrs
rand1	Random	16M	64M	20	17	1
rand2	Random	32M	128M	18	17	1
rmat1	Scale Free	10M	100M	∞	1873	0
rmat2	Scale Free	20M	200M	∞	2525	0
road1(usa-ctr)	Road Network	14M	34M	3826	9	1
road2(usa-full)	Road Network	23M	58M	6261	9	1

Table 5.1. Inputs used for local computation algorithms

Results are shown for three cautious morph algorithms (SP, DMR and dynamic SSSP) and three local computation algorithms (SSSP, BFS and MST). `Falcon` achieves close to 2× and 5× reduction in the number of lines of code (see Table 5.2) for morph algorithms and local computation algorithms respectively compared to the hand-written code. Morph algorithms DMR and SP have a read function that a user is required to write in `Falcon`, which increases the code length. This could have been added as a function of the `Graph` class (as in LonestarGPU and Galois), but it differs much (reading triangles) from reading a normal graph which has just points and edges. If we look at the lines of code leaving out the code for the read function,

Algorithm	Falcon CPU	Green-Marl	Galois	Totem CPU	Falcon GPU	Lonestar GPU	Totem GPU
BFS	26	24	310	400	28	140	200
SSSP	35	24	310	60	38	170	330
MST	113	N.A.	590	N.A.	103	420	N.A.
DMR	302	N.A.	1011	N.A.	308	860	N.A.
SP	198	N.A.	401	N.A	185	420	N.A.
Dynamic SSSP	51	N.A.	N.A.	N.A.	56	165	N.A.

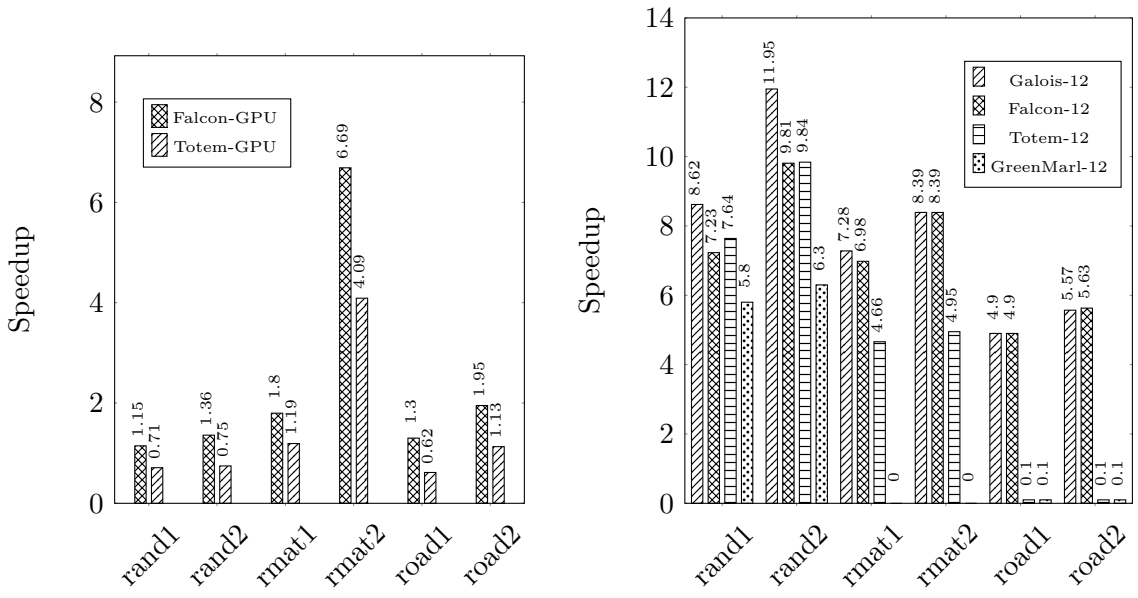
Table 5.2. Lines of codes for algorithm in different frameworks / DSL

there is a significant reduction in the size of `Falcon` code for morph algorithms also, when compared to hand written code. We have measured the running time from the beginning of the computation phase till its end. This includes the cost of communication between the `CPU` and the `GPU` during this period. We have not included the running time for reading and copying the `Graph` object to the `GPU` and for copying results from the `GPU`.

5.5.1 Local computation algorithms

Figure 5.2 shows the speedup of SSSP on GPU over LonestarGPU and on CPU over Galois-single. Figure 5.3 shows the speedup of BFS on GPU over LonestarGPU and on CPU over Galois-single. We experimented with several graph types (such as the Erdös-Rényi model graphs [35], road networks, and scale-free graphs) and have shown results for two representative graphs from each category, with several million edges. Details can be seen in Table 6.3. Road network graphs are real road networks of USA [33], have less variance in degree distribution, but have large diameter. Scale-free graphs have been generated using GTGraph [11] tool, have a large variance in degree distribution but exhibit small-world property. Random graphs have been generated using the graph generation tool available in Galois.

SSSP. The speedup for SSSP on `GPU` is shown for Totem and `Falcon` with respect to LonestarGPU in Figure 5.2(a). Results for SSSP on `GPU` have been plotted as speedup over best time reported by LonestarGPU variants (worklist based SSSP and Bellman-Ford style SSSP). `Falcon` also generates worklist bases and optimized Bellman-Ford algorithms. We find that `Falcon` SSSP (Algorithm 32, Section 4.2) is faster than LonestarGPU. This is due to the optimization used in the `Falcon` program using the *uptd* field, which eliminates many unwanted



(a) GPU-Speedup over LonestarGPU

(b) CPU-Speedup Over Galois Single

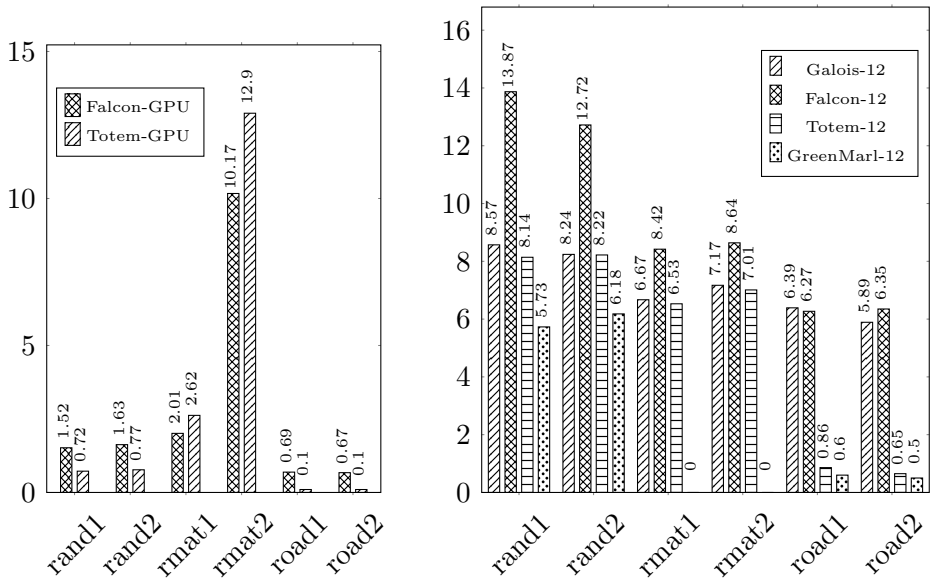
Figure 5.2: SSSP speedup on CPU and GPU

computations. For rmat2 input worklist based SSSP of LonestarGPU went out of memory and speedup shown is over slower Bellman-Ford style SSSP of LonestarGPU.

The results for SSSP on `CPU` are plotted as speedup over Galois single threaded code (Figure 5.2(b)). `Falcon` and Galois use a `Collection` based Δ-stepping implementation. Totem and GreenMarl do not have a Δ-stepping implementation. Hence, Totem and GreenMarl are always slower than Galois and `Falcon` for road network inputs. GreenMarl failed to run on rmat input giving a runtime error on `std::vector::reverse()`. It is important to note that Bellman-Ford variant of the SSSP code (Algorithm 32, Chapter 4) on `CPU` with 12 threads is about 8× slower than that of the same on `GPU`. It is the worklist based Δ-stepping algorithm which made `CPU` code fast. BFS and MST also benefit considerably from worklist based execution on `CPU`.

BFS. The speedup for BFS on `GPU` is shown for Totem and `Falcon` with respect to LonestarGPU in Figure 5.3(a). Results for BFS on `GPU` are compared as speedup over the best running times reported by LonestarGPU. We took the best running times reported by worklist based BFS and Bellman-Ford variant BFS implementations. The worklist based BFS performed faster only for road network input. `Falcon` also has a worklist based BFS on `GPU` which is slower by about 2× compared to that of LonestarGPU. Totem framework is too slow on road network due to lack of worklist based implementation.

`Falcon` BFS code on `CPU` always outperformed Galois BFS, due to our optimizations (Figure



(a) GPU-Speedup over LonestarGPU

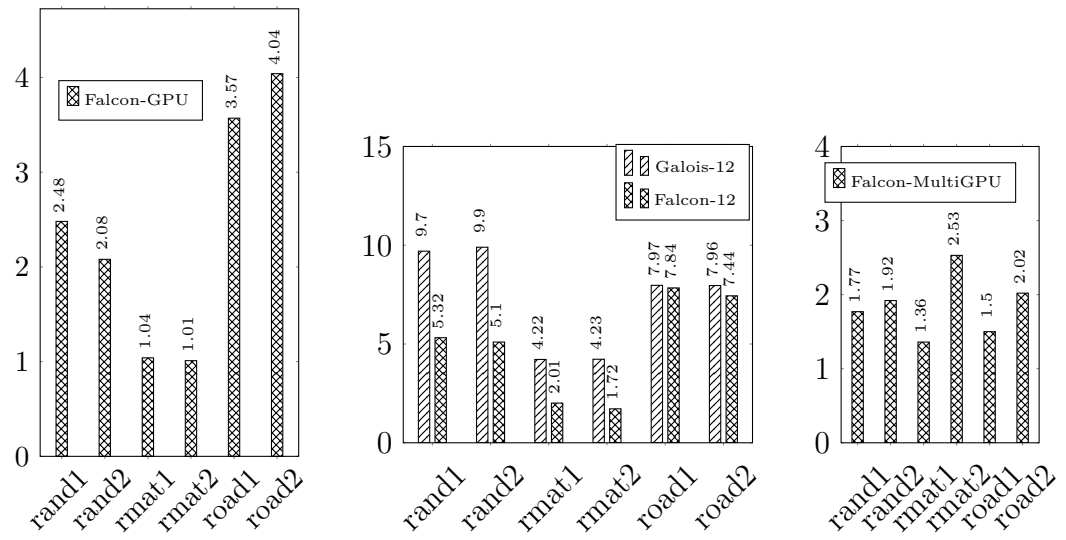
(b) CPU-Speedup over Galois single

Figure 5.3: BFS speedup on CPU and GPU

5.3(b)). Totem and GreenMarl are again slower on road inputs. Totem performed better than `Falcon` for scale free graphs on GPU. GreenMarl failed to run on rmat input giving a runtime error on `std::vector::reverse()`.

MST. The Speedup for MST on `GPU` is shown in Figure 5.4(a) and same for `CPU` is shown in Figure 5.4(b). LonestarGPU has a Union-Find based MST implementation. `Falcon GPU` code for MST always outperformed that of LonestarGPU for all inputs, with the help of a better implementation of Union-Find that `Falcon` has for `GPU`. But our `CPU` code showed a slowdown compared to Galois (about 2× slowdown). Galois has a better Union-Find implementation based on object location as key.

Multi-GPU. Figure 5.4(c) shows the speedup of `Falcon` when algorithms BFS, SSSP and MST are executed on three different `GPU`s in parallel for the same input, when compared to their separate executions on the same `GPU`. The running time of `Falcon` is taken as the maximum of the running times of BFS, SSSP and MST, while the running time of LonestarGPU is the sum of the running times of BFS, SSSP and MST. One should not get confused with speedup values in Figure 5.4(c) and values in Figures 5.2 and 5.3, because for road networks, SSSP running time was very high compared to the MST running time, and for other inputs(random, rmat) MST running time was higher. It is also possible to run algorithms on `CPU` and `GPU` in parallel



(a) GPU- MST Speedup over LonestarGPU

(b) CPU-MST speedup over Galois Single

(c) Speedup of `Falcon` on multi-GPU

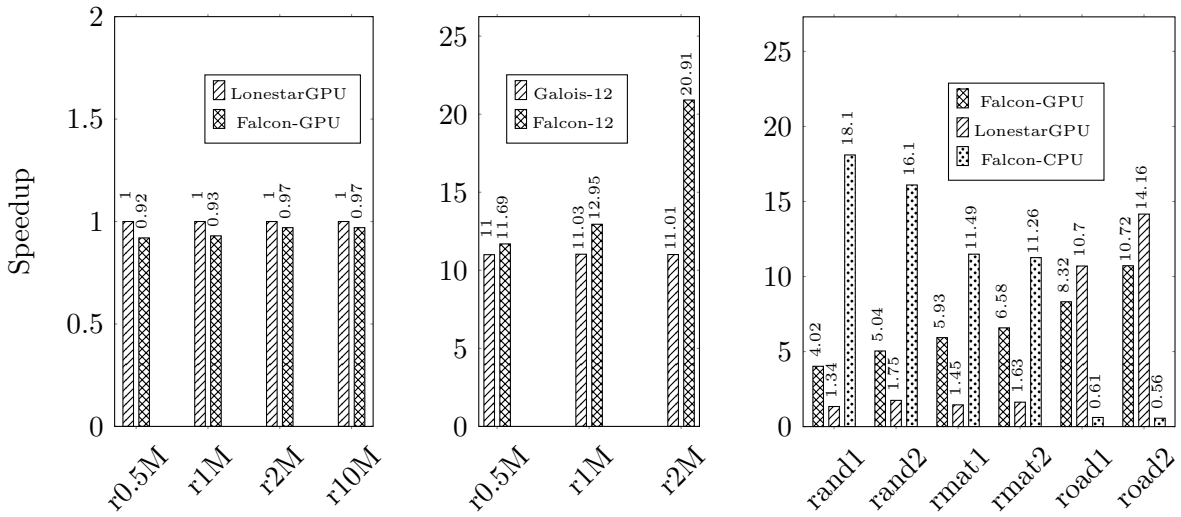
Figure 5.4: MST and Multi-GPU Results

using the `parallel sections` statement. A Programmer can decide where to run a program by allocating a Graph object on `GPU` or `CPU`, by giving proper command line arguments. He/She can then place appropriate `foreach` statements in each `section` of the `parallel sections` statement of `Falcon`. For example, SSSP on road network inputs can be run on `CPU` (because it is slow on GPU) and for random graph inputs, on `GPU`. The effort required to modify codes for `CPU` or `GPU` is minimal with `Falcon`.

