Large Scale Graph Processing in a Distributed Environment

Nitesh Upadhyay(✉), Parita Patel, Unnikrishnan Cheramangalath, and Y. N. Srikant

Indian Institute of Science, Bangalore, India
niteshu@iisc.ac.in

Abstract. Large graphs are widely used in real world graph analytics. Memory available in a single machine is usually inadequate to process these graphs. A good solution is to use a distributed environment. Typical programming styles used in existing distributed environment frameworks are different from imperative programming and difficult for programmers to adapt. Moreover, some graph algorithms having a high degree of parallelism ideally run on an accelerator cluster. Error prone and lower level programming methods (memory and thread management) available for such systems repel programmers from using such architectures. Existing frameworks do not deal with the accelerator clusters.

We propose a framework which addresses the previously stated deficiencies. Our framework automatically generates implementations of graph algorithms for distributed environments from the intuitive shared memory based code written in a high-level Domain Specific Language (DSL), *Falcon*. The framework analyses the intermediate representation, applies a set of optimizations and then generates Giraph code for a CPU cluster and MPI+OpenCL code for a GPU cluster. Experimental evaluations show efficiency and scalability of our framework.

Keywords: Distributed architecture · Accelerator · Cross-platform Graph processing · DSL · Falcon

1 Introduction

Large scale graphs are generated and analyzed in various domains such as social networks, road networks, systems biology, and web graphs. Graph processing on a single machine becomes inefficient when the graph size exceeds the machine memory (due to high disk access latency). Graph processing is also inefficient because of irregular behaviour of graph algorithms. On the other hand, parallelism exhibited by graph algorithms improves performance [15]. To exploit the parallelism, modern distributed architectures such as multi-core CPU clusters and GPU clusters, are used.

In recent times, many frameworks for graph analytics in distributed environment such as Giraph [1], GraphLab [13], and PowerGraph [7], which target only

© Springer International Publishing AG, part of Springer Nature 2018
D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 465–477, 2018.
https://doi.org/10.1007/978-3-319-75178-8_38

multi-core CPU cluster, have been proposed. All these frameworks have their own specific unconventional programming style which is difficult for a programmer to comprehend and adopt.

To exploit the high degree of parallelism in graph algorithms, high performance compute resources such as GPUs are the most suitable targets. Moreover, large scale graph processing requires a distributed environment such as GPU clusters. Lower level APIs such as CUDA and OpenCL with Message Passing Interface (MPI) for communication among nodes facilitate programming in such environments. It is inconvenient for an amateur programmer to develop algorithms in such a language. A few challenges are summed up below.

– Thread management: Deciding the total number of threads and thread block size, and synchronizing threads.
– Memory management: Allocating and deallocating memory for a graph object and all its vertex and edge properties on GPU, copying a graph object to GPU from CPU and copying back the results.
– Debugging: Manual thread management and memory management make a program error prone and difficult to debug.
– Message passing: Deciding which data to communicate and to which nodes, preparing the data, and sending and receiving the data in appropriate buffers.
– Global variables: Absence of shared memory forces each node to keep a separate copy of each global variable and synchronize it whenever any node modifies its copy.

This paper presents a scalable framework which addresses the above discussed challenges of large scale graph processing in a distributed environment, i.e., CPU cluster and GPU cluster. The framework uses the front-end of Falcon DSL [18]. The framework traverses the Abstract Syntax Tree (AST) generated by Falcon [18], applies a set of optimizations and then generates Giraph [1] code for a CPU cluster and MPI+OpenCL code for a GPU cluster. Since our framework uses the constructs of Falcon [18], the programmer enjoys conventional, imperative, and shared memory programming style.

Our key contributions are as follows.

– We provide a multi-target code generator for any vertex-centric algorithm written in Falcon [18] that caters to CPU and GPU clusters.
– The framework analyses the DSL code and decides the graph object *properties* to be communicated, sends the messages, and synchronizes received message data with local data. Thus, it hides the complexity of message passing from the programmer.
– The framework applies a set of optimizations in order to minimize memory occupancy and communication latency.
– Experimental evaluations on CPU and GPU clusters shows scalability and efficiency of our framework.