5.5.2 Morph algorithms

We have specified three morph algorithms using `Falcon`: DMR, SP and dynamic SSSP. All these algorithms have been implemented as cautious algorithms and we have compared the results with implementations using LonestarGPU and Galois (other frameworks do not support mutation of graphs).

Delaunay Mesh Refinement (DMR). DMR implementation in LonestarGPU relies on a global barrier, which can be implemented either by returning to the CPU and launching another kernel, or by emulating a grid-barrier in software [100]. LonestarGPU uses the latter approach as it allows saving the state of the computation in local and shared memory across barriers inside the kernel (which is infeasible in the first approach where the kernel is terminated) and this approach is used in `Falcon` DSL code as well. Unfortunately, grid-level barriers pose a limit on the number of threads with which a kernel can be launched, as all the thread-



(a) DMR speedup over LonestarGPU

(b) DMR speedup over Galois single

(c) DynamicSSSP- Self relative speedup

Figure 5.5: Morph Algorithm Results -DMR and DynamicSSSP

blocks need to be resident and all the threads must participate in the barrier; otherwise, the kernel execution hangs. Therefore, both LonestarGPU and `Falcon`-generated codes restrict the number of launched threads, thereby limiting parallelism. This is also observable in other morph algorithm implementations needing a grid-barrier. Figure 5.5(a) and 5.5(b) show the performance comparison of DMR code for `GPU` and `CPU` on input meshes containing a large number of triangles in the range 0.5 to 10 million. Close to 50% of the triangles in each mesh are initially bad (that is, they need to be processed for refinement). Galois goes out of memory for 10 million triangles or more, and terminates. `Falcon` code is about 10% slower compared to LonestarGPU code and both used the same algorithm. This can be due to the inefficiency arising from conversion of DSL code to CUDA code as compared to the hand written codes of LonestarGPU. Speedup shown is for mesh refinement code (including communication involved during that time), after reading the mesh.

Survey Propagation (SP). Survey Propagation algorithm [17] deletes a node when its associated probability becomes close to zero and this makes SP a morph algorithm. In this implementation, we implemented the global barrier on a `GPU` by returning to the `CPU`, as no local state information needs to be carried across kernels (the carried state of variables is stored in global memory). A similar approach is used in LonestarGPU as well.

The first four rows of Table 5.3 show how SP works for a clause(M)-to-literal(N) ratio of 4.2 and 3 literals-per-clause(K) for different input sizes and the last three rows are for different

Input(K, N, M)	Galois 12 threads	Falcon 12 threads	Lonestar GPU	Falcon GPU
(3,1x10^6, 4.2x10^6)	67	46	26	23
(3,2x10^6,8.4x10^6)	147	76	55	47
(3,3x10^6,12.6x10^6)	232	114	86	69
(3,4x10^6,16.8x10^6)	322	147	117	93
(4,4x10^6,9.9x10^6)	1867	149	118	95
(5,1x10^6,21.1x10^6)	killed	356	414	314
(6,1x10^6,43.4x10^6)	killed	1322	1180	928

Table 5.3. Performance comparison for Survey Propogation (running time in seconds)

values for the clause(M)-to-literal(N) ratio. We observe that `Falcon`-generated code always performs better than both multi-core Galois with 12 threads and LonestarGPU. Note that performance has been compared with LonestarGPU-1.0 and Galois-2.1 codes. New versions of both these frameworks use a new algorithm, which is yet to be coded in `Falcon`. Multi-core Galois goes out-of-memory for higher values of (K, N, M), whereas LonestarGPU and `Falcon` versions complete successfully. LonestarGPU allocates each property of clause and literal in separate arrays whereas in `Falcon`, each property of clause and literal is put in structures, one each for clause and literal. Galois has a worklist based implementation of the algorithm. Also both Galois and LonestarGPU work by adding edges from *clauses* (Point in Graph) to each *literal* (Point in Graph) in the *clause*. But `Falcon` takes a *clause* as an extra property of the Graph (like *triangle* was used in DMR) and that property stores *literals* (Points) of the *clause* in it. So our Graph does not have any explicit edges, and *literals* of a *clause* (which correspond to edges) can be accessed very efficiently from the *clause* property of the Graph. We find that `Falcon` code runs faster than that of both Galois and LonestarGPU. Writing an algorithm that maintains a *clause* as a property of a Graph in LonestarGPU and Galois is not an easy task.

Dynamic SSSP. In the dynamic Single Source Shortest Path (SSSP) algorithm, edges can be added or deleted dynamically. A dynamic algorithm where only edges get added (deleted) is called as an incremental (decremental) algorithm, whereas algorithms where both insertion and deletion of edges happen are called fully dynamic algorithms [39]. We have implemented an incremental dynamic algorithm on `GPU` and `CPU` using `Falcon`. We have used a variant of the algorithm by [81]. Insertions are carried out in chunks and then SSSP is recomputed. We

found it difficult to add dynamic SSSP to the Galois system, because no Graph structure that allows efficient addition of a big chunk of edges to an existing Graph object was available in Galois. LonestarGPU code has been modified to implement dynamic SSSP, and we compare it with our `CPU` and `GPU` versions. `Falcon` looks at functions used in programs that modify a Graph structure (addPoint(), addEdge(), etc.) and converts a Graph read() function in `Falcon` to the appropriate read() function of the HGraph class. For dynamic SSSP, the read() function allocates more space to add edges for each Point and makes the algorithm work faster. LonestarGPU code has also been modified in the same way. Results are shown in Figure 5.5(c), which shows the speedup of the incremental algorithms with respect to their own initial SSSP computation. SSSP on `GPU` was an optimized Bellman-Ford style algorithm that processes all the elements and so does many unwanted computations, while `CPU` code is Δ-stepping algorithm.

Chapter 6

Code Generation for Distributed Systems

6.1 Introduction

This chapter explains how the `Falcon` compiler converts a `Falcon` DSL code to CUDA/C++ codes with MPI/OpenMPI library targeting distributed systems. Falcon supports the following types of distributed systems.

- CPU cluster - A set of inter-connected machines with each machine having a multi-core CPU.
- GPU cluster - A set of inter-connected machines with each machine having one or more GPUs used for computation and a multi-core CPU on which the operating system runs.
- Multi-GPU machine - A single machine with a multi-core CPU and two or more GPU devices.
- CPU+CPU cluster - A set of inter-connected machines with multi-core CPU and GPU and both used for computation.

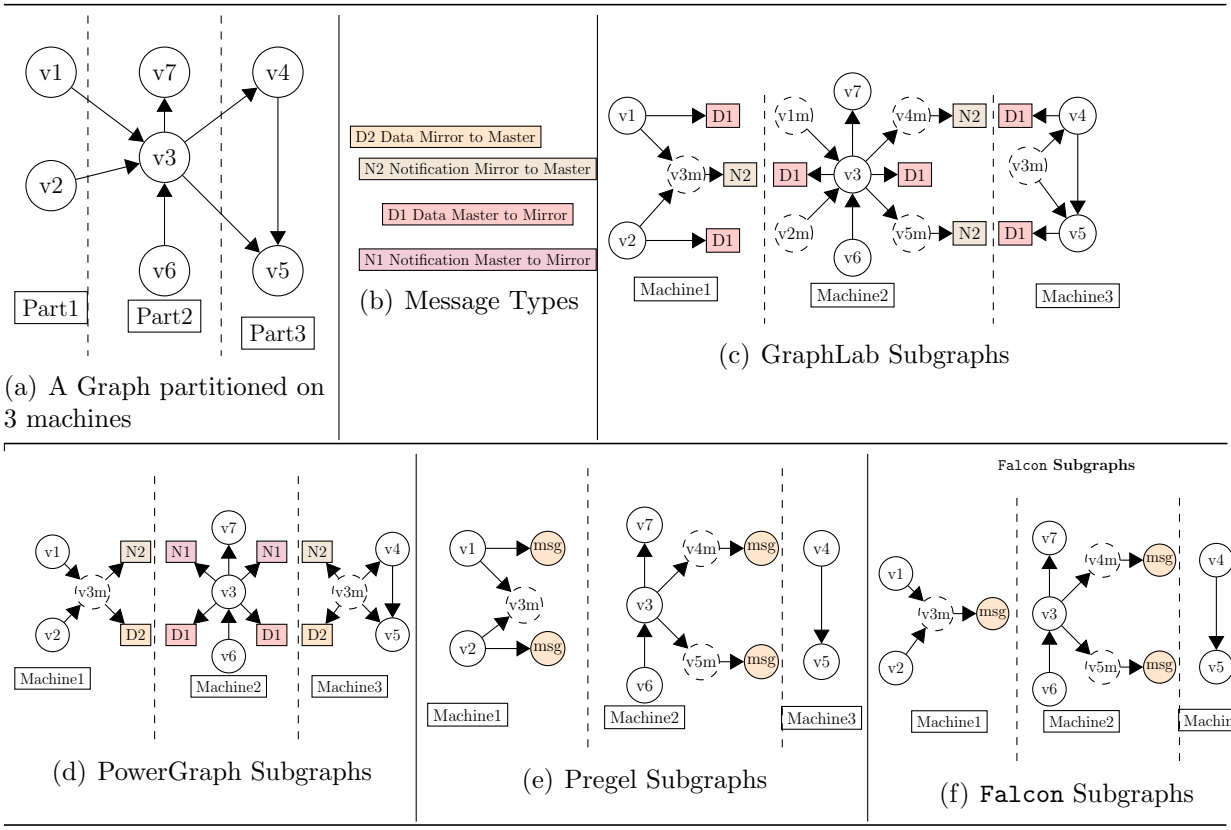
A graph is taken as a primary data structure for representing relationships in real world data and social network systems such as Twitter, Facebook etc. These graphs have billions of vertices and trillions of edges. Such large-scale graphs do not fit on a single machine and are stored and processed on a distributed computer system or clusters. Algorithms which process these distributed data must incure less communication overhead and work balance across machines to achieve good performance. There are many frameworks for large-scale graph processing

targeting only CPU clusters like Google's Pregel [65], Apache Giraph [87], GraphLab [64], PowerGraph [46]. Pregel and Giraph follows the Bulk Synchronous Parallel (BSP) model of execution, and GraphLab follows the asynchronous execution model. PowerGraph supports both synchronous and asynchronous execution with Gather-Apply-Scatter (GAS) model of execution. The important contributions of `Falcon` for large-scale graph processing are mentioned below.

- A programmer need not deal with the communication of data across machines as it is taken care by the `Falcon` compiler.
- The Message Passing Interface (MPI) library does not support distributed locking. `Falcon` provides support for distributed locking which is used in implementing the `single` construct of `Falcon`.
- The Union-Find `Set` data type has also been extended for distributed systems.
- To the best of our knowledge there is no DSL other than `Falcon` which targets heterogeneous distributed systems with multi-core CPU and GPU devices for large-scale graph processing.
- `Falcon` supports dynamic graph algorithms for distributed systems with multi-core CPU and/or GPU devices.

`Falcon` uses random *edge-cut* cut graph partitioning as optimal graph partitioning is an NP-Complete problem and similar methods have been used in other frameworks also (e.g, Pregel). A single DSL code with proper command line arguments gets converted to different high-level language codes (C++, CUDA) with the required library calls (OpenMP, MPI/OpenMPI) for the distributed systems by the `Falcon` compiler (see Figure 4.1). These codes are then compiled with the native compilers (`g++`, `nvcc`) and libraries to create the executables. For distributed targets, the `Falcon` compiler performs static analysis to identify the data that needs to be communicated between devices at various points in the program (see Sections 6.6.5 and 6.6.9).

The graph is partitioned and stored as subgraphs, namely *localgraph* on all the devices involved in the computation. The `Falcon` compiler generates code for communication between subgraphs after a parallel computation (if required), in addition to the code for parallel computation on each device which is almost similar to the strategies discussed Chapter 5. A `foreach` statement is converted to CUDA kernel call for GPU and `OpenMP` parallel loop for CPU. These codes will be preceded and/or succeeded by extra code (if required), which perform communication across devices involved in parallel computation.



(a) A Graph partitioned on 3 machines

(b) Message Types

(c) GraphLab Subgraphs

(d) PowerGraph Subgraphs

(e) Pregel Subgraphs

(f) `Falcon` Subgraphs

Figure 6.1: Comparison of `Falcon` and other distributed graph frameworks

Performance of `Falcon` is compared with PowerGraph for CPU clusters and Totem [44] for a multi-GPU machine. `Falcon` was able to match or outperform these frameworks and for some of the benchmarks, `Falcon` gave a speedup of up to 13× over them.

6.2 Requirements of large-scale graph processing and demerits of current frameworks

Distributed graph processing follows a common pattern:

- A vertex gathers values from its neighboring vertices on remote machines, and updates its own value.
- It then modifies property values of its neighboring vertices and edges.
- It broadcasts the modified values to the remote machines.

Figure 6.1 shows a comparison of GraphLab, PowerGraph, Pregel and `Falcon`, related to graph

storage and communication patterns on vertex $v3$ in the directed graph on Figure 6.1(a).

6.2.1 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 6.1(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).

6.2.2 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 6.1(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.

Item	PowerGraph	GraphLab	Pregel	Falcon
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	vertex-cut	edge-cut	edge-cut	edge-cut
Message Volume	high	high	low	low

Table 6.1. Comparison of various distributed frameworks

6.2.3 Pregel

The Pregel framework uses random edge-cut to partition the graph object (Figure 6.1(e)). Pregel follows the Bulk Synchronous Parallel (BSP) Model [95] 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 single message. This needs to be done by the programmer by overriding the *Combine*() method of the *Combiner* class [65] 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}.

6.2.4 Falcon

`Falcon` follows the BSP model of execution and uses random edge cut to partition graph objects (Figure 6.1(f)). The execution is carried out as a series of supersteps similar to Pregel. `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 `Falcon` compiler also requires that operations which modify mutable graph properties be commutative and associative.

Pregel and `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 6.1 compares the frameworks mentioned above.

6.3 Representation and usage of Falcon data types

Storage of `Falcon` data types for a distributed system differs from that for a single device (GPU or CPU) system.

6.3.1 Point and Edge

In a distributed setup, a `Point` object has a global-vertex-id, as well as a local or a remote vertex-id in a *localgraph* object. The programmer can only view and operate on a `Point` based on the global-vertex-id. The local vertex-id is used for storing and processing *localgraph* objects on each machine/device and the remote vertex-id is used for communication between machines/devices in each superstep of the BSP model of execution. Edges are stored in the *localgraph* with modified values for source and destination vertex-id, which will be the local or remote vertex-id within that *localgraph*.

6.3.2 Graph and distributed graph storage

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-node*$(u) = k$ (*m-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-node*$(p) = k$) or remote-vertex-id(*m-node*$(p) \neq k$).

The `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 66) instead of push-based computation. The `Falcon` performs static analysis to determine this, and generates efficient code with minimal communication overhead (See Section 6.6.9).

	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}
Machine1	$v_0(l_0)$	$v_1(l_1)$	$v_2(l_2)$	$v_3(l_3)$	$v_4(l_4)$	$v_6(r_5)$	$v_8(r_6)$	$v_9(r_7)$	$v_{11}(r_8)$	$v_{13}(r_9)$					
Machine2	$v_5(l_0)$	$v_6(l_1)$	$v_7(l_2)$	$v_8(l_3)$	$v_9(l_4)$	$v_2(r_5)$	$v_4(r_6)$	$v_{11}(r_7)$	$v_{13}(r_8)$						
Machine3	$v_{10}(l_0)$	$v_{11}(l_1)$	$v_{12}(l_2)$	$v_{13}(l_3)$	$v_{14}(l_4)$	$v_3(r_5)$	$v_5(r_6)$	$v_7(r_7)$							

offset	Machine0	Machine1	Machine2
positions (0 to 3)	(0, 5, 8, 10)	(0, 5, 7, 9)	(0, 5, 6, 8)

Table 6.2. Conversion of global vertex-id to local and remote vertex-id on three machines. l stands for *local* and r stands for remote.

Table 6.2 shows an example of how a global-vertex-id is converted to local vertex-id and remote vertex-id, when a graph which has 15 vertices and E edges is partitioned across three machines. The local vertices on each machine is given below, which is the master node for those vertices.

- Machine1 - v_0, v_1, v_2, v_3, v_4.
- Machine2 - v_5, v_6, v_7, v_8, v_9.

- Machine3 - v_{10}, v_{11}, v_{12}, v_{13}, v_{14}.

The 10 vertices of the *localgraph* on Machine1 are v_0, v_1, v_2, v_3, v_4, v_6, v_8, v_8, v_9, v_{11}, v_{13}. The first 5 vertices are local vertices as the master node of these vertices is Machine1. The master node of vertices v_6, v_8 and v_9 is Machine2 and that of the vertices v_{11} and v_{13} is Machine3. These are remote vertices for the *localgraph* on Machine1 and given vertex-id's from 5 to 9. Boundaries of beginning of local and remote vertices belonging to the *localgraph* on each machine are stored in an array $offset[]$ (last row, Table 6.2). The same is shown for Machine2 and Machine3 also in Table 6.2. A remote vertex rv becomes a part of the vertices in a machine M_i, if there are one or more edges from the local vertices of the subgraph in M_i with the other end point of the edge as the vertex rv, and master node of $rv \neq M_i$. The vertex-id in the $edges[]$ array of each *localgraph* is modified to have the local and remote vertex-id's.

Algorithm 63: Distributed-Union in `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 remote nodes
      perform union; update parent of each element
      send parent value to each remote node
4 }
```

6.3.3 Set

`Falcon` implements distributed Union-Find on top of the Union-Find of `Falcon` [24]. 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 the pseudo-code of Algorithm 63.

6.3.4 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

Algorithm 64: Collection synchronization in `Falcon`

```
1 foreach( item in Collection ){
    | if (item.master-node≠rank(node))
    |     add item to buffer[item.master-node] and delete item from Collection
2 }
  foreach (i ∈ remote-node) send buffer to remote-node(i)
  foreach ( i ∈ remote-node)receive buffer from remote-node(i)
  foreach( i ∈ remote-node ){
3 |   foreach( j ∈ buffer[i] ){
  |   |   update property values using buffer[i].elem[j]
  |   |   addtocollection(buffer[i].elem[j])
4 |   }
5 }
```

the appropriate master mode. `Collection` object is synchronized as shown in the pseudo-code of Algorithm 64.

6.4 Parallelization and synchronization constructs

6.4.1 `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 `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, and this may involve data communication. A global barrier is imposed after this step.

To iterate over all the edges of a *localgraph*, either the *points* or *edges* iterator can be used. If a *points* iterator is used, then a `foreach` statement using the *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, thereby minimizing thread divergence and improving GPU performance. For example, when the SSSP computation is performed on a partitioned twitter [59] input on a single machine with 8 GPUs, it showed a 10× speedup while iterating over *edges* compared to iterating over *points*. In the twitter input, half of the edges are covered by 1% of the vertices and the out-degree varies from 0 to 2,997,469.

6.4.2 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 [24]. This has been discussed in Section 4.6.3.

6.4.3 Single statement

`single` statement is the synchronization construct of `Falcon`. It can be be used to lock a single element or a `Collection` of elements in a distributed system. The `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 6.6.7 and it is used in our implementation of Boruvka's-MST algorithm [90].

Algorithm 65: Single Source Shortest Path in `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      Graph graph;
9      graph.addPointProperty(dist,int);
10     graph.read(argv[1]);
11     foreach (t In graph.points) t.dist=1234567890;
12     graph.points[0].dist=0;
13     while( 1 ){
14         changed = 0; //keep relaxing
15         foreach(t In graph.edges) relaxgraph(t,graph);
16         if(changed == 0)break;
17     }
18     for(int i=0;i<graph.npoints;i++)printf(i=%d dist=%d\n, i, graph.points[i].dist);
19 }
```

6.5 Examples: SSSP and Pagerank

Algorithm 65 shows the `Falcon` code for SSSP (Single Source Shortest Path) computation in `Falcon` for multiple platforms or target devices. An extra-property *dist* is added to the `Graph` object *graph* (Line 9). Before the parallel `foreach` call in Line 11 initializes the *dist* property of each vertex to ∞, the graph object is read from the disk (Line 10). The *read*() function

gets converted to different versions of *read*() based on the command line argument given for target system to the `Falcon` compiler. For example, if the target system is a GPU cluster, this converts to a *read*() function which reads the graph to the CPU memory and then partitions the graph and copies the *localgraph* object on each node from its CPU memory to its GPU device memory. The SSSP computation happens in the while loop (Lines 13–17), 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 for at least one vertex. The computation finishes when a fixed-point reached, which is checked in Line 16. The *relaxgraph()* function is called using the *edges* iterator (Line 15), which performs better on GPUs for large-scale graphs. Large-scale graphs follow power-law distribution and can create thread divergence when iterated over points, as the *out-degree* of vertices have a large variance. When iterated using the *edges* iterator each thread processes exactly one element, which eliminates thread divergence.

Pagerank code in `Falcon` is shown in Algorithm 66. This algorithm follows a pull-based computation by iterating over *innbrs* of each `Point`, where values are fetched by destination vertex of an edge from its source vertex. (Line 3, Algorithm 66). The *ADD()* function used in the algorithm (Line 4) is not atomic as `Point` *p* modifies its own value.

Algorithm 66: Pagerank in `Falcon`

```
1  pagerank(Point p, Graph graph) {
2      ddouble 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[]) {
7      Graph graph;
8      graph.addPointProperty(PR,double);
9      graph.read(argv[1]);
       foreach (t In graph.points) t.PR = 1 / graph.npoints;
       int cnt = 0;
10     while( cnt <ITERATIONS ){
11         foreach (t In graph.points) pagerank(t, graph);
12         ++cnt;
13     }
14 }
```

6.6 Code generation

6.6.1 Overview

The `Falcon` compiler takes a `Falcon` program and converts it to high-level language code for distributed systems, based on command line arguments. The generated high-level language code is then compiled with a native compiler (`nvcc`/`g++`) and libraries (OpenMPI/MPI, OpenMP). The target systems supported by `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. This allows *device* (GPU) variables as arguments to the MPI functions. For other distributed systems with GPUs, the `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 `Falcon` compiler implements distributed locking across multiple GPUs and CPUs. This is required for the synchronization statement (`single` statement) of `Falcon`.

6.6.2 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 the distributed system. `Falcon` uses random edge-cut partitioning for graph objects. The partitioning algorithm of `Falcon` takes the values $|V|$ and *degree* of each vertex and assigns each vertex, a master node randomly, and makes sure that each *localgraph* object has almost similar number of vertices and edges, but the number of inter-partition edges cannot be controlled by such a naive partitioning scheme. `Falcon` uses this algorithm, as available efficient partitioning tools [7] failed to partition large-scale graphs. `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 a *localpoint* mutable property value needs to be

scattered to remote nodes.

6.6.3 Important MPI functions used by Falcon

Most of the MPI function calls require a communicator of type `MPI_Comm` as an argument and MPI processes can only communicate using a shared communicator. The major MPI functions used by `Falcon` are listed below.

1. MPI_INT(`int` arg, `char` **argv) - Initializes the MPI execution environment.
2. MPI_Comm_rank(`MPI_Comm` comm, `int` *rank) - Used to find the rank of a process.
3. MPI_Comm_size(`MPI_Comm` comm, `int` *rank) - Finds the number of processes involved in program execution.
4. MPI_Isend(`const void` *buf, `int` cnt, `MPI_Datatype` dtype, `int` dst, `int` tag, `MPI_Comm` com, `MPI_Request` *req) - Send *cnt* number of elements of type *dtype* to the process with *rank dst* without blocking.
5. MPI_Recv(`void` *buf, `int` cnt, `MPI_Datatype` dtype, `int` src, `int` tag, `MPI_Comm` com, `MPI_Status` *stat) - Receives from process with *rank src*, a maximum of *cnt* elements of type *dtype* in *buf*. The status of the receive operation such as the number of elements received or buffer overflow can be found using the *stat* variable.
6. MPI_Get_count(`const MPI_Status` *stat, `MPI_Datatype` dtype, `int` *cnt) - Gets the number of elements of type *dtype* received in an $MPI\_Recv()$ operation to which the *stat* variable was given as an argument and stores it in variable *cnt*.
7. MPI_Barrier(MPI_Comm comm) - Barrier for all the processes.
8. MPI_Finalize( ) - All the processes must call this function before program termination.

6.6.4 Initialization for distributed execution

The distributed system codes are executed using the *mpirun* command, to which a user should specify the number processes, the number of processes per nodes e.t.c. These arguments are captured in the $argv[]$ array argument of $main()$ function. The initialization of distributed code in `Falcon` is shown in Algorithm 67.

The `Falcon` compiler uses variables of type MPI_Status and $MPI\_Request$ with names $FALCstatus$ and $FALCrequest$ respectively. The variables $FALCsize$ and $FALCrank$ are used in the program code for allocation of communication buffers, sending and receiving data across nodes etc. The $main()$ function calls the $FALCmpiinit()$ function at the beginning.

Algorithm 67: Initializing distributed code before algorithm execution

```
1  int FALCsize,FALCrank;
2  MPI_Status *FALCstatus;
3  MPI_Request *FALCrequest;
4  void FALCmpiinit(int argc,char **argv) {
5      MPI_Init(&argc,&argv);
6      MPI_Comm_rank(MPI_COMM_WORLD, &FALCrank);
7      MPI_Comm_size(MPI_COMM_WORLD, &FALCsize);
8      FALCstatus=(MPI_Status *)malloc(sizeof(MPI_Status)*FALCsize);
9      FALCrequest=(MPI_Request *)malloc(sizeof(MPI_Request)*FALCsize);
10     gethostname(FALChostname,255);
11     cudaMalloc(&FALCsendbuff,sizeof(struct FALCbuffer )*FALCsize);
12     cudaMalloc(&FALCrecvbuff,sizeof(struct FALCbuffer ));
13     cudaMalloc(&FALCsendsize,sizeof(int)*FALCsize);
14     cudaMalloc(&FALCrecvsize,sizeof(int)*FALCsize);
15     for( (int i=0;i<FALCsize;i++) ){
16         int temp=0;
17         cudaMemcpy(&FALCsendsize[i],&temp,sizeof(int),cudaMemcpyHostToDevice);
18         cudaMemcpy(&FALCrecvsize[i],&temp,sizeof(int),cudaMemcpyHostToDevice);
19     }
20 }
21 main(int argc,char *argv[]) {
22     ......//code not relevant for distributed execution.
23     FALCmpiinit(argc,argv);
24     ....
25 }
```

Variables $FALCsendbuff$ and $FALCrecvbuff$ (of type $FALCbuffer$) are used as variables for sending and receiving data in the auto generated MPI code.$FALCbuffer$ has fields (pointer variables) which are required for sending and receiving graph properties among processes. These fields are allocated later in the *main*() function. In Algorithm 67 lines 11-19 allocate and initialize the buffer for a $TARGET$, which is a GPU luster or a multi-GPU machine.

Algorithm 68: 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
```

6.6.5 Allocation of global variables

Line 1 of Algorithm 65 declares a global variable *changed*. This variable is accessed inside the function *relaxgraph()*, which is called inside the *foreach* statement in Line 15. 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 68 depending on the value of *TARGET*.

Algorithm 69: Global variable allocation in `Falcon`

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

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 68). The `Falcon` compiler generates CUDA and C++ version of the *relaxgraph()* function for the above target system. The CPU and GPU copies of the variable *changed* will be used in C++ and CUDA code (respectively).

The analysis carried out by the `Falcon` compiler for global variable allocation is shown in Algorithm 69. The `Falcon` compiler take each *parallel region* of code (target of outermost `foreach` statement) and each global variable stored in the *symbol table*. Algorithm checks whether a global variable is accessed inside a parallel region and if so the variable is allocated on the *target* device where the parallel code is run. A user gives (*target* as an input to the compiler while compiling the program.

Algorithm 70: 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 }
```

6.6.6 Synchronization of global variables

Line 16 of Algorithm 65 reads the global variable *changed* to check the exit condition, as shown below.