2 Related Work

Green-Marl [9] and its extensions [10,16] target multi-core CPU, NVIDIA GPU and CPU cluster (Pregel [14]). Galois [12] provides C++ APIs to implement graph algorithms on a multi-core CPU. LonestarGPU [4] is a CUDA framework for graph algorithms and targets NVIDIA GPU. All of these DSLs and frameworks are either limited to a single node or do not target GPU cluster.

TOTEM [6] is a graph processing engine which targets hybrid architectures on a single node. It partitions the graph on multiple GPUs and CPU of a single node. Medusa [19] is a framework which generates graph algorithms implementations for multiple GPUs of a single node. Both, Totem and Medusa limit the graph size to the memory of the single node. Parallel Boost Graph Library [8] provides implementations of graph algorithms in a distributed environment. But it does not target GPU clusters.

There are many frameworks for graph processing in distributed environments, such as Pregel [14], Giraph [1], GraphLab [13], PowerGraph [7], GoFFish [17] etc. Pregel [14] and Giraph [1] adopt the Bulk Synchronous Parallel (BSP) model [5] and are scalable. Unlike Pregel [14], GraphLab [13] provides asynchronous and adaptive computation. It is suitable for graph algorithms where different parts of graph converge with dissimilar rates. PowerGraph [7] is an extended version of GraphLab [13]. It is desirable for natural graphs whose degree distribution follows a power law. GoFFish [17] is a sub-graph centric programming abstraction for distributed clusters. All these frameworks adopt unconventional programming models and target only CPU clusters. We generated code in Giraph because it offers high scalability. However, several other frameworks can also be targeted.

3 Background

3.1 Giraph

Giraph [1] is a scalable, efficient and fault-tolerant open-source implementation of Google's Pregel [14]. The collection of JAVA APIs available in Giraph [1] is useful to a programmer to implement graph algorithms on a Hadoop cluster. Giraph [1] has the following features.

- Bulk Synchronous Parallel model (BSP) [5] model: Giraph framework is built on the BSP model.
- Vertex-centric: Giraph algorithms are implemented in the form of *computation over vertices*. Algorithmic logic is written in a *single local compute function* (`BasicCompute.compute()`) which runs on every vertex in parallel. This function gets executed iteratively, and the program terminates when no communication happens and all vertices become inactive.

However, it poses the following challenges from a programming perspective.

– Programming in Giraph requires exclusive handling of *global variables* and *graph object properties*. A user must register them through an `Aggregator` in the global compute function (`MasterCompute.compute()`). Afterwards, these global variables and properties can be accessed and modified from both local and global functions.
– Since Giraph allows only a *single* `BasicCompute.compute()`, managing *multi-functions* is a difficult task. A global variable has to be maintained in the `MasterCompute.compute()` function. Depending on the value of this variable apt function is chosen in the *local compute function*.
– Programmer must explicitly implement *message type* through `Message` class and *vertex properties* through `Vertex` class.

```
1  class RBMC extends BasicCompute{
2   void compute(Vertex v,Message m){
3    int fun_call = getAggregatedValue
4    (RBMM.current_fun);
5    switch(fun_call){
6     case 0: shake1(); break;
7     case 1: shake2(); break;
8     case 2: shake3(); break;
9    }
10  }
11  void shake1(Vertex v,Message m){...}
12  void shake2(Vertex v,Message m){...}
13  void shake3(Vertex v,Message m){...
14  aggregate(RBMM.count,1);
15  ...}
16 }
17 class RBMM extends MasterCompute{
18  String count = "count";
19  String current_fun = "current_fun";
20  public void initialize(){
21   register(count);
22   register(current_fun);
23  }
24 }
```

Fig. 1. Giraph random bipartite matching code

Figure 1 shows the partial implementation of random bipartite matching in Giraph. This algorithm is a three-step handshake algorithm. In the first step, the left vertex sends a message to the right vertex. In second step, the right vertex accepts the message from any one of the left vertices randomly, and sends an acknowledgement. One of the edges is matched in the third step. Global variable `count`, which can be modified by any vertex, is registered in the `RBMMasterCompute` class. Also, *multi-functions* call is handled through `phase` aggregator. Its value gets modified in `MasterCompute.compute()` method which is not shown here. We have also omitted the implementations of `Vertex` and `Message` class, and other methods for brevity.