`if`(changed==0)`break`;

Here the value of the variable *changed* should be synchronized across all the nodes before the read access. The code generated by `Falcon` for this access, for heterogeneous systems, follows

the code pattern in Algorithm 70. Algorithm 71 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 71: 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  }
```

Algorithm 72 shows how the *changed* variable is synchronized in a CPU cluster. The value of the variable *changed* is sent by each process to each other process using the $MPI\_ISend()$ function (Lines 6-8). Then each process receives the values in the variable compiler generated temporary variable *falctv4* using the $MPI\_Recv()$ function and received values are added to the *changed* variable (Line 11) in the `for` loop (Lines 9-12. If the value of the variable *changed* is zero, the `break` statement is executed (Line 13).

Code in the Algorithm 72 is for a CPU cluster. If it is a GPU cluster the send operation will be preceded by an operation of copying the value of variable *changed* from GPU to CPU and the receive operation will be succeeded by an operation of copying the value of *changed* to GPU, using *cudaMemcpy*. For multi-GPU devices, `Falcon` uses OpenMPI, which has *cuda-aware-mpi* support and allows device (GPU) addresses as arguments to send and receive operations. Hence `Falcon` auto-generated code for multi-GPU device will not have the *cudaMemcpy* operations.

6.6.7 Distributed locking using `single` statement

The usage of a `single` statement in a function *fun()* is shown in Algorithm 73. Assume that in a distributed execution, the point p may be present as a local vertex in one node and as a 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`) (see Section 5.3.4) operation in the generated code for each node, and exactly one 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 `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

Algorithm 72: CPU cluster code for synchronization of global variable *changed*

```
1  #define MCW MPI_COMM_WORLD
2  #define MSI MPI_STATUS_IGNORE
3  MPI_Status status[16];//assuming maximum processes is 16
4  MPI_Request request[16];
5  int falctv4;
6  for( (int i=0;i<NPARTS;i++) ){
7  |  if(i!=rank)MPI_Isend(&changed, 1, MPI_INT, i,messageno, MCW,&request[i]);
8  }
9  for( (int i=0;i<NPARTS;i++) ){
10 |  if(i!=rank)MPI_Recv(&falctv4,1, MPI_INT, i,messageno, MCW,MSI);
11 |  changed=changed+falcvt4;
12 }
13 if(changed==0)break;
```

Algorithm 73: single statement in `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 }
```

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, the process P_0 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, the process P_0 sends the modified *lock* value back to each remote node, and they update the *lock* value. In the second phase, the `single` statement will be executed with a `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 the value (MAX_INT-1) for the property

Algorithm 74: Code generation for single statement `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()* in order.

lock, after the second `CAS` operation. Otherwise value could be MAX_INT or a value less than the number of processes (*rank*) used in the program execution.

The pseudo code for distributed locking code generation is shown in Algorithm 74. Function *fun()* is duplicated to two versions, *fun1()* and *fun2()*. *fun1()* simply tries to get the *lock*. Code for combining the *lock* values of all the element to produce the minimum *rank* value (by the process P_0) follows. *fun2()* executes *stmt_block*{}, as now *lock* is given to the process with the 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's-MST implementation.

A sample code generated for distributed locking on a GPU cluster is shown in Algorithm 75 for the code shown in Algorithm 73. The `Falcon` compiler generates the functions *updatelock()* (Lines 1-10) and *sendlock()* (Lines 11-21). The function *sendlock()* is used to add the points whose lock value is modified. In the generated code, *lock* value is reset to MAX_INT and copied to *templock*[] array. Then *fun1()* (Lines 22-31) is called and it tries to make the lock value of the destination vertices of an edge equal to the rank of the process (Line 28). These functions are invoked from the *main()* function. Then using the *sendlock()* function, values which are

Algorithm 75: Code generated for GPU cluster for distributed locking

```
1  __global__ void updatelock(GGraph graph,struct buff1 buffrecv,int size) {
2      int id= blockIdx.x * blockDim.x + threadIdx.x;
3      if( id <size ){
4          int vid=buffrecv.vid[id];
5          int lock=buffrecv.lock[id];
6          if( ((struct struct_hgraph *)(graph.extra))->minppty[vid].lock>lock ){
7              ((struct struct_hgraph *)(graph.extra))->minppty[vid].lock=lock;
8          }
9      }
10 }
11 __global__ void sendlock(GGraph graph,int *templock,struct buff1 buff1send,int rank) {
12     int id= blockIdx.x * blockDim.x + threadIdx.x;
13     int temp;
14     if( id <graph.npoints ){
15         if( templock[id]!= ((struct struct_hgraph *)(graph.extra))->minppty[id].lock ){
16             temp=atomicAdd(&lock1sendsize,1);
17             buff1send.vid[temp]=id;
18             buff1send.lock[temp]=((struct struct_hgraph *)(graph.extra))->minppty[id].lock;
19         }
20     }
21 }
22 __global__ void fun1 ( GGraph graph, struct sendnode *sendbuff,int rank ) {
23     int id= blockIdx.x * blockDim.x + threadIdx.x;
24     if( id <graph.localpoints ){
25         int falct1=graph.index[id+1]-graph.index[id]; int falct2=graph.index[id]; for( int
           falct3=0;falct3¡falct1;falct3++ ){
26             int ut1=2*(falct2+falct3);
27             int ut2=graph.edges[ut1].ipe;
28             atomicCAS( &(((struct struct_hgraph
               *)(graph.extra))->minppty[ut2].lock),MAX_INT,rank);
29         }
30     }
31 }
32 main() {
33     .....
34     //copy current lock value to templock[] array.
35     fun1<<<graph.localpoints/TPB+1,TPB>>>fun1(graph, FALCsendbuff,FALCrank);
36     cudaDeviceSyncrhonize();
37     MPI_Barrier(MPI_COMMWORLD);
38     //synchronize lock var by all process sending lock value to rank zero process.
39     // Then rank zero process updates lock value of all points (minimum of received values)
       and send to all nodes.
40     fun2<<<graph.localpoints/TPB+1,TPB>>>fun1(graph, FALCsendbuff,FALCrank);
41     cudaDeviceSyncrhonize(); MPI_Barrier(MPI_COMMWORLD);
42 }
```

modified ($lock[id]! = templock[id]$) are added to $FALCsendbuff[]$ and sent to the rank zero process. The rank zero process collects requests from all the remote nodes and updates the lock value of a point to the minimum of the values received using the *updatelock()* function. Then rank zero process sends the modified values to each remote node. Remote nodes then update the lock value to the value received from the rank zero process. At this point, the *lock* value is the same for a point v which is present in multiple nodes, and the value will be equal to the rank of the least ranked process which succeeded in locking v. Then the `single` operation in function *fun2()* will be $if(CAS(\&lock, rank, MAX\_INT - 1) == rank)$, and this condition will be true for only that process.

6.6.8 Adding prefix and suffix codes for foreach Statement

Algorithm 76: Prefix and suffix code for *relaxgraph* call for CPU cluster (Algorithm 65, Section 6.5)

```
1  //prefixcode
2  #pragma omp parallel for num_threads(FALC_THREADS)
3  for( (int i=graph.nlocalpoints;i<graph.nremotepoints;i++) ){
4  |  tempdist[i]= (( struct struct_graph *)(graph.extra))->dist[i];
5  }
6  #pragma omp parallel for num_threads(FALC_THREADS)
7  for( (int i=0;i<graph.nlocaledges;i++) ){
8  |  relaxgraph(i,graph);
9  }
10 //suffixcode
11 for( (int kk=1;kk<FALCsize;kk++) ){
12 |  #pragma omp parallel for num_threads(FALC_THREADS)
13 |  for( (int i=graph.offset[kk-1];i<graph.offset[kk];i++) ){
14 |  |  addto_sendbuff(i,graph,FALCsendsize,FALCsendbuff,kk-1);
15 |  }
16 }
```

Extra code is generated by `Falcon` at the call site of parallel *foreach*, if it updates all the remote nodes. The *relaxgraph()* function (Line15, Algorithm 65, Section 6.5) updates *dist* value of destination point (T) of an edge E using MIN function (Lines 5) and T could be a remotepoint. The `Falcon` compiler adds extra code before and after the *relaxgraph()* parallel call, similar to the code in Algorithm 76. The code first copies current *dist* value to a temporary buffer *tempdist* (Line 4) and then calls *relaxgraph()* (Line 8). The number of *remotepoints* for a remote node kk is $(offset[kk] - offset[kk-1])$. The *add_to_sendbuff()* (Line 14) function

checks for the condition ($tempdist[i] != dist[i]$) and *remotepoints* of remote node(kk) which satisfy this condition are added to $FALCsendbuff[kk]$. $FALCsendbuff[kk]$ will be filled with ($dist$,$localid[kk]$) in remote node kk. Then these buffers are sent to the respective remote nodes. Each node receives these ($dist$,$localid[kk]$) pairs from all the remote nodes. and updates its *dist* value.This is done by sending the $sendbuff[i]$ values to remote node/process i, followed by a receive operation which receives each $sendbuff[]$ values sent by the remote processes in a $recvbuff[]$ and updating the *dist* value by taking the minimum of the current and the received values.

Algorithm 77: Update *dist* property for SSSP (Algorithm 65, Section 6.5)

```
1  #define MCW MPI_COMM_WORLD
2  #define MSI MPI_STATUS_IGNORE
3  #define graphep ((struct struct_graph * )(graph.extra)) int totsend=0;
4  for( (int i=0;i<FALCsize;i++) ){
5      if( (i!=FALCrank) ){
6          totsend+=sendsize[i];
7          MPI_Isend((sendbuff[i].vid), sendsize[i], MPI_INT, i ,messageno, MCW,&request[i]);
8          MPI_Isend((sendbuff[i].dist), sendsize[i], MPI_INT, i ,messageno+1, MCW,&request[i]);
9      }
10 }
11 for( (int kk=0;kk<(FALCsize);kk++) ){
12     if( (kk!=FALCrank) ){
13         MPI_Recv(recvbuff[0].vid,graph.hostparts[0].npoints, MPI_INT,i, messageno,
           MCW,&FALCstatus[i]);
14         MPI_Recv(recvbuff[0].dist,graph.hostparts[0].npoints, MPI_INT,i, messageno+1,
           MCW,MSI);
15         MPI_Get_count(&FALCstatus[kk], MPI_INT, &FALCnamount);
16         #pragma omp parallel for num_threads(FALC_THREADS)
17         for( (int i=0;i<FALCnamount;i++) ){
18             int vertex= FALCrecvbuff[0].vid[i];
19             if( graphep->dist[vertex] >FALCrecvbuff[0].dist[i])
20                 graphep->dist[vertex] = FALCrecvbuff[0].dist[i];
21         }
22     }
23 }
```

6.6.9 Optimized communication of mutable graph properties

The *relaxgraph()* function (Line 15, Algorithm 65, Section 6.5) updates *dist* value of the destination point which could be a remotepoint of an edge using the MIN function (Lines 5). This requires communication of *dist* properties from remote nodes of a vertex v to its master node

and update the $dist[v]$ value on master node of v to minimum of values at the remote and the master node. The code generated for updating *dist* property for a CPU cluster is shown in Algorithm 77. Each process sends the modified *dist* value of remote vertices and the local-id of the vertices in the corresponding remote node (Lines 4-10). Then the processes receives the values sent in a receive buffer one by one (Lines 13- 14) and updates the *dist* value to the minimum of the current value and the received value (Lines 17-21).

The `Falcon` compiler generates optimized code for such an update. The analysis is as shown in Algorithm 78. 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 (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). `Falcon` DSL code for SSSP and auto-generated code targeting multi-GPU devices can be found in the appendix. This reduces communication volume, as only the modified graph properties are communicated and out of the modified properties only the elements which are modified are communicated.

6.6.10 Storage Optimizations

The `Falcon` compiler generates code to store edge weights only if they are accessed in the program using *getWeight()* functions. Pagerank, K-CORE, and BFS computation do not use edge weights. This optimization saves space, as a weighted Graph G(V,E) requires $2\times|E|$ space for $edges[]$ array, while an unweighted graph requires only $|E|$ space. The receive buffer (to get updated values) is allocated only for one node, as updates are done synchronously. But the send buffer is allocated for all the remote nodes as the send operation is asynchronous. This optimization also saves space and the saved space depends on the number of processes involved in the computation.

6.7 Experimental evaluation

We have used large-scale graphs available in the public domain for result analysis and they are listed in Table 6.3. For checking scalability we have generated RMAT graphs of bigger size with the GT-Graph [11] tool with parameter values a=0.45, b=0.25, c=0.15 and d=0.15. Different RMAT graphs were used for GPU and CPU devices due to the considerable difference in memory size of each device, and the difference in the number of GPU devices and CPU devices used. RMAT graphs used for CPU and GPU clusters are shown in Tables 6.5 and 6.4 respectively.

Algorithm 78: Code generation for synchronization of graph properties in `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

Input	Type	Points ($\|V\|$)	Edges ($\|E\|$)	max *in-degree*	max *out-degree*	avg *degree*
ljournal [28]	social	5,363,260	77,991,514	19,409	2,469	14.73
arabic [14]	web	22,744,080	631,153,669	575,618	9,905	28.14
uk2005 [15]	web	39,460,000	921,345,078	1,776,852	5,213	23.73
uk2007 [15]	web	105,896,555	3,738,733,602	975,418	15,402	35.31
twitter [59]	social	41,652,230	1,468,364,884	770,155	2,997,469	35.13
frontier [102]	social	65,608,366	1,806,067,135	4,070	3,615	27.53

Table 6.3. Input graphs and their properties

6.7.1 Distributed machines used for the experimentation

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

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.

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.

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 multi-core CPU with 32 cores and 100 GB memory.

6.7.2 CPU cluster execution

6.7.2.1 Public inputs

Figure 6.2 shows the speedup of `Falcon` over PowerGraph on a sixteen node CPU cluster for public inputs in Table 6.3. The benchmarks used are Single 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 (two different ways of partitioning graphs) [46]. It is found that the amount of data communicated by PowerGraph is high and is up to 5× to 30× more compared to `Falcon`.

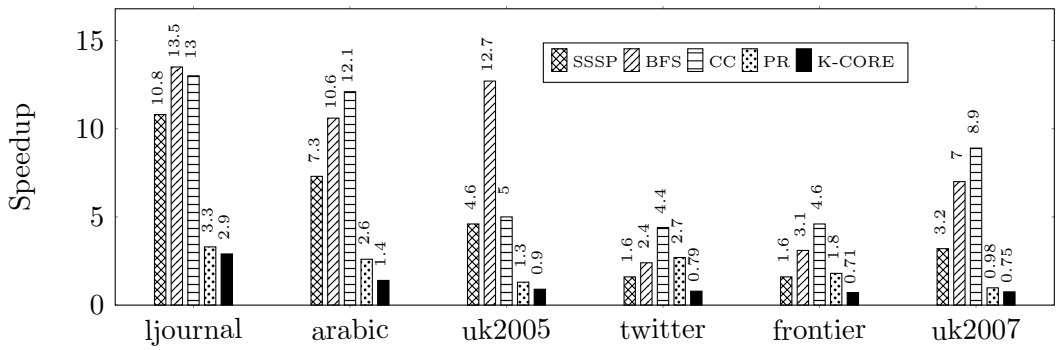
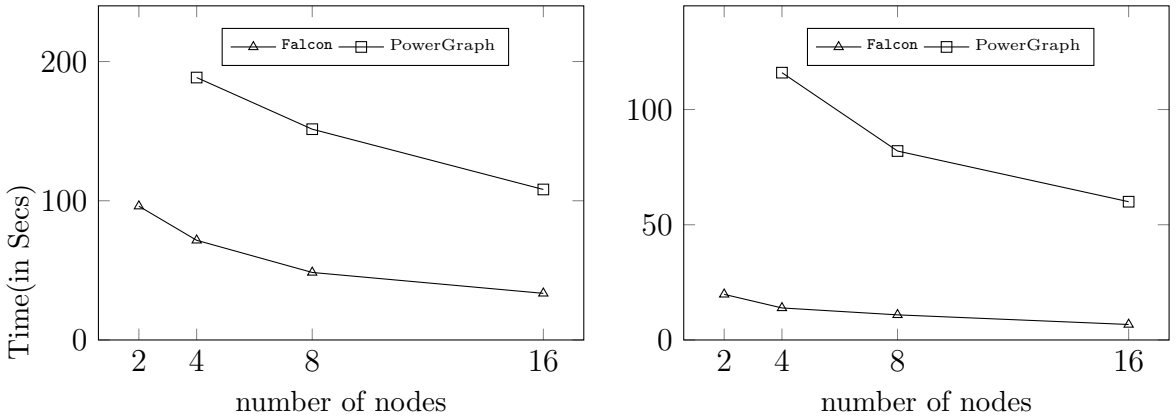


Figure 6.2: Speedup of Falcon over PowerGraph on 16 node CPU cluster



(a) SSSP

(b) Connected Components

Figure 6.3: `Falcon` Vs PowerGraph- on UK-2007

`Falcon` is able to outperform PowerGraph for most of the (benchmark, input) pairs. The major reason being the amount of data communicated by `Falcon` code much is lesser compared to PowerGraph as `Falcon` uses edge-cut partitioning and optimized communication. Pagerank and k-core implementations of `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. K-core running time is calculated as the average time of running the algorithm for 11 iterations for ($kmin = 10$) to ($kmax = 20$). The `Falcon` implementation of K-core also communicates more volume of data. Figure 6.3 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.