3.2 Falcon

Falcon [18], a graph DSL, extends C programming language and helps programmers to implement graph analysis algorithms intuitively. The front-end of the Falcon [18] compiler generates an AST. The back-end traverses the AST and generates OpenMP annotated C code for multi-core CPU and CUDA code for

NVIDIA GPU. Besides C data types, Falcon [18] has additional data types pertinent to graph algorithms such as *graph, vertex, edge, set* and *collection*. The user required to define parallelism explicitly through the parallel construct `foreach`.

4 Back-End of Our Framework

Our framework adopts the BSP model [5] where the computation of a graph algorithm occurs in a series of supersteps. Each superstep consists of the three following steps.

- Computation: Each node runs a *computation function* parallelly and independently.
- Communication: At the end of the computation, nodes communicate with each other.
- Synchronization: Each node synchronizes its local data with the received data.

Our framework uses the front-end of the Falcon compiler [18]. Front-end of Falcon parses the DSL code and generates an AST which is input to the back-end of our framework. Back-end compilation occurs in two phases. In the first phase, the AST is traversed to get some essential information such as the vertex or edge property to be communicated and the program location for the communication. In the second phase, optimizations are applied and efficient code is generated in Giraph for CPU cluster and MPI+OpenCL for heterogeneous cluster. The DSL programmer can enable or disable optimizations through command line switches.

Graph Storage and Partitioning for a Heterogeneous Cluster. Generated code stores the graph in Compressed Sparse Row (CSR) format. CSR format offers less storage overhead and favors memory coalescing compared to edge list and adjacency list representations. It keeps two arrays: `edges` and `indices`. `edges` array stores the destination vertices and weights of outgoing edges of each vertex. All the outgoing edges of each vertex are stored contiguously. `indices` array stores the index of the first outgoing edge (stored in `edges` array) of each vertex.

By default, our framework partitions the graph vertex-wise using the METIS tool [11]. A programmer can also use his/her own partitioning strategies. Each vertex belongs to a specific cluster node depending on its partition-id. Figure 2 shows graph partitioning on three nodes and graph storage at Node2 in CSR format. The input graph, which is shared among all nodes of a cluster, is read in parallel.

Multiple copies of a vertex property are stored in order to keep the vertex's property consistent across partitions. One *master copy* is stored at the node where the vertex belongs. Other copies (*duplicate copies*) reside at the nodes where the vertex is destination vertex of any inter-partition edge. As Fig. 2 suggests, the *master copies* of vertices v2 and v4 reside on Node2. Node2 also stores copies of vertices v3 and v5 whose *master copies* are stored at Node0 and Node1 respectively.

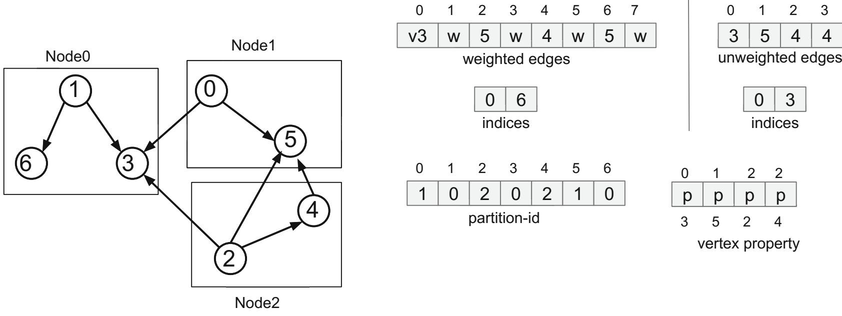

Fig. 2. Graph and its storage on Node2

4.1 Compilation for a GPU Cluster (MPI+OpenCL Code)

A heterogeneous cluster is composed of multiple nodes where each node can be any device such as CPU or accelerator, such as GPU, DSP, or FPGA. Usually, graph algorithms run on general purpose computing devices, i.e., CPU and GPU.

Typically, OpenMP is used to target the multi-core CPU nodes. CUDA is used to target the NVIDIA GPU nodes. But OpenCL is used to target both multi-core CPU and GPU devices of any vendor since it is an open, royalty-free, platform-agnostic and vendor-agnostic standard for programming on heterogeneous computation resources. Hence, we decided to generate code in OpenCL coupled with MPI which is a programming standard for a distributed architecture. Translations of some of the constructs are discussed below.

Parallel Regions: A Falcon programmer explicitly defines parallelism through the parallel construct `foreach`. Code enclosed in a `foreach` loop is translated to a kernel. The kernel will run in parallel on all local vertices. Since the execution of kernel comes under the *computation* step of the BSP model, it runs in parallel on all the nodes. Figure 3 shows the translation of `foreach` loop. `t` is a points iterator which iterates on all the vertices of the graph. Each vertex corresponds

```
1  //Falcon code
2  foreach(t In gr.points)
3   t.cc = Y;

1  //Giraph code
2  BasicCompute::compute(vertex){
3   if(getSuperStep==0)
4     vertex.getValue().cc = Y;
5  }
```

```
1  //OpenCL code
2  //kernel definition
3  __kernel void fun1(....){
4   int t_id = get_global_id(0);
5   cc[t_id+vertex_offset] = Y;
6  }
7  //kernel call
8  size = gr.no_of_part_vertices;
9  clEnqueueNDRangeKernel(..,&size,..);
```

Fig. 3. Falcon code and its equivalent generated Giraph and OpenCL code

to a thread, and all threads execute the loop body in parallel. The code enclosed in the `foreach` loop is translated to a kernel `fun1`. The kernel is called with the size of total number of vertices (lines 8–9).

Global Property and Variable: In a shared memory architecture, the scope of global variables and properties associated with a graph runs throughout the program. But in a distributed architecture, there is no memory which is accessible from all the nodes of the cluster. Hence, memory is allocated on all the nodes for all the global variables and properties. Whenever such a variable gets updated on any node, its value is broadcast to all the other nodes. If the operation applied on a global variable is `MIN`, `MAX`, `ADD` or `MUL`, then it can be applied in any order since these operations are associative and commutative. But, when `Assignment` operator is applied on a global variable, it gets written non-deterministically in any order.

Communication and Synchronization: In order to carry out communication in a distributed environment, a programmer needs to figure out data, program location, and source and destination nodes for the communication. Our framework automatically generates code to handle these issues from Falcon code. In the first phase of compilation, the AST is traversed to find out the required information for communication.

If any property is modified in a kernel, then that property needs to be communicated to the neighbours, and communication happens after the call to the kernel that modifies the property. Communication occurs between two nodes only if there is any inter-partition edge between them.

After the communication, received data is synchronized with local data based on the kind of operation applied on the property in the kernel code. This synchronization code is also generated by our framework.

4.2 Compilation for a CPU Cluster (Giraph Code)

Parallel Region: The code enclosed in a `foreach` loop is translated to the `BasicCompute.compute()` function of Giraph (as shown in Fig. 3). This function executes conceptually in parallel on all the vertices of graph.

Global Variables and Properties: Giraph offers `Aggregators` to handle global variables. Its value can be modified from either `BasicCompute.compute()` or `MasterCompute.compute()` function. `MasterCompute.compute()` function gets executed in the beginning of every superstep. Graph properties or global variables in Falcon are mapped to the `Aggregators` of Giraph. The first phase of compilation determines the type of `Aggregator` based on the type of the operation applied on global variable or graph object property. Table 1 depicts the operations and their equivalent `Aggregators`.

Table 1. Operations and their corresponding Aggregators

Expression	Type of Aggregator
MAX(gr.prop,k,change)	MaxAggregator
MIN(gr.prop,k,change)	MinAggregator
ADD(gr.prop,k)	SumAggregator
MUL(gr.prop,k)	ProductAggregator
gr.prop = k	OverwriteAggregator

Multi-functions: When multiple functions are called through parallel constructs in Falcon, they all are mapped to the same `BasicCompute.compute()` function. Our framework assigns numbers to all functions according to their calling order. Our framework also uses a `current_fun` variable (of type enum) to keep track of which function is being executed. Since `current_fun` is a global variable, it is modified through the `SumAggregator`. The `MasterCompute.compute()` function modifies the `current_fun` variable appropriately in the beginning of every superstep. Subsequently, `BasicCompute.compute()` function selects the function based on the value of `current_fun` variable.

Communication and Synchronization: Communication and synchronization are managed as stated in the GPU cluster compilation (Sect. 4.1).