Input G(V,E)	rmat100	rmat200	rmat250
$\|V\|$	100M	200M	250M
$\|E\|$	1000M	2000M	2500M

Table 6.4. Input for GPU cluster and Multi-GPU machine scalability Test

Input G(V,E)	rmat300	rmat600	rmat900	rmat1200
$\|V\|$	300M	600M	900M	1200M
$\|E\|$	3000M	6000M	9000M	12000M

Table 6.5. Input for CPU cluster scalability Test

6.7.2.2 Scalability test

`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 (see Table 6.5). Scalability was compared for the benchmarks SSSP, BFS and CC and the results are shown in Table 6.6. The PowerGraph framework failed to run on the rmat1200 input. Here also, `Falcon` was able to outperform PowerGraph.

Algorithm	Framework	rmat300	rmat600	rmat900	rmat1200
SSSP	PowerGraph	158.2	358.9	442.4	segfault
	`Falcon`	112	238.9	384.7	478.7
CC	PowerGraph	107	305	324	segfault
	`Falcon`	34.9	72.9	92.4	188.8
BFS	PowerGraph	24.8	49.7	93.2	segfault
	`Falcon`	15.8	33.2	42.9	75.1

Table 6.6. Running time (in Secs) of rmat graph on fixed 16 node CPU cluster.

6.7.3 GPU execution

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

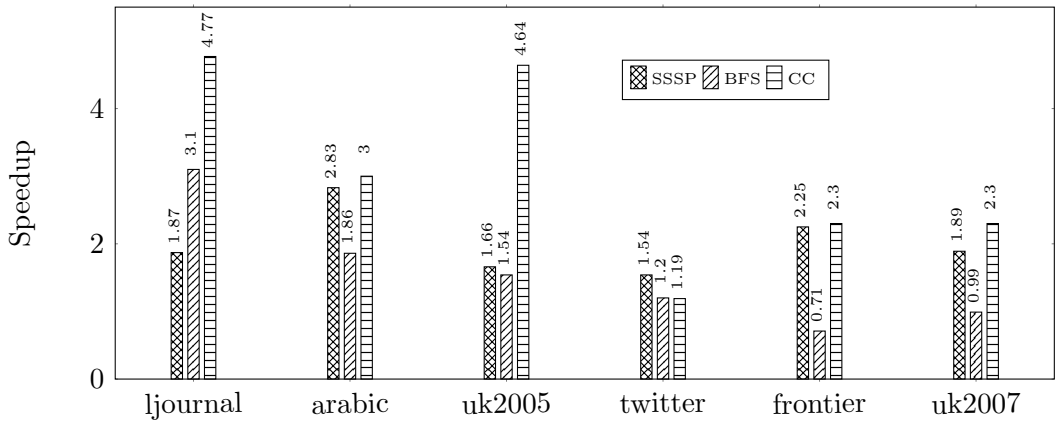


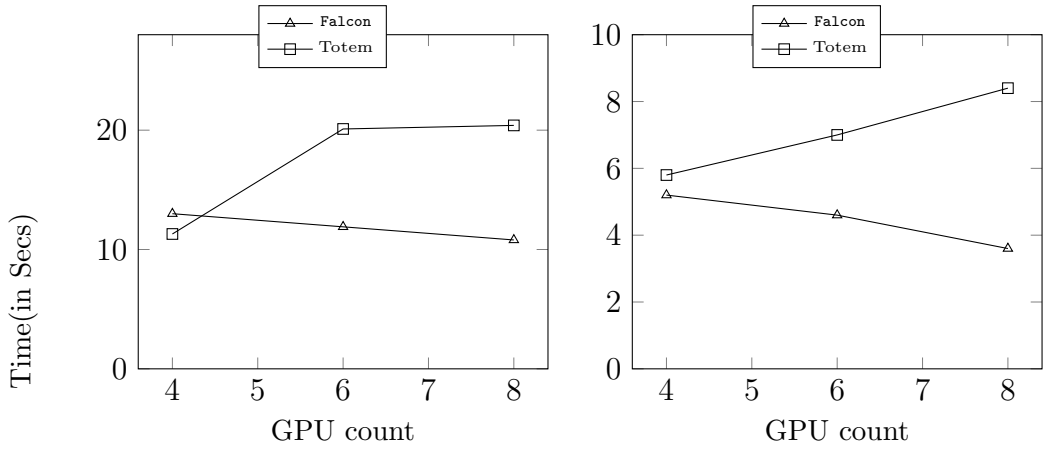
Figure 6.4: Speedup of Falcon over Totem

6.7.3.1 Multi-GPU machine

`Falcon` codes were executed for all the public inputs and `Falcon` performance is compared with Totem [44]. The results are shown in Figure 6.4 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 have *peer-access* capability. Totem showed a sharp increase in running time when the number of GPUs is changed from four to five and thereby showing non-linear scalability as it uses *peer-access* capability which is possible between all the devices when the number of GPUs is four or less. The `Falcon` compiler does not use the *peer-access* capability and showed linear capability. So if the inputs fit within four GPUs, Totem was able to achieve better performance for some inputs and benchmarks compared to `Falcon`. The `Falcon` compiler uses OpenMPI with cuda_aware_mpi feature for communication between GPUs. `Falcon` allows iterating over edges in a localgraph object using the *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 6.5 for SSSP on uk-2007 input and for CC on frontier input. Increasing the number of GPUs from x to y, will have a huge impact on the running time, when the input graph object which fits on p GPUs where $p \geq x$, has enough size and parallelism to use y GPUs. We saw in Chapter 4 that road networks perform poorly on GPUs due to the lack of parallelism. Parallelism normally decreases as the number of GPUs is increased.

6.7.3.2 GPU cluster

Figure 6.6 shows relative speedup of `Falcon` on an 8 node for GPU cluster over an 8 node CPU cluster, on public inputs. BFS algorithm shows less speedup on the GPU cluster as BFS does not have a compute-bound kernel and communication between GPUs on different nodes



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

Figure 6.5: Running time- public inputs on 8 GPU machine

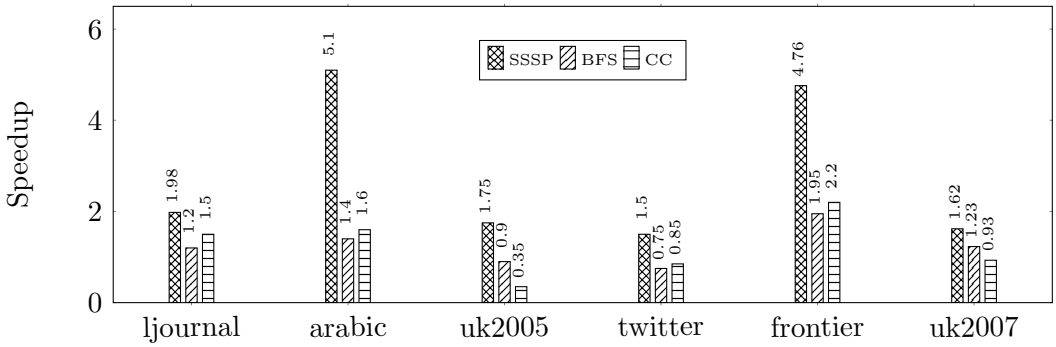


Figure 6.6: Relative speedup of 8 node GPU cluster over 8 node CPU cluster

has to go through the CPU. In BFS, nearly 90% time is spent on communication on the 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 (on an average) 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.

6.7.4 Scalability test

We conducted scalability tests on multi-GPU machines, GPU clusters, and CPU+GPU clusters. 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 equal to ten times of number of vertices (see Table6.4). Table 6.7 shows the running time for different systems on

the benchmarks BFS, CC and SSSP. First two columns show the running time of `Falcon` and Totem on a multi-GPU machine. The other two columns show the running time on distributed systems with each machine having one GPU (column III) and one GPU or CPU (column IV). The multi-GPU system has a 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 different nodes is very high. The CPU+CPU cluster has the worst performance as there is mismatch in computation time for parallel code in GPU and CPU devices.

Input	Algorithm	multi-GPU	Totem	GPUcluster	GPU+CPUcluster
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 6.7. Running time (in Secs) of rmat graph on fixed 8 devices (8 GPUs or four GPU+ four CPU).

6.7.5 Boruvka's MST

The Boruvka's MST algorithm uses the Union-Find `Set` data type of `Falcon`. This algorithm also uses the `single` statement of `Falcon`. The `single` statement is used to add only one edge connecting two disconnected components among the many possible edges with the same weight. Running time of the algorithm for public inputs is shown below in Table 6.8. The outermost `foreach` statements used the `points` iterator and the kernel with `single` statement was similar to the one given Algorithm 74. The twitter input has a similar running time on a multi-GPU machine and a GPU cluster as the iterator `points` was used and it created thread divergence. A code with `edges` iterator can be written like the SSSP example in Algorithm 65, 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 6.8. Running time of MST (in Seconds) on different distributed Systems.

6.7.6 Dynamic graph algorithms

The `Falcon` compiler allows mutation of graph objects and hence supports programming dynamic graph algorithms. The `Falcon` compiler looks at algorithms which add edges and points to the graph object and allocates more space to store edges for each vertex. A 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 the second highest in the 3-tuple (*min*, *max*, *outdegree*) for each vertex. The deletion of edges and points is done using marking (Section 5.4).

6.7.6.1 Dynamic-SSSP

The incremental dynamic-SSSP gives a speedup of around 4.5× on GPU cluster, 11× on multi-GPU machine and 7.5× on CPU cluster. The rmat graphs of Table 6.7 and Table 6.6 were used for GPU and CPU systems respectively. After the initial SSSP computation upto 5% edges were added during the experimentation to the RMAT graphs and SSSP was computed incrementally from previous computation. Other vertex-centric incremental dynamic algorithms can also be programmed in `Falcon` in a similar fashion.

6.7.6.2 Delaunay Mesh Refinement (DMR)

We have implemented distributed DMR based on the PCDM algorithm [27]. The DMR algorithm has a graph with a mesh of triangles. This algorithm is totally different from the other algorithms discussed above, where the graph is collection of triangles. So in the distributed implementation of DMR in `Falcon`, each triangle in the *localgraph*, which contains a constrained edge is added to a `Collection` object *coll*1. 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 *coll*2. The *coll*2 object will be synchronized by `Falcon` using *coll*1. Then triangles in *coll*2 will be refined in the remote node

which contains the same constrained edge. The refinement algorithm is the same as that of the PCDM algorithm. The triangular mesh is partitioned using the ParMetis [7] Tool, which provides a 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 6.9 shows the running time on different systems for the 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 6.9. Running time of DMR (in seconds) on different distributed systems.

Chapter 7

Conclusions and Future Directions

7.1 Conclusions

In this research, we have introduced the `Falcon` graph manipulation language, which targets heterogeneous systems which include multi-core CPUs, GPUs, multi-GPU machines, CPU clusters, GPU clusters. *To the best of our knowledge this is the first framework which facilitates graph manipulation for such wide range of targets.* The attractive features of `Falcon` are listed below.

- It supports a wide range of target systems.
- A `Falcon` DSL program can be converted to these heterogeneous targets, without modifying the program. The programmer is required to give only different command line arguments to the compiler to emit the codes for different target systems. This increases productivity.
- It supports mutation of graph objects, barrier for GPU Kernels and distributed locking on clusters, all of which enable programming complex graph algorithms.
- It supports mesh refinement algorithms, which are widely used in computational geometry.
- `Falcon` compiler generates very high quality code. Performance of `Falcon` programs were compared with the state-of-the-art frameworks and it matched those frameworks for most of the benchmarks and outperformed some of them.

Table 7.1 compares `Falcon` with the state-of-the-art graph frameworks.

Properties	GreenMarl	Elixir	Totem	PowerGraph	GraphLab	Lonestar GPU	Falcon
CPU	√	√	√	√	√	X	√
GPU	X	X	√	X	X	√	√
multi-GPU	X	X	√	X	X	X	√
CPU cluster	√	X	X	√	√	x	√
GPU custer	X	X	X	X	X	X	√
DSL	√	√	X	X	X	X	√
Dynamic Algorithms	X	X	X	√	√	√	√

Table 7.1. Comparison of Falcon with other DSLs/Frameworks

7.2 Future directions

- `Falcon` supports GPU devices. But there are no GPU-specific optimizations provided in `Falcon`. For example, RMAT graphs or social graphs which follow the power-law will have thread divergence when all the edges are processed using two `foreach` statements, with the outermost iterator being *points* and the inner one iterating over *outnbrs*. This can be avoided by iterating over edges (see Algorithm 65 ,Section 6.5). This forces the programmer to write different codes for the same algorithm (for GPU devices), for different input graph classes (see Section 2.8). An optimization which converts *points*+*outnbrs* iterator pair to *edges* iterator will avoid writing separate codes. In the future we wish to explore similar optimizations which benefit some types of graph classes.

- Similarly, the current implementation of distributed locking using a `single` statement, and distributed union of `Set` data type in `Falcon` have high overhead as all the requests are collected at the first process ($rank == 0$). The possibility of optimizing these operations is also part of future work.

- Currently, mesh based algorithms are programmed in `Falcon` using the *addProperty()* function of the `Graph` class. Providing data types `Mesh`, `Hypergraph` etc., and adding support in the compiler for writing algorithms for such data types is desirable.