5 Optimizations

5.1 Execute Only Active Vertices

Programmers often tend to write simple and unoptimized parallel code for traversal based graph algorithms that underperform and waste resources. However, better manual or optimized Falcon code can also be written.

In traversal-based graph algorithms, all the vertices do not remain active in each iteration. In the generated code, active vertices can be tracked and kernel can be executed only on those vertices. In order to achieve this, the compiler associates a boolean property `is_Active` with each vertex that stores the status of the vertex. Initially, all the vertices have their `is_Active` flags set to false except for the starting point. Whenever a vertex's property gets updated, the status of that vertex is modified to active. When the execution of an active vertex is finished, its status is changed to inactive (false). The execution time of each iteration gets reduced because the number of active vertices is less than the total number of vertices. As a result, total execution time of the algorithm gets reduced significantly.

5.2 Discarding the Weight if Not Required

When a weighted graph is given as input to an algorithm that does not use the weight of an edge, storing the weight of edges does not serve any purpose.

Falcon uses the function `getWeight()` to access the weight of an edge. While traversing the AST, our compiler notes whether this function is called anywhere or not. Subsequently, this information is propagated to the code generator. The code generator generates code with the graph storage format accordingly and also alters the `READ` function appropriately. $|V| + 2 * |E|$ units of memory are required to store the graph G(V,E) in weighted CSR format while unweighted CSR format requires only $|V| + |E|$ units of memory. Our compiler optimization saves $|E|$ units of memory when the algorithm does not use weight of an edge. Thus larger graph can be accommodated on the given limited amount of memory. Furthermore, unweighted CSR format stores neighbors contiguously, unlike weighted CSR, where neighbors are stored alternatively. Thereby, it provides better cache locality and improves execution time.

5.3 Communicate Selective Data

Vertex's property has multiple copies as discussed in Sect. 4. One *master copy* is stored at the node where the vertex belongs. Other copies reside at the nodes where the vertex is the destination vertex of any inter-partition edge (referred here as *duplicate copy*). Whenever any *duplicate copy* is updated, that must be communicated to the *master copy* in order to preserve consistency. Some algorithms such as Pagerank update the *duplicate copy* in every iteration. However, other algorithms such as Single Source Shortest Path (SSSP) or Breadth First Search (BFS) do not update each *duplicate copy* in each iteration. Communicating all *duplicate copies* in each iteration is costly. Our optimization algorithm determines the updated duplicate copies and sends out only these. The compiler creates an extra copy of the property which stores previous value and compares it with the current value of the property to find out whether it is updated or not.

If an algorithm requires update of almost every *duplicate copy* in each iteration, then this optimization will not add any value and ends up performing slower than the unoptimized one (as illustrated in Table 4). Hence, this optimization has been made optional (through a command line switch).

6 Experimental Evaluations

Our framework generates Giraph code from Falcon code for a CPU cluster. It generates MPI+OpenCL for a heterogeneous cluster (OpenCL version 1.1). Table 2 shows the number of lines of code handwritten for Falcon, Giraph and MPI+OpenCL. Experiments on a GPU cluster have been carried out for five algorithms: BFS, SSSP, WCC, PR and RBM (Table 2). We used supercomputer Cray XC40 for experiments of the GPU cluster, where each node is having a NVIDIA Tesla K40 clocked at 706 MHz, with 2880 cores and 12 GB global memory. GPU cluster code is compiled with NVCC version 7.5. Experiments on CPU cluster are done for five algorithms: BFS, SSSP, PR, RBM and K-Core (Table 2). Our framework can generate code for any vertex-centric algorithm. An algorithm involving message-pulling can be indirectly supported in Giraph by storing all

the in-neighbours of vertices. Graph mutation for GPU clusters is not supported in our framework. CPU cluster used for experiments consists of AMD CPUs, running Hadoop version 2.6.0. Each AMD CPU is an Opteron 6376 clocked at 1.40 GHz, with 8 cores and 32 GB RAM. K-Core algorithm mutates the graph structure and our framework for hybrid cluster does not support mutation of the graph. WCC algorithm is not implemented for CPU cluster because it involves message pulling, and Giraph does not directly support it.

Table 2. Graph algorithms and their lines of code [2]

Graph algorithm	Falcon	Giraph	MPI+OpenCL
Breadth First Search (BFS)	21	81	499
Single Source Shortest Path (SSSP)	21	81	310
Weakly Connected Components (WCC)	49	-	300
Pagerank (PR)	25	60	263
Random Bipartite Matching (RBM)	45	147	360
K-Core	32	82	-

Table 3. Graph inputs used in experiments

Graph	Type	No of nodes (in millions)	No of edges (in millions)
RD-1	Random	64	256
RD-2	Random	128	512
RM-1	Scale free (R-MAT)	60	300
RM-2	Scale free (R-MAT)	80	400
BP-1	Bipartite	64	256
BP-2	Bipartite	128	512
BP-3	Bipartite	256	1024

Table 3 shows the input graphs used for experiments. Random and R-MAT graphs have been generated using the GTgraph tool [3]. We synthesized bipartite graphs using the random function.

Figure 4 shows experimental evaluations on a GPU cluster with two, four and eight nodes. Time measured here for all the benchmarks includes only the computation and communication times after distributing the graph to the nodes. Speedup is shown with respect to two nodes. Figure 5 shows experimental evaluations on a CPU cluster with four, six, eight and ten nodes. Speedup is shown with respect to four nodes. Figures 4 and 5 show scalability of our framework. The scalability is not linear because with the increase in number of nodes of

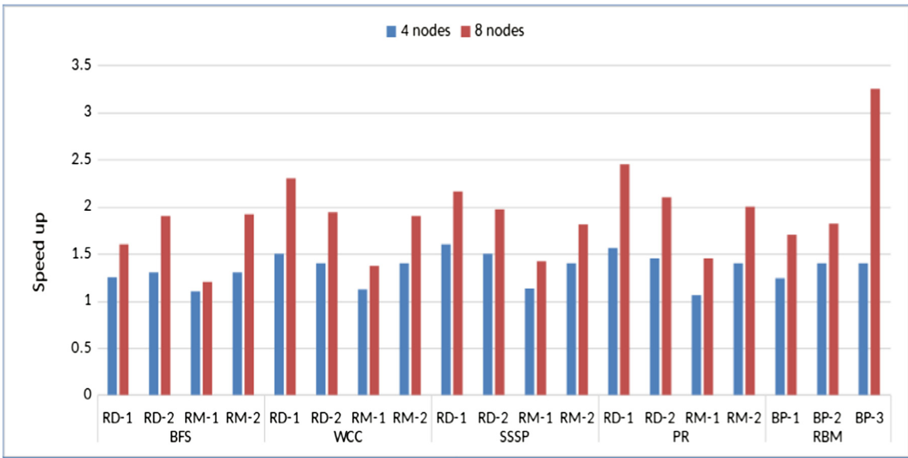

Fig. 4. Speedup over 2 nodes (MPI+OpenCL)

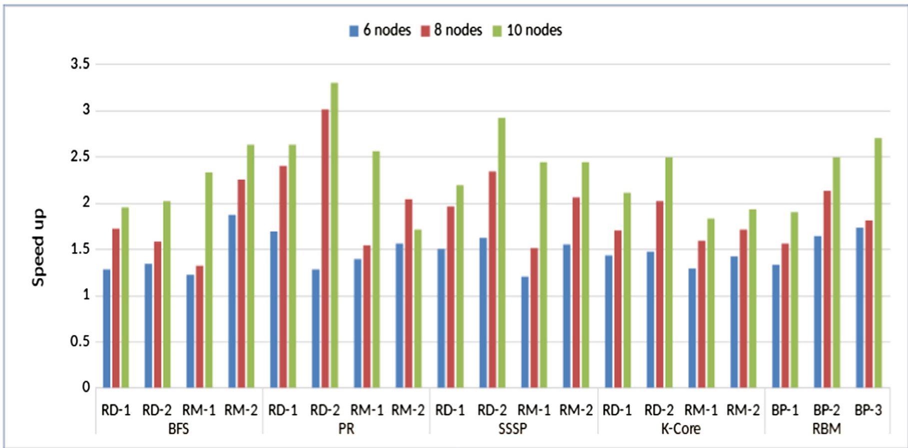

Fig. 5. Speedup over 4 nodes (Giraph)

cluster, the computation time decreases but the communication time increases. Also, we have compared compiler generated code with manual implementations of the above mentioned algorithms for both CPU and heterogeneous cluster and found that they perform similarly.