- `Falcon` currently supports only *cautious morph* algorithms, which we would like to extend to speculations which require rollbacks to the previous consistent state on mis-speculation. Targeting other devices like Xeon-Phi coprocessor, FPGAs are also on the cards.

Chapter 8

Appendix

8.1 Absolute Running Time- Single Machine CPU and GPU

Input	Copy Graph to GPU			From initialization to end of while			While loop			CopyResult to CPU		
	Falcon GPU	Lonestar GPU	Totem GPU	Falcon GPU	Lonestar GPU	Totem GPU	Falcon GPU	Lonestar GPU	Totem GPU	Falcon GPU	Lonestar GPU	Totem GPU
rand1	192.1	210.7	418.7	404.7	463.8	651.4	401.8	459.1	591.7	42.84	43.5	55.7
rand2	460.3	541.7	848.2	845.7	1149	1303	840	1142.7	1190	84.33	85.3	137.6
rmat1	246.77	252.5	250.1	697	1252.7	1051.3	695.26	1248.4	1011.3	26	25.8	35.4
rmat2	489.5	508.8	498.7	1558.9	10424.7	2549.9	1555.7	10371.7	2483	51.7	62.3	73.1
road1	158.7	225.9	464	333086	45647.7	74220.5	33083.9	45646.7	74047.5	31.7	26.15	163
road2	192.2	324.3	7596.6	78127.3	152321.5	134688	78123.6	1522817	134613	45.9	48.7	70

Table 8.1. Running Time(in Ms) SSSP on GPU(Falcon-GPU,LonestarGPU,Totem-GPU)

Input	Copy Graph to GPU			From initialization to end of while			While loop			CopyResult to CPU		
	Falcon GPU	Lonestar GPU	Totem GPU	Falcon GPU	Lonestar GPU	Totem GPU	Falcon GPU	Lonestar GPU	Totem GPU	Falcon GPU	Lonestar GPU	Totem GPU
rand1	192	495.3	418	100.84	153.1	215.3	99.15	149.7	176.1	51.8	45	40.2
rand2	433	664	839	199	323,7	419	196.3	318,7	350.4	85.77	85.7	63.6
rmat1	279.1	568.2	251,4	158.6	324.5	121.6	157.1	319	97.4	26.0	25.8	20.1
rmat2	474.7	704.7	498.7	319.2	3246.4	250.7	317.3	3244.6	219.8	51.8	45	40.2
road1	151.7	379.7	462.7	380.1	263.7	5304	379.1	260.5	5191	35.3	26.7	39.7
road2	424.4	785.3	597.3	601.4	405	11492.5	632.3	400	11523	45.8	47.1	38.92

Table 8.2. Running Time(in Ms) BFS on GPU(Falcon-GPU,LonestarGPU,Totem-GPU)

Input	Copy Graph to GPU		From initialization to end of while		While loop		CopyResult to CPU	
	Falcon GPU	Lonestar GPU	Falcon GPU	Lonestar GPU	Falcon GPU	Lonestar GPU	Falcon GPU	Lonestar GPU
rand1	632.5	837.7	1521.3	3763.7	1512	different code	179	different code
rand2	1298.7	1217	4511	9379.2	4495	different code	440	different code
rmat1	663	653.1	3641.1	3782.4	3632.6	different code	238.8	different code
rmat2	1262.2	1241.1	7889.7	7907	7873	different code	525	different code
road1	566.9	189.3	881.4	3146.4	874.3	different code	88.5	different code
road2	923.6	743.7	1072.67	4333.4	1061.3	different code	155.3	different code

Table 8.3. Running Time(in Ms) MST on GPU(Falcon-GPU,LonestarGPU)

Input	Galois-1	Galois-12	Falcon-CPU	Totem-CPU	Green-Marl
rand1	20025	2314.3	2743.3	2620.7	3451
rand2	52283	4375.7	5327	5322	7578
rmat1	18683	2566	2677	4011	killed
rmat2	47779	5693	5691.3	9663	killed
road1	3600	735	734	103981	122651
road2	5699	1023	1012	183619	197512

Table 8.4. Running Time(in Ms) for SSSP on CPU

Input	Galois-1	Galois-12	Falcon-CPU	Totem-CPU	Green-Marl
rand1	5832.5	680.5	420.5	716	1017
rand2	12074.5	1464.3	949.3	1468.1	1952
rmat1	3345	501.4	397.6	512.1	killed
rmat2	7133.5	995	825.9	1009.1	killed
road1	2042	320	325.5	2387.7	1225.2
road2	3063.5	520	482.7	4686.2	6127

Table 8.5. Running Time(in Ms) for BFS on CPU

Input	Galois-1	Galois-12	Falcon-CPUl
rand1	49861.8	5036	9795
rand1	110804	11419	20827
rmat1	29188	6910	14503
rmat2	63372	14972	36784
road1	11180	1402	1421
road2	12013	1510	1615

Table 8.6. Running Time(in Ms) for MST on CPU

8.2 Absolute Running Time- Distributed Systems

Input	BenchMark	Falcon	PowerGraph
ljournal	BFS	0.6	8.1
ljournal	SSSP	1.32	14.3
ljournal	CC	0.8	9
arabic	BFS	1.08	11.5
arabic	SSSP	6.8	49.8
arabic	CC	1.32	16
uk2005	BFS	3.36	42.6
uk2005	SSSP	22	101
uk2005	CC	5.16	26
uk2007	BFS	4.9	32.2
uk2007	SSSP	33.5	108.1
uk2007	CC	6.72	60
twitter	BFS	3.1	7.3
twitter	SSSP	9.5	4.8
twitter	CC	5.04	22
frontier	BFS	5.63	17.2
frontier	SSSP	31.4	49.6
frontier	CC	10.56	49

Table 8.7. Absolute Running Time in Seconds on 16 node CPU cluster Falcon and PowerGraph

Input	BFS	SSSP	CC
ljournal	0.4	0.9	0.4
arabic	1.6	5.1	1.45
uk2005	12.4	16.8	10
uk2007	8.1	30.1	8.9
twitter	13.1	25.3	10.9
frontier	3.5	9.4	3.2

Table 8.8. Absolute Running Time in Seconds of Falcon 8 node GPU cluster

Input	BenchMark	Falcon	Totem
ljournal	BFS	0.13	.4
ljournal	SSSP	.44	.84
ljournal	CC	0.15	.7
arabic	BFS	.24	.45
arabic	SSSP	1.9	5.4
arabic	CC	0.7	2.12
uk2005	BFS	.75	1.15
uk2005	SSSP	5.3	8.76
uk2005	CC	.94	7.85
uk2007	BFS	1.3	1.3
uk2007	SSSP	10.7	20.4
uk2007	CC	6.8	15.8
twitter	BFS	2.7	3.2
twitter	SSSP	2.67	4.1
twitter	CC	2.69	3.2
frontier	BFS	1.2	.87
frontier	SSSP	10.6	23.9
frontier	CC	10.56	49

Table 8.9. Absolute Running In Seconds on Multi-GPU machine with 8 GPUs Falcon and Totem

8.3 Example Falcon Programs

This section gives some sample Falcon programs.

More programs can be found in, Unnikrishnan Cheramangalath, Rupesh Nasre, and Y. N. Srikant. Falcon Graph DSL Compiler Version 1, Jan 2017. [Online]. Available at https://github.com/falco graphdsl/ .

8.3.1 SSSP in `Falcon`

Algorithm 79: Bellman-Ford SSSP code in `Falcon`

```
1  int changed=0;
2  relaxgraph( Point p,Graph graph) {
3  |   foreach( t In p.outnbrs ){
4  |   |   MIN(t.dist,p.dist+graph.getWeight(p,t),changed);
5  |   }
6  }
7  SSSP(char *name) {
8  |   Graph hgraph;
9  |   hgraph.addPointProperty(dist,int);
10 |   hgraph.read(name);
11 |   foreach(t In hgraph.points)t.dist=1234567890;
12 |   hgraph.points[0].dist=0;
13 |   while( 1 ){
14 |   |   changed=0;
15 |   |   foreach(t In hgraph.points)relaxgraph(t,hgraph);
16 |   |   if(changed==0) break;
17 |   }
18 |   for(int i=0;i<hgraph.npoints;i++)
19 |        printf("%d\n",hgraph.points[i].dist);
20 |   return;
21 }
22  main(int argc,char *argv[]) {
23 |   SSSP(argv[1]);
24 }
```

The Algorithm 80 is the optimized variant program shown in the Algorithm 79. Here the code is optimized to mak sure that only the nodes on which there is a need to take atomic MIN operation will do the same. To be more precise, all points which have the *uptd* property `false` will not execute the code inside *relaxgraph()* function, reducing running time. The optimized code has extra-properties *dist*, *olddist*, *uptd*. So this code will produce correct output, but takes less time than that of algorithm given in 8.3.1.

Algorithm 80: Optimized Bellman-Ford SSSP code in `Falcon`

```
1  int changed = 0;
2  relaxgraph(Point p, Graph graph) {
3  |   foreach( t In p.outnbrs ){
4  |   |   MIN(t.dist, p.dist + graph.getWeight(p, t), changed);
5  |   |   if(t.dist<t.olddist)t.uptd=true;
6  |   }
7  }
8  reset( Point t, Graph graph) {
9  |   t.dist=t.olddist=1234567890; t.uptd=false;
10 }
11 reset1( Point t, Graph graph) {
12 |   if(t.uptd==true && t.dist==t.olddist)t.uptd=false;
13 |   t.olddist=t.dist;
14 }
15 main(int argc, char *argv[]) {
16 |   Graph graph; // graph on CPU
17 |   graph.addPointProperty(dist, int);
18 |   graph.addPointProperty(upt, bool);
19 |   graph.addPointProperty(olddist, int);
20 |   graph.read(argv[1]) // read graph on CPU
21 |   // initialize graph object properties
22 |   foreach (t In graph.points)reset(t,graph);
23 |   graph.points[0].dist = 0; // source has dist 0
24 |   graph.points[0].uptd=true;
25 |   while( 1 ){
26 |   |   changed = 0; //keep relaxing
27 |   |   foreach(t In graph.points)
28 |   |         (t.uptd) relaxgraph(t,graph);
29 |   |   if(changed == 0)break;//reached fix point
30 |   |   foreach(t In graph.points)reset1(t,graph)
31 |   }
32 |   for(int i = 0; i <graph.npoints; ++i)
33 |       printf("i=%d dist=%d\n", i, graph.points[i].dist);
34 }
```

Algorithm 81: Δ-stepping SSSP in `Falcon`

```
1  Graph hgraph;
2  struct node {
3      Point(hgraph) n1;
4      int w;
5  };
6  Collection pred[struct node];
7  relaxEdge(Point p,Graph hgraph,Point p1,int weight,Collection pred[struct node]) {
8      int changed=0;
9      int ddata=hgraph.points[p].dist;
10     int newdist=hgraph.points[p1].dist+weight;
11     while( newdist <(olddist=hgraph.points[p].dist) ){
12         MIN(hgraph.points[p].dist,newdist,changed);
13         struct node tt1;
14         tt1.w=newdist;
15         tt1.n1=p;
16         pred.add(tt1);
17     }
18 }
19 relaxNode1(struct node req,Graph hgraph,Collection pred[struct node]) {
20     Point (hgraph) p1;
21     struct node temp;
22     temp=req;
23     p1=temp.n1;
24     foreach( t In p1.outnbrs ){
25         int weight=hgraph.getWeight(p1,t);
26         relaxEdge(t,hgraph,p1,weight,pred);
27     }
28 }
29 int main(int argc,char *argv[]) {
30     hgraph.addPointProperty(dist,int);
31     Point (hgraph) p;
32     hgraph.read(argv[3]);
33     pred.OrderByIntValue(w,10);
34     foreach(t In hgraph.points)t.dist=1234567890;
35     p=hgraph.points[0];
36     hgraph.points[p].dist=0;
37     foreach( t In p.outnbrs ){
38         int weight=hgraph.getWeight(p,t);
39         relaxEdge(t,hgraph,p,weight,pred);
40     }
41     foreach(t In pred)relaxNode1(t,hgraph,pred);
42     int maxdist=0;
43     for( int i=0;i<hgraph.npoints;i++ ){
44         if(hgraph.points[i].dist>maxdist)maxdist=hgraph.points[i].dist;
45     }
46     printf("MAX DIST=%d \n", maxdist);
47 }
```

Algorithm 82: Worklist Based SSSP in `Falcon`

```
1  int changed=0, coll1cnt=0, coll2cnt=0,hchanged;
2  relaxgraph(Point p,Graph graph, Collection coll1[Point(graph)] ,Collection coll2[Point(graph)] ,
   int val) {
3      int ch;
4      foreach( t In p.outnbrs ){
5          int newdist=graph.getWeight(p,t);
6          if( t.dist>newdist+p.dist ){
7              MIN(t.dist,newdist+p.dist,ch);
8              coll2.add(t);
9              changed=1;
10         }
11     }
12 }
13 SSSP(char *name) {
14     Graph graph;
15     graph.addPointProperty(dist,int);
16     int xx=0,temp=0;
17     graph.read(name);
18     Collection coll1[Point(graph)],coll2[Point(graph)],coll3[Point(graph)];
19     foreach(t In graph.points)t.dist=1234567890;
20     coll1.add(graph.points[0]);
21     graph.points[0].dist=0;
22     foreach(t In coll1)relaxgraph(t,graph,coll1,coll2,xx);
23     while( 1 ){
24         changed=0;
25         coll3=coll1;
26         coll1=coll2;
27         coll2=coll3;
28         temp=coll2.size;
29         coll1.size=temp;
30         temp=0;
31         coll2.size=temp;
32         foreach(t In coll1)relaxgraph(t,graph,coll1,coll2,xx);
33         if(changed==0)break;
34     }
35     int maxdist=0;
36     for( int i=0;i<graph.npoints;i++ ){
37         if(maxdist <graph.points[i].dist) maxdist=graph.points[i].dist;
38     }
39     printf("MAXDIST=%d \n",maxdist);
40 }
41 int main(int argc, char *argv[]) {
42     SSSP(argv[1]);
43 }
```

8.3.2 BFS in `Falcon`

Algorithm 83: BFS code in `Falcon`

```
1  int changed=0;
2  BFS(Point p,Graph graph) {
3      foreach(t In p.outnbrs)MIN(t.dist,p.dist+1,changed);
4  }
5  main(int argc, char *name[]) {
6      Graph hgraph;
7      hgraph.addPointProperty(dist,int);
8      hgraph.read(name[1]);
9      foreach(t In hgraph.points)t.dist=1234567890;
10     hgraph.points[0].dist=0;
11     while( 1 ){
12         changed=0;
13         foreach(t In hgraph.points)BFS(t,hgraph);
14         if(changed==0)break;
15     }
16     for(int i=0;i<hgraph.npoints;i++)
17          printf("%d\n",hgraph.points[i].dist);
18     return;
19 }
```

Algorithm 84 shows the optimized BFS code in `Falcon`. There is not much difference from code for BFS given in Algorithm 83. Here code is optimized for running time. In first invocation of *relaxgraph()* only source node will reduce *dist* value of its neighbours as `foreach` call is with the condition ($t.dist == leve$) (Line 18, Algorithm84) . If we say in other words the i^{th} invocation will reduce distance of neighbours of nodes whose distance value is $(i-1)$. So we make foreach call to have a conditional to make sure that no unwanted computation take place and there is no atomic operations in the code. This will produce correct output and will run faster.

Algorithm 84: Atomic free Optimized BFS code in `Falcon`

```
1  int changed=0,lev=0;
2  BFS(Point p,Graph graph) {
3      foreach( t In p.outnbrs ){
4          if( t.dist>p.dist+1 ){
5              t.dist=p.dist+1;
6              changed=1;
7          }
8      }
9  }
10 main(int argc, char *name[]) {
11     Graph hgraph;
12     hgraph.addPointProperty(dist,int);
13     hgraph.read(name[1]);
14     foreach(t In hgraph.points)t.dist=1234567890;
15     hgraph.points[0].dist=0;
16     while( 1 ){
17         changed=0;
18         foreach(t In hgraph.points)(t.dist==lev)BFS(t,hgraph);
19         if(changed==0)break;
20         lev++;
21     }
22     for(int i=0;i<hgraph.npoints;i++)printf("%d\n",hgraph.points[i].dist);
23     return;
24 }
```

8.3.3 MST code in `Falcon`

The worlist based algorithm uses `FalconCollection` data type. The `Collection` gets converted to `Galois::InsertBag` data structure, which is a worklist. This code will be converted to a code similar to the one which can be found in Galois-2.2 Boruvka MST code.

Algorithm 85: Boruvka MST(all targets part1

```
1  struct node{
2  int lock,weight;
3  Point set,src,dst;
4  };
5  int hchanged, changed;
6   void reset(Point p, Graph graph,Set set[Point(graph)]) {
7       p.minppty.set.reset();//reset sets value to with MAX_INT
8       p.minppty.src.reset();//replaced with reset()
9       p.minppty.dst.reset();//replaced with reset()
10      p.minppty.weight=99999999;
11      p.minedge=99999999;
12      p.minppty.lock=0;
13 }
14 void minset(Point p, Graph graph,Set set[Point(graph)]) {
15      int ch;
16      Point (graph) t1;
17      Point (graph) t2;
18      foreach( t In p.outnbrs ){
19          t1=set.find(p);
20          p.minedge=99999999;
21          t2=set.find(t);
22          if( t1!=t2 ){
23              MIN(t1.minppty.weight,graph.getWeight(p,t),ch);//find minimum edge out going
                from t1
24              MIN(t2.minppty.weight,graph.getWeight(p,t),ch);//find minimum edge out going
                from t2.
25          }
26      }
27 }
28 void mstunion(Point p,Graph graph,Set set[Point(graph)]) {
29      Point (graph)t1,(graph)t2;
30      int t3,t4;
31      t1=set.find(p);
32      t2=t1.minppty.set;t3=t1.minppty.lock;t4=t2.minppty.lock;
33      if( t1!=t2 && t3==1 ){
34          set.Union(t1,t2);
35          changed=1;
36      }
37 }
38 initmark(Edge e, Graph graph {
39      e.mark=999999999;
40 }
```

Algorithm 86: Boruvka MST (all targets) part2

```
1  void Minedge(Point p,Graph graph,Set set[Point(graph)]) {
2      Point(graph) t1,(graph)t2;
3      int t3;
4      Edge (graph) e;
5      foreach( t In p.outnbrs ){
6          t1=set.find(p);
7          t2=set.find(t);
8          t3=graph.getWeight(p,t);
9          if( t1!=t2 ){
10             if( t3==t1.minppty.weight ){
11                 single( t1.minppty.lock ){
12                     e=graph.getedge(p,t);
13                     e.mark=true;////add edge to mst
14                     t1.minppty.src=p;
15                     t1.minppty.dst=t;
16                     t1.minppty.weight=t3;
17                     t1.minppty.set=t2;
18                 }
19             }
20             if( t3==t2.minppty.weight ){
21                 single( t2.minppty.lock ){
22                     e=graph.getedge(p,t);
23                     e.mark=true;////add edge to mst
24                     t2.minppty.src=p;
25                     t2.minppty.dst=t;
26                     t2.minppty.weight=t3;
27                     t2.minppty.set=t2;
28                 }
29             }
30         }
31     }
32 }
```

Algorithm 87: Boruvka MST (all targets) part3

```
1  int main(int argc,char *argv[]) {
2      Graph hgraph;
3      hgraph.addPointProperty(minppty,struct node);
4      hgraph.addEdgeProperty(mark,bool);
5      hgraph.addNodeProperty(minedge,int);
6      hgraph.getType() graph;
7      hgraph.read(argv[1]);
8      Set hset[Point(hgraph)],set[Point(graph)];
9      graph=hgraph;
10     set=hset;
11     foreach(t In graph.edges)initmark(t,graph);
12     while( 1 ){
13         changed=0;
14         foreach(t In graph.points)reset(t,graph,set);
15         foreach(t In graph.points)minset(t,graph,set);
16         foreach(t In graph.points)Minedge(t,graph,set);
17         foreach(t In graph.points)mstunion(t,graph,set);
18         if(changed==0)break;
19     }
20     hgraph.mark=graph.mark;
21     unsigned long int mst=0;
22     foreach( t In hgraph.edges ){
23         if(t.mark==1)mst=mst+t.weight;
24     }
25 }
```

Algorithm 88: Worklist based MST in `Falcon` for CPU device part1

```
1  Graph hgraph;
2  Set hset[Point(hgraph)];
3  struct node{ Point (hgraph) src,Point (hgraph) dst;
4  int weight;
5  };
6  int glimit,int bcnt;
7  struct workitem{
8  Point (hgraph) src,Point (hgraph) dst;
9  int weight,int cur;
10 };
11 Collection WL1[struct workitem],WL2[struct workitem],WL3[struct workitem],WL4[struct
   workitem];
12 Collection mst[struct node],*current[struct workitem],*next[struct workitem],*pending[struct
   workitem];
13 Collection *temp[struct workitem];
14 findLightest(struct workitem req,Graph hgraph,Collection pred [struct workitem],Collection
   next[struct workitem],Collection pending[struct workitem],int useLimit,Set hset[Point(hgraph)])
   {
15     struct workitem req1=req;
16     Point (hgraph) src=req1.src;
17     int cur=req1.cur;
18     foreach( t In src.outnbrs ){
19         int weight=hgraph.getWeight(src,t);
20         if( useLimit && weight>glimit ){
21             struct workitem tt;
22             tt.src=src; tt.dst=t; tt.weight=weight; tt.cur=cur;
23             pending.add(tt);
24             return;
25         }
26         Point (hgraph) rep=hset.find(src);
27         Point (hgraph) dst;
28         int old,ch;
29         if( rep!=hset.find(t) ){
30             struct workitem tt;
31             tt.src=src;tt.dst=t;tt.weight=weight;tt.cur=cur;
32             next.add(tt);
33             while(weight <(old=hgraph.points[rep].minedge)){ MIN(rep.minedge,weight,ch);
34             } return;
35         }
36     }
37 }
```

Algorithm 89: Worklist based MST in `Falcon` for CPU device part2

```
1  findLightest1(struct workitem req,Graph hgraph,Collection pred [struct workitem],Set
   hset[Point(hgraph)]) {
2      Point (hgraph) src=req.src;
3      Point (hgraph) rep=hset.find(src);
4      int cur=req.cur;
5      if( rep<hgraph.npoints && hgraph.points[rep].minedge==req.weight ){
6          Point (hgraph) dst=req.dst;
7          if( dst<hgraph.npoints && (rep=hset.Union(rep,dst)) ){
8              hgraph.points[rep].minedge=1234567890;
9              struct node tt;
10             tt.src=src;
11             tt.dst=dst;
12             tt.weight=req1.weight;
13             mst.add(tt);
14         }
15     }
16 }
```

Algorithm 90: Worklist based MST in `Falcon` for CPU device part2

```
1  findLightest2(Point p,Graph hgraph,Collection pred [struct workitem],Collection next[struct
   workitem],Collection pending[struct workitem],int useLimit,Set hset[Point(hgraph)]) {
2      struct workitem req;
3      req.src=p;
4      req.cur=0;
5      findLightest(req, hgraph,pred,next, pending, useLimit, hset);
6  }
7   int main(int argc,char *argv[]) {
8      hgraph.addPointProperty(minedge,int);
9      int a;
10     hgraph.read(argv[3]);
11     Set hset[Point(hgraph)];
12     for(int i=0;i<hgraph.npoints;i++)hgraph.sortEdges(i);
13     glimit=2000;
14     current=&WL1;
15     next=&WL2;
16     pending=&WL3;
17     int bcnt=0;
18     for(int i=0;i<hgraph.npoints;i++){ hgraph.points[i].minedge=1234567890;
19     } foreach(t In hgraph.points)findLightest2(t,hgraph,*current,*next,*pending,bcnt,hset);
20     bcnt=1;
21     while( 1 ){
22         while( 1 ){
23             foreach(t In *current)
24             findLightest1(t,hgraph,*current,hset);
25             temp=current;current=next; next=temp;
26             foreach(t In *current)
27             findLightest(t,hgraph,*current,*next,*pending,bcnt,hset);
28             if(*next.empty())break;
29         }
30         temp=current;current=pending; next=temp;
31         if(*pending.empty())break;
32     }
33 }
```

References

[1] *J.B. Kruskals. On the shortest spanning subtree of a graph and the traveling salesman problem*, 1956. Proceedings of the American Mathematical Society. URL http://www.ams.org/journals/proc/1956-007-01/S0002-9939-1956-0078686-7/home.html. 8

[2] *R.C. Prim. Shortest connection networks and some generalizations*, 1957. Bell System Techn. J. 36. URL http://ieeexplore.ieee.org/document/6773228/?arnumber=6773228&tag=1. 8

[3] *R Bellan. On a routing problem*, 1958. Quart. Appl. Math. 16. URL http://www.ams.org/mathscinet-getitem?mr=0102435. 8

[4] Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. In *Internet Mathematics*, 2008. 20

[5] Built-in functions for atomic memory access in GCC. 2009. URL https://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Atomic-Builtins.html. 76, 96

[6] Gacl: A vertex-centric cuda/c++ api for large graph analytics on gpus using the gather-apply-scatter abstraction. 2013. 55

[7] Parmetis - parallel graph partitioning and fill-reducing matrix ordering, 2013. URL http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview. 118, 137

[8] Amine Abou-Rjeili and George Karypis. Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 124–124, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 1-4244-0054-6. URL http://dl.acm.org/citation.cfm?id=1898953.1899055. 20

[9] A. Davidson and. Baxter, M. Garland, and J. D. Owens. Work-Efficient Parallel GPU Methods for Single Source Shortest Paths. In *Proceedings of the 2014 IEEE 28th International Symposium on Parallel and Distributed Processing*, IPDPS 2014. 47, 53, 61

[10] D. Bader and K. Madduri. Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *Proceedings of the 2008 IEEE 22th International Symposium on Parallel and Distributed Processing*, IPDPS 2008. 61

[11] David A. Bader and Kamesh Madduri. Gtgraph: A synthetic graph generator suite, 2006. 100, 128

[12] Albert-Laszlo Barabasi. *Linked: How Everything Is Connected to Everything Else and What It Means*. Plume, 2003. ISBN 0452284392. 25

[13] A. Bickle. *The K-cores of a Graph*. Western Michigan University, 2010. URL https://books.google.co.in/books?id=MJ_mZwEACAAJ. 9

[14] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Softw. Pract. Exper.*, 34(8):711–726, July 2004. ISSN 0038-0644. doi: 10.1002/spe.587. URL http://dx.doi.org/10.1002/spe.587. 130

[15] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A large time-aware web graph. *SIGIR Forum*, 42(2):33–38, November 2008. ISSN 0163-5840. doi: 10.1145/1480506.1480511. URL http://doi.acm.org/10.1145/1480506.1480511. 130

[16] John Adrian Bondy. *Graph Theory With Applications*. Elsevier Science Ltd., Oxford, UK, UK, 1976. ISBN 0444194517. iv, 6

[17] A. Braunstein, M. Mzard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms*, 27(2):201–226, 2005. ISSN 1098-2418. doi: 10.1002/rsa.20057. URL http://dx.doi.org/10.1002/rsa.20057. 104

[18] Alfredo Braunstein, Marc Mézard, and Riccardo Zecchina. Survey propagation: an algorithm for satisfiability. *CoRR*, cs.CC/0212002, 2002. URL http://arxiv.org/abs/cs.CC/0212002. 9

[19] Alain Bretto. *Hypergraph Theory: An Introduction*. Springer Publishing Company, Incorporated, 2013. ISBN 3319000799, 9783319000794. 26

[20] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, September 1973. ISSN 0001-0782. doi: 10.1145/362342.362367. URL http://doi.acm.org/10.1145/362342.362367. 10

[21] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010. ISSN 2150-8097. doi: 10.14778/1920841.1920881. URL http://dx.doi.org/10.14778/1920841.1920881. 60

[22] Martin Burtscher and Keshav Pingali. CUDA Implementation of the Tree-based Barnes Hut n-Body Algorithm. In *GPU Computing Gems Emerald Edition*, pages 75–92. Morgan Kaufmann, 2011. ISBN 978-0-12-384988-5. URL http://iss.ices.utexas.edu/Publications/Papers/burtscher11.pdf. 47, 61

[23] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex cover. *J. Algorithms*, 41(2):280–301, November 2001. ISSN 0196-6774. doi: 10.1006/jagm.2001.1186. URL http://dx.doi.org/10.1006/jagm.2001.1186. 9

[24] Unnikrishnan Cheramangalath, Rupesh Nasre, and Y. N. Srikant. Falcon: A graph manipulation language for heterogeneous systems. *ACM Trans. Archit. Code Optim.*, 12(4):54:1–54:27, December 2015. ISSN 1544-3566. doi: 10.1145/2842618. URL http://doi.acm.org/10.1145/2842618. 62, 114, 116

[25] Unnikrishnan Cheramangalath, Rupesh Nasre, and Y. N. Srikant. Dh-falcon: A language for large-scale graph processing on distributed heterogeneous systems. 2017. 62