Table 4 shows the execution time for two implementations of the SSSP and the PR algorithms, one with *communicate-selective-data* optimization and the other without it.

Table 4. Execution time (in seconds) for generated optimized and unoptimized SSSP and PR algorithms on 2, 4 and 8 nodes of a GPU cluster

Graph	Optimized SSSP			Unoptimized SSSP			Optimized PR			Unoptimized PR		
	2	4	8	2	4	8	2	4	8	2	4	8
RD-1	14	11	8	90	57	41	144	105	78	142	95	62
RD-2	29	23	16	164	110	83	296	219	157	292	203	130
RM-1	11	11	9	49	44	35	110	182	109	103	108	78
RM-2	22	17	12	87	64	49	208	176	139	194	158	102

7 Conclusion

We proposed a framework for large scale graph processing on a distributed environment. It reuses the front-end of Falcon and generates Giraph implementations for a CPU cluster and MPI+OpenCL code for a heterogeneous cluster. Empirical results show scalability and efficiency of our framework.

References

1. Apache giraph project. http://giraph.apache.org
2. Distributed-falcon. https://github.com/niteshupadhyay/distributed-framework/
3. Bader, D.A., Madduri, K.: GTgraph: a synthetic graph generator suite. Atlanta, GA, February 2006
4. Burtscher, M., Nasre, R., Pingali, K.: A quantitative study of irregular programs on GPUs. In: Workload Characterization (IISWC), pp. 141–151. IEEE (2012)
5. Gerbessiotis, A.V., Valiant, L.G.: Direct bulk-synchronous parallel algorithms. In: Nurmi, O., Ukkonen, E. (eds.) SWAT 1992. LNCS, vol. 621, pp. 1–18. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55706-7_1
6. Gharaibeh, A., Reza, T., Santos-Neto, E., Costa, L.B., Sallinen, S., Ripeanu, M.: Efficient large-scale graph processing on hybrid CPU and GPU systems. arXiv preprint arXiv:1312.3018 (2013)
7. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: distributed graph-parallel computation on natural graphs. In: OSDI, vol. 12, p. 2 (2012)
8. Gregor, D., Lumsdaine, A.: The parallel BGL: a generic library for distributed graph computations. In: POOSC (2005)
9. Hong, S., Chafi, H., Sedlar, E., Olukotun, K.: Green-Marl: a DSL for easy and efficient graph analysis. In: ACM SIGARCH Computer Architecture News, vol. 40, pp. 349–362. ACM (2012)
10. Hong, S., Salihoglu, S., Widom, J., Olukotun, K.: Simplifying scalable graph processing with a domain-specific language. In: CGO. ACM (2014)
11. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1), 359–392 (1998)
12. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L.P.: Optimistic parallelism requires abstractions. ACM SIGPLAN Not. **42**(6), 211–222 (2007)

13. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (2012)
14. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD ICMD, pp. 135–146. ACM (2010)
15. Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M.A., Kaleem, R., Lee, T.H., Lenharth, A., Manevich, R., Méndez-Lojo, M., et al.: The tao of parallelism in algorithms. ACM Sigplan Not. **46**(6), 12–25 (2011)
16. Shashidhar, G., Nasre, R.: `LightHouse`: an automatic code generator for graph algorithms on GPUs. In: Ding, C., Criswell, J., Wu, P. (eds.) LCPC 2016. LNCS, vol. 10136, pp. 235–249. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52709-3_18
17. Simmhan, Y., Kumbhare, A., Wickramaarachchi, C., Nagarkar, S., Ravi, S., Raghavendra, C., Prasanna, V.: *GoFFish*: a sub-graph centric framework for large-scale graph analytics. In: Silva, F., Dutra, I., Santos Costa, V. (eds.) Euro-Par 2014. LNCS, vol. 8632, pp. 451–462. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09873-9_38
18. Unnikrishnan, C., Nasre, R., Srikant, Y.: Falcon: a graph manipulation language for heterogeneous systems. ACM TACO **12**(4), 54:1–54:27 (2016)
19. Zhong, J., He, B.: Medusa: simplified graph processing on GPUs. TPDS **25**(6), 1543–1552 (2014)