[26] L. Paul Chew. Guaranteed-quality Mesh Generation for Curved Surfaces. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, SCG '93, pages 274–280, New York, NY, USA, 1993. ACM. ISBN 0-89791-582-8. doi: 10.1145/160985.161150. URL http://doi.acm.org/10.1145/160985.161150. 4, 23, 67, 68, 70

[27] L. Paul Chew, Nikos Chrisochoides, and Florian Sukup. Parallel constrained delaunay meshing, 1997. 136

[28] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 219–228, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-495-9.

doi: 10.1145/1557019.1557049. URL http://doi.acm.org/10.1145/1557019.1557049. 130

[29] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, August 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824077. URL http://dx.doi.org/10.14778/2824032.2824077. 59

[30] Shane Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013. ISBN 9780124159334, 9780124159884. 28

[31] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511. iv, 8

[32] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998. ISSN 1070-9924. doi: 10.1109/99.660313. URL http://dx.doi.org/10.1109/99.660313. 27

[33] DIMACS. 9th dimacs implementation challenge. 2009. URL http://www.dis.uniroma1.it/~challenge9/download.shtml. 100

[34] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8. doi: 10.1145/1851476.1851593. URL http://doi.acm.org/10.1145/1851476.1851593. 60

[35] P. Erds and A Rnyi. On the Evolution of Random Graphs. In *PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES*, pages 17–61, 1960. 25, 100

[36] Rajiv Gupta Farzad Khorasani. *High Performance Vertex-Centric Graph Analytics on GPUs*. PhD thesis, 2016. 54

[37] Min Feng, Rajiv Gupta, and Laxmi N. Bhuyan. Speculative Parallelization on GPGPUs. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of*

Parallel Programming, PPoPP '12, pages 293–294, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145860. URL http://doi.acm.org/10.1145/2145816.2145860. 55

[38] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994. 31

[39] Daniele Frigioni, Mario Ioffreda, Umberto Nanni, and Giulio Pasqualone. Experimental analysis of dynamic algorithms for the single source shortest paths problem. *J. Exp. Algorithmics*, 3, September 1998. ISSN 1084-6654. doi: 10.1145/297096.297147. URL http://doi.acm.org/10.1145/297096.297147. 21, 105

[40] Zhisong Fu, Michael Personick, and Bryan Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, pages 2:1–2:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2982-8. doi: 10.1145/2621934.2621936. URL http://doi.acm.org/10.1145/2621934.2621936. 55

[41] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004. 32

[42] Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Comput. Surv.*, 23(3):319–344, September 1991. ISSN 0360-0300. doi: 10.1145/116873.116878. URL http://doi.acm.org/10.1145/116873.116878. 86

[43] Philippe Galinier and Alain Hertz. A survey of local search methods for graph coloring. *Comput. Oper. Res.*, 33(9):2547–2562, September 2006. ISSN 0305-0548. doi: 10.1016/j.cor.2005.07.028. URL http://dx.doi.org/10.1016/j.cor.2005.07.028. 9

[44] Abdullah Gharaibeh, Lauro Beltrão Costa, Elizeu Santos-Neto, and Matei 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*, PACT '12, pages 345–354, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1182-3. doi: 10.1145/2370816.2370866. URL http://doi.acm.org/10.1145/2370816.2370866. 51, 61, 99, 109, 133

[45] Abdullah Gharaibeh, Elizeu Santos-Neto, Lauro Beltrão Costa, and Matei Ripeanu. The Energy Case for Graph Processing on Hybrid CPU and GPU Systems. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, IA3 13, pages 2:1–2:8, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2503-5. doi: 10.1145/2535753.2535755. URL http://doi.acm.org/10.1145/2535753.2535755. 51, 61, 99

[46] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387883. 2, 57, 61, 108, 130

[47] Douglas Gregor and Andrew Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC)*, 2005. 61

[48] Chris Groër, Blair D. Sullivan, and Steve Poole. A mathematical analysis of the r-mat random graph generator. *Netw.*, 58(3):159–170, October 2011. ISSN 0028-3045. doi: 10.1002/net.20417. URL http://dx.doi.org/10.1002/net.20417. 26

[49] Pawan Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. 47, 61

[50] Pawan Harish, Vibhav Vineet, and P. J. Narayanan. Large Graph Algorithms for Massively Multithreaded Architectures. Technical report, 2009. 47, 61

[51] Mark Harris. Optimizing parallel reduction in cuda. *http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf*, 2007. 96

[52] Keld Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126:106–130, 2000. 10

[53] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle 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*, ASPLOS XVII, pages 349–362, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2151013. URL http://doi.acm.org/10.1145/2150976.2151013. 3, 38, 61, 66, 99

[54] Sungpack Hong, Semih Salihoglu, Jennifer Widom, and Kunle Olukotun. Simplifying Scalable Graph Processing with a Domain-Specific Language. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 208:208–208:218, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2670-4. doi: 10.1145/2544137.2544162. URL http://doi.acm.org/10.1145/2544137.2544162. 60, 61

[55] Donald B. Johnson. A note on dijkstra's shortest path algorithm. *J. ACM*, 20(3):385–388, July 1973. ISSN 0004-5411. doi: 10.1145/321765.321768. URL http://doi.acm.org/10.1145/321765.321768. 8, 45

[56] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. Adaptive heterogeneous scheduling for integrated gpus. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 151–162, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2809-8. doi: 10.1145/2628071.2628088. URL http://doi.acm.org/10.1145/2628071.2628088. 55

[57] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465369. URL http://doi.acm.org/10.1145/2465351.2465369. 60

[58] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. CuSha: Vertex-centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High performance Parallel and Distributed Computing*, HPDC '14, pages 239–252, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2749-7. doi: 10.1145/2600212.2600227. URL http://doi.acm.org/10.1145/2600212.2600227. 55, 61

[59] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: 10.1145/1772690.1772751. URL http://doi.acm.org/10.1145/1772690.1772751. 115, 130

[60] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012.

USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387884. 60

[61] R. Motwani L. Page, S. Brin and T. Winograd. The pagerank citation ranking: Bringing order to the web. In *Technical Report 1999-66, Stanford InfoLab*, November 1999. URL http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf. 8

[62] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization. *SIGPLAN Not.*, 44(4):101–110, February 2009. ISSN 0362-1340. doi: 10.1145/1594835.1504194. URL http://doi.acm.org/10.1145/1594835.1504194. 55, 61

[63] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol.*, 8(1):1:1–1:20, July 2016. ISSN 2157-6904. doi: 10.1145/2898361. URL http://doi.acm.org/10.1145/2898361. 46

[64] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012. ISSN 2150-8097. doi: 10.14778/2212351.2212354. URL http://dx.doi.org/10.14778/2212351.2212354. 56, 61, 108

[65] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807184. URL http://doi.acm.org/10.1145/1807167.1807184. 2, 58, 61, 108, 111

[66] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU Implementation of Inclusion-based Points-to Analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 107–116, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145831. URL http://doi.acm.org/10.1145/2145816.2145831. 47, 61

[67] Jaikrishnan Menon, Marc De Kruijf, and Karthikeyan Sankaralingam. iGPU: Exception Support and Speculative Execution on GPUs. In *Proceedings of the 39th Annual*

International Symposium on Computer Architecture, ISCA '12, pages 72–83, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4503-1642-2. URL http://dl.acm.org/citation.cfm?id=2337159.2337168. 55

[68] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *SIGPLAN Not.*, 47(8):117–128, February 2012. ISSN 0362-1340. doi: 10.1145/2370036.2145832. URL http://doi.acm.org/10.1145/2370036.2145832. 47

[69] Ulrich Meyer and Peter Sanders. Delta-Stepping: A Parallel Single Source Shortest Path Algorithm. In *Proceedings of the 6th Annual European Symposium on Algorithms*, ESA '98, pages 393–404, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-64848-8. URL http://dl.acm.org/citation.cfm?id=647908.740136. 15

[70] Technical University of Denmark Morten Stockel, Soren Bog. Concurrent datastructres(pages 37 to 51 ,bachelor thesis. Technical report. 69

[71] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Data-Driven Versus Topology-driven Irregular Computations on GPUs. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 463–474, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4971-2. doi: 10.1109/IPDPS.2013.28. URL http://dx.doi.org/10.1109/IPDPS.2013.28. 61

[72] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph Algorithms on GPUs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 147–156, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442531. URL http://doi.acm.org/10.1145/2442516.2442531. 61

[73] Rupesh Nasre, Martin Burtscher, and Keshav Pingali. Morph Algorithms on GPUs. *SIGPLAN Not.*, 48(8):147–156, February 2013. ISSN 0362-1340. doi: 10.1145/2517327.2442531. URL http://doi.acm.org/10.1145/2517327.2442531. 47, 95

[74] Jared Hoberock (NVIDIA) Nathan Bell (NVIDIA). Thrust: A Productivity-Oriented Library for CUDA. Technical report, October 2001. 70

[75] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, March 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500. URL http://doi.acm.org/10.1145/1365490.1365500. 76, 96

[76] Sreepathi Pai and Keshav Pingali. A compiler for throughput optimization of graph algorithms on gpus. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 1–19, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984015. URL http://doi.acm.org/10.1145/2983990.2984015. 53

[77] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The Tao of Parallelism in Algorithms. *SIGPLAN Not.*, 46(6):12–25, June 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993501. URL http://doi.acm.org/10.1145/1993316.1993501. 24, 41, 61, 99

[78] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. EigenCFA: Accelerating Flow Analysis with GPUs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 511–522, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926445. URL http://doi.acm.org/10.1145/1926385.1926445. 47, 61

[79] Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Elixir: A System for Synthesizing Concurrent Graph Programs. *SIGPLAN Not.*, 47(10):375–394, October 2012. ISSN 0362-1340. doi: 10.1145/2398857.2384644. URL http://doi.acm.org/10.1145/2398857.2384644. 3, 44, 61, 66

[80] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462176. URL http://doi.acm.org/10.1145/2491956.2462176. 55

[81] G. Ramalingam and Thomas Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996. 21, 105

[82] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY,

USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522740. URL http://doi.acm.org/10.1145/2517349.2522740. 46

[83] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1921-8. doi: 10.1145/2484838.2484843. URL http://doi.acm.org/10.1145/2484838.2484843. 60, 61

[84] Mehrzad Samadi, Amir Hormati, Janghaeng Lee, and Scott Mahlke. Paragon: Collaborative speculative loop execution on gpu and cpu. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 64–73, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1233-2. doi: 10.1145/2159430.2159438. URL http://doi.acm.org/10.1145/2159430.2159438. 55

[85] Ahmet Erdem Sariyüce, Kamer Kaya, Erik Saule, and Ümit V. Çatalyürek. Betweenness Centrality on GPUs and Heterogeneous Architectures. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 76–85, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2017-7. doi: 10.1145/2458523.2458531. URL http://doi.acm.org/10.1145/2458523.2458531. 47

[86] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. *GraphIn: An Online High Performance Incremental Graph Processing Framework*, pages 319–333. Springer International Publishing, Cham, 2016. 60

[87] Roman Shaposhnik, Claudio Martella, and Dionysios Logothetis. *Practical Graph Analytics with Apache Giraph*. Apress, Berkely, CA, USA, 1st edition, 2015. ISBN 1484212525, 9781484212523. 59, 61, 108

[88] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, February 2013. ISSN 0362-1340. doi: 10.1145/2517327.2442530. URL http://doi.acm.org/10.1145/2517327.2442530. 46

[89] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming, Vol. 2, Inter Process Communication (IPC)*. Prentice Hall, 2 edition, aug 1998. 33

[90] R. Rivest T. Cormen, C. Leiserson and editors C. Stein. Introduction to Algorithms. MIT Press, 2001. 8, 9, 69, 88, 116

[91] R. E Tarjan. Depth-first search and linear graph algorithms. pages 146–160, 1972. URL http://epubs.siam.org/doi/10.1137/0201010. 9

[92] Chen Tian, Min Feng, Vijay Nagarajan, and Rajiv Gupta. Copy or Discard Execution Model for Speculative Parallelization on Multicores. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 330–341, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2836-6. doi: 10.1109/MICRO.2008.4771802. URL http://dx.doi.org/10.1109/MICRO.2008.4771802. 46, 61

[93] Chen Tian, Changhui Lin, Min Feng, and Rajiv Gupta. Enhanced Speculative Parallelization via Incremental Recovery. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 189–200, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. doi: 10.1145/1941553.1941580. URL http://doi.acm.org/10.1145/1941553.1941580. 46, 61

[94] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 837–846, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-495-9. doi: 10.1145/1557019.1557111. URL http://doi.acm.org/10.1145/1557019.1557111. 9

[95] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8): 103–111, August 1990. ISSN 0001-0782. doi: 10.1145/79173.79181. URL http://doi.acm.org/10.1145/79173.79181. 2, 33, 58, 111

[96] Guozhang Wang, Wenlei Xie, Alan Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. 60

[97] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. *SIGPLAN Not.*, 51(8):11:1–11:12, February 2016. ISSN 0362-1340. doi: 10.1145/3016078.2851145. URL http://doi.acm.org/10.1145/3016078.2851145. 50

[98] D. F. Watson. Computing the n-dimensional delaunay tessellation with application to voronoi polytopes*. *The Computer Journal*, 24(2):167, 1981. doi: 10.1093/comjnl/24.2.167. URL +http://dx.doi.org/10.1093/comjnl/24.2.167. 23

[99] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596521979, 9780596521974. 60

[100] S. Xiao and W. c. Feng. Inter-block gpu communication via fast barrier synchronization. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, April 2010. doi: 10.1109/IPDPS.2010.5470477. 55, 94, 103

[101] 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*, 27(6):1647–1659, June 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2015.2453978. 60, 61

[102] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *2012 IEEE 12th International Conference on Data Mining*, pages 745–754, Dec 2012. doi: 10.1109/ICDM.2012.138. 130

[103] Kaiyuan Zhang, Rong Chen, and Haibo Chen. Numa-aware graph-structured analytics. *SIGPLAN Not.*, 50(8):183–193, January 2015. ISSN 0362-1340. doi: 10.1145/2858788.2688507. URL http://doi.acm.org/10.1145/2858788.2688507. 46

[104] Jianlong Zhong and Bingsheng He. Medusa: Simplified Graph Processing on GPUs. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1543–1552, June 2014. ISSN 1045-9219. doi: 10.1109/TPDS.2013.111. URL http://dx.doi.org/10.1109/TPDS.2013.111. 47, 61

[105] Andy Diwen Zhu, Xiaokui Xiao, Sibo Wang, and Wenqing Lin. Efficient single-source shortest path and distance queries on large graphs. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 998–1006, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2174-7. doi: 10.1145/2487575.2487665. URL http://doi.acm.org/10.1145/2487575.2487665. 8